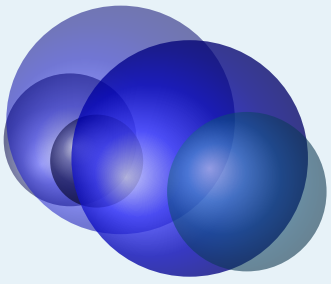
tkz-elements 5.00c

AlterMundus

tkz-elements 5.00c

Euclidean Geometry



Alain Matthes

February 17, 2026 Documentation V.5.00c

http://altermundus.fr

tkz-elements

AlterMundus

Alain Matthes

This document gathers notes on tkz-elements, a comprehensive **Lua** library designed to handle all necessary computations for constructing objects in Euclidean geometry. The document must be compiled with LuaLaTeX.

With tkz-elements, all definitions and calculations are carried out exclusively in Lua, following an object-oriented programming model. Core geometric entities such as points, lines, triangles, and conics are implemented as object classes.
Once the computations are complete, graphical rendering is typically handled using tkz-euclide, although you may also use TikZ directly if desired.

I discovered Lua and object-oriented programming while developing this package, so it's possible that some parts could be improved. If you'd like to contribute or offer suggestions, feel free to contact me by email.

☞ Acknowledgements: I received much valuable advice, remarks, and corrections from Nicolas Kisselhoff, David Carlisle, Roberto Giacomelli and Qrrbrbirlbel.
Special thanks to Wolfgang Büchel for his invaluable contribution in correcting the examples.

☞ I would also like to extend my gratitude to Eric Weisstein, creator of MathWorld.

☞ You can find some examples on my site and a french documentation: altermundus.fr.

Overview

Attributes Overview

Class	Attributes
point	see table
line	see table
triangle	see table
circle	see table
occs	see table
conic	see table
quadrilateral	see table
square	see table
rectangle	see table
parallelogram	see table
regular	see table
vector	see table
matrix	see table
angle	see table
list_point	see table

Methods Overview

Class	Methods
point	see table
line	see table 1
line	see table 2
triangle	see table
circle	see table
circle	see table 2
occs	see table
conic	see table
path	see table
quadrilateral	see table
square	see table
rectangle	see table
parallelogram	see table
regular	see table
vector	see table
matrix	see table
angle	see table
list_point	see table
fct	see table
pfct	see table

Metamethods Overview

Class	Methods
point	see table
occs	see table
vector	see table
matrix	see table
path	see table

Contents

Part I.

Geometry Core

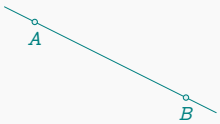
1. Getting started

1.1. The first code

A quick introduction to get you started. We assume that the packages tkz-euclide and tkz-elements are installed. Compile the following code using the **lualatex** engine; you should obtain a straight line passing through points A and B.

Note. The package tkz-elements performs all geometric definitions and computations in Lua, while tkz-euclide is mainly used for drawing. For this reason, the mini option of tkz-euclide is recommended. If a compilation problem occurs, simply load tkz-euclide without this option.

```
% !TEX TS-program = lualatex
\documentclass{article}
\usepackage[mini]{tkz-euclide}
\usepackage{tkz-elements}
\begin{document}
\directlua{
  init_elements()
  z.A = point(0, 1)
  % or z.A = point:new(0, 1)
  z.B = point(2, 0)
}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawLine(A,B)
 \tkzDrawPoints(A,B)
 \tkzLabelPoints(A,B)
\end{tikzpicture}
\end{document}
```



1.2. Key points

The following points are essential for most of the codes in this documentation.

- % !TEX TS-program = lualatex. Compilation must be performed with **lualatex**. This line is optional if your editor is already configured accordingly.
- \usepackage[mini]{tkz-euclide}. The use of tkz-euclide is optional; however, if it is loaded, the **mini** option is recommended. It loads only the drawing macros, while all computations are handled by **lua**. **Important:** at the current stage, some drawing macros are not yet fully independent of computation macros. If a compilation problem occurs, simply load tkz-euclide without the **mini** option.
- \usepackage{tkz-elements}. This package is required. It provides the Lua-based geometry engine and support macros used together with tkz-euclide.
- \directlua{...}. All geometric definitions and computations are placed inside this macro (or within a tkzelements environment).
- init_elements(). This function should be called at the beginning of each Lua section. It resets internal tables and clears global variables used by tkz-elements.
- \tkzGetNodes. This macro must be placed at the beginning of the **tikzpicture** environment. It transfers the points defined in Lua to TikZ as usable nodes.

1.3. Testing

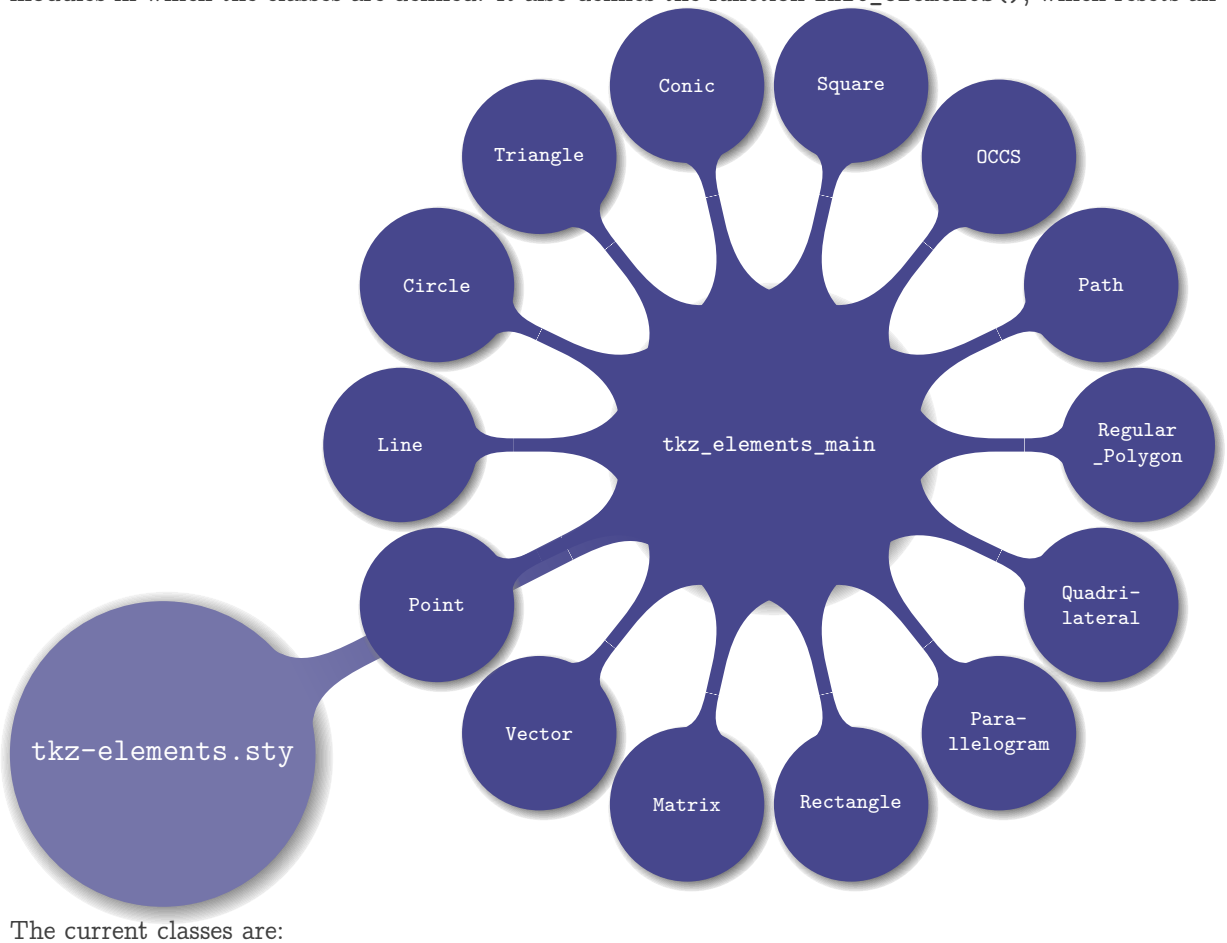
To test your installation and follow the examples in this documentation, you need to load two packages: tkz-euclide and tkz-elements. The first package automatically loads TikZ, which is necessary for all graphical rendering.

The **Lua** code is provided as an argument to the **\directlua** macro. We will often refer to this code block as the **Lua part** [1]. This part depends entirely on the tkz-elements package.
A crucial component in the **tikzpicture** environment is the macro **\tkzGetNodes**. This macro transfers the points defined in **Lua** to TikZ by creating the corresponding nodes. All such points are stored in a table named **z** and are accessed using the syntax **z.label**. These labels are then reused within tkz-euclide.
When you define a point by assigning it a label and coordinates, it is internally represented as a complex number — the affix of the point. This representation allows the point to be located within an orthonormal Cartesian coordinate system.
If you want to use a different method for rendering your objects, this is the macro to modify. For example, Section 37 presents **\tkzGetNodesMP**, a variant that enables communication with **MetaPost**.
Another essential element is the use of the function **init_elements()**, which clears internal tables[2] when working with multiple figures.
If everything worked correctly with the previous code, you're ready to begin creating geometric objects. Section 10 introduces the available options and object structures.
Finally, it is important to be familiar with basic drawing commands in tkz-euclide, as they will be used to render the objects defined in **Lua**.

1 This code can also be placed in an external file, e.g., file.lua.
2 All geometric objects are stored in Lua tables. These tables must be cleaned regularly, especially when creating multiple figures in sequence.

2. Structure

The package defines two macros **\tkzGetNodes** and **\tkzUseLua**.
Additionally, the package loads the file tkz_elements_main.lua. This file initializes all the tables used by the modules in which the classes are defined. It also defines the function **init_elements()**, which resets all tables.



The current classes are:
point (z), **circle** (C), **line** (L), **matrix** (M), **occs** (O), **parallelogram** (P), **quadrilateral** (Q), **rectangle** (R), **square** (S), **triangle** (T), **vector** (V), **conic** (CO), **regular_polygon** (RP) and **path** (PA).

The variable in parentheses indicates the name typically assigned to the table containing objects of the corresponding class.
If **name** refers to a class, its definition can be found in the file **tkz_elements_name.lua**.

3. Why tkz-elements?

The tkz-elements package was created to address two key challenges in geometric work with TeX. First, TeX alone lacks the numerical precision and flexibility required for complex geometric calculations. By leveraging **Lua**, tkz-elements provides accurate and stable computations for operations such as intersections, projections, and transformations.
Second, the package introduces an object-oriented approach to geometry. Geometric entities—such as points, vectors, lines, circles, and conics—are modeled as structured objects with associated methods. This allows users to express geometric relationships in a more natural and readable way, similar to mathematical language.
Beyond these two core ideas, tkz-elements offers several additional benefits:

- Simplified computations: **Lua**'s floating-point arithmetic and expressive syntax make geometric calculations much easier to write and understand than in pure TeX.
- Extensibility: Thanks to its modular design and use of classes, new geometric objects and methods can be added with minimal effort, making the package scalable and adaptable.
- Interoperability: Calculated points and data integrate smoothly with Ti*k*Z or other TeX drawing tools, allowing users to combine computation and visualization effectively.
- Pedagogical clarity: With its structured and expressive syntax, tkz-elements is well-suited for educational purposes, where clarity of construction and notation is essential.

Altogether, tkz-elements provides a modern, object-oriented, and Lua-powered framework for geometric constructions within the TeX ecosystem.

3.1. Calculation accuracy

3.1.1. Calculation accuracy in TikZ

With Ti*k*Z, the expression `veclen(x,y)` calculates the expression $\sqrt{x^2+y^2}$. This calculation is achieved through a polynomial approximation, drawing inspiration from the ideas of `Rouben Rostamian`.

```
pgfmathparse{veclen(65,72)} \pgfmathresult
```

☞ $\sqrt{65^2+72^2} \approx 96.9884$.

3.1.2. Calculation accuracy in Lua

A `luaveclen` macro can be defined as follows:

```
\def\luaveclen#1#2{\directlua{tex.print(string.format(
'\percentchar.5f',math.sqrt((#1)*(#1)+(#2)*(#2))))}}
```

and

```
\luaveclen{65}{72}
```

gives:
☞ $\sqrt{65^2+72^2} = 97$!!

The error, though insignificant when it comes to the placement of an object on a page by a hundredth of a point, becomes problematic for the results of mathematical demonstrations. Moreover, these inaccuracies can accumulate and lead to erroneous constructions.

To address this lack of precision, I initially introduced the fp, followed by the package xfp. More recently, with the emergence of LuaLaTeX, I incorporated a **Lua** option aimed at performing calculations with **Lua**.
This was the primary motivation behind creating the package, with the secondary goal being the introduction of object-oriented programming (OOP) and simplifying programming with Lua. The concept of OOP persuaded me to explore its various possibilities further.

At that time, I had received some Lua programming examples from tkzPoptNicolas Kisselhoff, but I struggled to understand the code initially, so I dedicated time to studying Lua patiently. Eventually, I was able to develop **tkz-elements**, incorporating many of his ideas that I adapted for the package.

3.1.3. Using objects

Subsequently, I came across an article by Roberto Giacomelli[3] on object-oriented programming using **Lua** and TikZ tools. This served as my second source of inspiration. Not only did this approach enable programming to be executed step-by-step, but the introduction of objects facilitated a direct link between the code and geometry. As a result, the code became more readable, explicit, and better structured.

3.1.4. Example: Apollonius circle (new version 2025/05/12)

> Problem: The objective is to identify an inner tangent circle to the three exinscribed circles of a triangle.

For additional information, you can consult the corresponding entry on MathWorld. This example was used as a reference to test the tkz-euclide package. Initially, the results obtained with basic methods and available tools lacked precision. Thanks to tkz-elements, more powerful and accurate tools are now available — and they are also easier to use. The core principles of figure construction with tkz-euclide remain unchanged: definitions, calculations, drawings, and labels, all following a step-by-step process that mirrors classical compass-and-straightedge constructions. This version takes advantage of the simplest construction method enabled by **Lua**.

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(0.8, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 z.N = T.ABC.eulercenter
 z.S = T.ABC.spiekercenter
 T.feuerbach = T.ABC:feuerbach()
 z.Ea, z.Eb, z.Ec = T.feuerbach:get()
 T.excentral = T.ABC:excentral()
 z.Ja, z.Jb, z.Jc = T.excentral:get()
 C.JaEa = circle(z.Ja, z.Ea)
% C.ortho = circle:radius(z.S, math.sqrt(C.JaEa:power(z.S)))
 C.ortho = C.JaEa:orthogonal_from(z.S) % 2025/05/12
 z.a = C.ortho.through
 C.euler = T.ABC:euler_circle()
 C.apo = C.ortho:inversion(C.euler)
 z.O = C.apo.center
 z.xa, z.xb, z.xc = C.ortho:inversion(z.Ea, z.Eb, z.Ec)}
```

Creating an object involves defining its attributes and methods — that is, its properties and the actions it can perform. Once created, the object is associated with a name (or reference) by storing it in a table. This table acts as an associative array, linking a **key** (the reference) to a **value** (the object itself). These concepts will be explored in more detail later.
For instance, suppose **T** is a table that associates the triangle object with the key **ABC**. Then **T.ABC** refers to another table containing the attributes of the triangle — such as its vertices, sides, or angles — each accessible via a specific key. These attributes are predefined within the package to support geometric operations.

```
z.N = T.ABC.eulercenter
```

3 Grafica ad oggetti con LuaTEX

N is the name of the point, **eulercenter** is an attribute of the triangle. [4]

```
T.excentral = T.ABC:excentral()
```

In this context, **excentral** is a method associated with the **T.ABC** object. It defines the triangle formed by the centers of the exinscribed circles.

Deprecated Code

Note: This code has been replaced by a more elegant version.
Two lines were particularly noteworthy. The first demonstrates how the exceptional precision of **Lua** allows a radius to be defined through a complex computation.
The radius of the radical circle is given by:

$$\sqrt{\Pi(S,C(Ja,Ea))}$$

(the square root of the power of the point S with respect to the exinscribed circle centered at Ja and passing through Ea).

```
C.ortho  = circle: radius (z.S,math.sqrt(C.JaEa: power(z.S)))
```

Revised Code

Note: The following code replaces the previous version with a more elegant and object-oriented approach. The previous code demonstrated the kinds of calculations that can be performed manually. However, tkz-elements offers a wide range of methods associated with the various geometric objects. Among the methods related to circles is one that allows you to define a circle orthogonal to another, given a center point.
We want to obtain the circle orthogonal to the three exinscribed circles of the triangle. Its center (S) is the radical center of these three circles. It suffices to compute a circle orthogonal to one of them [a], taking as center the point S.
The second important line performs an inversion with respect to this orthogonal circle.

```
C.ortho  = C.JaEa:orthogonal_from(z.S)
```

a Given a main circle and two secondary circles, a specific method exists for defining the radical circle orthogonal to all three.

Lastly, it's worth noting that the inversion of the Euler circle with respect to the radical circle yields the Apollonius circle.[5]
This transformation requires an object as a parameter. The method automatically detects the object type (as all objects in the package are typed) and selects the appropriate algorithm accordingly.

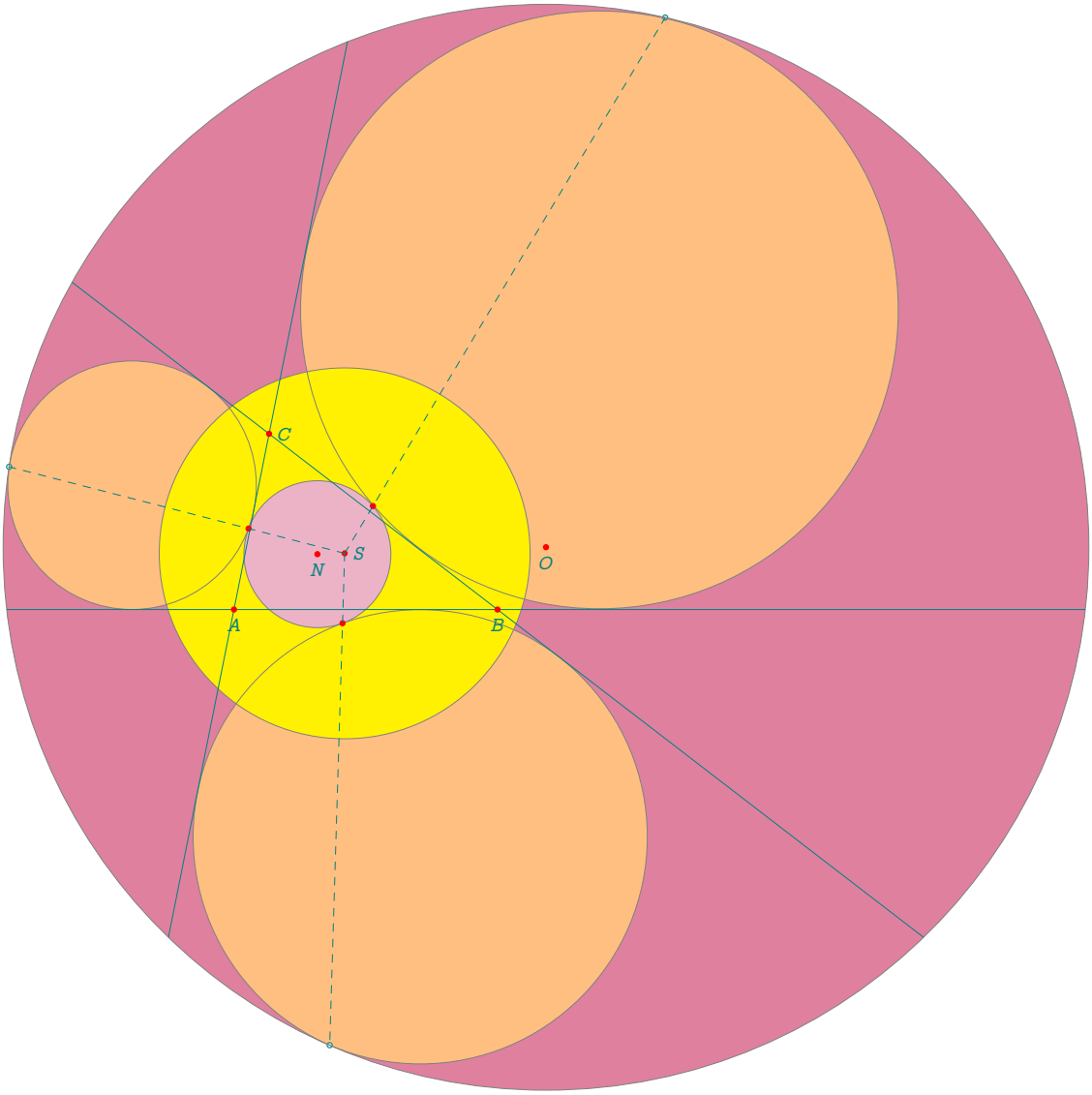
```
C.apo   = C.ortho:inversion(C.euler)
```

Now that all relevant points have been defined, it is time to begin drawing the geometric paths. To do this, the corresponding nodes must first be created. This is precisely the role of the macro **\tkzGetNodes**.
The subsequent section exclusively deals with drawings, and is managed by tkz-euclide.

```
\begin{tikzpicture}
   \tkzGetNodes
   \tkzFillCircles[green!30](O,xa)
   \tkzFillCircles[teal!30](Ja,Ea Jb,Eb Jc,Ec)
```

4 The center of the Euler circle, or center of the nine-point circle, is a characteristic of every triangle.
5 The nine-point circle, also known as Euler's circle, is externally tangent to the three exinscribed circles. The points of tangency form the Feuerbach triangle.

```
   \tkzFillCircles[lightgray](S,a)
   \tkzFillCircles[green!30](N,Ea)
   \tkzDrawPoints(xa,xb,xc)
   \tkzDrawCircles(Ja,Ea Jb,Eb Jc,Ec S,a O,xa N,Ea)
   \tkzClipCircle(O,xa)
   \tkzDrawLines[add=3 and 3](A,B A,C B,C)
   \tkzDrawPoints(O,A,B,C,S,Ea,Eb,Ec,N)
   \tkzDrawSegments[dashed](S,xa S,xb S,xc)
   \tkzLabelPoints(O,N,A,B)
   \tkzLabelPoints[right](S,C)
\end{tikzpicture}
```



4. Presentation

4.1. Geometric Construction Philosophy

tkz-elements is built around the principles of classical Euclidean geometry, emphasizing constructions achievable with an unmarked straightedge and a compass. The library favors a declarative and geometric approach over algebraic or numeric definitions.
This philosophy means that all geometric objects — points, lines, circles, triangles, and so on — are primarily defined by other points. Explicit numerical values such as coordinates, distances, or angles are deliberately avoided unless they are essential to the construction. This is in line with a traditional ruler-and-compass mindset, where geometric reasoning emerges from visual configurations and not from numbers.
For example:

- A circle is defined by its center and a point on the circumference,
- A perpendicular bisector is constructed using midpoint and symmetry,
- An angle is inferred from three points rather than given as a numeric value.

This design decision also explains why constructors based on scalar values, such as OCCS (One Center and a Scalar), are not part of the default interface. Their use would introduce numerical dependency at odds with the intended geometric abstraction.

While tkz-elements promotes a high-level, point-based approach to geometry, it relies internally on **Lua** for performing all necessary calculations. Lua is not exposed to the user as a scripting tool, but rather serves as a powerful computational engine that enables precise and robust geometric operations behind the scenes.
This separation of concerns allows users to focus entirely on geometric constructions while benefiting from Lua's computational precision and performance. In the next section, we will explain how Lua integrates with LaTeX in the context of this package — not as a programming layer, but as an invisible partner enabling advanced constructions.

4.2. With Lua

The primary purpose of tkz-elements is to perform geometric computations and define points using **Lua**. You can think of it as a computational kernel that can be used by either tkz-euclide or directly with TikZ.
Lua code can be executed in two ways: directly, using the **\directlua** primitive, or within a **tkzelements** environment based on the luacode package (which must be loaded separately). When using **\directlua**, especially in complex documents, you can reset internal data structures — such as coordinate tables and scaling factors — using the **init_elements()** function.

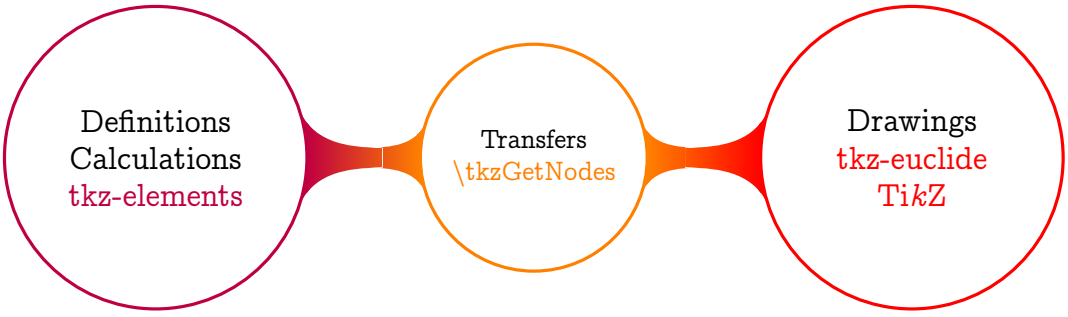
The key points are:

- The source file must be ☞ UTF-8 encoded.
- Compilation must be performed with ☞ LuaLaTeX.
- You need to load tkz-euclide and **tkz-elements**.
- All definitions and calculations are carried out in an **occs** (orthonormal cartesian coordinate system,) using Lua either via the macro **\directlua** or within the **tkzelements** environment.

On the right, you will find a minimal template.
The code is divided into two main parts: the Lua code (executed using **\directlua** or placed inside a **tkzelements** environment), and the **tikzpicture** environment, where drawing commands — typically from tkz-euclide — are issued.
When using tkz-euclide, the **mini** option is recommended, as it loads only the macros required for drawing. However, at the current stage, some drawing macros are not yet completely independent of computation macros; if a compilation problem occurs, simply load tkz-euclide without this option.
Within the **Lua** section, it is best practice to systematically call the **init_elements()** function. This function resets internal tables and clears internal data structures. **Important:** from this point on, all scaling operations must be avoided in the **Lua** code.

```
% !TEX TS-program = lualatex
% Created by Alain Matthes
\documentclass{standalone}
\usepackage[mini]{tkz-euclide}
% or simply TikZ
\usepackage{tkz-elements}
begin{document}

\directlua{
init_elements()
% definition of some points
z.A = point(   ,   )
z.B = point(   ,   )
% or
z.A = point:new(   ,   )
z.B = point:new(   ,   )

 ...code...
}

\begin{tikzpicture}
% points transfer to Nodes
% from Lua to tikz (tkz-euclide)
\tkzGetNodes

\end{tikzpicture}
\end{document}
```

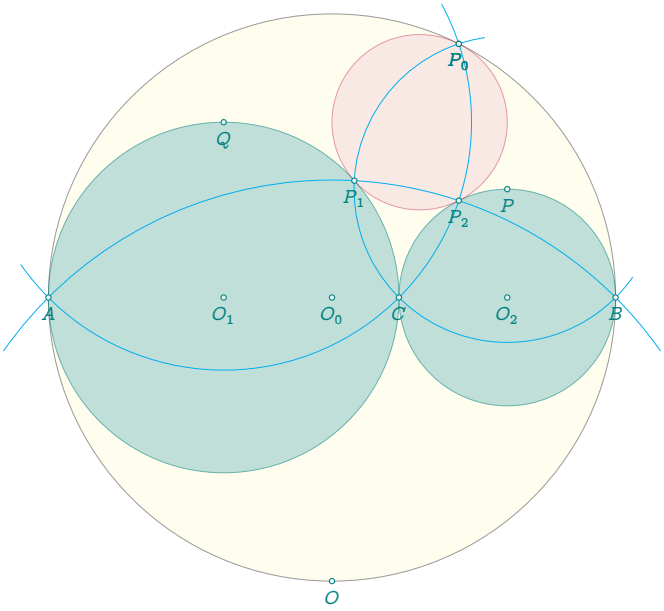
4.3. The main process



After obtaining all the necessary points for the drawing, they must be transformed into **nodes** so that TikZ or tkz-euclide can render the figure. This is accomplished using the macro **\tkzGetNodes**. This macro iterates through all the elements of the table **z** using the key (which is essentially the name of the point) and retrieves the associated values, namely the coordinates of the point (node).

4.4. Complete example: Pappus circle

4.4.1. The figure



4.4.2. The code

```
% !TEX TS-program = lualatex
\documentclass{article}
\usepackage[mini]{tkz-euclide} % mini = only tracing function
\usepackage{tkz-elements}
\begin{document}

\directlua{
  init_elements()  % Clear tables
  z.A = point(0, 0)
  z.B = point(10, 0)  %  creation of two fixed points $A$ and $B$
  L.AB = line(z.A, z.B)
  z.C = L.AB:gold_ratio()       %  use of a method linked to "line"
  z.O_0 = line(z.A, z.B).mid % midpoint of segment with an attribute of "line"
  z.O_1 = line(z.A, z.C).mid %  objects are not stored and cannot be reused.
  z.O_2 = line(z.C, z.B).mid
  C.AB = circle(z.O_0, z.B)  %  new object "circle" stored and reused
  C.AC = circle(z.O_1, z.C)
  C.CB = circle(z.O_2, z.B)
  z.P = C.CB.north            %  "north" atrributes of a circle
  z.Q = C.AC.north
  z.O = C.AB.south
  z.c = z.C:north(2)   %"north" method of a point (needs a parameter)
  C.PC = circle(z.P, z.C)
  C.QA = circle(z.Q, z.A)
  z.P_0 = intersection(C.PC,C.AB)   %  search for intersections of two circles.
  z.P_1 = intersection(C.PC,C.AC)   %   idem
  _,z.P_2 = intersection(C.QA,C.CB)   %  idem
  T.P = triangle(z.P_0, z.P_1, z.P_2)
  z.O_3 = T.P.circumcenter  % circumcenter attribute of "triangle"
}
```

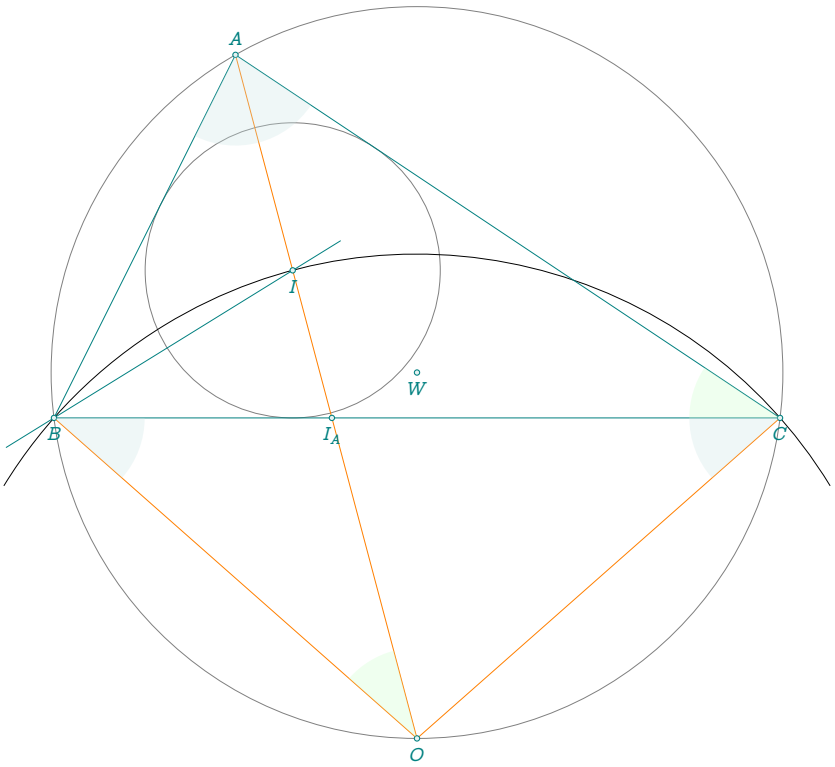
```
\begin{tikzpicture}
```

```
  \tkzGetNodes
  \tkzDrawCircle[black,fill=yellow!20,opacity=.4](O_0,B)
  \tkzDrawCircles[teal,fill=teal!40,opacity=.6](O_1,C O_2,B)
  \tkzDrawCircle[purple,fill=purple!20,opacity=.4](O_3,P_0)
  \tkzDrawArc[cyan,delta=10](Q,A)(P_0)
  \tkzDrawArc[cyan,delta=10](P,P_0)(B)
  \tkzDrawArc[cyan,delta=10](O,B)(A)
  \tkzDrawPoints(A,B,C,O_0,O_1,O_2,P,Q,P_0,P_0,P_1,P_2,O)
  \tkzLabelPoints(A,B,C,O_0,O_1,O_2,P,Q,P_0,P_0,P_1,P_2,O)
\end{tikzpicture}
\end{document}
```

4.5. Another example with comments: South Pole

Here's another example with comments

```
% !TEX TS-program = lualatex
\documentclass{standalone}
\usepackage[mini]{tkz-euclide}
\usepackage{tkz-elements}
\begin{document}
\directlua{
 init_elements()                    % Clear tables
 z.A = point(2, 4)
 z.B = point(0, 0)                  % three fixed points are used
 z.C = point(8, 0)                  %
 T.ABC = triangle(z.A, z.B, z.C)    % we create a new triangle object
 C.ins = T.ABC:in_circle()          % we get the incircle of this triangle
 z.I = C.ins.center                 % center is an attribute of the circle
 z.T = C.ins.through                % through is also an attribute
 z.I, z.T = C.ins:get()             % get() is a shortcut
 C.cir = T.ABC:circum_circle()      % we get the  circumscribed circle
 z.W = C.cir.center                 % we get the center of this circle
 z.O = C.cir.south                  % now we get the south pole of this circle
 L.AO = line(z.A, z.O)              % we create an object "line"
 L.BC = T.ABC.bc                    % we get the line (BC)
 z.I_A = intersection(L.AO, L.BC)   % we search the intersection of the last lines
}
```



Here's the **tikzpicture** environment to obtain the drawing:

```
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircles(W,A I,T)
\tkzDrawArc(O,C)(B)
\tkzDrawPolygon(A,B,C)
\tkzDrawSegments[new](A,O B,O C,O)
\tkzDrawLine(B,I)
\tkzDrawPoints(A,B,C,I,I_A,W,O)
\tkzFillAngles[green!20,opacity=.3](A,O,B A,C,B)
\tkzFillAngles[teal!20,opacity=.3](O,B,C B,C,O B,A,O O,A,C)
\tkzLabelPoints(I,I_A,W,B,C,O)
\tkzLabelPoints[above](A)
\end{tikzpicture}
```

5. Writing Convention, best practices and common mistakes

5.1. Assigning a Name to an Object

Note: Certain variables such as **z**, **L**, **T**, etc. are reserved for storing geometric objects. Reassigning one of them (e.g., **L = 4**) will overwrite its content and disrupt the system.
Points are a special case: they must be stored in the table **z**. For all other geometric objects, it is strongly recommended to follow the predefined variable names. Avoid mixing naming conventions or inventing your own identifiers, as this may lead to unpredictable behavior.

☞Warning: Do not reuse names such as `point`, `line`, `circle`, `path`, `triangle`, etc., as variable names. These identifiers are reserved internally by `tkz-elements` to represent object constructors.

Below is the list of reserved classes and the default variable names associated with them:

angle –> A
circle –> C
conic –> CO
fct –> F
pfct –> PF
line –> L
list_point –> LP
matrix –> M
occs –> O
parallelogram –> P
path –> PA
point –> z
quadrilateral –> Q
rectangle –> R
regular_polygon –> RP
square –> S
triangle –> T
vector –> V

5.2. Best practices

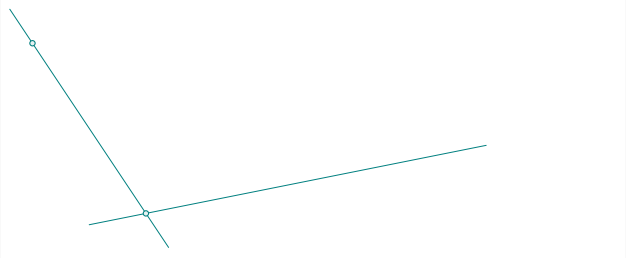
It is preferable to use methods—or better yet, object attributes—rather than standalone functions.
For example, to determine the midpoint of a segment AB, iit was previously possible to write:
`z.m = midpoint(z.A, z.B)`. This is now considered incorrect: you must use
`z.m = tkz.midpoint(z.A, z.B)`.
However, the recommended approach is to define the segment first with:
`L.AB = line(z.A, z.B)`, and then use the attribute:
`z.m = line(z.A, z.B).mid`. If you don't reuse the line, a one-liner like `z.m = line(z.A, z.B).mid` is acceptable.

Similarly, it may have been tempting to write `L.bi = tkz.bisector(z.A, z.B, z.C)` to obtain the angle bisector at vertex A. As in the previous example, this is no longer valid. You must now use:
`L.bi = tkz.bisector(z.A, z.B, z.C)`, as all standalone functions have been grouped under the `tkz` module.
A better practice is to define the triangle ABC using:
`T.ABC = triangle(z.A, z.B, z.C)` and then access the bisector via:
`L.bi = T.ABC:bisector(z.A)` or directly:
`L.bi = triangle(z.A, z.B, z.C):bisector(z.A)`.

5.3. Common mistakes and best practices

☞Second Warning: Let's examine the consequences of incorrect assignments. The following example is very simple: we define two points, the line passing through them, and a line orthogonal to the first at one of the

points. To define these objects, we use two tables (or classes): **z** and **L**.
Here is the correct code:



```
\directlua{
  z.A = point(1, 2)
  z.B = point(3, -1)
  L.AB = line(z.A, z.B)
  -- blunders
  L.ortho = L.AB:ortho_from(z.B)
  z.C = L.ortho.pb
}
\begin{tikzpicture}[scale = .75]
  \tkzGetNodes
  \tkzDrawLines(A,B B,C)
  \tkzDrawPoints(A,B)
\end{tikzpicture}
```

- **First blunder: `z = nil` or `z = 4`**. You've reassigned the **z** variable. It no longer refers to a table, but to a number. Its type has changed, and the system can no longer access the points.
- **`z.A = 4`** is equally problematic: you've overwritten point *A*. If your intention is to remove point *A*, then use **`z.A = nil`** instead.
- **`L = 4`**. This might seem convenient to store a length, but doing so will erase the entire table of lines.
- **`L.ortho = ortho`**. A more subtle mistake: if **ortho** is not defined, you lose your line. If it is defined but of the wrong type, an error will occur.
- For objects other than points, incorrect assignments at the end of the process may not affect the figure. It is possible to clean up tables before plotting. However, the **z** table must not be altered. Only points not used for plotting should be deleted.

These precautions help ensure consistency in the system and prevent unpredictable behavior.

We'll now explain how to assign variable names for each type of object.

5.4. Assigning a Name to a Point

Points must be stored in the table **z** if you intend to use them later in TikZ or tkz-euclide. In tkz-elements, all points follow the convention **z.name**, where **name** is the intended node label.

When transferring points to TikZ using **\tkzGetNodes**, each entry **z.name** produces the corresponding coordinate:

```
\coordinate (name) at (x,y);
```

Some naming cases require attention, particularly those involving primes.

Prime notation. The macro **\tkzGetNodes** converts certain terminal patterns in Lua names into prime or double-prime notation in TikZ.

- A name ending in **p** becomes a single prime:

  ```
  z.Bp = point(...) → node (B')
  ```

- A name ending in **pp** becomes a double prime:

  ```
  z.Bpp = point(...) → node (B'')
  ```

This convention is convenient for geometric constructions but may lead to unexpected renaming if the final **p** or **pp** is not intended as prime notation.

Warning

Avoid names ending with **p** or **pp** unless you explicitly want prime or double-prime notation in Ti*k*Z.

You can work with all these names in the **lua** section, but an error will occur when transferring points to nodes with the `\tkzGetNodes` macro. To avoid errors, use one of the solutions in the following paragraph. For **z.p**, simply using **z.P** is sufficient. The lowercase letter can be retrieved with Ti*k*Z.

Intermediate names. For readability or when working with long or descriptive names in Lua, you may use an intermediate variable and optionally assign a short label for the drawing.
For instance:

```
local euler = point(...) z.E = euler
```

Here, the point is internally stored under the clear name **euler**, but it appears as node **(E)** in Ti*k*Z.
If you prefer to keep the long name in both Lua and Ti*k*Z, simply write:

```
z.euler = point(...)
```

Alternatively, you may assign an alias at the Ti*k*Z level:

```
\pgfnodealias{E}{euler}
```

Possible also, Before the transfer, we use an accepted name such as **A** and remove the point **z.apollonius** with `z.apollonius = nil`.

```
z.A = z.apollonius z.apollonius = nil
```

Valid examples.

- `z.A = point(1,2)` → node `(A)`
- `z.Bp = point(3,4)` → node `(B')`
- `z.Cpp = point(5,1)` → node `(C'')`
- `z.H_a = T.ABC:altitude()` → node `(H_a)`
- `z.euler = T.ABC.eulercenter` → node `(euler)`

Names may contain letters, digits, and "_". Other characters should be avoided because they may not form valid TikZ node names.

Lua reminder. Fields of the table **z** may be accessed as:

- `z.A` (sugar syntax), or
- `z["A"]`

Assigning `z.A = nil` removes the point entirely.

5.5. Assigning a Name to Other Objects

For all geometric objects other than points, you are free to choose names. However, adopting consistent conventions greatly improves code readability and maintainability. To be more precise, each Lua code block is usually preceded by the call to the function `init_elements()`, whose role is to clear and reset the various tables (such as `L`, `C`, etc.). If you choose to use custom variable names or structures, you are free to do so — but in that case, you are responsible for managing and cleaning your variables and tables manually to avoid conflicts or unintended behavior.
In this documentation, the following naming strategy is used:
Objects are stored in dedicated tables, each associated with a specific class. These tables are represented by variables such as:

- `L` for lines and segments
- `C` for circles
- `T` for triangles
- `CO` for conics (including ellipses, hyperbolas, and parabolas)
- etc. See the list of reserved words [5.1]

Here are some examples of naming conventions used:

- Lines (L):
  The name reflects the two points defining the line.
  Example: `L.AB = line(z.A, z.B)` – line through A and B
- Circle (C)
  You can name a circle based on its defining points or its purpose. Examples:
  - C.AB –> Circle centered at A, passing through B
  - C.euler –> Euler circle
  - C.external –> External circle
- Triangles (T)
  Use vertex labels, or descriptive names when appropriate. Examples:
  - T.ABC –> Triangle with vertices A, B, C
  - T.feuerbach –> Feuerbach triangle
- Conics (CO)
  The table CO can store various conics; use meaningful keys to indicate type and role.

Note: *While you may choose other variable names or formats, following these conventions ensures that your code remains clear and easy to follow, especially when working with more complex figures.*

5.6. Writing conventions for attributes, methods.

You must follow standard Lua conventions when accessing attributes or invoking methods:

- **Attributes** are accessed using the dot notation: `object.attribute`.
  - For example, to access the coordinates of point `A`, use `z.A.re` for the abscissa and `z.A.im` for the ordinate.
  - To get the type of the object, write `z.A.type`.
  - To retrieve the south pole of the circle `C.OA`, write `C.OA.south`.
- **Methods** are invoked using the colon notation: `object:method()`.
  - For example, to compute the incircle of triangle `ABC`, use: `C.incircle = T.ABC:in_circle()`.
  - If a method requires a parameter, include it in parentheses. For instance, to compute the distance from point `C` to the line `(AB)`: `d = L.AB:distance(z.C)`.
- **Discarding results**: If a function returns multiple values and you only need one, use _ to ignore the rest.
  - For example, to retrieve only the second point of intersection between a line and a circle: `_, z.J = intersection(`

5.7. Miscellaneous

- Units and coordinates: As in tkz-euclide, the default unit is the centimeter. All points are placed in an orthonormal Cartesian coordinate system.
- Numerical variables: Real numbers follow the standard Lua conventions for notation.
- Complex numbers: Similar to real numbers, but you must define them using the point constructor. For example:

  ```
  za = point(1, 2)
  ```

  This corresponds mathematically to $1+2i$. You can print the complex number using:

  ```
  tex.print(tostring(za))
  ```

- Boolean values:
  - In Lua: `bool = true` or `bool = false`
  - You can use the following **Lua** code:

    ```
    if bool == true then ... else ... end
    ```

  - In LaTeX, after loading the ifthen package, you can write:

    ```
    \ifthenelse{\equal{\tkzUseLua{bool}}{true}}{<true code>}{<false code>}
    ```

- Strings:
  - Example in Lua: `st = "Euler's formula"`
  - In LaTeX: `\tkzUseLua{st}` will display `Euler's formula`

6. Work organization

Here is a sample organization for working with tkz-euclide and LuaLaTeX.

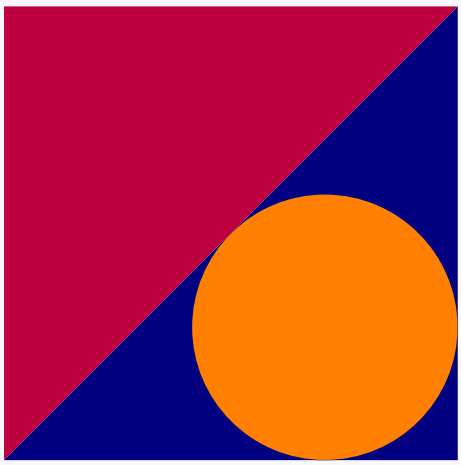
- Compilation: Add the line:

  ```
  % !TEX TS-program = lualatex
  ```

  to ensure that your document compiles with LuaLaTeX.
- Document Class: The standalone class is recommended when the goal is simply to create a figure, as it avoids unnecessary overhead.
- Package Loading: You can load tkz-euclide in two ways:
  - **\usepackage{tkz-euclide}** gives you full access to the entire package.
  - The recommended method is to use the mini option, which loads only the necessary modules for drawing. You still retain the ability to draw with Ti*k*Z if desired.
- Conditionals: The package ifthen is useful when you need to evaluate Boolean conditions within your document.
- Lua Code Organization: While you can embed Lua code directly with \directlua, externalizing the code offers several advantages:
  - Better syntax highlighting and code presentation in editors that support **Lua** and LaTeX.
  - Simplified commenting: **Lua** uses --, while LaTeX uses %. Keeping Lua in a separate file avoids confusion.
  - Reusability: external code files can be reused across multiple documents or figures.

For simplicity, this documentation uses embedded Lua code in most cases. However, in some examples, external files are used to show you how it's done.

```
% !TEX TS-program = lualatex
% Created by Alain Matthes on 2024-01-09.
\documentclass[margin = 12pt]{standalone}
\usepackage[mini]{tkz-euclide}
\usepackage{tkz-elements,ifthen}

\begin{document}
\directlua{
 init_elements() % Clear tables
 dofile ("lua/sangaku.lua") % Load the lua code
}
\begin{tikzpicture}[ scale = .75]
 \tkzGetNodes
 \tkzDrawCircle(I,F)
 \tkzFillPolygon[color = purple](A,C,D)
 \tkzFillPolygon[color = blue!50!black](A,B,C)
 \tkzFillCircle[color = orange](I,F)
\end{tikzpicture}
\end{document}
```



And here is the code for the **Lua** part: the file `ex_sangaku.lua`

```
z.A = point(0, 0)
z.B = point(8, 0)
L.AB = line(z.A, z.B)
S.AB = L.AB:square()
_, _, z.C, z.D = S.AB:get()
z.F = S.ac:projection(z.B)
L.BF = line(z.B, z.F)
T.ABC = triangle(z.A, z.B, z.C)
L.bi = T.ABC:bisector(2)
z.c = L.bi.pb
L.Cc = line(z.C, z.c)
z.I = intersection(L.Cc, L.BF)
```

6.1. Scaling Policy Update

In previous versions, it was recommended to apply scaling within the Lua part of the code. However, this guidance has now changed.
Since all geometric computations are handled in **Lua**, applying scaling in Ti*k*Z no longer presents any issues. On the contrary, performing scaling in **Lua** has led to several complications—particularly with the recent implementation of conic-related functions, which involve numerous distance calculations using real numbers.
These challenges prompted a review of several functions, during which some bugs related to Lua-side scaling were identified and resolved.

New Recommendation: From now on, scaling should be applied exclusively in the Ti*k*Z part. The **Lua** code should operate in an unscaled, consistent coordinate system to ensure the reliability of all geometric computations.
The following documentation uses only scaling in the **tikzpicture** environment.

7. Coordinates

This section outlines the various coordinate systems available to users. Given the removal of scaling operations from the **Lua** layer, such clarification seems more necessary than ever.

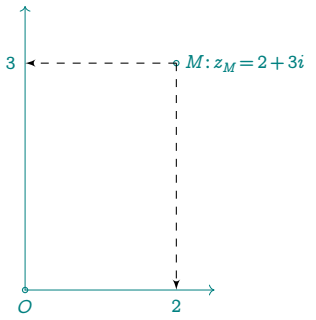
7.1. Common Use of Coordinates

As with tkz-euclide, tkz-elements is based on a two-dimensional orthonormal Cartesian coordinate system, using centimeters as the default unit.
It would be inconsistent to use what we will see as an *occs* (orthogonal coordinate coordinate system) in a context focused on Euclidean geometry.
Moreover, the concept of a point in tkz-elements is tied to the affix of a complex number. To maintain code clarity and consistency, the option to modify units within Lua has been deliberately omitted.

Conclusion: For any figure created with tkz-elements, all points are placed within a 2D orthonormal system using centimeters.

Coordinates of M

```
z.A = point(2, 3)
```



7.2. Use of barycentric coordinates.

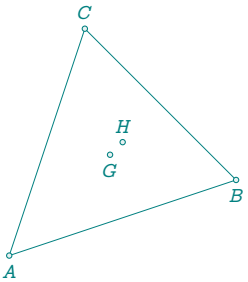
A barycentric coordinate system describes the position of a point relative to a reference triangle. Any point in the plane can be expressed with barycentric coordinates, which are defined up to a scalar multiple (homothety). Alternatively, they may be normalized so their sum equals 1.
Barycentric coordinates are particularly useful in triangle geometry, especially when analyzing properties invariant under affine transformations—those not dependent on angles.
Consider a triangle ABC. One can define key points like the centroid and orthocenter using barycentric coordinates.

`About the Orthocenter.`
Computing barycentric coordinates for the orthocenter requires knowledge of the triangle's angles. These are stored as attributes in the table `T.ABC`:

- `T.ABC.alpha` — angle $\hat{A}$ at vertex A
- `T.ABC.beta` — angle $\hat{B}$ at vertex B
- `T.ABC.gamma` — angle $\hat{C}$ at vertex C

See [14.6.3]

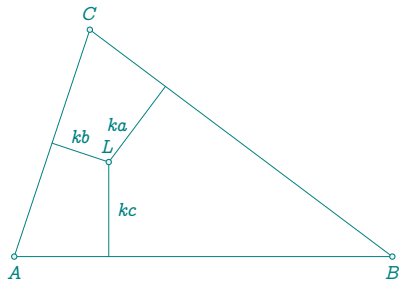
`Retrieving Barycentric Coordinates.`
It is possible to compute the barycentric coordinates of a given point with respect to a triangle. The returned values are automatically normalized. See Section [14.5.1] for usage.




```
\directlua{
z.A = point(0, 0)
z.B = point(3, 1)
z.C = point(1, 3)
T.ABC = triangle(z.A, z.B, z.C)
z.G = T.ABC:barycentric(1, 1, 1)
z.H = T.ABC:barycentric(math.tan(T.ABC.alpha),
                        math.tan(T.ABC.beta),
                        math.tan(T.ABC.gamma))}
```

7.3. Use of Trilinear Coordinates

The trilinear coordinates of a point P with respect to a triangle ABC are a triple of values proportional to the directed distances from P to each of the triangle's sides. These coordinates are homogeneous and typically written as $x:y:z$ or (x,y,z).
Since only the ratio between the coordinates is relevant, trilinear coordinates are particularly well suited for expressing geometric relationships that are invariant under scaling.
So $a':b':c' = ka:kb:kc$ in the next example $(a = BC, b = AC, c = AB)$.




```
\directlua{
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(1, 3)
T.ABC = triangle(z.A, z.B, z.C)
z.L = T.ABC:trilinear(T.ABC.a,
                      T.ABC.b,
                      T.ABC.c)
z.a, z.b, z.c = T.ABC:projection(z.L)}
```

7.4. OCCS

Objects in this class are *orthonormal Cartesian coordinate system*. They are obtained from the reference system by translation and rotation. They can be used to simplify certain expressions and coordinates. See [15]

8. Numerical Tolerance

8.1. Floating-Point Arithmetic

All computations in tkz-elements are performed using floating-point arithmetic. As a consequence, exact comparisons between real numbers are unreliable.
For example, a point theoretically lying on a line may produce a very small non-zero value due to rounding errors.

8.2. Global Tolerance: tkz.epsilon

To ensure numerical robustness, tkz-elements uses a global tolerance parameter:
`tkz.epsilon = 1e-10`
This value defines the admissible numerical error in geometric tests (collinearity, incidence, equality of distances, etc.).

By default, this tolerance is set to a small positive value. It can be adjusted by advanced users if needed.

8.3. Usage in Position Tests

All membership and position tests are *EPS-aware*. When a method accepts an optional argument **EPS**, the following rule applies:

- If **EPS** is provided, it overrides the global tolerance.
- Otherwise, **tkz.epsilon** is used.

This design ensures consistency across all geometric objects.

9. Geometric Relations API

9.1. General Principle

In tkz-elements, geometric objects provide a unified method

```
object:position(other_object[, EPS])
```

to classify a geometric relation.
The result depends on the type of the second argument (**other_object**).

All position tests are tolerance-aware. If not specified, the optional parameter **EPS** defaults to the global value **tkz.epsilon**.

Convention: All classification results are returned as *uppercase symbolic strings*, intended for logical comparison in Lua code.

9.2. Point vs Object

When the tested object is a point, the result describes membership relative to a region (or a boundary).

Possible return values:

`"IN"`	strictly inside the region,
`"ON"`	on the boundary (within tolerance),
`"OUT"`	strictly outside the region.

Examples:

- **circle:position(point)** (`"IN"`, `"ON"`, `"OUT"`)
- **triangle:position(point)** (`"IN"`, `"ON"`, `"OUT"`)
- **conic:position(point)** (`"IN"`, `"ON"`, `"OUT"`)
- **line:position(point)** (`"ON"` or `"OUT"`)

9.3. Object vs Object

When both arguments are geometric objects, the result describes their mutual geometric relation.

Line vs Line

`"INTERSECT"`	lines intersect at one point,
`"PARALLEL"`	distinct parallel lines,
`"IDENTICAL"`	identical lines.

Line vs Circle

`"DISJOINT"`	no intersection,
`"TANGENT"`	exactly one common point,
`"SECANT"`	two intersection points.

Circle vs Circle

`"DISJOINT_EXT"`	exterior disjoint circles,
`"TANGENT_EXT"`	exterior tangency,
`"SECANT"`	two intersection points,
`"TANGENT_INT"`	interior tangency,
`"DISJOINT_INT"`	one circle strictly inside the other,
`"CONCENTRIC"`	same center, different radii,
`"IDENTICAL"`	identical circles.

9.4. Triangle and Conic

For **triangle** and **conic**, the method **position()** currently supports point arguments only.

Triangle vs Point

`"IN"`	strictly inside the triangle,
`"ON"`	on an edge or a vertex,
`"OUT"`	outside the triangle.

Conic vs Point

`"IN"`	inside the conic region (according to its type),
`"ON"`	on the conic curve,
`"OUT"`	outside the associated region.

`Note:` For hyperbolas and parabolas, the meaning of `"IN"` and `"OUT"` depends on the region associated with the conic type and its construction data (focus/directrix or equivalent representation).

9.5. Design Philosophy

- A single polymorphic method centralizes geometric classification.
- Uppercase symbolic results ensure clarity and stability.
- Numerical tolerance improves robustness near boundary cases.
- The API is extensible: additional object types may be supported over time.
- Backward compatibility wrappers may be provided when older boolean methods existed in previous versions.

10. Class and Object

10.1. Class

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. An object is a data structure that contains both attributes (data) and methods (operations), which together define its behavior.

A class is a user-defined data type that serves as a blueprint for creating objects. It specifies the structure and shared behavior of a category of objects, including default values for attributes and common implementations of methods[6].

10.2. Object

An object is an instance of a class. Each object encapsulates attributes and methods. Attributes store information or properties specific to the object (typically as fields in a data table), while methods define how the object behaves or interacts with other objects.

All objects in the package are typed. The currently defined and used types are: **point**, **line**, **circle**, **triangle**, **conic**, **quadrilateral**, **square**, **rectangle**, **parallelogram**, **regular_polygon**, **occs** and **path**.

10.2.1. Creating an object

Objects are generally created using the method **new**, by providing points as arguments.

- The **point** class requires two real numbers (coordinates),
- The **regular_polygon** class requires two points and an integer (the number of sides),
- The **occs** class requires a line and a point,
- The **path** class requires a table of points written as strings.

Each object is usually assigned a name and stored in a table according to its type. For example:

- points are stored in the global table **z**,
- lines in **L**, circles in **C**, triangles in **T**, and so on.

This convention allows easy access and reusability across computations and drawings. For example:

```
z.A = point(1, 2)
z.B = point(4, 5)
L.AB = line(z.A, z.B)
```

Here, **z.A** and **z.B** store points in table **z**, while the line defined by these points is stored as **L.AB** in table **L**.

☞ Note:
From version 4 onwards, object creation has been streamlined. Instead of calling **object:new(arguments)**, you can simply use **object(arguments)** — the shorter form is equivalent.

```
z.A = point(1, 2)           -- short form
-- equivalent to:
z.A = point:new(1, 2)
```

Objects can also be generated by applying methods to existing objects. For instance, **T.ABC:circum_circle()** produces a new **circle** object. Some object attributes are themselves objects: **T.ABC.bc** returns a **line** representing side BC of triangle ABC.

6 An action that an object can perform.

☞ Important:
All these named objects are stored in global tables. To avoid conflicts or residual data between figures, it is strongly recommended to call the function **at** the beginning of each construction. This resets the environment and ensures a clean setup. See the next section

10.2.2. Initialization: init_elements

Before performing geometric constructions or calculations, it is important to initialize the system. The function **init_elements()** resets internal tables and parameters to prepare for a new figure. This step ensures a clean environment and avoids interference from previously defined objects.

```
init_elements()
```

Purpose. The function **init_elements** clears global tables such as:

- **z** — for storing points,
- **L**, **C**, **T**, etc. — for lines, circles, triangles,
- and other geometric structures.

It also (re)sets default values for internal constants such as the number of decimal digits and the floating-point tolerance.

When to use it:
This function should be called:

- at the beginning of each new TikZ figure using Lua,
- or any time you need to reset the environment manually.

Example usage.
Here is a typical usage:

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(3, 4)
  L.AB = line(z.A, z.B)}
```

Note.
Calling **init_elements** is optional if you manage object names carefully, but it is highly recommended in iterative workflows or automated figure generation to avoid unwanted data persistence.

10.2.3. Attributes

Attributes are accessed using the standard method. For example, `T.pc` retrieves the third point of the triangle, and `C.OH.center` retrieves the center of the circle. Additionally, I have added a method **get()** that returns the points of an object. This method applies to straight lines (pa and pc), triangles (pa, pb, and pc), and circles (center and through).

Example usage: : `z.O, z.T = C.OT:get()` retrieves the center and a point of the circle.

10.2.4. Methods

A method is an operation (function or procedure) associated (linked) with an object.
Example: The point object is used to vertically determine a new point object located at a certain distance from it (here 2). Then it is possible to rotate objects around it.

```
\directlua{
      init_elements()
      z.A = point(1, 0)
      z.B = z.A:north(2)
      z.C = z.A:rotation (math.pi / 3, z.B)
      tex.print(tostring(z.C))
}
```

The coordinates of C are: -0.73205080756888 and 1.0

11. Class point

A point in tkz-elements is internally represented as a complex number (its affix). This choice enables concise computations (translation, rotation, scaling) while remaining compatible with TikZ/tkz-euclide for rendering. The variable **z** holds a table used to store points. It is **mandatory** and is automatically initialized by the package (e.g.,**z = {}**).

The **point** class forms the foundation of the entire framework. It is a hybrid class, representing both points in the plane and complex numbers. The underlying principle is as follows:

- The plane is equipped with an orthonormal basis (OCCS See [15]), allowing us to determine a point's position via its abscissa and ordinate.
- Similarly, any complex number can be seen as an ordered pair of real numbers (its real and imaginary parts).
- Therefore, the plane can be identified with the complex plane, and a complex number $x+iy$ is represented by a point in the plane with coordinates (x, y).

Thus, a point such as A is stored as the object **z.A**, with its coordinates and associated properties encapsulated within this object.

The **point** object possesses several attributes:

- **re** –> the real part (abscissa),
- **im** –> the imaginary part (ordinate),
- **type** –> the type of the object (in this case, always **"point"**),
- **arg** –> the argument of the complex number (angle with respect to the x-axis),
- **mod** –> the modulus of the complex number (distance from the origin).

11.1. Creating a point

Points are created by providing their coordinates in the current orthonormal Cartesian coordinate system (OCCS). The recommended form is:

```
z.A = point(x, y)
```

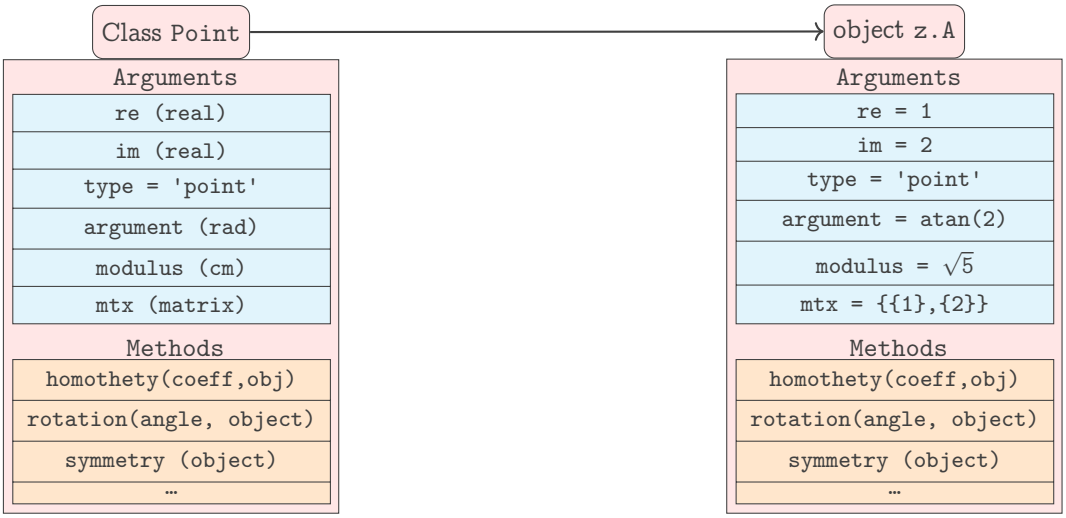
where **x** and **y** are real numbers corresponding to the x and y coordinates of the point.

Internally, this creates a complex number $x+iy$ and stores it in the table **z** under the key **"A"**. The table **z** is used to reference all points by their label.

Alternatively, the more explicit syntax is also available:

```
z.A = point:new(x, y)
```

Both forms are equivalent. The shorthand constructor is available since version 4 and preferred for readability and consistency.



11.2. Attributes of a point

Creation
`z.A = point(1, 2)`

The point A has coordinates $x = 1$ and $y = 2$. If you use the notation `z.A`, then A will be referenced as a node in Ti*k*Z or in `tkz-euclide`.
This is the creation of a fixed point with coordinates 1 and 2 and which is named A. The notation **`z.A`** indicates that the coordinates will be stored in a table assigned to the variable **`z`** (reference to the notation of the affixes of the complex numbers) that A is the name of the point and the key allowing access to the values.

Table 1: Point attributes.

Attributes	Description	Example / Reference
type	Object type name, always "point"	
re	Real part (i.e., x-coordinate)	[10.2.4]
im	Imaginary part (i.e., y-coordinate)	[10.2.4]
argument	Argument of the affix (angle in radians)	≈ 0.785398... [11.2.1]
modulus	Modulus of the affix (distance to origin)	$\sqrt{5} \approx$ 2.2360... [11.2.1]
mtx	Matrix representation as column vector	**z.A.mtx = {{1},{2}}** [11.2.1]

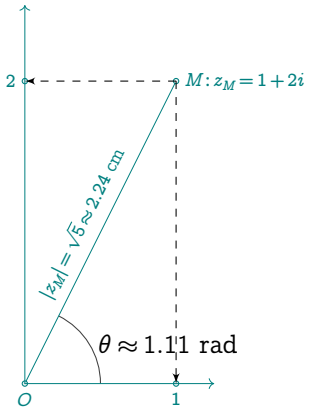
11.2.1. Example: point attributes

```
\directlua{
   init_elements()
   z.M = point(1, 2)}

\begin{tikzpicture}[scale = 1]
\pgfkeys{/pgf/number format/.cd,std,precision=2}
\let\pmpn\pgfmathprintnumber
\tkzDefPoints{2/4/M,2/0/A,0/0/O,0/4/B}
\tkzLabelPoints(O)
\tkzMarkAngle[fill=gray!30,size=1](A,O,M)
\tkzLabelAngle[pos=1,right](A,O,M){%
   $\theta \approx \pmpn{\tkzUseLua{z.M.argument}}$ rad}
\tkzDrawSegments(O,M)
\tkzLabelSegment[above,sloped](O,M){%
    $|z_M| =\sqrt{5}\approx \pmpn{%
```

```
   \tkzUseLua{z.M.modulus}}$ cm}
\tkzLabelPoint[right](M){$M: z_M = 1 + 2i$}
\tkzDrawPoints(M,A,O,B)
\tkzPointShowCoord(M)
\tkzLabelPoint[below,teal](A){$\tkzUseLua{z.M.re}$}
\tkzLabelPoint[left,teal](B){$\tkzUseLua{z.M.im}$}
\tkzDrawSegments[->,add = 0 and 0.25](O,B O,A)
\end{tikzpicture}
```

Attributes of `z.M`

- `z.M.re` $= 1$
- `z.M.im` $= 2$
- `z.M.type` $=$ point
- `z.M.argument` $= \theta \approx 1.11$ rad
- `z.M.modulus` $= |z_M| = \sqrt{5} \approx 2.24$ cm
- `z.M.mtx` $= \begin{bmatrix} 1 \\ 2 \end{bmatrix}$



11.2.2. Attribute mtx

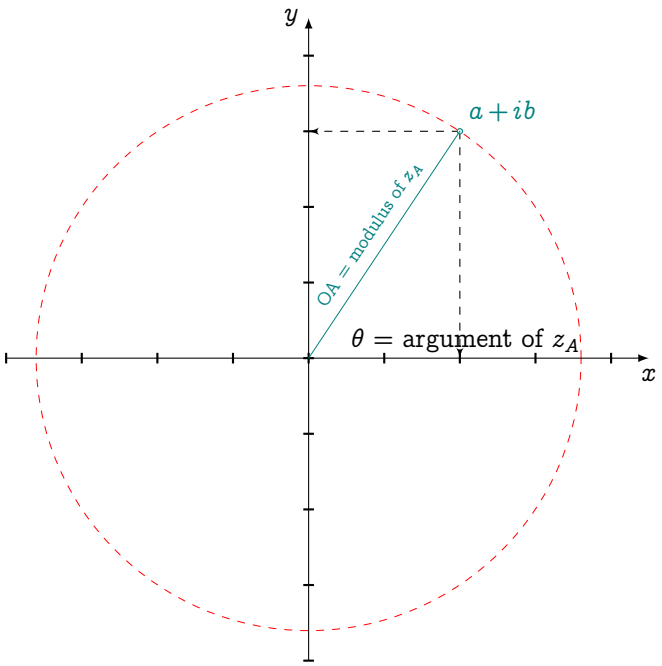
This attribute allows the point to be used in conjunction with matrices.

```
  \directlua{
  z.A = point(2, -1)
  z.A.mtx:print()}
```

$$\begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

11.2.3. Argand diagram

```
\directlua{
  init_elements()
  z.A = point(2, 3)
  z.O = point(0, 0)
  z.I = point(1, 0)}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzInit[xmin=-4,ymin=-4,xmax=4,ymax=4]
   \tkzDrawCircle[dashed,red](O,A)
   \tkzPointShowCoord(A)
   \tkzDrawPoint(A)
   \tkzLabelPoint[above right](A){\normalsize $a+ib$}
   \tkzDrawX\tkzDrawY
   \tkzDrawSegment(O,A)
   \tkzLabelSegment[above,anchor=south,sloped](O,A){ OA = modulus of $z_A$}
  \tkzLabelAngle[anchor=west,pos=.5](I,O,A){$\theta$ = argument of $z_A$}
\end{tikzpicture}
```


y
x
a + ib
OA = modulus of z_A
θ = argument of z_A

11.3. Methods of the class point

The methods listed in the following table are standard and commonly used throughout the examples at the end of this documentation. Each of these methods returns a **point** object.

For more advanced operations using complex numbers and operator overloading, see section 38.2.1, which describes the available metamethods.

Table 2: Functions and Methods of the **class** point.

Methods	Reference
Constructors	
new(r,r)	[11.3.1; 11.3.6
polar(d,an)	[11.3.3]
polar_deg(d,an)	[11.3.4]
Methods Returning a Real Number	
get()	[11.3.2]
Methods Returning a Point	
north(r)	[10.2.4]
south(r)	
east(r)	
west(r)	
normalize()	[11.3.6]
normalize_from(pt)	[11.3.6]
orthogonal(d)	[11.3.7]
at()	[11.3.8]
shift_orthogonal_to(pt, d)	[11.3.11]
shift_collinear_to(pt, d)	[11.3.12]
Methods Returning a Circle	
PPP(a,b)	[11.3.9]
Methods Returning a Object	
symmetry(obj)	[11.3.15]
rotation(an, obj)	[11.3.13]
homothety(r,obj)	[11.3.16]
Utilities	
identity(pt)	[11.3.14]
print()	[11.3.15]

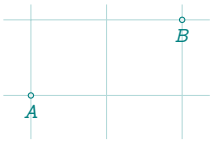
11.3.1. Method new(r, r)

This method creates a point in the plane using Cartesian coordinates. The **shorthand constructor (r, r)** is available since version 4 and preferred for readability and consistency.

It takes two real numbers as arguments: the first represents the *abscissa* (real part), and the second the *ordinate* (imaginary part). Internally, the point is treated as a complex number and stored in the global table **z**.

The resulting object is of type **point**, and can be used in further geometric constructions or displayed with tkz-euclide.

Note: The default unit is the centimeter, in accordance with the conventions of tkz-euclide. All coordinates are interpreted in an orthonormal Cartesian coordinate system.




```
\directlua{
  init_elements()
  z.A  = point(0, 0)
  z.B  = point(2, 1)}
\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawPoints(A,B)
  \tkzLabelPoints(A,B)
\end{tikzpicture}
```

11.3.2. Method get()

The **get** method is used to retrieve the Cartesian coordinates of a point.

Let I be the intersection point of two lines. You can obtain its coordinates in two equivalent ways:

- using the point's attributes directly:

  x = z.I.re and **y = z.I.im**

- or using the **get()** method:

  x, y = z.I:get()

This method improves code readability and makes it easier to pass coordinates to functions that expect numerical values.

x_I =2.7272727272727
y_I =-0.54545454545455

```
\directlua{
 init_elements()
 z.A, z.B = point(0, 0), point(5, -1)
 z.C, z.D = point(1, -4), point(4, 2)
 L.AB = line(z.A, z.B)
 L.CD = line(z.C, z.D)
 z.I = intersection(L.AB, L.CD)
 x, y = z.I:get()
 tex.print("$x_I = $"..x)
 tex.print('\\\\')
 tex.print("$y_I = $"..y)}
```

11.3.3. Function polar(r, an)

This method creates a point in the plane using polar coordinates.

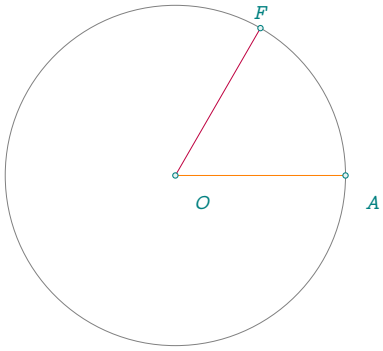
It takes two arguments:

- **r** —> the modulus (distance from the origin),
- **an** —> the argument (angle in radians).

Internally, the point is represented as a complex number: **r * exp(i * an)**. This method is particularly useful for constructing points on circles or for defining points in terms of angle and distance.

Note: The default unit is the centimeter, consistent with the conventions of tkz-euclide. All coordinates are interpreted in an orthonormal Cartesian coordinate system.

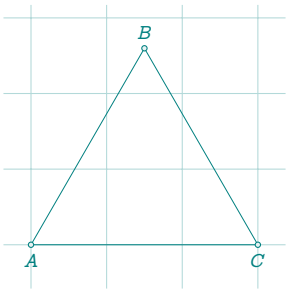
```
z.B = polar(2, math.pi / 4)
or
z.B = point:polar(2, math.pi / 4)
```




```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = point(3, 0)
 z.F = polar(3, math.pi / 3)}
\begin{center}
\begin{tikzpicture}[scale=.75]
   \tkzGetNodes
   \tkzDrawCircle(O,A)
   \tkzDrawSegments[new](O,A)
   \tkzDrawSegments[purple](O,F)
   \tkzDrawPoints(A,O,F)
   \tkzLabelPoints[below right=6pt](A,O)
   \tkzLabelPoints[above](F)
\end{tikzpicture}
\end{center}
```

11.3.4. Method polar_deg(d,an)

Identical to the previous one, except that the angle is given in degrees.



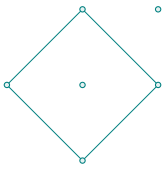

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = polar_deg(3, 60)
 z.C = polar_deg(3, 0)}
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,C)
  \tkzLabelPoints[above](B)
  \end{tikzpicture}
\end{center}
```

11.3.5. Method north(d)

This method creates a new point located at a vertical distance from the given point, along the line passing through it and directed upward (toward the north).

It is particularly useful when you want to construct a point offset by a specific distance above a reference point—for example, to place a label or construct a geometric configuration with a known height.

The optional argument **d** represents the vertical distance. If omitted, a default value of 1 is used.



```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = z.O:east()
 z.Ap= z.O:east(1):north(1)
 z.B = z.O:north()
 z.C = z.O:west()
 z.D = z.O:south()}
\begin{center}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawPolygon(A,B,C,D)
   \tkzDrawPoints(A,B,C,D,O,A')
\end{tikzpicture}
\end{center}
```

11.3.6. Method normalize()

This method returns a new point located on the segment from the origin to the current point, at a distance of 1 from the origin. It is typically used to extract the direction of a vector and normalize its length to one.

You can also use this method to construct a point at a fixed distance from another point along a given direction. For example:

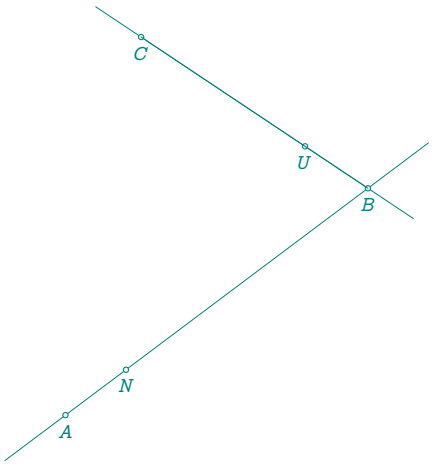
```
z.U = (z.C - z.B):normalize() + z.B
```

Here, the vector $\overrightarrow{BU}$ has length 1, and U lies on the segment $[BC]$ in the direction from B to C.

There are two equivalent ways to achieve the same result:

```
z.U = z.C:normalize_from(z.B)
z.U = L.BC:normalize()
```

The second approach requires prior creation of the line object **L.BC**.




```
\directlua{
  init_elements()
  z.A  = point(0, 0)
  z.B  = point(4, 3)
  z.C  = point(1, 5)
  L.AB = line(z.A, z.B)
  L.BC = line(z.B, z.C)
  z.N = z.B:normalize()
  z.U = z.C:normalize_from(z.B)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(A,B  B,C)
  \tkzDrawPoints(A,B,C,U,N)
  \tkzLabelPoints(A,B,C,U,N)
  \tkzDrawSegment(B,C)
\end{tikzpicture}
```

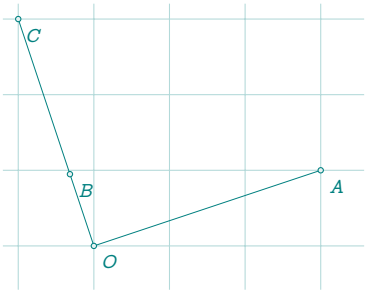
11.3.7. Method orthogonal(d)

Let O be the origin of the plane, and let A be a point distinct from O. This method constructs a new point B such that the vectors $\overrightarrow{OB}$ and $\overrightarrow{OA}$ are orthogonal:

$$\overrightarrow{OB} \perp \overrightarrow{OA}$$

By default, the point B is chosen so that `OB` $=$ `OA`. If the optional argument **d** is provided, then the point B is constructed so that `OB` $= d$.

This method is useful for constructing perpendicular vectors or generating points on circles orthogonal to given directions.




```
\directlua{
  init_elements()
  z.A = point(3, 1)
  z.B = z.A:orthogonal(1)
  z.O = point(0, 0)
  z.C = z.A:orthogonal()}
\begin{center}
  \begin{tikzpicture}[gridded]
    \tkzGetNodes
    \tkzDrawSegments(O,A O,C)
    \tkzDrawPoints(O,A,B,C)
    \tkzLabelPoints[below right](O,A,B,C)
  \end{tikzpicture}
\end{center}
```

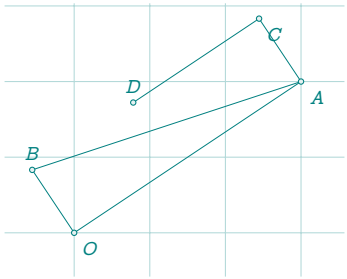
11.3.8. Method at(pt)

This method complements the **orthogonal** method. Instead of constructing a point B such that $\overrightarrow{OB} \perp \overrightarrow{OA}$ (with O as the origin), it constructs a point B such that:

$$\overrightarrow{AB} \perp \overrightarrow{OA}$$

In this case, the reference direction remains $\overrightarrow{OA}$, but the orthogonal vector is constructed from point A, not the origin. The result is a point B lying on a line orthogonal to (OA) and passing through A.

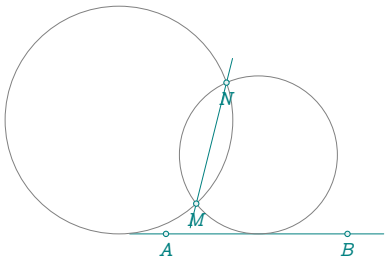
This method is useful when working with local orthogonal directions, such as when constructing altitudes in a triangle or defining perpendicular vectors anchored at a given point.




```
\directlua{
init_elements()
z.O = point(0, 0)
z.A = point(3, 2)
z.B = z.A:orthogonal(1)
z.C = z.A + z.B
z.D =(z.C - z.A):orthogonal(2):at(z.C)}
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints[below right](O,A,C)
  \tkzLabelPoints[above](B,D)
  \tkzDrawSegments(O,A A,B A,C C,D O,B)
  \tkzDrawPoints(O,A,B,C,D)
  \end{tikzpicture}
\end{center}
```

11.3.9. Method PPP(a,b)

This method is presented in the document **Geometry Euclidean** [AlterMundus]




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, 0)
  z.M = point(1, 1)
  z.N = point(2, 5)
  L.AB = line(z.A, z.B)
  PA.center,
  PA.through, n = L.AB:LPP(z.M, z.N)
  tkz.nodes_from_paths(PA.center,
                       PA.through, "O", "T")
}

\begin{tikzpicture}[scale = .4]
  \tkzGetNodes
  \tkzDrawLines(A,B M,N)
  \tkzDrawCircles(O1,T1 O2,T2)
  \tkzDrawPoints(A,B,M,N)
  \tkzLabelPoints(A,B,M,N)
\end{tikzpicture}
```

11.3.10. Method rotation(obj)

— First example
This method performs a rotation of one or more points around the current point, which serves as the center of rotation.

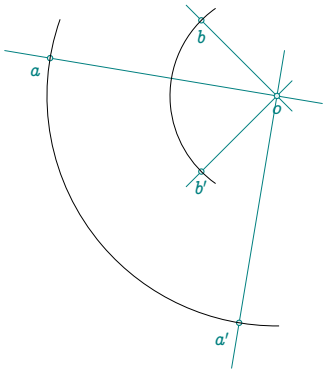
Arguments
The arguments are:

– the angle of rotation, expressed in radians;

– a point or a list of points to rotate.

Return value
The result is a point (or a list of points) obtained by rotating each given point around the center by the specified angle.

Example usage
In the following example, a list of points is rotated about a given center.




```
\directlua{
 init_elements()
 z.a = point(0, -1)
 z.b = point(4, 0)
 z.o = point(6, -2)
 z.ap,
 z.bp = z.o:rotation(math.pi / 2, z.a, z.b)}
\begin{center}
\begin{tikzpicture}[scale = .5]
 \tkzGetNodes
 \tkzDrawLines(o,a o,a' o,b o,b')
 \tkzDrawPoints(a,a',b,b',o)
 \tkzLabelPoints(b,b',o)
 \tkzLabelPoints[below left](a,a')
 \tkzDrawArc(o,a)(a')
 \tkzDrawArc(o,b)(b')
\end{tikzpicture}
\end{center}
```

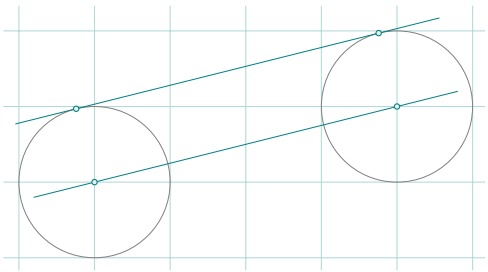
11.3.11. Method shift_orthogonal_to(pt, dist)

Syntax: z.P = z.A:shift_orthogonal_to(z.B, 2)

Return value This method returns the point obtained by shifting the current point A in the direction orthogonal to the line (AB), at a signed distance dist. A positive distance corresponds to a rotation of the vector $\overrightarrow{AB}$ by $+90°$ around the point A, while a negative distance corresponds to $-90°$.

z.Q = z.B:shift_orthogonal_to(z.A, 2)

A positive distance corresponds to a rotation of the vector $\overrightarrow{BA}$ by $+90°$, around the point B.



```
\directlua{
z.A = point(3, 2)
z.a = point(3, 3)
z.B = point(7, 3)
z.b = point(7, 4)
C.A = circle(z.A, z.a)
C.B = circle(z.B, z.b)
z.TA = z.A:shift_orthogonal_to(z.B, C.A.radius)
z.TB = z.B:shift_orthogonal_to(z.A, -C.A.radius)
}
\begin{tikzpicture}[gridded]
\tkzGetNodes
\tkzDrawCircles(A,a B,b)
\tkzDrawLines(A,B TA,TB)
\tkzDrawPoints(A,B,TA,TB)
\end{tikzpicture}
```

11.3.12. Method shift_collinear_to(pt, dist)

This method returns the point obtained by shifting the current point A in the same direction that line (AB), at a signed distance dist.

Syntax: z.P = z.A:shift_collinear_to(z.B, 2)

11.3.13. Method rotation(an, obj)

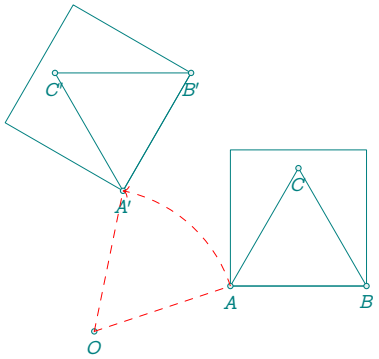
Purpose:
This method rotates a geometric object by a given angle around a specified point.
In this example, the triangle is rotated by an angle of $\pi/3$ around the point O, which serves as the center of rotation.

Syntax: `new_obj = z.I:rotation(an, obj)`

Arguments:
The arguments are:

- **an** —> the angle of rotation, in radians;
- **obj** —> the geometric object to be rotated (e.g., a triangle).

Returns: The method returns a new object of the same type, rotated accordingly.




```
\directlua{
 init_elements()
 z.O  = point(-1, -1)
 z.A  = point(2, 0)
 z.B  = point(5, 0)
 L.AB  = line(z.A, z.B)
 T.ABC = L.AB:equilateral()
 S.fig = L.AB:square()
 _,_,z.E,z.F = S.fig:get()
 S.new = z.O:rotation(math.pi / 3, S.fig)
 _,_,z.Ep,z.Fp = S.new:get()
 z.C = T.ABC.pc
 T.ApBpCp = z.O:rotation(math.pi / 3, T.ABC)
 z.Ap,z.Bp,
 z.Cp = T.ApBpCp:get()}
\begin{center}
\begin{tikzpicture}[scale = .6]
 \tkzGetNodes
 \tkzDrawPolygons(A,B,C A',B',C' A,B,E,F)
 \tkzDrawPolygons(A',B',E',F')
 \tkzDrawPoints(A,B,C,A',B',C',O)
 \tkzLabelPoints(A,B,C,A',B',C',O)
 \begin{scope}
  \tkzDrawArc[delta=0,->,dashed,red](O,A)(A')
  \tkzDrawSegments[dashed,red](O,A O,A')
 \end{scope}
\end{tikzpicture}
\end{center}
```

11.3.14. identity(pt)

Syntax: `z.A:identity(z.B)`

Purpose: Check whether two points are identical within the numerical precision defined by **tkz.epsilon**.

Parameters: The point to be compared with **self**.

Return value: True if the two points are numerically identical, false otherwise.

```
true
1e-10
```

```
\directlua{
 z.A = point(0, 0)
 z.B = point(0.0000000001, 0)
 tex.print(tostring(z.A:identity(z.B)))
 tex.print('\\\\')
 tex.print(tkz.epsilon)}
 %% true if tkz.epsilon > 1e-10
```

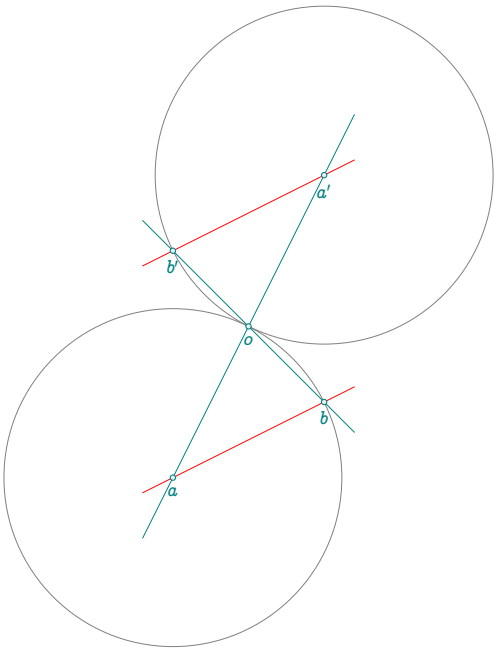
Remark:
This method compares two points by computing the Euclidean distance between them and checking if it is smaller than the global tolerance **tkz.epsilon**. It is useful to test equality of points in floating-point computations.

11.3.15. Method symmetry(obj)

Description: This method performs a central symmetry (point reflection) of a geometric object with respect to the given point.

Arguments
The argument **obj** can be a point, a line, a triangle, a circle, or any other supported geometric object. Each element of the object is reflected through the center point, producing a new object of the same type.

Example usage
The following example shows how to apply central symmetry to an object (e.g., a triangle) using a reference point as the center.




```
\directlua{
  init_elements()
  z.a = point(0, -1)
  z.b = point(2, 0)
  L.ab = line(z.a, z.b)
  C.ab = circle(z.a, z.b)
  z.o = point(1, 1)
  z.ap, z.bp = z.o:symmetry(C.ab):get()}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircles(a,b a',b')
  \tkzDrawLines(a,a' b,b')
  \tkzDrawLines[red](a,b a',b')
  \tkzDrawPoints(a,a',b,b',o)
  \tkzLabelPoints(a,a',b,b',o)
  \end{tikzpicture}
\end{center}
```

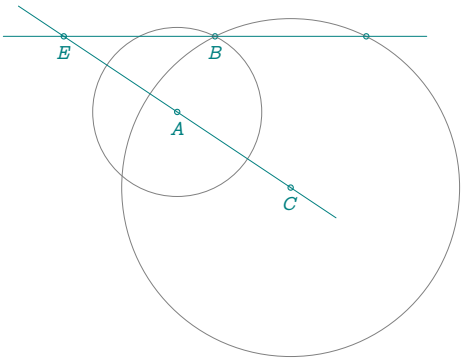
11.3.16. Method homothety(k, obj)

Purpose: This method performs a homothety (dilation or contraction) of a geometric object with respect to the current point, which serves as the center of the transformation.

Arguments:

- **k** — the homothety ratio (a real number),
- **obj** — the object to be transformed, which can be:
  1. a single point,
  2. a list of points,
  3. or a geometric object (line, triangle, circle, etc.).

A positive ratio **k** produces a scaling centered at the point, while a negative ratio also reflects the object through the center.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(1, 2)
 z.E = point(-3, 2)
 z.C, z.D = z.E:homothety(2, z.A, z.B)}
\begin{center}
\begin{tikzpicture}[scale = .5]
  \tkzGetNodes
  \tkzDrawPoints(A,B,C,E,D)
  \tkzLabelPoints(A,B,C,E)
  \tkzDrawCircles(A,B C,D)
  \tkzDrawLines(E,C E,D)
\end{tikzpicture}
\end{center}
```

This method converts the point's coordinates to a formatted string that can be displayed directly in the text.

The number of decimal places is controlled by the global variable **tkz_dc**, which is set to 2 by default in the **init_elements()** function. You can override it by assigning a new value before calling **print()**:

```
tkz_dc = 0
```

This is particularly useful when displaying coordinates, sums, products, or intermediate results in mathematical expressions.

Example usage:

```
\directlua{
  init_elements()
  z.A = point(1, 2)
  z.B = point(1, -1)
  z.a = z.A + z.B
  z.m = z.A * z.B
  tkz_dc = 0}
  The respective affixes of points $A$ and $B$ being
   \tkzUseLua{z.A:print()} and \tkzUseLua{z.B:print()},
  their sum is \tkzUseLua{z.a:print()} and
  their product \tkzUseLua{z.m:print()}.
```

The respective affixes of points A and B being 1+2.00i and 1-i, their sum is 2+i and their product 3+i.

12. Class line

The variable `L` holds a table used to store line objects. It is optional, and users are free to choose their own variable name (e.g., **`Lines`**). However, for consistency and readability, it is recommended to use `L`. The function `init_elements()` reinitializes this table automatically.

12.1. Creating a line

To define a line passing through two known points, use the following constructor:

```
L.AB = line(z.A, z.B)          (short form, recommended)
L.AB = line:new(z.A, z.B)      (explicit form)
```

This creates a line object **`L.AB`** representing:

- the infinite line passing through points **`z.A`** and **`z.B`**, and
- the segment $[AB]$, which is used to compute attributes such as the midpoint or direction.

Internally, this object stores the two defining points and derives several geometric properties from them.

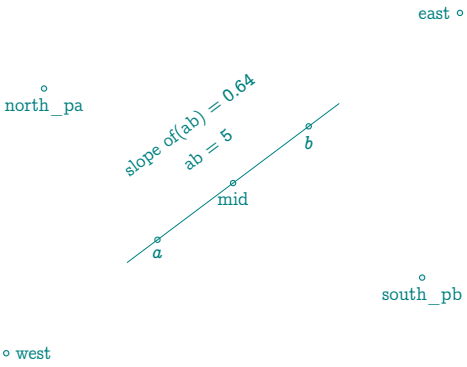
12.2. Attributes of a line

Let's consider `L.AB = line(z.A, z.B)`
A line object provides access to the following attributes:

Table 3: Line attributes.

Attribute	Meaning	Reference
`type`	Always **`"line"`**	
`pa`	First point (e.g., **`z.A`**)	
`pb`	Second point (e.g., **`z.B`**)	
`mid`	Midpoint of segment $[AB]$	
`slope`	Angle with respect to the horizontal axis	[12.2.1]
`length`	Euclidean distance AB	12.2.1]
`vec`	Vector $B - A$	[22]
`north_pa`	Auxiliary point north of **`pa`**	[12.2.1]
`north_pb`	Auxiliary point north of **`pb`**	–
`south_pa`	Auxiliary point south of **`pa`**	–
`south_pb`	Auxiliary point south of **`pb`**	–
`east`	Auxiliary point east of the segment	–
`west`	Auxiliary point west of the segment	–

12.2.1. Example: attributes of class line




```
\directlua{
  init_elements()
  z.a = point(1, 1)
  z.b = point(5, 4)
  L.ab = line(z.a, z.b)
  z.m = L.ab.mid
  z.w = L.ab.west
  z.e = L.ab.east
  z.r = L.ab.north_pa
  z.s = L.ab.south_pb
  sl = L.ab.slope
  len = L.ab.length}
```

12.2.2. Note on line object attributes

To recover the original defining points of a line object **L.name**, use either of the following:

- via the method **get(n)**, as in **z.A, z.B = L.name:get()** See [12.7.1,
- or directly via its attributes **L.name.pa** and **L.name.pb**.

12.3. Methods of the class line

Here's the list of methods for the **line** object. The results can be real numbers, points, lines, circles or triangles. The triangles obtained are similar to the triangles defined below.

Table 4: Methods of the class line.(part 1)

Methods	Reference
Constructor	
new(pt, pt)	Note[a]; [12.1; 12.3.1]
Methods Returning a Real Number	
distance(pt)	[12.4.1]
Methods Returning a Boolean	
on_line(pt)	[12.5.1]
on_segment(pt)	[12.5.2]
is_parallel(L)	[12.5.3]
is_orthogonal(L)	[12.5.4]
is_equidistant(pt)	[12.5.5]
Methods Returning a String	
position(pt)	[12.6.1]
position_segment(pt)	[12.6.2
where_on_line(pt)	[12.6.3]
Methods Returning a Point	
get(n)	[12.7.1]
random()	[12.7.14]
gold_ratio()	[12.7.10; 4.4]
normalize()	[12.7.11]
normalize_inv()	[12.7.11]
barycenter(r,r)	[12.7.3]
point(r)	[12.7.4]
midpoint()	[12.7.5]
harmonic_int(pt)	[12.7.6]
harmonic_ext(pt)	[12.7.7]
harmonic_both(r)	[12.7.8]
harmonic(mode, pt)	[12.7.9]
report(d,pt)	[12.7.2]
collinear_at(pt,k)	[ex. 12.7.12]
Methods Returning a Line	
ll_from(pt)	[12.8.1]
ortho_from(pt)	[12.8.3]
mediator()	Note[b] ; [12.8.5]
swap_line()	[12.8.7 ; 27.5.2]
orthogonal_at()	12.7.13

a line(pt, pt) (short form, recommended)
b You may use as a synonym.

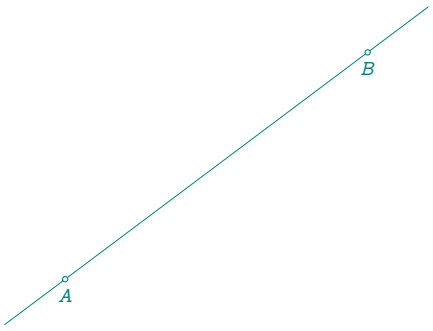
Table 5: Methods of the class line.(part 2)

Methods	Reference
Methods Returning a Triangle	
equilateral(<'swap'>)	Note[a]; [12.9.1; 11.3.13]
isosceles(d,<'swap'>)	[12.9.2]
two_angles(an,an)	Note [b] [12.9.5]
school(<'swap'>)	[12.9.3]
half(<'swap'>)	[12.9.4]
s_s(r,r<,'swap'>)	[12.9.6]
sa_(r,an<,'swap'>)	[12.9.7]
_as(r,an<,'swap'>)	[12.9.8]
a_s(r,an<,'swap'>)	[12.9.10]
s_a(r,an<,'swap'>)	[12.9.9]
Methods Returning a Sacred Triangle	
gold(<'swap'>)	[12.10.1]
golden(<'swap'>)	or sublime [12.10.2]
golden_gnomon(<'swap'>)	[12.10.3]
pythagoras(<'swap'>)	or egyptian [12.10.4]
Methods Returning a Circle	
circle()	
apollonius(r)	[12.11.1]
LPP	or c_l_pp(pt, pt) [12.11.2]
LLP	or c_ll_p(L, pt) [12.11.3]
LLL(L, L)	[12.11.4]
Methods Returning a Square	
square()	Note [c]; [11.3.13]
Methods Returning a Object	
reflection(obj)	[12.13.5]
translation(obj)	[12.13.4]
projection(obj)	[12.13.1]
projection_ll(L, pts)	[12.13.2]
affinity_ll(L, k, pts)	[12.13.3]
Methods Returning a Path	
path(n)	[12.13.6]

a By default, triangles are oriented positively (counter-clockwise). Use **"swap"** for clockwise orientation.
b The given side is between the two angles
c `_,_,z.C,z.D = S.AB:get()`

12.3.1. Method new(pt,pt)

It is preferable to use syntax such as **L.xx** and it's also preferable to use the short form **line(pt, pt)**.



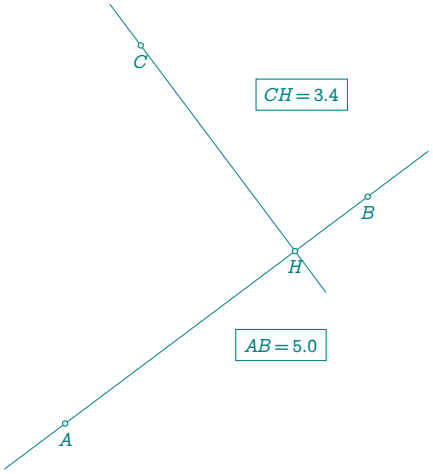

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 3)
  L.AB = line(z.A, z.B)}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLine(A,B)
  \tkzDrawPoints(A,B)
  \tkzLabelPoints(A,B)
\end{tikzpicture}
\end{center}
```

12.4. Returns a real number

Only one method in this category

12.4.1. Method distance(pt)

This method gives the distance from a point to a straight line.

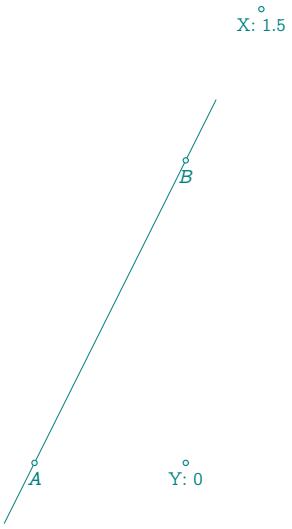



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 3)
  z.C = point(1, 5)
  L.AB = line(z.A, z.B)
  d = L.AB:distance(z.C)
  l = L.AB.length
  z.H = L.AB:projection(z.C)}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawLines(A,B C,H)
\tkzDrawPoints(A,B,C,H)
\tkzLabelPoints(A,B,C,H)
\tkzLabelSegment[above right=2em,draw](C,H){%
  $CH = \tkzUseLua{d}$}
\tkzLabelSegment[below right=1em,draw](A,B){%
  $AB = \tkzUseLua{l}$}
\end{tikzpicture}
\end{center}
```

12.5. Returns a boolean

12.5.1. Method on_line(pt)

This method is a boolean wrapper of **position(pt)**.
It returns **true** if the point lies on the line and **false** otherwise.
The method can also be called using **L:on_line(pt)**.
Note: This method is kept for backward compatibility.

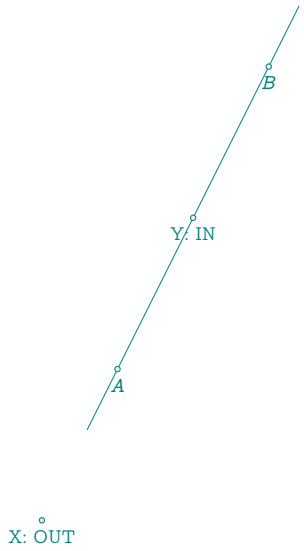



```
\directlua{
  init_elements()
  local  function calc_distance(L, p)
    if L:on_line(p) then
      return point.abs(p - L.pa) / L.length
    else
      return 0
    end
  end
  z.A = point(0, 0)
  z.B = point(2, 4)
  z.X = point(3, 6)
  z.Y = point(2, 0)
  L.AB = line(z.A, z.B)
  dx = calc_distance(L.AB, z.X)
  dy = calc_distance(L.AB, z.Y)}
\begin{center}
  \begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawLine(A,B)
   \tkzDrawPoints(A,B,X,Y)
   \tkzLabelPoints(A,B)
   \tkzLabelPoint(X){X: \tkzUseLua{dx}}
   \tkzLabelPoint(Y){Y: \tkzUseLua{dy}}
  \end{tikzpicture}
\end{center}
```

12.5.2. Method on_segment(pt)

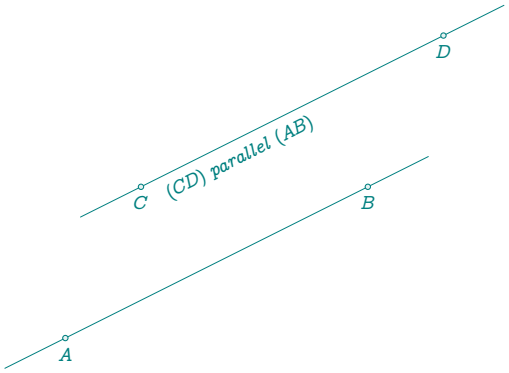
Variant of the previous method; indicates whether a point is on or off a segment.
Boolean wrapper of **position_segment(pt)**.
Returns: **true** if **position_segment(pt)** returns **"ON"**, and **false** otherwise.
Note: Kept for backward compatibility.
Returns:

- **true** if the point lies on the segment $[pa, pb]$,
- **false** otherwise.



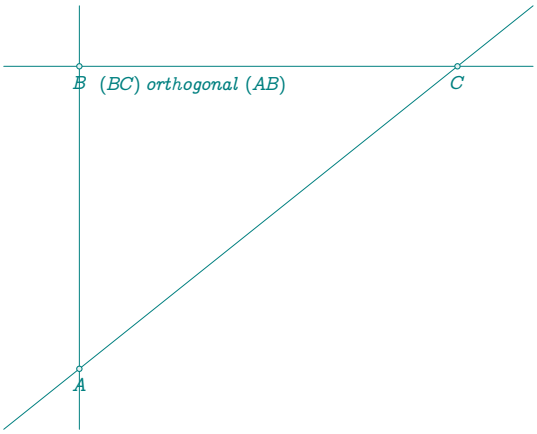

```
\directlua{
  init_elements()
  local  function inseg(L, p)
    if L:on_segment(p) then
      return "IN"
    else
      return "OUT"
    end
  end
  z.A = point(0, 0)
  z.B = point(2, 4)
  z.X = point(-1,-2)
  z.Y = point(1, 2)
  L.AB = line(z.A, z.B)
  dx = inseg(L.AB, z.X)
  dy = inseg(L.AB, z.Y)}
\begin{center}
  \begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawLine(A,B)
   \tkzDrawPoints(A,B,X,Y)
   \tkzLabelPoints(A,B)
   \tkzLabelPoint(X){X: \tkzUseLua{dx}}
   \tkzLabelPoint(Y){Y: \tkzUseLua{dy}}
  \end{tikzpicture}
\end{center}
```

12.5.3. Method is_parallel(L)




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 2)
 L.AB = line(z.A, z.B)
 z.C = point(1, 2)
 z.D = point(5, 4)
 L.CD = line(z.C, z.D)
 if L.AB:is_parallel(L.CD)
 then tkztxt = "parallel"
 else tkztxt = "no parallel"
 end}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(A,B C,D)
  \tkzDrawPoints(A,B,C,D)
  \tkzLabelPoints(A,B,C,D)
  \tkzLabelSegment[sloped,pos=.3](C,D){%
   $(CD)\ \tkzUseLua{tkztxt}\ (AB)$}
\end{tikzpicture}
\end{center}
```

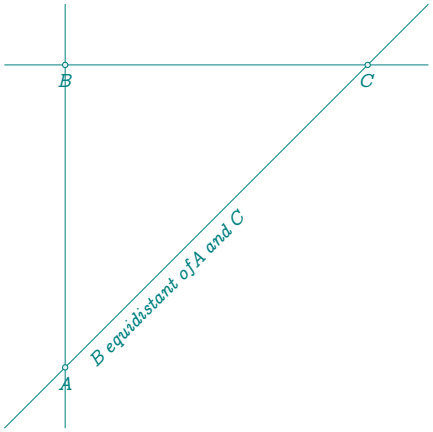
12.5.4. Method is_orthogonal(L)




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(0, 4)
 L.AB = line(z.A, z.B)
 z.C = point(5, 4)
 L.BC = line(z.B, z.C)
 if L.AB:is_orthogonal(L.BC) then
   tkztxt = "orthogonal"
 else
   tkztxt = "no orthogonal"
 end}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(A,B B,C A,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B,C)
  \tkzLabelSegment[sloped,pos=.3](B,C){%
    $(BC)\ \tkzUseLua{tkztxt}\ (AB)$}
\end{tikzpicture}
\end{center}
```

12.5.5. Method is_equidistant(pt)

Is a point equidistant from the two points that define the line?




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(0, 4)
  z.C = point(4, 4)
  L.AC = line(z.A, z.C)
  if L.AC:is_equidistant(z.B) then
    tkztxt = "equidistant"
  else
    tkztxt = "no equidistant"
  end}
\begin{center}
    \begin{tikzpicture}
    \tkzGetNodes
    \tkzDrawLines(A,B B,C A,C)
    \tkzDrawPoints(A,B,C)
    \tkzLabelPoints(A,B,C)
    \tkzLabelSegment[sloped,pos=.3](A,C){%
      $B \ \tkzUseLua{tkztxt}\ of A\ and\ C$}
    \end{tikzpicture}
\end{center}
```

12.6. Returns a string

12.6.1. Method position(obj[,EPS])

This method classifies the relative position between the current line and another object. The returned value is a *symbolic string* whose meaning depends on the type of **obj**. All tests are performed with a numerical tolerance.

Arguments:

- **obj**: a **point**, **line**, or **circle**;
- **EPS** (optional): numerical tolerance. If omitted, **tkz.epsilon** is used.

Returns:

- If **obj** is a **point**: returns **"ON"** if the point lies on the line (within tolerance), otherwise **"OUT"**.
- If **obj** is a **line**: returns the result of **line_line_position_(self,obj,EPS)**.
- If **obj** is a **circle**: returns the result of **line_circle_position_(self,obj,EPS)**.

Note: The method is a dispatcher based on **getmetatable(obj)** and uses **tkz.epsilon** by default.

12.6.2. Method position_segment(pt)

This method determines the position of a point relative to the segment $[pa,pb]$ defined by the line object.
Arguments: **pt**: a point.
Returns:

- **"ON"** if the point lies on the segment (within tolerance),
- **"OUT"** otherwise.

Note: The method uses the numerical tolerance **tkz.epsilon**.

12.6.3. Method where_on_line(pt)

This method determines the *oriented position* of a point **pt** along a line, considered as oriented from its first defining point **pa** to its second one **pb**.

The method assumes that the line is oriented from **pa** to **pb**. It first checks whether the point lies on the line (within a tolerance). If this is not the case, the method returns **nil**.

Arguments.

- **pt**: a point.

Returns.

- **"BEFORE"** if **pt** lies on the line *before* **pa**;
- **"BETWEEN"** if **pt** lies on the segment $[pa,pb]$ (i.e. between **pa** and **pb**);
- **"AFTER"** if **pt** lies on the line *after* **pb**;
- **nil** if **pt** does not lie on the line.

Notes.

- Reversing the order of the defining points **pa** and **pb** reverses the meaning of "BEFORE" and "AFTER".
- The method uses a numerical tolerance **tkz.epsilon**.

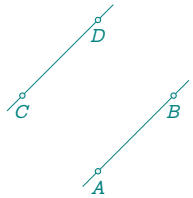
12.7. Returns a point

12.7.1. Method get()

This method retrieves the two points that define the given line. It is useful, for example, when constructing a line through a specific point and parallel to another: you need two points to define the direction.

- **L.AB:get()** returns the two points **pa** and **pb**.
- **L.AB:get(1)** returns the first point **pa**.
- **L.AB:get(2)** returns the second point **pb**.

This method is available for most geometric objects that are defined by two or more points (e.g., lines, triangles, circles). It allows for easy reuse and composition of geometric constructions.




```
\directlua{
  init_elements()
  z.A = point(1, 1)
  z.B = point(2, 2)
  L.AB = line(z.A, z.B)
  z.C = L.AB.north_pa
  z.D = L.AB:ll_from(z.C):get(2)}

\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(A,B C,D)
  \tkzDrawPoints(A,B,C,D)
  \tkzLabelPoints(A,B,C,D)
\end{tikzpicture}
```

12.7.2. Method report(r,<pt>)

Purpose: This method constructs and returns a **point** located at a given distance r from a reference point along the direction of the line.

It generalizes the idea of "reporting a length" on a line segment. Depending on whether a reference point is specified, the method either measures from the first endpoint of the line or from a user-defined origin.
Syntax:

`L:report(r)`	→	returns the point at distance r from `L.pa` along `L.pb`
`L:report(r, pt)`	→	returns the point at distance r from `pt` parallel to `L`

Arguments:

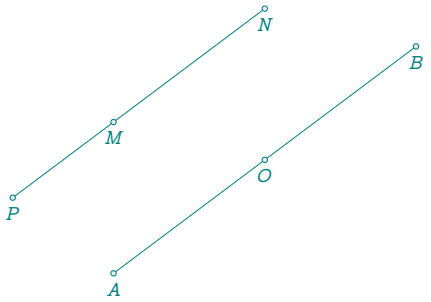
r (number) — the signed distance to report along the line. If $r > 0$, the point lies in the direction from **pa** to **pb**; if $r < 0$, it lies in the opposite direction.

pt (optional **point**) — an optional reference point from which to start measuring the distance.

Special cases:

- If the line has zero length (i.e., both endpoints coincide), an error is raised.
- Negative distances produce points on the opposite extension of the line.
- The method works consistently with open or oriented lines (not just finite segments).

Alias: The synonym **point_at_distance** is provided for readability, especially when the geometrical intent is to "locate a point at a given distance along a direction".
Returned value: A new **point** object corresponding to the computed location.

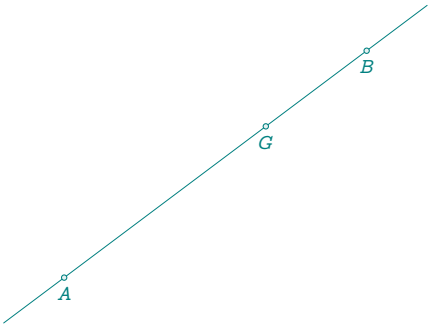



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 3)
  L.AB = line(z.A, z.B)
  z.M = point(0, 2)
  z.N = L.AB:report(2.5, z.M)
  z.O = L.AB:report(2.5)
  z.P = L.AB:report(-L.AB.length/3,z.M)}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawSegments(A,B P,N)
  \tkzDrawPoints(A,B,M,N,O,P)
  \tkzLabelPoints(A,B,M,N,O,P)
  \end{tikzpicture}
\end{center}
```

12.7.3. Method barycenter(ka, kb)

This method returns the barycenter of the two points that define the line, weighted by the coefficients **ka** and **kb**.

Geometrically, the barycenter lies on the line segment $[AB]$ (or its extension) and divides it internally or externally, depending on the sign and ratio of the coefficients.



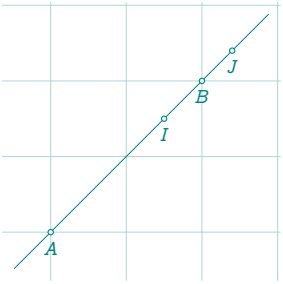

```
\directlua{
 init_elements()
 z.A = point(0, -1)
 z.B = point(4, 2)
 L.AB = line(z.A, z.B)
 z.G = L.AB:barycenter(1, 2)}
\begin{center}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawLine(A,B)
   \tkzDrawPoints(A,B,G)
   \tkzLabelPoints(A,B,G)
\end{tikzpicture}
\end{center}
```

12.7.4. Method point(r)

This method is very useful: it allows you to place a point on the line, based on a real parameter **r**.

- If **r = 0**, the result is the point **pa**.
- If **r = 1**, the result is the point **pb**.
- If **r = 0.5**, the result is the midpoint of the segment $[AB]$.
- Any value of **r** is allowed: a negative value places the point before **pa**, a value greater than 1 places it beyond **pb**.

This method is implemented for all objects that are defined by at least two points, except for quadrilaterals.




```
\directlua{
  init_elements()
  z.A = point(-1, -1)
  z.B = point(1, 1)
  L.AB = line(z.A, z.B)
  z.I = L.AB: point(0.75)
  z.J = L.AB: point(1.2)}
```

12.7.5. Method midpoint()

This method has been replaced by **tkz.midpoint** [See 29.2].
You can obtain the midpoint of a segment using an attribute of the **line** object.

```
z.M = L.AB.mid
```

However, when the line object has not been created and its creation is unnecessary, the standalone function **tkz.midpoint(z.A, z.B)** remains convenient and efficient.

12.7.6. Method harmonic_int(pt)

Given a point on the line but located *outside* the segment $[AB]$, this method returns a point on the segment that maintains a harmonic ratio.

Let AB be a line, and let D be a point such that $D \in (AB)$ but $D \notin [AB]$. The method returns a point $C \in [AB]$ such that:

$$\frac{AC}{BC} = \frac{AD}{BD}$$

This construction is particularly useful in projective geometry and when working with harmonic divisions.

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.D = point(8, 0)
 L.AB = line(z.A, z.B)
 z.C = L.AB:harmonic_int(z.D)}
\begin{center}
  \begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawLine(A,D)
   \tkzDrawPoints(A,B,C,D)
   \tkzLabelPoints(A,B,C,D)
  \end{tikzpicture}
\end{center}
```

12.7.7. Method harmonic_ext(pt)

This method returns a point located outside the segment $[AB]$ that satisfies a harmonic relation with a given point on the segment.

Let AB be a line segment, and let C be a point on $[AB]$ (but not its midpoint). The method returns a point D lying on the line (AB) but *outside* the segment $[AB]$, such that:

$$\frac{AC}{BC} = \frac{AD}{BD}$$

This is the inverse of the operation performed by the method **harmonic_int(pt)**.

12.7.8. Method harmonic_both(k)

This method returns two points on the line defined by AB:

- one point C lying *inside* the segment $[AB]$,
- one point D lying *outside* the segment $[AB]$,

such that the following harmonic ratio holds:

$$\frac{AC}{BC} = \frac{AD}{BD} = k$$

The parameter **k** represents the desired ratio between the distances. This method is useful when you want to construct both the internal and external harmonic conjugates of a given segment.

The method returns the two corresponding points in the order: **internal, external**.

```
  \directlua{
   init_elements()
   z.A = point(0, 0)
   z.B = point(6, 0)
   L.AB = line(z.A, z.B)
   z.C, z.D = L.AB:harmonic_both(5)}
\begin{center}
  \begin{tikzpicture}
     \tkzGetNodes
     \tkzDrawLine(A,D)
     \tkzDrawPoints(A,B,C,D)
     \tkzLabelPoints(A,B,C,D)
  \end{tikzpicture}
\end{center}
```

12.7.9. Method harmonic(mode, arg)

This method provides a unified interface for harmonic division.

- **mode = "internal"** returns the internal harmonic point.
- **mode = "external"** returns the external harmonic point.
- **mode = "both"** returns both harmonic points.

The argument **arg** is:

- a point when **mode** is **"internal"** or **"external"**,
- a ratio when **mode** is **"both"**.

Example

```
H1      = L.AB:harmonic("internal",  z.P)
H2      = L.AB:harmonic("external",  z.P)
Hi, He  = L.AB:harmonic("both", 2)
```

12.7.10. Methode gold_ratio

This method returns a point C on the segment $[AB]$ that divides it according to the **golden ratio** φ:

$$\frac{AC}{CB} = \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

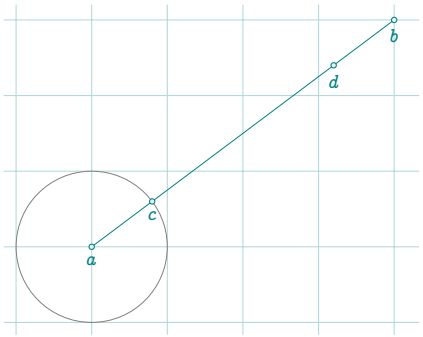
This construction is useful in design, geometry, and aesthetics where harmonic proportions are desired.

AC/BC = 1.6180339887499 $\varphi = 1.6180339887499$

```
\directlua{
   init_elements()
   z.A = point(0, 0)
   z.B = point(6, 0)
   L.AB = line(z.A, z.B)
   z.C = L.AB:gold_ratio()
   AC = tkz.length(z.A, z.C)
   BC = tkz.length(z.B, z.C)}

 AC/BC = \tkzUseLua{AC / BC}

 $\varphi = \tkzUseLua{(math.sqrt(5)+1)/2}$

\begin{center}
  \begin{tikzpicture}
     \tkzGetNodes
     \tkzDrawLine(A,B)
     \tkzDrawPoints(A,B,C)
     \tkzLabelPoints(A,B,C)
  \end{tikzpicture}
\end{center}
```

12.7.11. Method normalize() and normalize_inv()

$ac = 1$ and $c \in [ab]$
$bd = 1$ and $d \in [ab]$




```
\directlua{
  init_elements()
  z.a = point(1, 1)
  z.b = point(5, 4)
  L.ab = line(z.a, z.b)
  z.c = L.ab:normalize()
  z.d = L.ab:normalize_inv()
}
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawSegments(a,b)
  \tkzDrawCircle(a,c)
  \tkzDrawPoints(a,b,c,d)
  \tkzLabelPoints(a,b,c,d)
  \end{tikzpicture}
\end{center}
```

12.7.12. Method collinear_at(pt,<r>)

This method returns a point located on a line parallel to (AB) and passing through a given point. The resulting point is placed at a distance proportional to the length of AB, in the same direction.

If the second argument ⟨r⟩ is omitted, it defaults to 1. In that case, the method produces a segment of the same length and direction as $[AB]$.

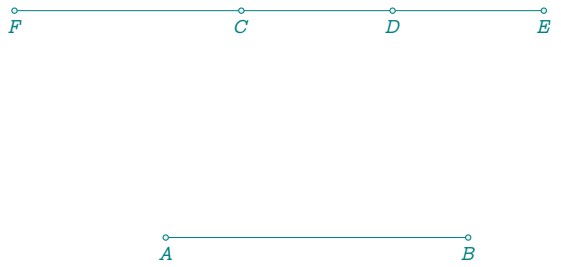
In addition to the scale-factor form, this method also accepts an option table **{length = ℓ}**, allowing the distance CD to be specified *absolutely*, independently of AB.

Example interpretations:

- If **L.AB:collinear_at(z.C)** produces point E, then $CE = AB$ and $(AB) \parallel (CE)$.
- If **L.AB:collinear_at(z.C, 0.5)** produces point D, then $CD = 0.5 \cdot AB$ and $(AB) \parallel (CD)$.

– If **L.AB:collinear_at(z.C, {length = 2})** produces point F, then $CF = 2$ (in the current unit) and $(AB) \parallel (CF)$. It is also possible to use a negative value.

This is particularly useful for replicating a vector or projecting a direction onto another position in the plane.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0)
  z.C = point(1, 3)
  L.AB = line(z.A, z.B)
  z.D = L.AB:collinear_at(z.C, .5)
  z.E = L.AB:collinear_at(z.C)
  z.F = L.AB:collinear_at(z.C, {length = -3})}
\begin{center}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawSegments(A,B C,E C,F)
 \tkzDrawPoints(A,B,C,D,E,F)
 \tkzLabelPoints(A,B,C,D,E,F)
\end{tikzpicture}
\end{center}
```

12.7.13. Method orthogonal_at(pt,<r>)

This method returns a point located on a line **perpendicular** to the line (AB), passing through a given point. The resulting point is placed at a distance proportional to the length of AB, in a direction orthogonal to it. By default, if the second argument is omitted, the factor is 1.

In addition to the scale factor form, this method also accepts an option table **{length = ℓ}**, allowing the distance CD to be specified *absolutely*, independently of AB.

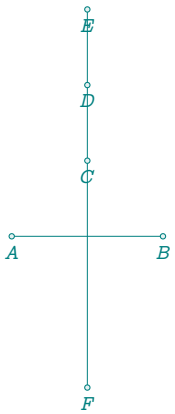
Example interpretations:

– If **L.AB:orthogonal_at(z.C)** gives point E, then $CE = AB$ and $(AB) \perp (CE)$.

– If **L.AB:orthogonal_at(z.C, 0.5)** gives point D, then $CD = 0.5 \cdot AB$ and $(AB) \perp (CD)$.

– If **L.AB:orthogonal_at(z.C, {length = 2})** gives point F, then $CF = 2$ (in the current unit) and $(AB) \perp (CF)$. It is also possible to use a negative value.

This method is useful for constructing perpendicular vectors or building geometric figures such as rectangles or perpendicular bisectors.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(2, 0)
  z.C = point(1, 1)
  L.AB = line(z.A, z.B)
  z.D = L.AB:orthogonal_at(z.C, .5)
  z.E = L.AB:orthogonal_at(z.C)
  z.F = L.AB:orthogonal_at(z.C, {length = -3})}
\begin{center}
  \begin{tikzpicture}
    \tkzGetNodes
    \tkzDrawSegments(A,B C,E C,F)
    \tkzDrawPoints(A,B,C,D,E,F)
    \tkzLabelPoints(A,B,C,D,E,F)
  \end{tikzpicture}
\end{center}
```

12.7.14. Method random()

The method returns a point belonging to the segment.

Syntax: `z.M = L.AB:random()`

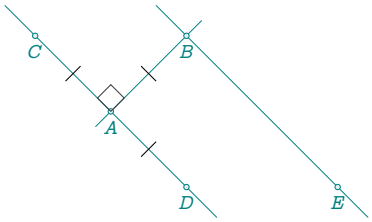
12.8. Returns a line

12.8.1. Method ll_from(pt)

Alias: **parallel_from**
Purpose: This method constructs a new object of type **line**, parallel to the original line and passing through the given point.

Unlike the **collinear_at** method, which simply returns a point translated along the same direction, **ll_from** returns a full line object. This allows further operations such as intersections, projections, or constructions involving new points along that line.

Use this method when you need a new line object rather than a point..
The choice between **ll_from** and **collinear_at** depends on the context and on whether you need to work with the resulting direction as a geometric object.




```
\directlua{
  init_elements()
  z.A = point(1, 1)
  z.B = point(2, 2)
  L.AB = line(z.A, z.B)
  z.C = L.AB.north_pa
  z.D = L.AB.south_pa
  L.CD = line(z.C, z.D)
  _, z.E = L.CD:ll_from(z.B):get()}
```

12.8.2. Comparison between collinear_at and ll_from methods

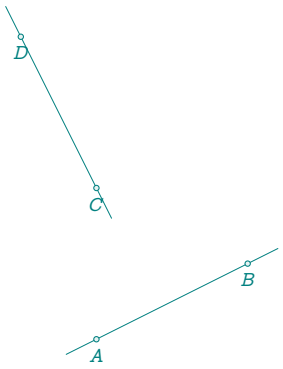
Aspect	collinear_at(pt, r)	ll_from(pt)
Return type	**point**	**line**
Purpose	Translated point	Parallel line
Input	Point, scalar (optional)	Point
Default scalar	**r = 1**	N/A
Use case	Vector displacement	Parallel construction

12.8.3. Method ortho_from(pt)

Alias: **orthogonal_from**
Purpose: This method constructs a new object of type **line**, perpendicular to the original line and passing through the given point.

Unlike the **orthogonal_at** method, which returns a **point** located at a given distance in the perpendicular direction, **ortho_from** returns a full **line** object. This makes it suitable for further geometric constructions such as intersections, projections, or drawing perpendiculars.

Use this method when a perpendicular line is needed as a geometric object.
Choose between **orthogonal_at** and **ortho_from** depending on whether you need a point or a line.




```
\directlua{
init_elements()
z.A = point(1, 1)
z.B = point(3, 2)
L.AB = line(z.A, z.B)
z.C = point(1, 3)
L.CD = L.AB:ortho_from(z.C)
z.D = L.CD.pb}
```

12.8.4. Comparison between orthogonal_at and ortho_from

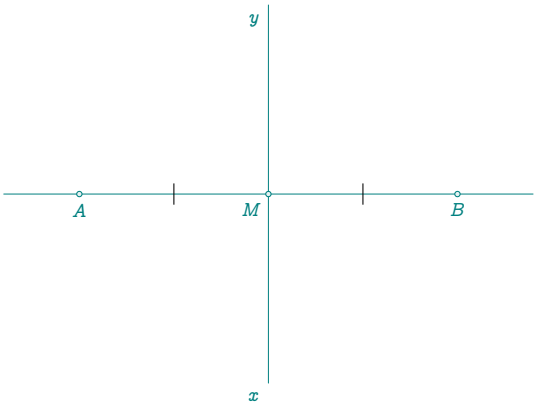
Aspect	`orthogonal_at(pt, r)`	`ortho_from(pt)`
Return type	**`point`**	**`line`**
Purpose	Perpendicular displacement	Perpendicular line
Input	Point, scalar (optional)	Point
Default scalar	**`r = 1`**	N/A
Use case	Construct a perpendicular point	Build a perpendicular line

12.8.5. Method mediator()

In mathematical literature (e.g., *MathWorld*), the *mediator* of a segment—also known as the *perpendicular bisector*—is defined as the line that passes through the midpoint of a segment and is perpendicular to it. It is sometimes called a *mediating plane* in 3D geometry.
The term *mediator* has historical roots, and was notably used by J. Neuberg (see Altshiller-Court, 1979, p. 298). In this package, I have chosen to adopt the French term *médiatrice*, translated here as **`mediator`**.

The method returns a new **`line`** object that is perpendicular to the original and passes through its midpoint.

`Alias:` You may also call this method using the alternative names **`perpendicular_bisector()`** or **`bisector()`**.




```
\directlua{
  init_elements()
  z.A = point(0,0)
  z.B = point(5,0)
  L.AB = line(z.A, z.B)
  L.med = L.AB:mediator()
  z.M = L.AB.mid
  z.x, z.y = L.med:get()}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLine(A,B)
  \tkzDrawSegments(x,y)
  \tkzDrawPoints(A,B,M)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[below left](x,y,M)
  \tkzMarkSegments(A,M M,B)
\end{tikzpicture}
\end{center}
```

12.8.6. Method collinear_at_distance(d)

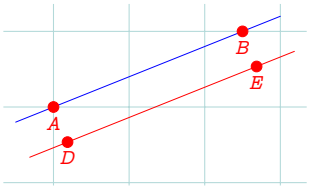
`Description:` This method creates a new line parallel to the current one, at a signed distance `d`. The sign of `d` follows the usual convention: `d > 0` shifts the line to the *left* of the oriented segment $\overrightarrow{pa\,pb}$, and `d < 0` to the *right*.

`Syntax:` `L' = L:collinear_at_distance(d)`

Arguments: d — signed offset distance (real number, same unit as coordinates).

Return value:
A new **line** object parallel to L at distance d.

Example usage:




```
\directlua{
  z.A = point(0, 0)
  z.B = point(5, 2)
  L.AB = line (z.A, z.B)
  L.n = line(collinear_at_distance_(z.A, z.B, -1))
  z.D, z.E = L.n:get()}
\begin{tikzpicture}[gridded,scale=.5]
\tkzGetNodes
\tkzDrawLines[blue](A,B)
\tkzDrawLines[red](D,E)
\tkzDrawPoints[red,size=4](A,B,D,E)
\tkzLabelPoints[red](A,B,D,E)
\end{tikzpicture}
```

12.8.7. Method swap_line

Purpose: This method is not intended for frequent use, but it can be useful in specific contexts. When a line is created, it is defined by two points—**pa** and **pb**—which determine the direction of the line. In certain geometric constructions (notably involving conics), the orientation of the line can affect the outcome of computations or methods.

The **swap_line()** method returns a new **line** object with the order of the defining points reversed, effectively reversing the direction of the line.

This is particularly relevant when direction-sensitive operations (e.g., projections, asymptotes, or oriented intersection tests) are involved.


```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(2, -1)
 L.dir = line(z.A, z.B)
 L.dir = L.dir:swap_line()
 z.a = L.dir.pa
 z.b = L.dir.pb}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawSegments[cyan,thick,->](a,b)
  \tkzLabelPoints[below](A,B)
\end{tikzpicture}
```

12.9. Returns a triangle

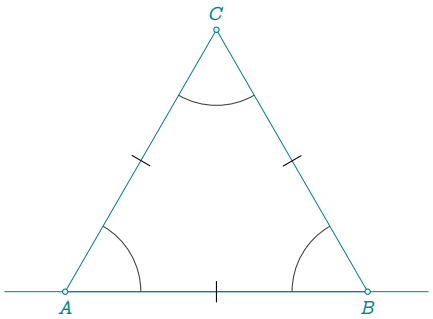
12.9.1. Method equilateral(<'swap'>)

This method constructs an equilateral triangle using the segment defined by the line as its base.

The triangle has three equal sides, and its base is the segment from **pa** to **pb**. The construction respects the orientation of the base.

- By default, the triangle is built in the *direct (counterclockwise)* orientation: the triangle has vertices A, B, C where AB is the base.
- If the optional argument **"swap"** is provided, the orientation is reversed: the triangle becomes A, B, C in indirect (clockwise) order.

This method returns a **triangle** object.




```
\directlua{
 init_elements()
 z.A = point(0,0)
 z.B = point(4,0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:equilateral()
 z.C = T.ABC.pc}
```

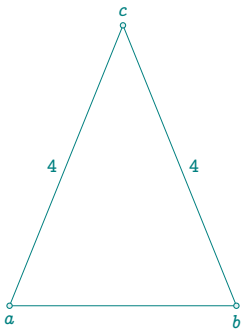
12.9.2. Method isosceles(d, <'swap'>)

This method constructs an isosceles triangle having the segment defined by the line as its base.

The argument **d** specifies the length of the two equal sides extending from each endpoint of the base. The triangle is constructed in the default orientation determined by the direction from **pa** to **pb**.

If the optional argument ⟨"swap"⟩ is provided, the triangle is constructed in the opposite orientation (i.e., the apex is placed on the other side of the base).

The method returns a **triangle** object.



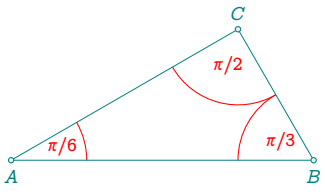

```
\directlua{
  init_elements()
  z.a = point(1, 2)
  z.b = point(3, 1)
  L.ab = line(z.a, z.b)
  T.abc = L.ab:isosceles(4)
  z.c = T.abc.pc}
```

12.9.3. Method school(<'swap'>)

The **school** triangle is a right triangle with angles measuring 30°, 60°, and 90°—commonly used in elementary geometric constructions.

By default, the 30° angle is at the first point (**pa**) of the line, and the 60° angle is at the second point (**pb**). Using the optional argument ⟨"swap"⟩ reverses this placement, exchanging the two base angles.

To reverse the triangle's orientation (i.e., construct it in the indirect direction), simply reverse the order of the points defining the line, for example: **L.AB = line(z.B, z.A)**.



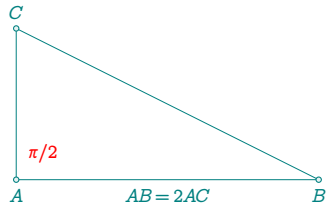

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0)
  L.AB = line(z.A, z.B)
  T.ABC = L.AB:school()
  z.C = T.ABC.pc}
```

12.9.4. Method half(<'swap'>)

This method constructs a right triangle in which the two sides adjacent to the right angle are in the ratio 1 : 2.

By default, the right angle is located at the first point of the line (**pa**). If the optional argument ⟨"swap"⟩ is provided, the right angle is placed at the second point (**pb**).

This type of triangle is useful for constructing simple geometric configurations or illustrating special triangle cases.



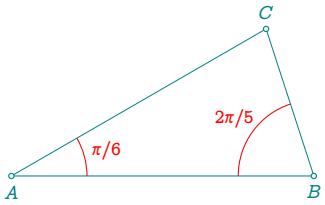

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0)
  L.AB = line(z.A, z.B)
  T.ABC = L.AB:half('swap')
  z.C = T.ABC.pc}
```

12.9.5. Method two_angles(an, an)

This method constructs a triangle by specifying two angles located at each endpoint of the given segment. The triangle is completed by determining the third vertex so that the sum of the interior angles is 180°.

The two given angles are applied at the endpoints **pa** and **pb** of the line. The resulting triangle is determined by extending the sides from these angles until they meet.

This method corresponds to the classical ASA construction (*Angle–Side–Angle*) and may also be called using the alternative names **asa** or **a_a**.



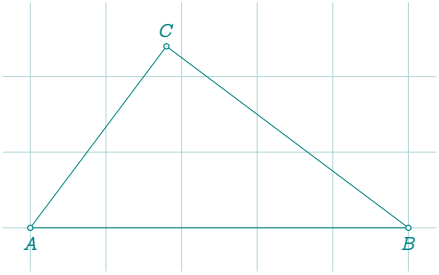

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:two_angles(math.pi / 6,2 *  math.pi / 5)
 z.C = T.ABC.pc}
```

12.9.6. Method s_s(d, d)

This method constructs a triangle given the lengths of its three sides: the base and the two sides adjacent to it.

The base of the triangle is defined by the line object (e.g., **L.AB**), which determines the segment between **pa** and **pb**. The two arguments **d** represent the lengths of the remaining sides.

This construction corresponds to the classical **SSS** configuration (*Side–Side–Side*), and the method may also be called using the aliases **s_s** or **sss**.



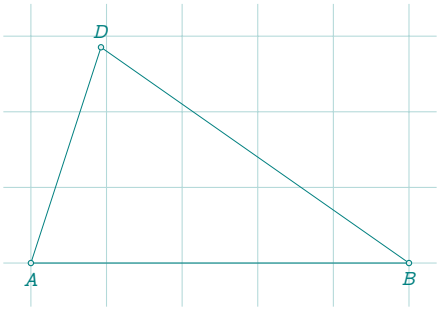

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  L.AB = line(z.A, z.B)
  T.ABC = L.AB:s_s(3, 4)
  z.C = T.ABC.pc}
```

12.9.7. Method sa_(d, an, <'swap'>)

This method constructs a triangle from a given base (the line), a side length, and the angle between that side and the base.

It corresponds to the classical **SAS** configuration (*Side–Angle–Side*). The side of length **d** is placed at the first point **pa** of the base, and the angle **an** (in degrees) is formed between this side and the base.

If the optional argument ⟨"swap"⟩ is provided, the side and angle are instead applied from the second point **pb**, effectively mirroring the triangle across the base.



12.9.8. Method _as(d, an,<'swap'>)

This method is the counterpart of **sa_**. It constructs a triangle using a given base (the line), a side of length **d**, and the angle **an** adjacent to that side, but this time measured from the second point **pb** of the base.

The construction still corresponds to the classical **SAS** configuration (*Side–Angle–Side*), but applied in reverse order from the base's endpoint.

If the optional argument ⟨"swap"⟩ is provided, the triangle is built in indirect (clockwise) orientation, which may be useful for constructing mirrored configurations.

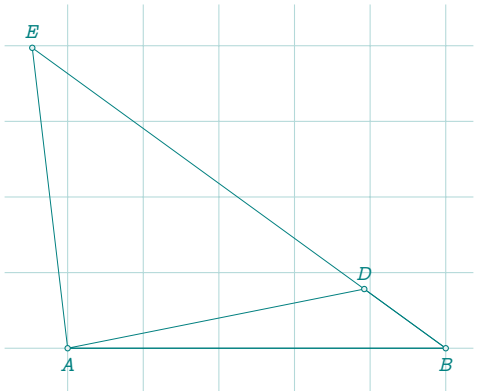
12.9.9. Method s_a(d, an, <'swap'>)

This method constructs a triangle from a given base (the line), a side of length **d**, and an angle **an** opposite that side.

The construction corresponds to the classical **SSA** configuration (*Side–Side–Angle*), which is known to be ambiguous: depending on the values of the side and angle, the result may be:

- no triangle (invalid configuration),
- exactly one triangle,
- or two possible triangles.

When two triangles are possible, the method returns the one with the larger angle at the base by default. If the optional argument ⟨"swap"⟩ is provided, the method returns the second solution.



12.9.10. Method a_s(d,an,<'swap'>)

12.9.11. Summary of triangle construction methods from a line

Method	Type	Parameters	Angle configuration
`equilateral()`	Equilateral	none	60° – 60° – 60°
`isosceles(d)`	Isosceles	one side	Two equal angles
`school()`	Right	none	30° – 60° – 90°
`half()`	Right	none	One angle with $\tan\theta = 1/2$
`two_angles(a,b)`	ASA	two angles	Two angles at endpoints
`s_s(d,d)`	SSS	two sides	Three known sides
`sa_(d,an)`	SAS	side + angle at `pa`	Included angle at first point
`_as(d,an)`	SAS	side + angle at `pb`	Included angle at second point
`s_a(d,an)`	SSA	side + opposite angle	Possible ambiguity

12.10. Returns a sacred triangle

The names attributed to these triangles are traditional or symbolic and may differ from those used in standard mathematical literature.

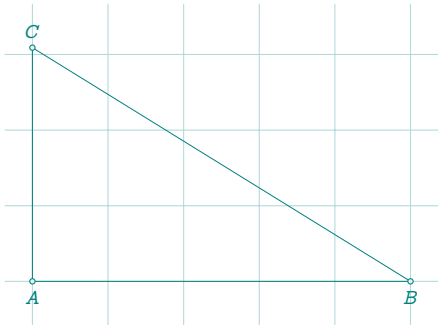
Name (Method)	Definition / Properties
`gold()`	Right triangle with $c/b = \sqrt{\varphi}$; half of the golden rectangle
`golden()`	Isosceles triangle with $b/c = \varphi$, angles $\alpha = \beta = \dfrac{2\pi}{5}$; Also called: sublime triangle, Euclid's triangle
`golden_gnomon()`	Isosceles triangle with $b/c = 1/\varphi$, angles $\alpha = \beta = \dfrac{\pi}{5}$
`pythagoras()`	Right triangle with sides $a = 5k$, $b = 4k$, $c = 3k$; Also known as the Egyptian or Isis triangle

12.10.1. Method gold()

This method constructs a right triangle in which the ratio of the lengths of the two sides adjacent to the right angle is equal to φ, the golden ratio:

$$\frac{c}{b} = \sqrt{\varphi}$$

This triangle corresponds to half of a golden rectangle.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  L.AB = line(z.A, z.B)
  T.ABC = L.AB:gold()
  z.C = T.ABC.pc}
```

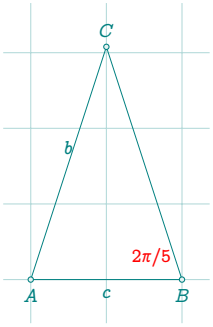
12.10.2. Method golden()

A golden triangle—also known as a *sublime triangle*—is an isosceles triangle in which the equal sides are in the golden ratio φ to the base.

In this construction, the ratio of a duplicated side b to the base c satisfies:

$$\frac{b}{c} = \varphi$$

This triangle appears in many classical constructions, notably in pentagonal geometry and Euclidean aesthetics.



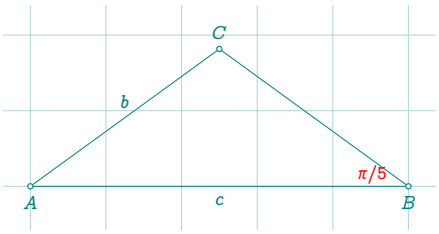

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(2, 0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:golden()
 z.C = T.ABC.pc}
```

12.10.3. Method golden_gnomon()

The **golden_gnomon**, also called the *divine triangle*, is an obtuse isosceles triangle in which the ratio of the side length to the base is equal to the inverse of the golden ratio:

$$\frac{b}{c} = \frac{1}{\varphi} = \varphi - 1$$

This triangle has two angles measuring 36° and one angle of 108°. It can be constructed geometrically from a regular pentagon and frequently appears in golden ratio-based constructions.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(5, 0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:divine()
 z.C = T.ABC.pc}
```

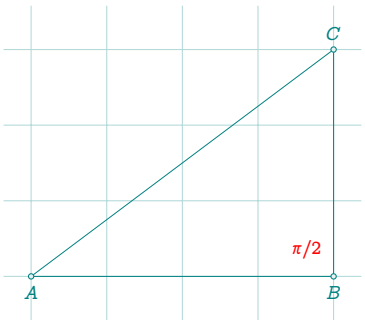
12.10.4. Method pythagoras()

This method constructs the classical right triangle whose side lengths are in the ratio 3:4:5.

Also known as the *Egyptian triangle* or *Isis triangle*, it is a Pythagorean triangle with integer side proportions:

$$a = 3k, \quad b = 4k, \quad c = 5k$$

It is one of the most fundamental triangles in Euclidean geometry and appears in many historical constructions.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(5, 0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:egyptian()
 z.C = T.ABC.pc}
```

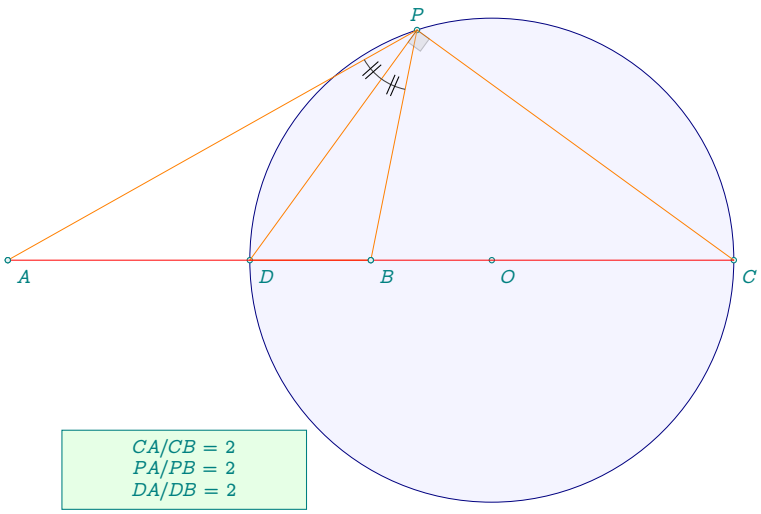
12.11. Returns a circle

12.11.1. Method apollonius(d)

Given two points A and B, this method constructs the Apollonius circle: the locus of points M such that the ratio of distances to A and B is constant:

$$\frac{MA}{MB} = d$$

The resulting object is a circle that does not pass through A or B (unless $d = 1$, in which case it becomes the perpendicular bisector of segment $[AB]$). The method returns a **circle** object.

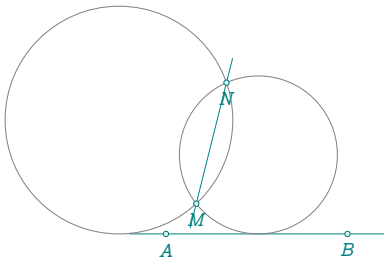



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, 0)
  L.AB = line(z.A, z.B)
  C.apo = L.AB:apollonius(2)
  z.O,z.C = C.apo:get()
  z.D = C.apo:antipode(z.C)
  z.P = C.apo:point(0.30)}

\begin{center}
\begin{tikzpicture}[scale=.8]
  \tkzGetNodes
  \tkzFillCircle[blue!20,opacity=.2](O,C)
  \tkzDrawCircle[blue!50!black](O,C)
  \tkzDrawPoints(A,B,O,C,D,P)
  \tkzDrawSegments[orange](P,A P,B P,D B,D P,C)
  \tkzDrawSegments[red](A,C)
  \tkzDrawPoints(A,B)
  \tkzLabelCircle[draw,fill=green!10,%
      text width=3cm,text centered,left=24pt](O,D)(60)%
    {$CA/CB=2$\\$PA/PB=2$\\$DA/DB=2$}
    \tkzLabelPoints[below right](A,B,O,C,D)
    \tkzLabelPoints[above](P)
  \tkzMarkRightAngle[opacity=.3,fill=lightgray](D,P,C)
  \tkzMarkAngles[mark=||](A,P,D D,P,B)
\end{tikzpicture}
\end{center}
```

Remark: \tkzUseLua{tkz.length(z.P,z.A)/tkz.length(z.P,z.B)} = 2.0

12.11.2. Method LPP(p, p)

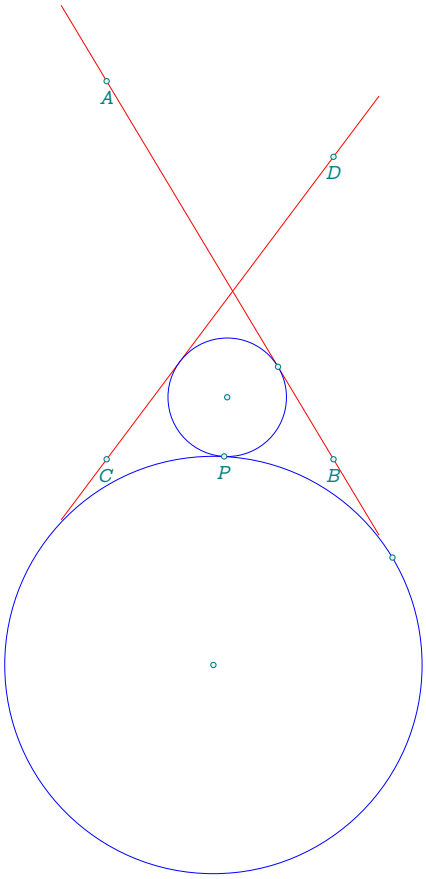
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, 0)
  z.M = point(1, 1)
  z.N = point(2, 5)
  L.AB = line(z.A, z.B)
  PA.center,
  PA.through, n = L.AB:LPP(z.M, z.N)
  tkz.nodes_from_paths(PA.center,
                       PA.through, "O", "T")
}

\begin{tikzpicture}[scale = .4]
  \tkzGetNodes
  \tkzDrawLines(A,B M,N)
  \tkzDrawCircles(O1,T1 O2,T2)
  \tkzDrawPoints(A,B,M,N)
  \tkzLabelPoints(A,B,M,N)
\end{tikzpicture}
```

12.11.3. Method LLP(L, p)

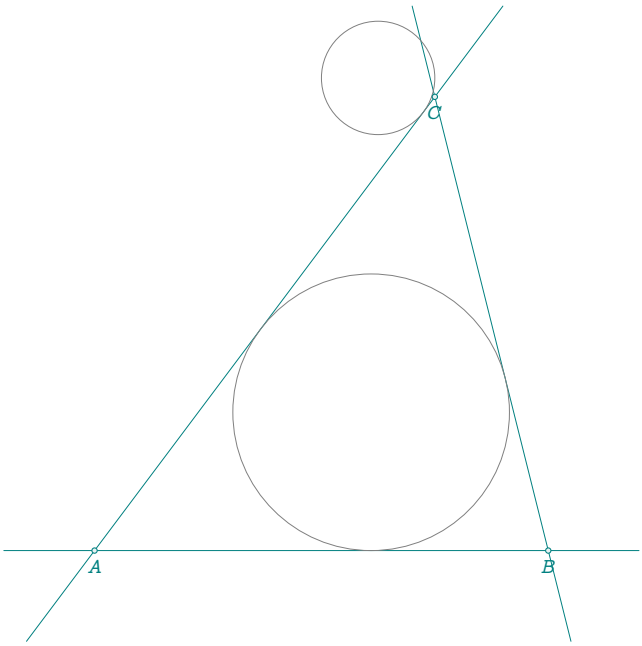
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(-1, 2)
  z.B = point(2, -3)
  z.C = point(-1, -3)
  z.D = point(2, 1)
  L.AB = line(z.A, z.B)
  L.CD = line(z.C, z.D)
  z.S = intersection(L.AB, L.CD)
  L.SI = tkz.bisector(z.S, z.A, z.D)
  z.P = L.SI:point(-1)
  local centers,
  throughs, n = L.AB:LLP(L.CD, z.P)
  tkz.nodes_from_paths(centers, throughs)
}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[red](A,B C,D)
  \tkzDrawCircles[blue](w1,t1 w2,t2)
  \tkzDrawPoints(A,B,C,D,P,w1,t1,w2,t2)
  \tkzLabelPoints(A,B,C,D,P)
\end{tikzpicture}
\end{center}
```

12.11.4. Method LLL(L1,L2[,which])

This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, 0)
  z.C = point(4.5, 6)
  T.ABC = triangle(z.A, z.B, z.C)
  L.AB = line(z.A, z.B)
  L.AC = line(z.A, z.C)
  L.med = L.AB:mediator ()
  z.M = L.AB.mid
  z.x, z.y = get_points(L.med)
  z.H = L.AB:projection(z.C)
  L.ortho = L.AB:orthogonal_from(z.C)
  PA.center,
  PA.through = L.AC:LLL(L.med, L.ortho)
  z.w = PA.center:get(1)
  z.t = PA.through:get(1)
  PA.center,
  PA.through = L.AB:LLL(L.AC, T.ABC.bc)
  z.o = PA.center:get(1)
  z.h = PA.through:get(1)}
\begin{center}
  \begin{tikzpicture}
\tkzGetNodes
\tkzDrawLines(A,B A,C B,C)
\tkzDrawCircles(w,t o,h)
\tkzDrawPoints(A,B,C)
\tkzLabelPoints(A,B,C)
\end{tikzpicture}
  \end{center}
```

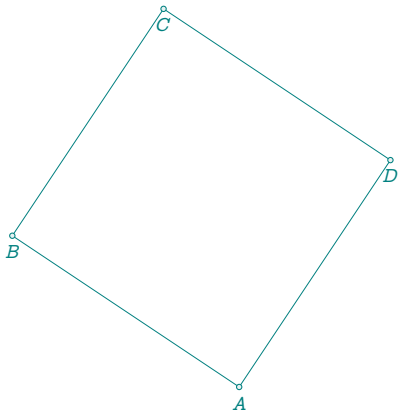
12.12. The result is a square

12.12.1. Method square(<'swap'>)

This method constructs a square using the segment defined by the line as one of its sides. The resulting square shares the segment $[AB]$ as a base and constructs the remaining two vertices so that all angles are right angles and all sides are equal in length.

By default, the square is built in the direct orientation from A to B. If the optional argument ⟨"swap"⟩ is provided, the square is built on the opposite side, reversing its orientation.

The result is a **square** object, and its vertices can be accessed using the **get()** method.




```
  \directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(-3, 2 )
  L.AB = line(z.A, z.B)
  S.AB = L.AB:square("swap")
  _,_,z.C,z.D = S.AB:get()}

\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,...,D)
  \tkzDrawPoints(A,...,D)
  \tkzLabelPoints(A,...,D)
\end{tikzpicture}
```

12.13. Transformations: the result is an object

This section presents geometric transformations whose result is a new geometric object. The nature of the result depends on both the transformation and the type of argument passed.

12.13.1. Method projection(obj)

This method projects one or more points orthogonally onto the line.

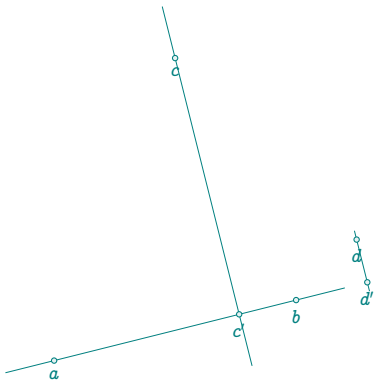
The argument **obj** may be:

- a single point object (e.g., **z.P**),
- a list (Lua table) of point objects.

The result is either:

- a single projected point, if the input is a point,
- a table of projected points, if the input is a list.

This method is useful for constructing foots of perpendiculars or projecting configurations onto a supporting line.




```
\directlua{
 init_elements()
 z.a = point(0, 0)
 z.b = point(4, 1)
 z.c = point(2, 5)
 z.d = point(5, 2)
 L.ab = line(z.a, z.b)
 z.cp, z.dp = L.ab:projection(z.c, z.d)}
\begin{center}
\begin{tikzpicture}[scale  = .8]
  \tkzGetNodes
  \tkzDrawLines(a,b c,c' d,d')
  \tkzDrawPoints(a,...,d,c',d')
  \tkzLabelPoints(a,...,d,c',d')
\end{tikzpicture}
\end{center}
```

12.13.2. Method projection_ll(L, obj)

This method performs a projection of a point or a group of points onto a line that is parallel to another reference line.

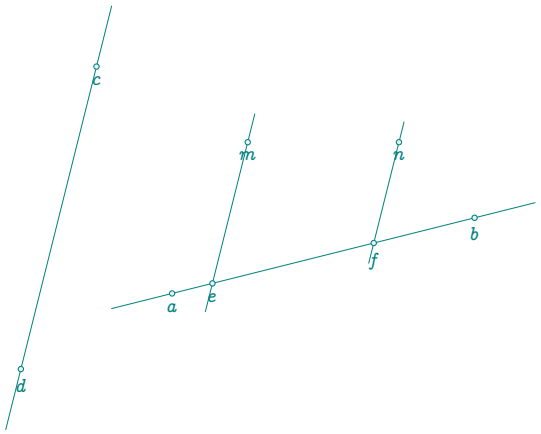
Unlike the standard orthogonal projection (using the line itself), this method projects onto a line that is parallel to the reference line **L**, and not necessarily coincident with it.

The method accepts the following arguments:

- **L** — a **line** object that defines the direction of the projection,
- **obj** — a point or a list of points to be projected.

The projection is computed along lines parallel to **L**. The result is:

- a projected point, if the input is a single point,
- a list of projected points, if the input is a table.




```
\directlua{
  init_elements()
  z.a = point(0, 0)
  z.b = point(4, 1)
  z.c = point(-1, 3)
  z.d = point(-2, -1)
  z.m = point(1, 2)
  z.n = point(3, 2)
  L.ab = line(z.a, z.b)
  L.cd = line(z.c, z.d)
  z.e,
  z.f = L.ab:projection_ll(L.cd, z.m, z.n)}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(a,b c,d e,m f,n)
  \tkzDrawPoints(a,...,f,m,n)
  \tkzLabelPoints(a,...,f,m,n)
\end{tikzpicture}
\end{center}
```

12.13.3. Method affinity(L, k, obj)

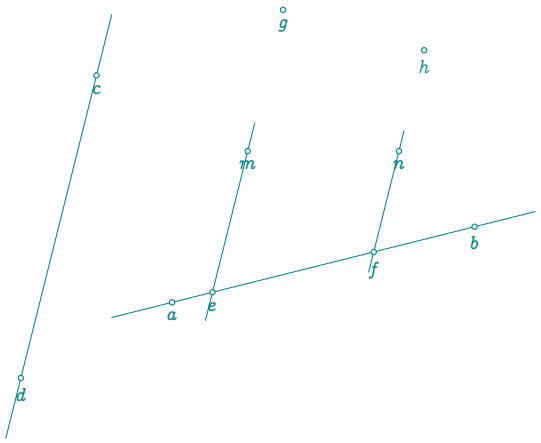
The introduction of parallel projection onto a given direction allows the definition of a new geometric transformation: *affinity*.

This method applies an affine transformation to a point or a group of points, using the line **L** to define the direction of the transformation. The points are projected onto the line associated with the current object along lines parallel to **L**.

Accepted arguments:

- **L** — a **line** object defining the direction of the affinity,
- **obj** — a point or a list of points to transform.

This transformation preserves parallelism and the ratio of distances along lines parallel to the axis of affinity.




```
\directlua{
  init_elements()
  z.a = point(0, 0)
  z.b = point(4, 1)
  z.c = point(-1, 3)
  z.d = point(-2, -1)
  z.m = point(1,2)
  z.n = point(3,2)
  L.ab = line(z.a, z.b)
  L.cd = line(z.c, z.d)
  z.e,
   z.f = L.ab:projection_ll(L.cd, z.m, z.n)
  z.g,
   z.h = L.ab: affinity(L.cd,2, z.m, z.n)
}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(a,b c,d e,m f,n)
  \tkzDrawPoints(a,...,h,m,n)
  \tkzLabelPoints(a,...,h,m,n)
 \end{tikzpicture}
\end{center}
```

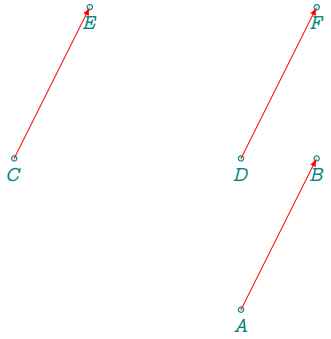
12.13.4. Method translation(obj)

This method performs a translation based on the vector defined by the segment from **pa** to **pb**, the two endpoints of the line.

The argument **obj** can be:

- a point,
- a line,
- a triangle,
- a circle.

The method applies the same translation vector to all components of the object. Other object types may be supported in future versions.




```
\directlua{
 init_elements()
 z.A  = point(0, 0)
 z.B  = point(1, 2)
 z.C  = point(-3, 2)
 z.D  = point(0, 2)
 L.AB = line(z.A, z.B)
 z.E, z.F = L.AB:translation(z.C, z.D)}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPoints(A,...,F)
  \tkzLabelPoints(A,...,F)
  \tkzDrawSegments[->,red,> =latex](C,E D,F A,B)
  \end{tikzpicture}
\end{center}
```

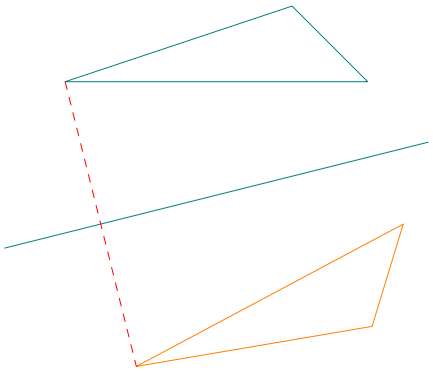
12.13.5. Method reflection(obj)

This method performs an axial (orthogonal) reflection with respect to the line. It corresponds to a symmetry with respect to the axis defined by the segment from **pa** to **pb**.

The argument **obj** can be:

- a point,
- a line,
- a triangle,
- a circle.

The method returns the reflected object across the line. Support for additional geometric objects may be added in future versions.



```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 1)
 z.E = point(0, 2)
 z.F = point(3, 3)
 z.G = point(4, 2)
 L.AB = line(z.A, z.B)
 T.EFG = triangle(z.E, z.F, z.G)
 T.new = L.AB:reflection(T.EFG)
 z.Ep, z.Fp, z.Gp = T.new:get()}

\begin{center}
  \begin{tikzpicture}
     \tkzGetNodes
     \tkzDrawLine(A,B)
     \tkzDrawPolygon(E,F,G)
     \tkzDrawPolygon[new](E',F',G')
     \tkzDrawSegment[red,dashed](E,E')
  \end{tikzpicture}
\end{center}
```

12.13.6. Method path(n): Creating a path

A line object can be sampled into a sequence of two points using the method **path(n)**. n indicates the number of segments between the two ends of the segment. This is typically used to create TikZ-compatible paths or intermediate constructions. This **path** exists mainly for compatibility and association with other **paths**.

Note:
By default, only the two endpoints are used to define the path, c'est à dire $n = 1$. This is suitable for filling or clipping regions. For smooth curves, a larger number of subdivisions is recommended.

```
L.AB = line(z.A, z.B)
PA.line = L.AB:path()
```

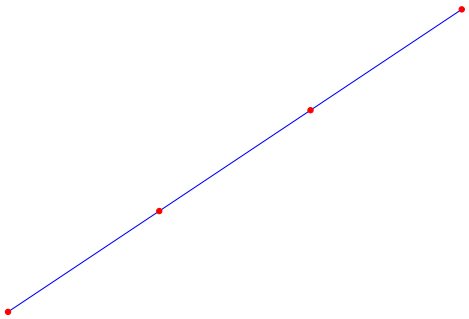
You can then draw the path using:

```
  \tkzDrawCoordinates(PA.line)
```

or draw the points with

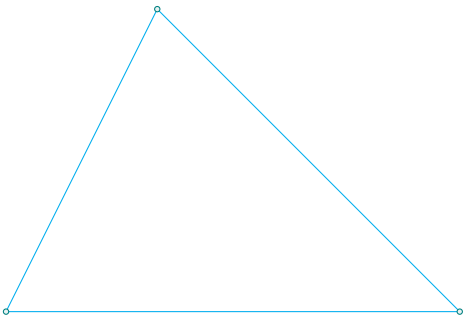
```
\tkzDrawPointsFromPath[red,size=2](PA.line)
```

Examples Three points and four intervals !



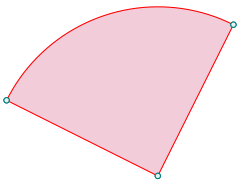
```
\directlua{
 z.A = point(0, 0)
 z.B = point(6, 4)
 L.AB = line(z.A, z.B)
 PA.AB = L.AB:path(3)
 }
\begin{tikzpicture}
\tkzDrawCoordinates[blue](PA.AB)
\tkzDrawPointsFromPath[red,size=2](PA.AB)
\end{tikzpicture}
```

Given three points, let's look at how, using the **path** method, we can draw the triangle defined by these three points. In this case, only the two ends of each side are required:



```
\directlua{
 z.O = point(0, 0)
 z.P = point(6, 0)
 z.Q = point(2, 4)
 T.OPQ = triangle(z.O, z.P, z.Q)
 PA.OP = T.OPQ.ab:path(2)
 PA.PQ = T.OPQ.bc:path(2)
 PA.QO = T.OPQ.ca:path(2)
 PA.zone = PA.OP +  PA.PQ + PA.QO}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[cyan](PA.zone)
 \tkzDrawPoints(O,P,Q)
\end{tikzpicture}
```

With `\tkzDrawCoordinates[cyan,fill = orange!10](PA.zone)`, you can fill inside the path.
In the following example, if you want to trace the contour of an angular sector, you'll need to use more points to define the various paths. Otherwise, distortions will appear at path junctions.



```
\directlua{
  z.A  = point(0, 0)
  z.B  = point(1, 2)
  C.AB = circle(z.A, z.B)
  z.E = C.AB:point(1 / 6)
  z.C = C.AB:point(0.25)
  z.D = C.AB:point(0.5)
  L.AB = line(z.A, z.B)
  L.CA = line(z.C, z.A)
  PA.arcBC = C.AB:path(z.B, z.C, 20)
  PA.AB = L.AB:path(5)
  PA.CA = L.CA:path(5)
  PA.sector = PA.AB + PA.arcBC + PA.CA
}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCoordinates[smooth,red,
  fill = purple!20](PA.sector)
\tkzDrawPoints(A,B,C)
\tkzDrawPoints(A,B,C)
\end{tikzpicture}
```

13. Class circle

The C variable is a table reserved for storing circle objects. Although its use is optional and any valid variable name may be used (e.g., **Circles**), it is strongly recommended to adopt the standard name C to ensure consistency and readability. If a custom variable is used, it must be initialized manually. The `init_elements()` function will reset the circleC table if it has already been defined.

13.0.1. Creating a circle

A circle is defined by two points:

- the center of the circle,
- a point lying on the circle.

To create a circle, use the following syntax:

```
C.OA = circle(z.O, z.A)
```

The newly created circle object stores various geometric attributes such as its radius, diameter, and notable points on the circumference (e.g., north, east, south, and west poles).

13.1. Attributes of a circle

A circle object stores various geometric attributes that can be accessed for further computation or drawing. These attributes are automatically computed at creation.

Table 6: Circle attributes.

Attributes	Reference
type	Type of circle, always "circle"
center	Center of the circle
through	Point through which the circle passes
radius	[13.1.1]
north	[13.1.1]
south	[13.1.1]
east	[13.1.1]
west	[13.1.1]
opp	[13.1.1]
ct	[13.1.1]
perimeter	[13.1.2]
area	[13.1.2]

13.1.1. Example: circle attributes

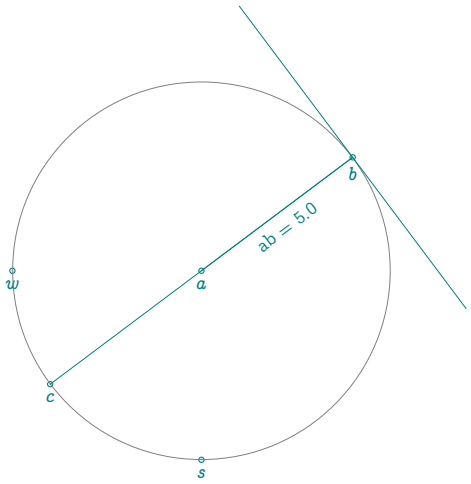
Several attributes of the **circle** class are illustrated in the following example.

The point diametrically opposite a given point on the circle can be obtained using the method:

```
z.Mp = C.OA:antipode(z.M)
```

When a circle object is created using **circle**, the point diametrically opposite the defining point (the one through which the circle passes) is automatically generated and stored in the attribute **opp**.

In addition, the line passing through the center and that point is also created and accessible via the attribute **ct** (center-to-through). This is particularly useful when dealing with a circle returned by a function, and you don't know in advance which points were used to construct it.




```
\directlua{
 init_elements()
 z.a = point(1, 1)
 z.b = point(5, 4)
 C.ab = circle(z.a, z.b)
 z.s = C.ab.south
 z.w = C.ab.west
 r = C.ab.radius
 z.c = C.ab.opp
 z.r,
 z.t = C.ab.ct:ortho_from(z.b):get()}
\begin{center}
\begin{tikzpicture}[scale=.5]
  \tkzGetNodes
  \tkzDrawPoints(a,b,c,s,w)
  \tkzLabelPoints(a,b,c,s,w)
  \tkzDrawCircle(a,b)
  \tkzDrawSegments(a,b r,t b,c)
  \tkzLabelSegment[sloped](a,b){ab = \tkzUseLua{r}}
  \end{tikzpicture}
\end{center}
```

13.1.2. Attributes perimeter and area

```
  \directlua{
  z.A = point(1, 2)
  z.B = point(4, 3)
  C.AB = circle(z.A, z.B)
  p = C.AB.perimeter
  a = C.AB.area}

Let be two points $A$ and $B$. The circle of center $A$ passing
through $B$ has perimeter \pmpn{\tkzUseLua{p}} $cm$
and area \pmpn{\tkzUseLua{a} }$cm^2$.
```

Let be two points A and B. The circle of center A passing through B has perimeter 19.8692 cm and area 31.4159cm^2.

13.2. Methods of the class circle

The circle class offers a wide range of methods, which can be grouped according to the type of value they return: numbers, booleans, strings, points, lines, or circles. These methods allow for computations such as checking inclusion, finding tangents, computing inversions, and more.

Table 7: Circle methods.

Methods	Reference
Constructor	
new(O,A)	Note[7]; [13.2.1; 13.0.1]
radius(O,r)	Deprecated; See [13.2.3]
diameter(A,B)	Deprecated; See [13.2.4]
Methods Returning a Real Number	
power(pt)	13.3.4]
Methods Returning a String	
circles_position(C1)	[13.5.1]
position(obj)	[13.3.5]
lines_position(L)	[13.3.8]
Methods Returning a Boolean	
is_tangent(L)	[13.3.1]
is_secant(L)	[13.3.2]
is_disjoint(L)	[13.3.3]
Methods Returning a Point	
get(i)	[13.6.1]
antipode(pt)	[13.6.2]
midarc(pt,pt)	[13.6.3]
point(r)	[13.6.4]
random_pt(<'inside'>)	[13.6.5]
inversion(obj)	[13.10.1; 13.10.1 13.10.1]
inversion_neg(obj)	[13.10.2]
internal_similitude(C)	[13.6.6]
external_similitude(C)	[13.6.7]
similitude(mode, C)	[13.6.8]
radical_center(C1<,C2>)	[13.6.9]
pole(L)	[13.6.10]
Methods Returning a Line	
radical_axis(C)	[13.7.6 ; 13.7.6; 13.7.6; 13.7.6; 13.7.6]
tangent_at(pt)	[13.7.1]
tangent_from(pt)	[13.7.2]
tangent_parallel(L)	[13.7.3]
common_tangent(C)	[13.7.4]
polar()	[13.7.5]

Table 8: Circle methods

Methods Returning a Circle	
orthogonal_from(pt)	[13.8.1]
orthogonal_through(pta,ptb)	[13.8.2]
midcircle(C)	[13.9.3; (ii)]
radical_circle(C1<,C2>)	[13.8.3]
CPP(pt,pt)	[13.8.4]
CCP(C,pt)	[13.8.5]
CLP(L,pt,<'inside'>)	[13.8.6]
CLL	13.8.7
CCL(C2, L)	13.9
CCC(C2, C3[, opts])	13.9.1
CCC_gergonne(C2, C3)	13.9.2
Methods Returning a Path	
path(pt,pt,nb)	[13.10.3]

Table 9: Auxiliary functions for circle construction.

Functions Returning two Points	
through(pt,r,<angle>)	See [13.2.2]
diameter(pt,pt,<'swap'> or <angle>)	See [13.2.2]

Each method can be called using the dot syntax, such as **C.OA:antipode(z.P)** or **C.OA:radical_axis(C1)**. Deprecated methods are retained for compatibility but should be avoided in new code.

13.2.1. Method new(pt, pt)

This method creates a new circle object defined by two points:

- the *center* of the circle,
- a *point on the circumference*, referred to as **through**.

The radius is computed as the Euclidean distance between the two points. Internally, the circle object stores both the center and the defining point, and computes several useful attributes for further geometric constructions.

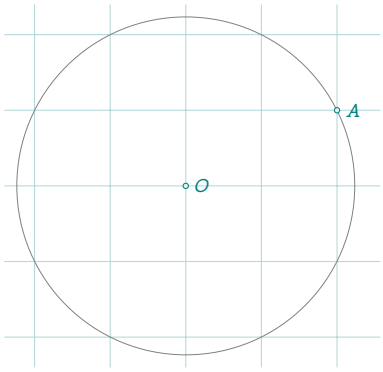
Short form:
The function **circle(center, through)** is equivalent to **circle:new(center, through)** and is recommended for ease of use.

```
C.OA = circle(z.O, z.A)
-- same as:
C.OA = circle:new(z.O, z.A)
```

Derived attributes:
Once created, the circle object contains:

- **radius** — the distance from the center to the point,
- **diameter** — twice the radius,
- **opp** — the point diametrically opposite to **through**,
- **ct** — the directed segment from center to through point.

These attributes are computed automatically and accessible via dot notation (e.g., **C.OA.radius**, **C.OA.opp**).




```
\directlua{
  init_elements()
  z.O = point(0, 0)
  z.A = point(2, 1)
  C.OA = circle(z.O, z.A)}
\begin{center}
\begin{tikzpicture}[gridded]
 \tkzGetNodes
 \tkzDrawCircle(O,A)
 \tkzDrawPoints(A,O)
 \tkzLabelPoints[right](A,O)
\end{tikzpicture}
  \end{center}
```

13.2.2. Functions through and diameter

These two utility functions assist in defining circles based on minimal data. They do not return a circle object themselves but instead return a pair of points: the *center* and a *point through which the circle passes*. These can be directly passed to **circle(...)**, as shown below.

```
C.OT = circle(through(z.O, 5))                      -- default angle = 0
C.OT = circle(through(z.O, 5, math.pi / 3))          -- point at 60°
C.OT = circle(diameter(z.A, z.B))                    -- through = B
C.OT = circle(diameter(z.A, z.B, "swap"))            -- through = A
```

1. Function **through(center, radius, <angle>])**

Constructs a circle based on:

- a center point,
- a radius (positive real),
- an optional angle (in radians) specifying the position of the point on the circumference.

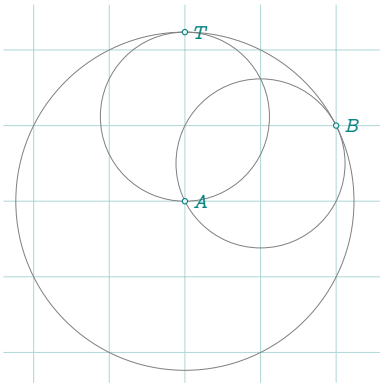
If the angle is omitted, the point lies on the positive x-axis from the center (i.e., angle $= 0$).

2. Function **diameter(A, B, <'swap'> or <angle>)**

Constructs a circle using two diametrically opposed points. The function returns:

- the midpoint of $[AB]$ as the center,
- one of the two points as the **through** point.

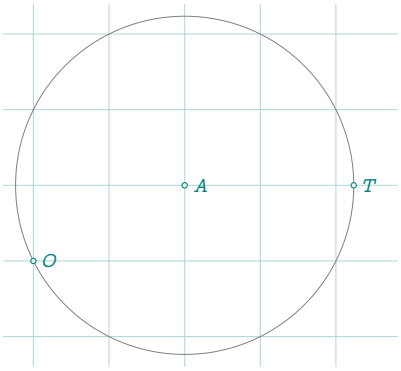
By default, the second argument (**B**) is returned as the **through** point. If the optional third argument **"swap"** is provided, the first point (**A**) is used instead. The optional angle (in radians) specifying the position of the point "through" on the circumference.




```
\directlua{
  z.A = point(0, 0)
  z.B = point(2, 1)
  C.T = circle(through(z.A, math.sqrt(5), math.pi / 2))
  C.a = circle(diameter(z.A, z.B))
  z.T = C.T.through
  C.b = circle(diameter(z.A, z.T, "swap"))
  z.w = C.a.center
  z.t = C.a.through
  z.u = C.b.center
  z.v = C.b.through}
\begin{tikzpicture}[gridded]
\tkzGetNodes
 \tkzDrawCircles(A,T w,t u,v)
 \tkzDrawPoints(A,B,T)
 \tkzLabelPoints[right](A,B,T)
\end{tikzpicture}
```

13.2.3. Method radius(pt,r)

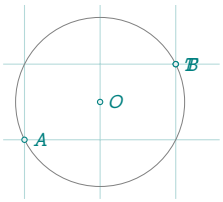
(Deprecated see above [13.2.2])
This method has been retained for backward compatibility but is no longer recommended. It has been replaced by a more flexible and construction-oriented approach.
We define a circle with its centre and radius.




```
\directlua{
  init_elements()
  z.O = point(0, 0)
  z.A = point(2, 1)
  C.T = circle:radius(z.A, math.sqrt(5))
  % better C.T = circle(from_radius(z.A, math.sqrt(5)))
  z.T = C.T.through }
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawCircles(A,T)
  \tkzDrawPoints(A,O,T)
  \tkzLabelPoints[right](A,O,T)
\end{tikzpicture}
\end{center}
```

13.2.4. Method diameter(pt,pt)

(Deprecated see above [13.2.2])
This method has been retained for backward compatibility but is no longer recommended. It has been replaced by a more flexible and construction-oriented approach.
A circle is defined by two points at the ends of one of its diameters.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(2, 1)
  C.T = circle:diameter(z.A, z.B)
  % better C.T = from_diameter(z.A, z.B)
  z.O = C.T.center
  z.T = C.T.through}
\begin{center}
\begin{tikzpicture}[gridded]
 \tkzGetNodes
 \tkzDrawCircles(O,T)
 \tkzDrawPoints(A,B,O,T)
 \tkzLabelPoints[right](A,B,O,T)
\end{tikzpicture}
\end{center}
```

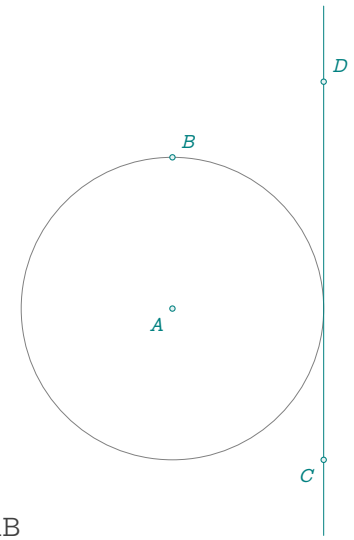
13.3. Returns a boolean value

13.3.1. Method is_tangent(L)

This method checks whether a given line is tangent to the circle. It returns a boolean value: **true** if the line is tangent, and **false** otherwise.

This is useful for logical tests and conditional constructions in Lua. The method uses geometric distance and precision tolerance to determine tangency.

Example:



L.CD tangent to C.AB

```
\directlua{
  z.A = point(0, 0)
  z.B = point(0, 2)
  C.AB = circle(z.A, z.B)
  z.C = point(2, -2)
  z.D = point(2, 3)
  L.CD = line(z.C, z.D)
  if C.AB:is_tangent(L.CD) then
    tex.print("L.CD tangent to C.AB")
  else
    tex.print("L.CD no tangent to C.AB")
  end}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircle(A,B)
  \tkzDrawLines(C,D)
  \tkzDrawPoints(A,B,C,D)
  \tkzLabelPoints[below left](A,C)
  \tkzLabelPoints[above right](B,D)
\end{tikzpicture}
```

13.3.2. Method is_secant(L)

see the example above

13.3.3. Method is_disjoint(L)

see the example above

13.3.4. Method power(pt)

The **power** method computes the *power of a point* with respect to the given circle. It is defined as:

$$\text{power}(P) = \texttt{OP}^2 - r^2$$

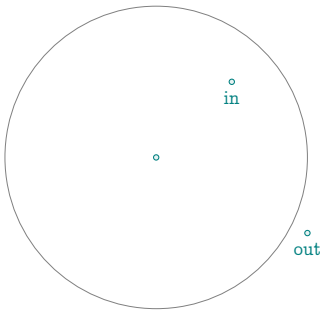
where O is the center of the circle and r its radius.

The sign of the result allows us to determine the relative position of the point P:

- positive: the point lies outside the circle,
- zero: the point lies on the circle,
- negative: the point lies inside the circle.

Here is an example using a function to print this information:

```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.R = point(2, 0)
 z.A = point(1, 1)
 z.B = point(2, -1)
 C.OR = circle(z.O, z.R)
  function position(pt)
 if C.OR:power(pt) > 0
 then
   return tex.print("out")
  else
   return tex.print("in")
 end
  end}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircle(O,R)
  \tkzDrawPoints(A,O,B)
  \tkzLabelPoint(A){\tkzUseLua{position(z.A)}}
  \tkzLabelPoint(B){\tkzUseLua{position(z.B)}}
\end{tikzpicture}
```



13.3.5. Method position(obj)

This method determines the geometric relation between a circle and another object. An optional argument **EPS** may be provided to adjust the numerical tolerance. By default, **EPS** is the global value **tkz.epsilon**.

Syntax:
`result = C:position(obj [, EPS])`

Supported object types and return values

- Point

  Returns one of:

  - **"IN"** — point strictly inside the disk,
  - **"ON"** — point on the circumference,
  - **"OUT"** — point outside the disk.

- Line

  Returns one of:

  - **"DISJOINT"** — no intersection,
  - **"TANGENT"** — exactly one common point,
  - **"SECANT"** — two intersection points.

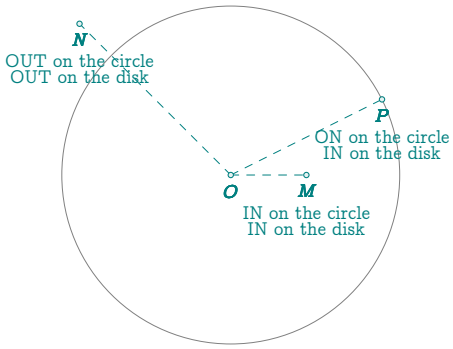
- Circle

  Returns one of:

  - **"DISJOINT_EXT"** — exterior disjoint circles,
  - **"TANGENT_EXT"** — exterior tangency,
  - **"SECANT"** — two intersections,
  - **"TANGENT_INT"** — interior tangency,
  - **"DISJOINT_INT"** — one circle strictly inside the other,
  - **"CONCENTRIC"** — same center, different radii,
  - **"COINCIDENT"** — identical circles.

Remarks

- Returned values are symbolic uppercase strings.
- Results are tolerance-aware.
- Unsupported object types raise an error.




```
\directlua{
  init_elements()
  z.O = point(0, 0)
  z.A = point(1 ,2)
  C.OA = circle(z.O, z.A)
  z.N = point(-2, 2)
  z.M = point(1, 0)
  z.P = point(2, 1)

  PCm = C.OA:position(z.M)
  PDm = C.OA:position_disk(z.M)
  PCn = C.OA:position(z.N)
  PDn = C.OA:position_disk(z.N)
  PCp = C.OA:position(z.P)
  PDp = C.OA:position_disk(z.P)
}

\def\tkzPosPoint#1#2#3#4{%
  \tkzLabelPoints(O,M,N,P)
  \tkzLabelPoint[below=#4pt,
  font=\scriptsize](#2){\tkzUseLua{#1} on the #3}%
}

\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawSegments[dashed](O,M O,N O,P)
  \tkzDrawCircle(O,A)
  \tkzDrawPoints(O,M,N,P)

  \tkzPosPoint{PCm}{M}{circle}{8}
  \tkzPosPoint{PCn}{N}{circle}{8}
  \tkzPosPoint{PCp}{P}{circle}{8}

  \tkzPosPoint{PDm}{M}{disk}{14}
  \tkzPosPoint{PDn}{N}{disk}{14}
  \tkzPosPoint{PDp}{P}{disk}{14}
  \end{tikzpicture}
\end{center}
```

13.3.6. Method in_out for circle and disk; Deprecated

The following methods **in_out()**, **in_out_disk()**, **in_disk(p)**, **in_out_disk(p)**, **in_disk_strict(p)** and **out_disk_strict(p)** have been replaced by **position** and **position_disk**. See [13.3.5]
Note: They still exist, but it is preferable to use **position**.

13.3.7. Method line_position(L)

Purpose:
This method classifies the position of a line relative to a circle.
It returns whether the line is disjoint, tangent or secant.

Syntax: pos = C.AB:line_position(L.CD)

Arguments:
L — a **line** object to be tested against the circle.

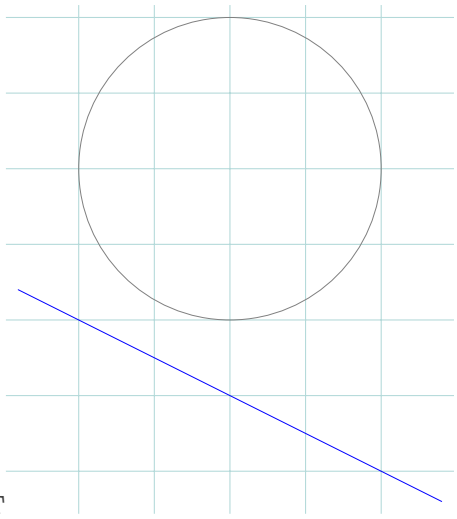
Return value:
A string describing the relative position of the line:

- "DISJOINT" — no intersection between line and circle,
- "TANGENT" — line tangent to the circle (one point of contact),
- "SECANT" — line intersecting the circle at two points.

Remarks:

- The radius is computed from the center and the defining point (self.through) to ensure numerical stability.
- The tolerance **tkz.epsilon** is used to handle rounding errors near tangency.

Example usage:




```
\directlua{
  z.A = point(0, 0)
  z.B = point(4, -2)
  L.AB  = line (z.A, z.B )
  z.O = point(2, 2)
  z.X = z.O + point(0,-2)
  C.OX = circle(z.O, z.X)
  tex.print(C.OX:line_position(L.AB))}
\begin{tikzpicture}[gridded]
 \tkzGetNodes
 \tkzDrawLines[blue](A,B)
 \tkzDrawCircle(O,X)
\end{tikzpicture}
```

13.3.8. Method lines_position(L1, L2, mode)

Purpose:
This method determines the relative position of a circle with respect to two lines. It analyzes which angular or strip sectors are touched by the circle depending on whether the lines are parallel or secant. The result is returned as a list of integers corresponding to the sectors intersected by the circle.
Syntax: sectors = C:lines_position(L1, L2 [, mode])
Arguments:

- L1, L2 — two **line** objects.
- mode — an optional string:
  - "parallel" — if the lines are known to be parallel;
  - any other value (or omitted) — default, for secant lines.

Return value:
A Lua table (list of integers) representing the indices of the sectors touched by the circle.

For secant lines:

- sectors are numbered counterclockwise from the first line to the second: 1, 2, 3, 4;
- for example, {1} means the circle is entirely inside the first sector, and {4, 1, 2} means the circle touches three adjacent sectors.

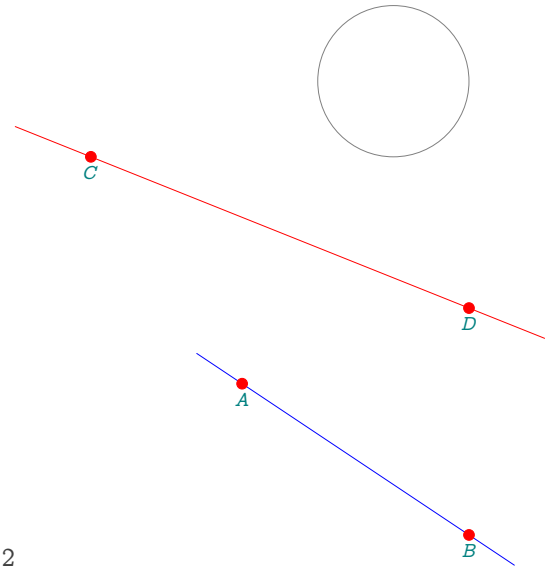
For parallel lines:

- sectors are numbered as: 1 – between the lines, 2 – outside on the side of L1, 3 – outside on the side of L2;
- examples: {1} → circle entirely between the lines; {1,2} → tangent to L1; {1,2,3} → circle large enough to cross both lines.

Remarks:

- The classification is purely geometric and does not compute intersection points.
- The tolerance **tkz.epsilon** is used to handle numerical approximations in tangency detection.

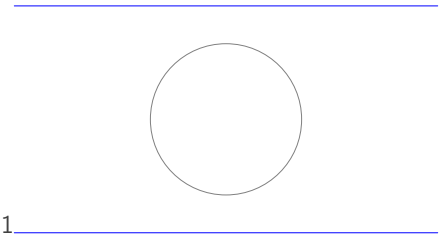
Example (secant lines)



2

```
\directlua{
  z.A = point(0, 0)
  z.B = point(3, -2)
  L.AB  = line (z.A, z.B)
  z.C = point(-2, 3)
  z.D = point(3, 1)
  L.CD  = line (z.C, z.D)
  z.O = point(2, 4)
  z.X = z.O + point(0, -1)
  C.OX = circle(z.O, z.X)
  tex.print(C.OX:lines_position(L.AB, L.CD))}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawLines[blue](A,B)
 \tkzDrawLines[red](C,D)
 \tkzDrawCircle(O,X)
 \tkzDrawPoints[red,size=4](A,B,C,D)
 \tkzLabelPoints(A,B,C,D)
\end{tikzpicture}
```

Example (parallel lines)



1

```
\directlua{
  z.A = point(0,0)
  z.B = point(4,0)
  L.AB = line(z.A, z.B)
  z.C = point(0,3)
  z.D = point(4,3)
  L.CD = line(z.C, z.D)
  z.O = point(2,1.5)
  z.T = z.O + point(0,1)
  C.OT = circle(z.O, z.T)
  local s = C.OT:lines_position(L.AB,
                    L.CD, "parallel")
  tex.print(s)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[blue](A,B C,D)
  \tkzDrawCircle(O,T)
\end{tikzpicture}
```

13.4. Returns a real number

13.4.1. Method power(pt)

The *power of a point A* with respect to a circle of radius r and center O is a classical notion in Euclidean geometry.

It is defined as the scalar quantity:

$$p = \overline{AP} \cdot \overline{AQ} = AM^2 - r^2$$

where:

- P and Q are the points of intersection of the circle with a line passing through A,
- M is the center O of the circle (so $AM^2 = \texttt{AO}^2$),
- and AT is a tangent from A to the circle, satisfying $AT^2 = AM^2 - r^2$.

Geometrically:

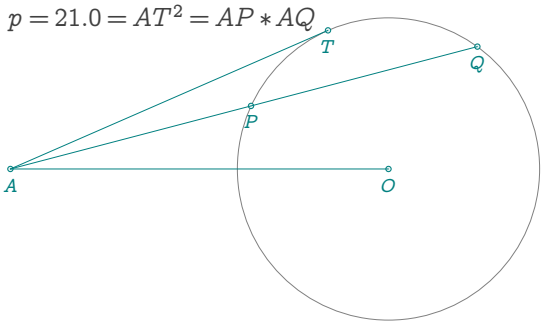
- If A lies outside the circle, the power is positive.
- If A lies on the circle, the power is zero.
- If A lies inside the circle, the power is negative.

In Lua, this is evaluated by calling:

```
p = C.XY:power(z.A)
```

This value can be used for algebraic computations or to test point inclusion via its sign.

```
\directlua{
init_elements()
z.O = point(5, 0)
z.A = point(0, 0)
z.R = point(7, 0)
C.OR = circle(z.O, z.R)
z.Q = C.OR:point(0.15)
L.AQ = line(z.A, z.Q)
_, z.P = intersection(C.OR,L.AQ)
L.T = C.OR:tangent_from(z.A)
z.T = L.T.pb}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircle(O,T)
\tkzDrawPoints(A,O,P,Q,T)
\tkzDrawSegments(A,O A,Q A,T)
\tkzLabelPoints(A,O,P,Q,T)
\tkzText(2,2){$p =\tkzUseLua{%
  C.OR:power(z.A)} =AT^2=AP * AQ$}
\end{tikzpicture}
```



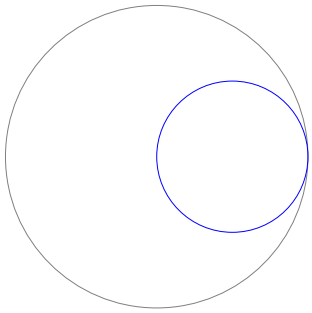
13.5. Returns a string

13.5.1. Method circles_position

This function returns a string indicating the position of the circle in relation to another. Useful for creating a function. Cases are:

- **outside**
- **outside tangent**
- **inside tangent**
- **inside**
- **intersect**

```
\directlua{
init_elements()
   z.A = point(0, 0)
   z.a = point(3, 0)
   z.B = point(2, 0)
   z.b = point(3, 0)
   C.Aa = circle(z.A, z.a)
   C.Bb = circle(z.B, z.b)
   position = C.Aa:circles_position(C.Bb)
   if position == "inside tangent"
   then color = "orange"
   else color = "blue" end}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircle(A,a)
  \tkzDrawCircle[color=\tkzUseLua{color}](B,b)
\end{tikzpicture}
```



13.6. Returns a point

13.6.1. Method get()

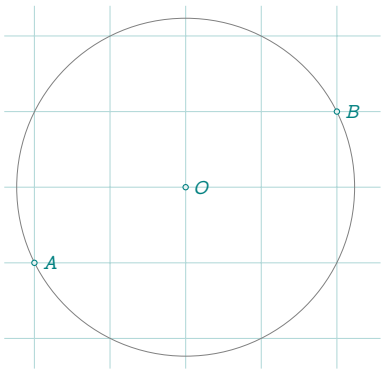
This method retrieves the defining points of a circle. A circle is determined by its center and one point on its circumference.

- **C:get()** returns the center and a point through which the circle passes.
- **C:get(1)** returns the center of the circle.
- **C:get(2)** returns the point on the circumference.

This method is useful whenever the geometric definition of a circle needs to be reused, for instance when constructing tangents, radical axes, or when passing circle data to other constructions. It provides a simple and consistent way to access the fundamental points that define a circle.

13.6.2. Method antipode

This method is used to define a point that is diametrically opposed to a point on a given circle.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.O = point(2, 1)
  C.OA = circle(z.O, z.A)
  z.B = C.OA:antipode(z.A)}
\begin{center}
\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawCircles(O,A)
  \tkzDrawPoints(A,B,O)
  \tkzLabelPoints[right](A,B,O)
\end{tikzpicture}
\end{center}
```

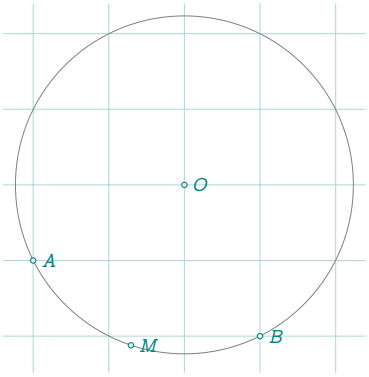
13.6.3. Method midarc

The classical definition, as found in MathWorld, is the following:

> The mid-arc points of a triangle, as defined by Johnson (1929), are the points on the circumcircle of the triangle that lie halfway along each of the three arcs determined by the triangle's vertices. These points arise in the definitions of the Fuhrmann circle and Fuhrmann triangle, and they lie on the extensions of the perpendicular bisectors of the triangle sides drawn from the circumcenter.

In the present context, the definition is generalized. The method returns the point on a circle that divides a given arc into two equal arcs in terms of angular measure. In other words, it computes the midpoint (in angle) of the arc defined by two points on the circle.

This method is applicable to any pair of points on a circle, not just those associated with triangle vertices or circumcircles.



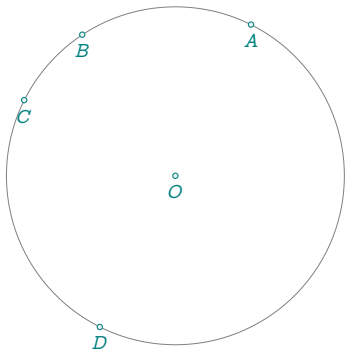

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.O = point(2, 1)
  C.OA = circle(z.O, z.A)
  z.B = C.OA:point(0.25)
  z.M = C.OA:midarc(z.A, z.B)}
\begin{center}
\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawCircles(O,A)
  \tkzDrawPoints(A,B,O,M)
  \tkzLabelPoints[right](A,B,O,M)
\end{tikzpicture}
  \end{center}
```

13.6.4. Method point(r)

Let C be a circle with centre O and passing through A such that `z.A = C.through`. This method defines a point M on the circle from A such that the ratio of the length of $\overset{\frown}{AM}$ to the circumference of the circle is equal to r.

In the next example, $r = \dfrac{1}{6}$ corresponds to $\dfrac{\pi/3}{2\pi}$, so the angle $\widehat{AOE}$ has the measure $\pi/3$.

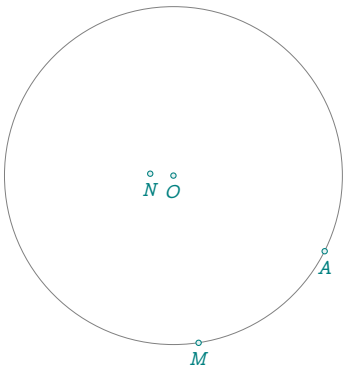
If $r = .5$ the defined point is diametrically opposed to A, the angle $\widehat{AOD}$ has the measure π.




```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = point(1, 2)
 C.OA = circle(z.O, z.A)
 z.B = C.OA:point(1 / 6)
 z.C = C.OA:point(0.25)
 z.D = C.OA:point(0.5)}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircle(O,A)
\tkzDrawPoints(A,...,D,O)
\tkzLabelPoints(A,...,D,O)
\end{tikzpicture}
\end{center}
```

13.6.5. Method random(<'inside'>)

Produces a point on the circle or inside the disc with the 'inside' option.




```
\directlua{
  init_elements()
  z.O = point(0, 2)
  z.A = point(2, 1)
  C.OA = circle(z.O, z.A)
  z.M = C.OA:random()
  z.N = C.OA:random('inside')}
```

13.6.6. Method internal_similitude(C)

This method computes the **internal center of similitude** of two given circles.

Two circles (in general position) always admit two homothetic centers:

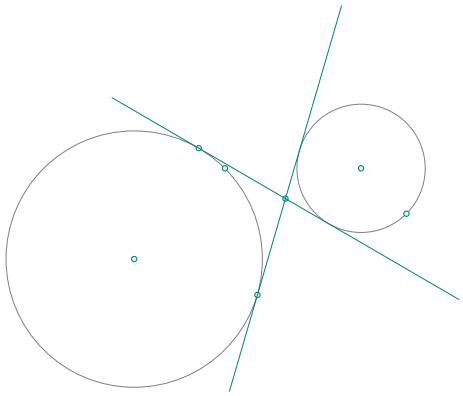
- the **internal center of similitude**, which lies between the two centers and divides the segment joining them internally in the ratio of their radii,
- the **external center of similitude**, which lies outside the segment and divides it externally in the same ratio.

These centers lie on the line joining the centers of the two circles, called the *line of centers*.

This method returns the internal homothetic center — the point from which the two circles appear in the same direction and proportion, as though one were scaled into the other internally.

Note: Degenerate cases (e.g., equal centers, equal or zero radii) are handled separately.

(See also: Wikipedia — Homothetic center)



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.a = point(2, 2)
  z.B = point(5 , 2)
  z.b = point(6 , 1)
  C.Aa = circle(z.A, z.a)
  C.Bb = circle(z.B, z.b)
  z.I = C.Aa:internal_similitude(C.Bb)
  L.TA1, L.TA2 = C.Aa:tangent_from(z.I)
  z.A1 = L.TA1.pb
  z.A2 = L.TA2.pb}
 \begin{center}
 \begin{tikzpicture}[ scale =.6]
   \tkzGetNodes
   \tkzDrawCircles(A,a B,b)
  \tkzDrawPoints(A,a,B,b,I,A1,A2)
  \tkzDrawLines[add = 1 and 2](A1,I A2,I)
 \end{tikzpicture}
 \end{center}
```

13.6.7. Method external_similitude(C)

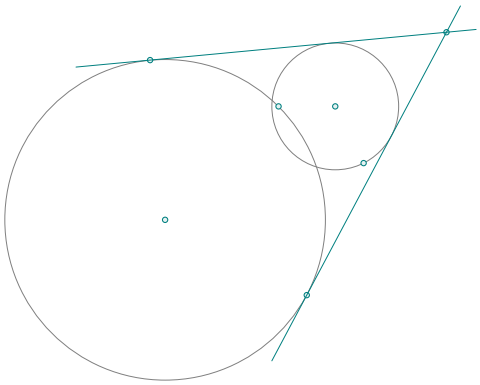
This method computes the **external center of similitude** of two given circles.

As with the internal case, two circles in general position admit two homothetic centers. The external center of similitude lies on the *line of centers*, but outside the segment joining the two centers.

It divides this line *externally* in a ratio equal to that of the radii of the circles. This center is the unique point from which the two circles appear as scaled images of each other in opposite orientations.

This construction is essential in many geometric transformations (e.g., inversion, similarity, coaxal systems) and in the theory of midcircles and Apollonius circles.

Note: As with the internal center, degenerate cases such as identical centers or zero radius are handled separately.



```
\directlua{
  init_elements()
  z.A = point (0 , 0 )
  z.a = point (2 , 2)
  z.B = point (3 , 2 )
  z.b = point(3.5, 1 )
  C.Aa = circle(z.A, z.a)
  C.Bb = circle(z.B, z.b)
  z.I = C.Aa:external_similitude(C.Bb)
  L.TA1,L.TA2 = C.Aa:tangent_from(z.I)
  z.A1 = L.TA1.pb
  z.A2 = L.TA2.pb}
\begin{center}
\begin{tikzpicture}[scale = .75]
\tkzGetNodes
\tkzDrawCircles(A,a B,b)
\tkzDrawPoints(A,a,B,b,I,A1,A2)
\tkzDrawLines[add = .25 and .1](A1,I A2,I)
\end{tikzpicture}
\end{center}
```

13.6.8. Method similitude(C,mode)

This method unifies the two similitude-center functions.

- **mode = "internal"** returns the internal center of similitude.
- **mode = "external"** returns the external center of similitude.

Example:

```
S  = C1:similitude(C2, "internal")
Se = C1:similitude(C2, "external")
```

13.6.9. Method radical_center(C1, C2)

In classical geometry, the **radical center** (also called the *power center*) of three circles is the unique point of concurrency of their pairwise radical axes.

This fundamental result, attributed to Gaspard Monge (see Dörrie 1965, p.153), states that:

> The radical axes of any three circles (no two of which have the same center) intersect in a single point — the radical center.

(See also: MathWorld — Radical Center)

In this implementation, the method **radical_center(C1, C2)** also has a well-defined meaning when applied to only two circles. The point returned by

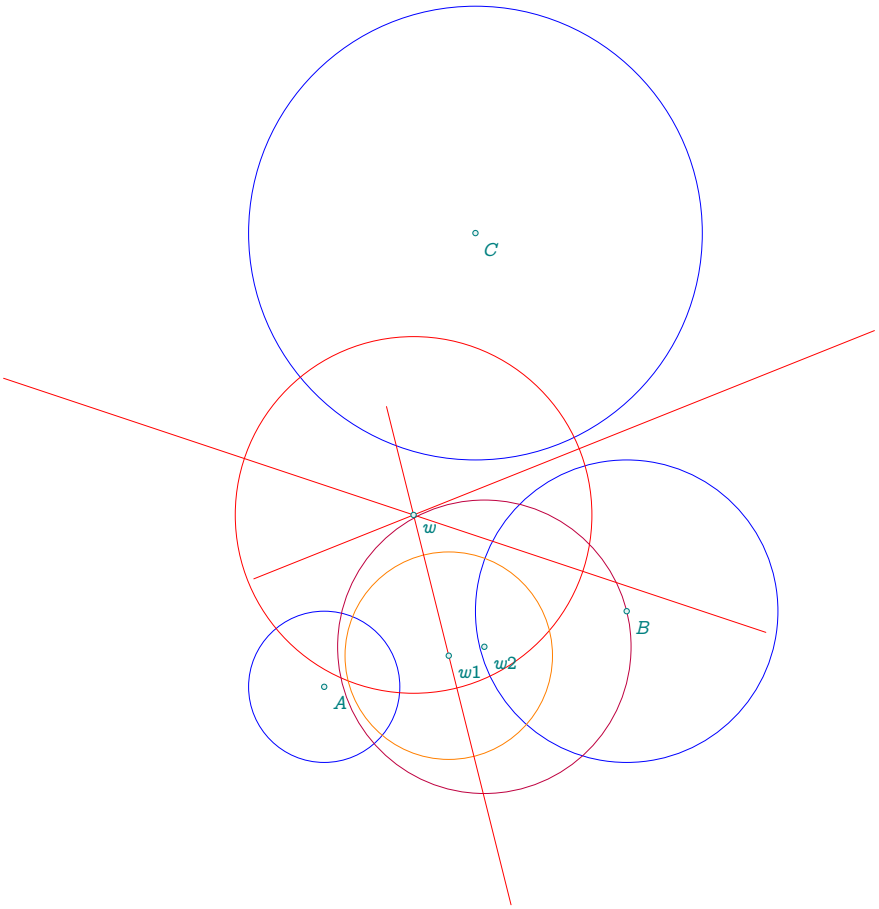
z.P = C1:radical_center(C2)

is the intersection of the *radical axis* of the two circles with the line joining their centres.
This point may be regarded as an *extended radical center*. It is the centre of a distinguished circle orthogonal to both given circles. This circle belongs to the coaxial pencil of circles orthogonal to **C1** and **C2**; its intersections with the line joining the centres coincide with the two fixed base points of this pencil.
The construction naturally extends to the case of a circle and a point. A point P is interpreted as a *circle-point*, that is, a circle of zero radius:

z.P = point(a,b), **C.P = circle(z.P,z.P)**.

In this setting, the radical center is the centre of the unique circle orthogonal to the given circle and passing through the point P.

This point is useful for constructions involving inversion, coaxal systems, and special configurations such as the Apollonius circle.
Example, with three circles, two circles and one circle and a point



```
\directlua{
 z.A = point(0, 0)
 z.a = z.A + point (1, 0)
 z.B = point (4, 1)
 z.b = z.B + point (2, 0)
 z.C = point (2, 6)
 z.c = z.C + point (3, 0)
 C.Aa = circle(z.A, z.a)
 C.Bb = circle(z.B, z.b)
 C.Cc = circle(z.C, z.c)
 L.rAB = C.Aa:radical_axis(C.Bb)
 L.rAC = C.Aa:radical_axis(C.Cc)
 L.rBC = C.Bb:radical_axis(C.Cc)
 z.w = C.Aa:radical_center(C.Bb, C.Cc)
 z.x1, z.y1 = L.rAB:get()
 z.x2, z.y2 = L.rAC:get()
 z.x3, z.y3 = L.rBC:get()
 z.t = C.Aa:radical_circle(C.Bb, C.Cc).through
 z.w1, z.t1 = C.Aa:radical_circle(C.Bb):get()
 C.B = circle(z.B, z.B)
 z.w2, z.t2 = C.Aa:radical_circle(C.B):get()}
\begin{center}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawCircles[blue](A,a B,b C,c)
   \tkzDrawCircles[red](w,t)
   \tkzDrawCircles[orange](w1,t1)
   \tkzDrawCircles[purple](w2,t2)
   \tkzDrawLines[red](x1,y1 x2,y2 x3,y3)
   \tkzDrawPoints(A,B,C,w,w1,w2)
   \tkzLabelPoints[below right](A,B,C,w,w1,w2)
\end{tikzpicture}
\end{center}
```

13.6.10. Method pole(L)

Syntax:

```
P = circle:pole(L)
```

Purpose: Compute the *pole* of a line L with respect to the current circle. The pole is the point whose polar is exactly the line L.

Arguments:

- **L** — a line object.

Returns:

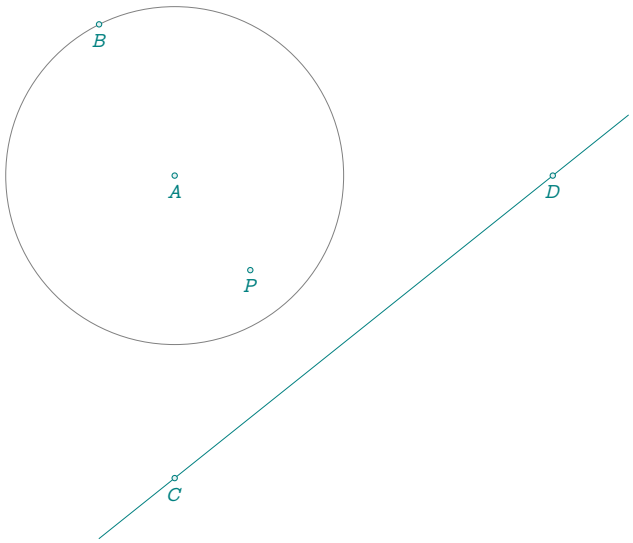
- the point P such that the polar of P w.r.t. the circle is L;
- or **nil,"INFINITE"** if the line passes through the circle's center.

Geometric principle: Let O be the center of the circle and R its radius. Denote by H the orthogonal projection of O onto the line L. If $H \neq O$, the inversion of H in the circle, satisfying $OH \cdot OP = R^2$, is the pole P of line L.

$$P = \mathrm{Inv}_{(O,R)}(H), \qquad \text{with } H = \mathrm{proj}_L(O)$$

If $H = O$ (i.e. the line passes through the center), the pole is an *ideal point* on the perpendicular to L through O.

Example:



13.7. Returns a line

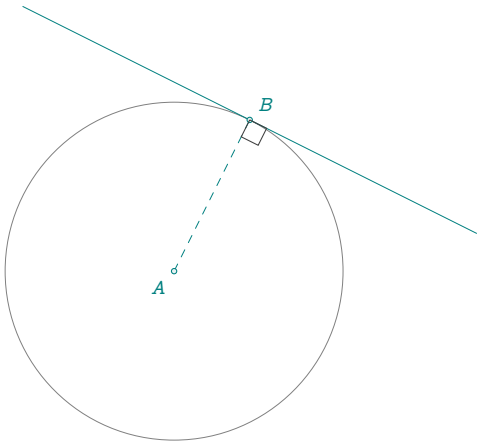
13.7.1. Method tangent_at(pt)

This method constructs the tangent to a circle at a given point on its circumference.

The result is a straight line segment centered at the point of contact. Its two endpoints are placed symmetrically at equal distances from the contact point, which facilitates drawing and labeling.

In the example below, the points **Tx** and **Ty** are the two endpoints of the tangent segment at the point of contact.

This method is useful for highlighting tangency in diagrams, constructing tangent directions, or defining orthogonal projections.




```
\directlua{
 init_elements()
 z.A = point(0,0)
 z.B = point(1,2)
 C.AB = circle(z.A, z.B)
 L.T = C.AB:tangent_at(z.B)
 z.Tx, z.Ty = L.T:get()}
```

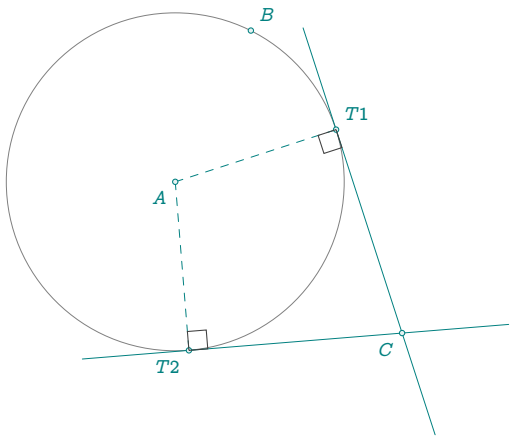
13.7.2. Method tangent_from(pt)

This method computes the two tangents from a point external to a given circle.

Given a point P lying outside the circle, there exist exactly two lines passing through P and tangent to the circle. This method returns these two tangent lines, each defined by P and the point of contact with the circle.

The points of contact are also accessible as the endpoints of these lines. These are the unique points where the tangents touch the circle and are orthogonal to the radius.

This construction is useful in many classical geometric configurations (radical axis, triangle incircle tangents, etc.).



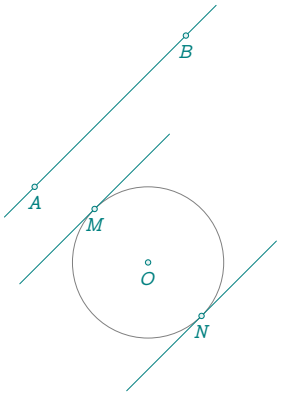

```
\directlua{
 init_elements()
 z.A = point(0,0)
 z.B = point(1,2)
 C.AB = circle(z.A, z.B)
 z.C = point(3,-2)
 L.T1,L.T2 = C.AB:tangent_from(z.C)
 z.T1 = L.T1.pb
 z.T2 = L.T2.pb}
```

13.7.3. Method tangent_parallel(line)

This method constructs the two tangents to a circle that are *parallel* to a given direction (provided as a line).

Each tangent is returned as a straight line segment centered at its point of contact with the circle. For convenience of drawing and labeling, the two endpoints of every tangent segment are placed symmetrically at equal distances from the contact point.

```
L1, L2 = C:tangent_parallel(Ldir)
```




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 4)
 L.AB = line(z.A, z.B)
 z.O = point(3, -2)
 z.T = point(3, 0)
 C.OT = circle(z.O, z.T)
 L.T1, L.T2 = C.OT:tangent_parallel(L.AB)
 z.x, z.y = L.T1:get()
 z.u, z.v = L.T2:get()
 z.M = L.T1.mid
 z.N = L.T2.mid}
\begin{center}
\begin{tikzpicture}[scale=.5]
\tkzGetNodes
\tkzDrawCircle(O,T)
\tkzDrawLines(A,B x,y u,v)
\tkzDrawPoints(O,A,B,M,N)
\tkzLabelPoints(O,A,B,M,N)
\end{tikzpicture}
\end{center}
```

13.7.4. Method commun_tangent(C)

This method constructs a **common tangent** to two circles. It used to return only external tangents; the new version supports three options for `mode`: `"external"` (default), `"internal"`, and `"both"`. The number of solutions depends on the relative position of the circles and ranges from 0 to 4.

Typical cases:

- **Disjoint (separate) circles:** 4 tangents (2 external, 2 internal).
- **Externally tangent circles (touch at one point):** 3 tangents (2 external, 1 internal).
- **Intersecting circles (two crossings):** 2 tangents (both external).
- **Internally tangent circles (one inside, touching):** 1 tangent (external).

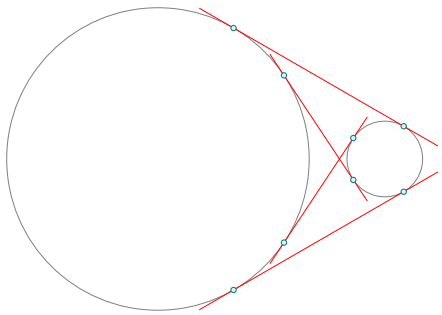
– **One circle strictly contained in the other (no contact):** 0 tangents.

Notes and conventions:

– `mode="external"` returns the external common tangents; `mode="internal"` returns the internal ones.

– Degenerate cases are handled explicitly: coincident circles yield infinitely many common tangents (undefined), concentric circles with distinct radii yield none.

– When radii are equal and circles are disjoint, there are still 4 tangents; the external homothety center is at infinity, which may influence the construction method but not the result.



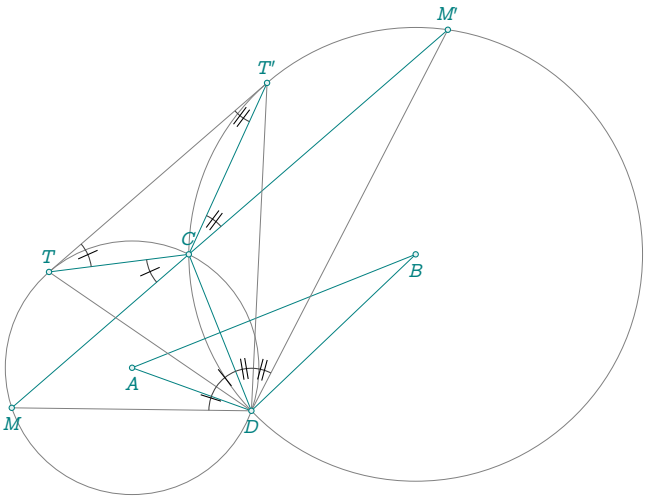
```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.a = point(4, 0)
 z.B = point(6, 0)
 z.b = point(5, 0)
 C.Aa = circle(z.A, z.a)
 C.Bb = circle(z.B, z.b)
 L.Tx, L.Ty = C.Aa:
    common_tangent(C.Bb,"external")
 z.x, z.y = L.Tx:get()
 z.xp, z.yp = L.Ty:get()
 L.Tu, L.Tv = C.Aa:
    common_tangent(C.Bb,"internal")
 z.u, z.v = L.Tu:get()
 z.up, z.vp = L.Tv:get()}
\begin{center}
\begin{tikzpicture}[scale =.5]
  \tkzGetNodes
  \tkzDrawCircles(A,a B,b)
  \tkzDrawLines[red](x,y x',y' u,v u',v')
  \tkzDrawPoints(x,y,x',y',u,v,u',v')
\end{tikzpicture}
\end{center}
```

Application:

Let T and T' be the points of tangency of a common external tangent to two circles, chosen such that T lies on the first circle and T' on the second, both on the same side. Consider a secant parallel to this tangent passing through a fixed point C (typically the center of one of the circles).

In this configuration, the segment $[TT']$ is seen from the other intersection point D (of the secant with the second circle) under an angle equal to *half the angle between the two given circles*.

This elegant geometric relationship is used in the construction of tangents and angle bisectors associated with pairs of circles.




```
\directlua{
init_elements()
z.A = point(0 , 0 )
z.B = point(5 , 2 )
L.AB = line(z.A, z.B)
z.C = point(1, 2)
C.AC = circle(z.A, z.C)
C.BC = circle(z.B, z.C)
L.TTp = C.AC:common_tangent(C.BC)
z.T, z.Tp = L.TTp:get()
z.M = C.AC:point(0.45)
L.MC =line(z.M, z.C)
z.Mp = intersection(L.MC, C.BC)
L.mm = L.TTp:ll_from(z.C)
_, z.M = intersection(L.mm, C.AC)
z.Mp = intersection(L.mm, C.BC)
_, z.D = intersection(C.AC,C.BC)}
```

13.7.5. Method polar(pt)

In projective geometry, the notions of *pole* and *polar* see a duality between points and lines with respect to a conic section.

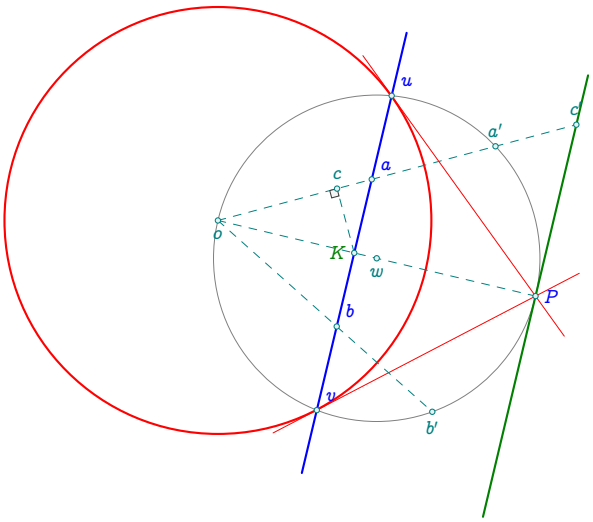
In the special case of a circle, this duality becomes particularly elegant. The method **polar(pt)** computes the **polar line** of a given point with respect to the circle.

This construction is based on **inversion** in the circle:

- The **polar line** of a point Q (not at the center) is defined as the line orthogonal to the radius through Q, passing through the *inverse* of Q with respect to the circle.
- Conversely, the **pole** of a line L (not passing through the center) is the inverse (in the circle) of the foot of the perpendicular from the center to the line.

This method currently only supports the case where a *point* is provided as argument. The result is a line — the polar of that point.

(See also: Wikipedia — Pole and polar)



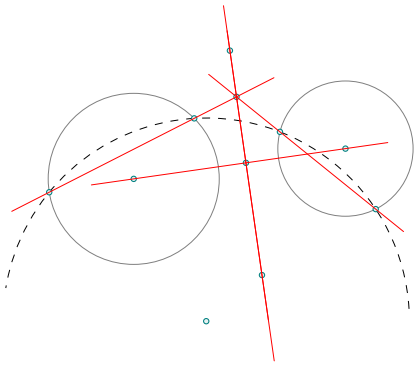

```
\directlua{
 init_elements()
 z.o = point(-1,1)
 z.t = point(1,3)
 z.P = point(3.2,0)
 C.o = circle(z.o, z.t)
 L.P = C.o:polar(z.P)
 z.a, z.b = L.P:get()
 z.u, z.v = intersection(C.o,L.P)
 z.K = L.P:projection(z.P)
 L.K = C.o:polar(z.K)
 z.ka, z.kb = L.K:get()
 C.wH = C.o:inversion(L.P)
 z.w, z.H = C.wH:get()
 z.ap, z.bp = C.o:inversion(z.a, z.b)
 L.oa = line(z.o, z.a)
 z.cp = intersection(L.K,L.oa)
 z.c = C.o:inversion(z.cp)}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircles[red,thick](o,t)
  \tkzDrawCircles(w,H)
  \tkzDrawLines[red](P,u P,v)
  \tkzDrawLines[blue,thick](u,v)
  \tkzDrawLines[add = 1 and 1,
     green!50!black,thick](ka,kb)
  \tkzDrawSegments[dashed](o,P o,c' o,b' K,c)
  \tkzMarkRightAngle[size=.1,
     fill=lightgray!15](o,c,K)
  \tkzDrawPoints(o,w,K,P,a,b,u,v,a',b',c',c)
  \tkzLabelPoints(o,w,b')
  \tkzLabelPoints[above right,blue](a,b,u,v)
  \tkzLabelPoints[above](c,a',c')
  \tkzLabelPoints[right,blue](P)
  \tkzLabelPoints[green!50!black,left](K)
 \end{tikzpicture}
\end{center}
```

13.7.6. Method radical_axis(C)

The radical line, also called the radical axis, is the locus of points of equal circle power with respect to two nonconcentric circles. By the chordal theorem, it is perpendicular to the line of centers (Dörrie 1965).

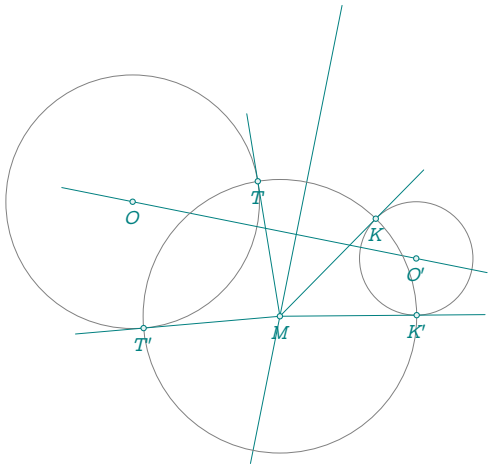
Weisstein, Eric W. "Radical Line." From MathWorld–A Wolfram Web Resource.

Radical axis v1



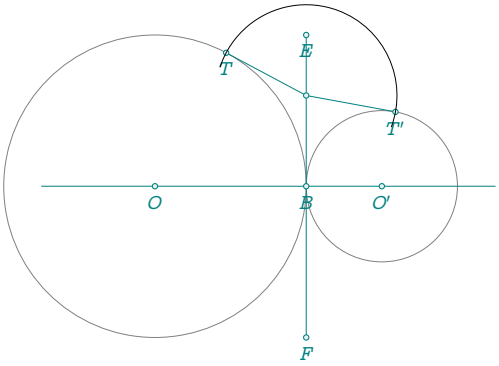
```
\directlua{
 init_elements()
 z.X = point(0,0)
 z.B = point(2,2)
 z.Y = point(7,1)
 z.Ap = point(8,-1)
 L.XY = line(z.X, z.Y)
 C.XB = circle(z.X, z.B)
 C.YAp = circle(z.Y, z.Ap)
 z.E,
 z.F = C.XB:radical_axis(C.YAp):get()
 z.A = C.XB:point(0.4)
 T.ABAp = triangle(z.A, z.B, z.Ap)
 z.O = T.ABAp.circumcenter
 C.OAp = circle(z.O, z.Ap)
 _, z.Bp = intersection(C.OAp,C.YAp)
 L.AB = line(z.A, z.B)
 L.ApBp = line(z.Ap, z.Bp)
 z.M = intersection(L.AB,L.ApBp)
 z.H = L.XY:projection(z.M)}
```

Radical axis v2



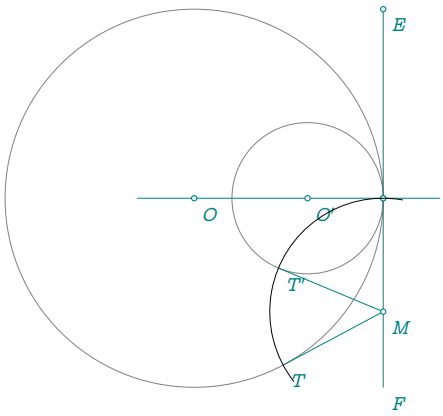

```
\directlua{
  init_elements()
  z.O = point(-1,0)
  z.Op = point(4,-1)
  z.B = point(0,2)
  z.D = point(4,0)
  C.OB = circle(z.O, z.B)
  C.OpD = circle(z.Op, z.D)
  L.EF = C.OB:radical_axis(C.OpD)
  z.E, z.F = L.EF:get()
  z.M = L.EF:point(.75)
  L.MT,L.MTp = C.OB:tangent_from(z.M)
  _, z.T = L.MT:get()
  _, z.Tp = L.MTp:get()
  L.MK,L.MKp = C.OpD:tangent_from(z.M)
  _, z.K = L.MK:get()
  _, z.Kp = L.MKp:get()}
```

Radical axis v3




```
\directlua{
 init_elements()
 z.O = point(0,0)
 z.B = point(4,0)
 z.Op = point(6,0)
 C.OB = circle(z.O, z.B)
 C.OpB = circle(z.Op, z.B)
 L.EF = C.OB:radical_axis(C.OpB)
 z.E, z.F = L.EF:get()
 z.M = L.EF:point(0.2)
 L.tgt = C.OB:tangent_from(z.M)
 _, z.T = L.tgt:get()
 L.tgtp = C.OpB:tangent_from(z.M)
 _, z.Tp =L.tgtp:get()}
```

Radical axis v4




```
\directlua{
 init_elements()
 z.O = point(0,0)
 z.B = point(5,0)
 z.Op = point(3,0)
 C.OB = circle(z.O, z.B)
 C.OpB = circle(z.Op, z.B)
 L.EF = C.OB:radical_axis(C.OpB)
 z.E, z.F = L.EF:get()
 z.H = L.EF.mid
 z.M = L.EF:point(.8)
 _,L.t = C.OB:tangent_from(z.M)
 _, z.T = L.t:get()
 _,L.tp = C.OpB:tangent_from(z.M)
 _, z.Tp = L.tp:get()}
```

13.8. Returns a circle

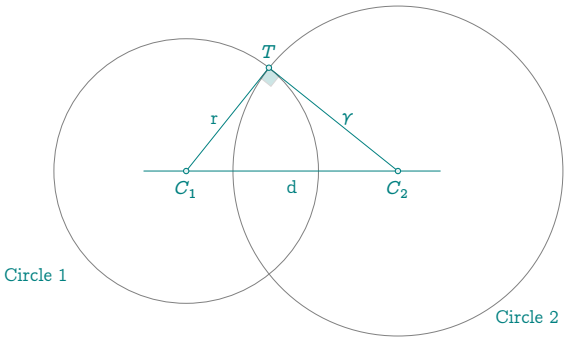
13.8.1. Method orthogonal_from(pt)

In geometry, two circles are said to be *orthogonal* if their tangent lines at each point of intersection meet at a right angle. (See also: Wikipedia — Orthogonal circles).

This method constructs a circle:

- centered at a given point,
- and orthogonal to a given circle.

The result is a circle that intersects the original one at right angles. The construction ensures that the condition of orthogonality is satisfied at all points of intersection.




```
\directlua{
 init_elements()
 z.C_1 = point(0,0)
 z.C_2 = point(8,0)
 z.A = point(5,0)
 C = circle(z.C_1, z.A)
 z.S,
 z.T = C:orthogonal_from(z.C_2):get()}
```

13.8.2. Method orthogonal_through(pt,pt)

This method constructs a circle that is orthogonal to a given circle and passes through two specified points.

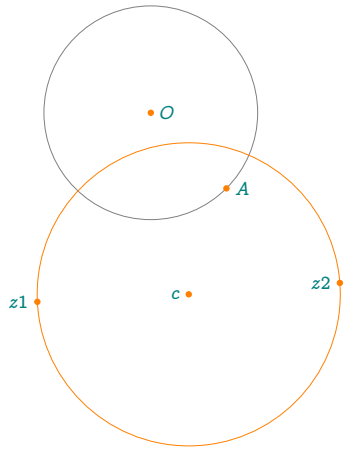
Geometrically, the resulting circle satisfies:

- it intersects the given circle orthogonally (i.e., the tangents at their intersection points are perpendicular),
- it passes through the two given points.

Special cases:

- Inverse points: If the two points are inverse with respect to the given circle, there is an infinite number of solutions, all with their centers lying on the perpendicular bisector of the segment joining the two points. The chosen solution is the one whose center is the midpoint of the two points.
- Collinear with center (non-inverse): If the two points are collinear with the center of the given circle but are not inverse points, no orthogonal circle exists.

– General case: In all other situations, there is a unique solution.




```
\directlua{
  init_elements()
  z.O = point(0,1)
  z.A = point(1,0)
  z.z1 = point(-1.5,-1.5)
  z.z2 = point(2.5,-1.25)
  C.OA = circle(z.O, z.A)
  C.z1 = C.OA:orthogonal_through(z.z1, z.z2)
  z.c = C.z1.center}
```

13.8.3. Method radical_circle(C,C)

Radical circle of three given circles.

This method constructs the *radical circle* associated with three given circles. It is defined as the circle:

– centered at the **radical center** (the common intersection point of the three radical axes),

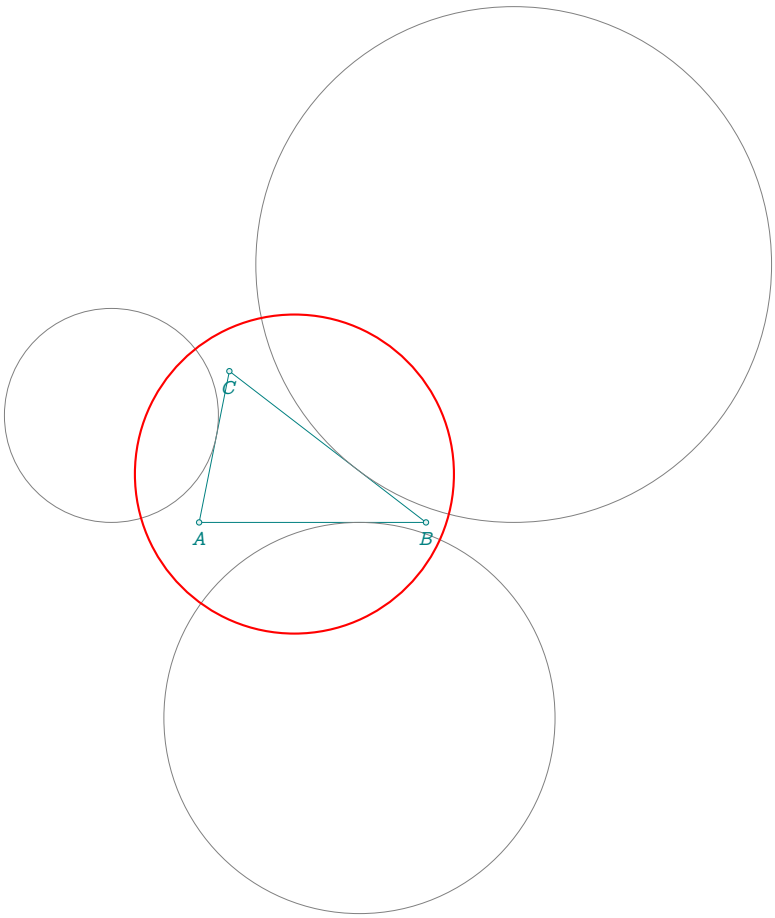
– and orthogonal to all three given circles.

An important geometric property is that a circle centered at the radical center and orthogonal to one of the original circles is necessarily orthogonal to the other two as well.

This construction plays a key role in advanced configurations such as coaxal systems, power of a point geometry, and triangle centers.

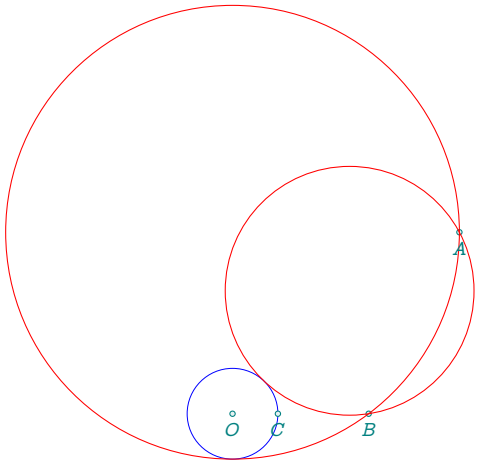
(Reference: Weisstein, Eric W. "Radical Circle." MathWorld)

```
\directlua{
 init_elements()
 z.A = point(0,0)
 z.B = point(6,0)
 z.C = point(0.8,4)
 T.ABC = triangle(z.A, z.B, z.C)
 C.exa = T.ABC:ex_circle()
 z.I_a, z.Xa = C.exa:get()
 C.exb = T.ABC:ex_circle(1)
 z.I_b, z.Xb = C.exb:get()
 C.exc = T.ABC:ex_circle(2)
 z.I_c, z.Xc = C.exc:get()
 C.ortho = C.exa:radical_circle(C.exb,C.exc)
 z.w, z.a = C.ortho:get()}
\begin{tikzpicture}[scale = .5]
   \tkzGetNodes
   \tkzDrawPolygon(A,B,C)
   \tkzDrawCircles(I_a,Xa I_b,Xb I_c,Xc)
   \tkzDrawCircles[red,thick](w,a)
   \tkzDrawPoints(A,B,C)
   \tkzLabelPoints(A,B,C)
\end{tikzpicture}
```



13.8.4. Method CPP

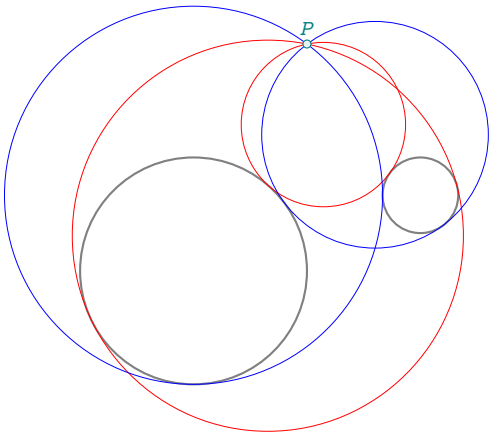
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(5,4)
  z.B = point(3,0)
  z.O = point(0,0)
  z.C = point(1,0)
  C.OC = circle(z.O, z.C)
  PA.center,
  PA.through,
  n = C.OC:CPP(z.A, z.B)
}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircle[blue](O,C)
  \tkzDrawPoints(A,B,C,O)
  \tkzDrawCirclesFromPaths[draw,
      red](PA.center,PA.through)
\end{tikzpicture}
```

13.8.5. Method CCP(C,p[,mode])

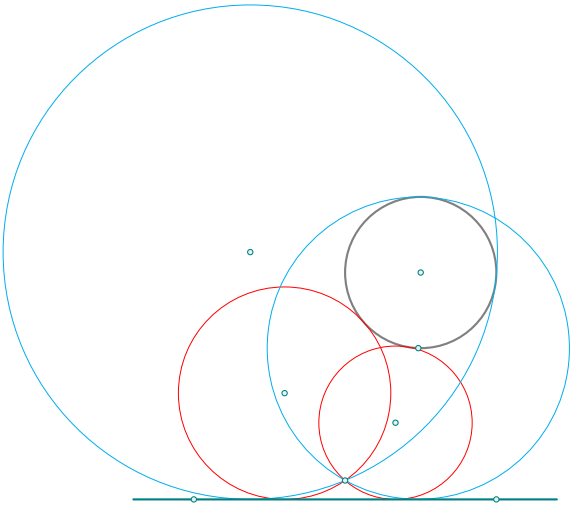
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.TA = point(3, 0)
  z.B = point(6, 2)
  z.TB = point(6, 1)
  z.P = point(3, 6)
  C.A = circle(z.A, z.TA)
  C.B = circle(z.B, z.TB)
  PA.center, PA.through  = C.A:CCP(C.B,
    z.P,"external")
  z.O1 = PA.center:get(1)
  z.O2 = PA.center:get(2)
  z.T1 = PA.through:get(1)
  z.T2 = PA.through:get(2)
  PA.center, PA.through = C.A:CCP(C.B,
    z.P,"internal")
  z.O3 = PA.center:get(1)
  z.O4 = PA.center:get(2)
  z.T3 = PA.through:get(1)
  z.T4 = PA.through:get(2)}
\begin{center}
  \begin{tikzpicture}[scale =.5]
 \tkzGetNodes
 \tkzDrawCircles[thick](A,TA B,TB)
 \tkzDrawCircles[red](O1,T1 O2,T2)
 \tkzDrawCircles[blue](O3,T3 O4,T4)
 \tkzDrawPoints[size=3](P)
 \tkzLabelPoints[above](P)
\end{tikzpicture}
\end{center}
```

13.8.6. Method CLP(L,p)

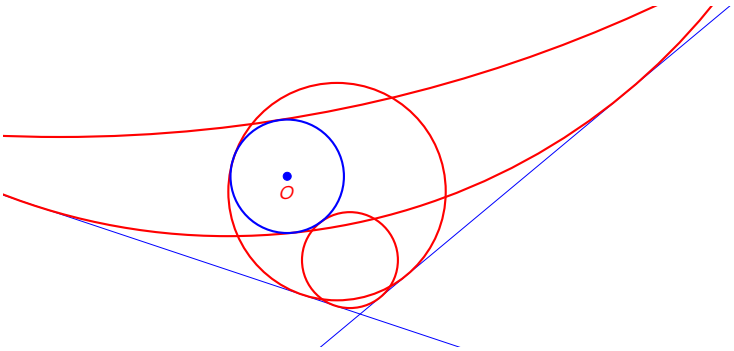
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0)
  L.AB = line (z.A, z.B)
  z.O = point(3, 3)
  z.T = point(3, 2)
  z.P = point(2, .25)
  C.OT = circle(z.O, z.T)
  PA.center, PA.through = C.OT:CLP(L.AB, z.P)
  z.O1 =  PA.center:get(1)
  z.O2 =  PA.center:get(2)
  PA.center,
  PA.through = C.OT:CLP(L.AB, z.P,'internal')
  z.O3 = PA.center:get(1)
  z.O4 = PA.center:get(2)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircles[thick](O,T)
  \tkzDrawCircles[red](O1,P O2,P)
  \tkzDrawCircles[cyan](O3,P O4,P)
  \tkzDrawLines[thick](A,B)
  \tkzDrawPoints[size = 2](P)
  \tkzDrawPoints(A,B,O,O1,O2,O3,O4)
\end{tikzpicture}
```

13.8.7. Method CLL(L,L,<choice,inside>)

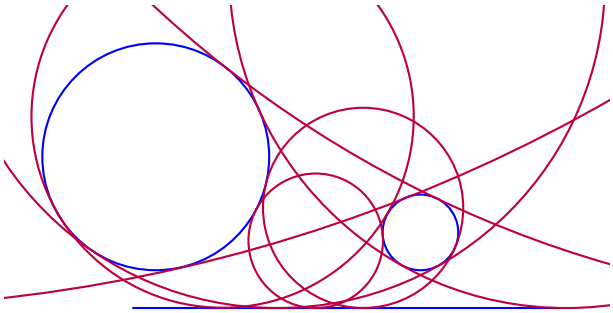
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, -2)
  L.AB  = line(z.A, z.B)
  z.C = point(-3, -4)
  z.D = point(3, 1)
  L.CD  = line(z.C, z.D)
  z.O = z.D + point(-3,1)
  z.X = z.O + point(0,1)
  C.OX = circle(z.O, z.X)
  PA.center, PA.through = C.OX:CLL(L.AB, L.CD,"all")
  tkz.nodes_from_paths(PA.center, PA.through)}
\begin{center}
\begin{tikzpicture}[scale=.75]
 \tkzGetNodes
 \tkzInit[xmin=-5,xmax=10,ymin=-1,ymax=5]
 \tkzClip
 \tkzDrawLines[blue,add =1 and .25](A,B)
 \tkzDrawLines[blue,add =.25 and 1](C,D)
 \tkzDrawPoints[blue,size=3](O)
 \tkzDrawCirclesFromPaths[draw,red,thick](PA.center,PA.through)
 \tkzDrawCircles[thick,blue](O,X)
 \tkzDrawPoints[red,size=3](w1)
 \tkzLabelPoints[red](O)
\end{tikzpicture}
\end{center}
```

13.9. Case: CCL(C2, L)

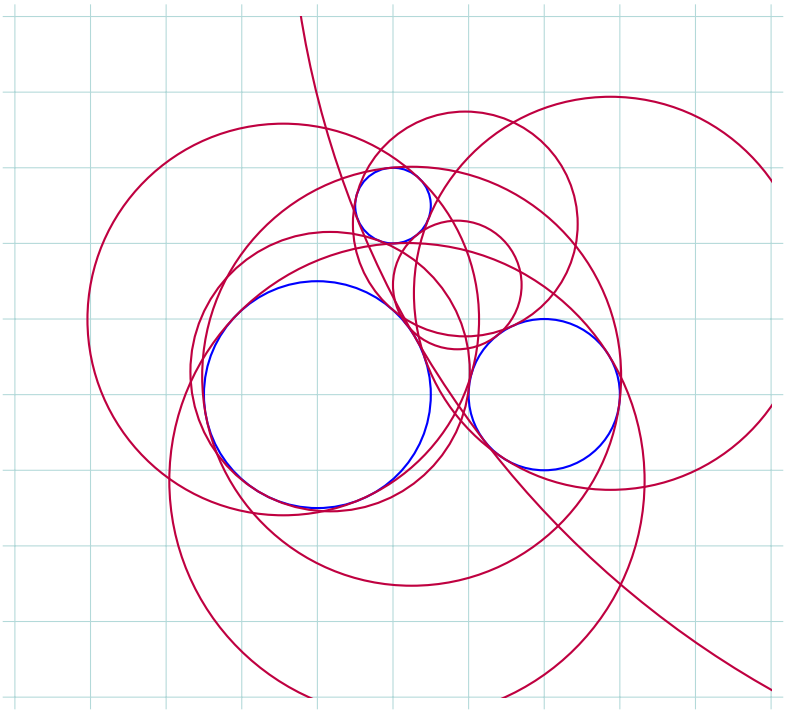
This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.



```
\directlua{
  init_elements()
  z.A = point(-4, 4)
  z.B = z.A + point(3, 0)
  C.AB  = circle(z.A, z.B)
  z.C = point(3,2)
  z.D = z.C + point(1,0)
  C.CD = circle(z.C, z.D)
  z.E = point(-3, 0)
  z.F = point(5, 0)
  L.EF  = line(z.E, z.F)
  PA.center, PA.through,n = C.AB:CCL(C.CD, L.EF)
  tkz.nodes_from_paths(PA.center, PA.through)
  }
\begin{center}
  \begin{tikzpicture}[scale=.5]
\tkzGetNodes
 \tkzInit[xmin=-8,xmax=8,ymin=-8,ymax=8]
 \tkzClip
 \tkzDrawLines[thick,blue](E,F)
 \tkzDrawCircles[thick,blue](A,B C,D)
 \tkzDrawCirclesFromPaths[draw,purple,thick](PA.center, PA.through)
\end{tikzpicture}
  \end{center}
```

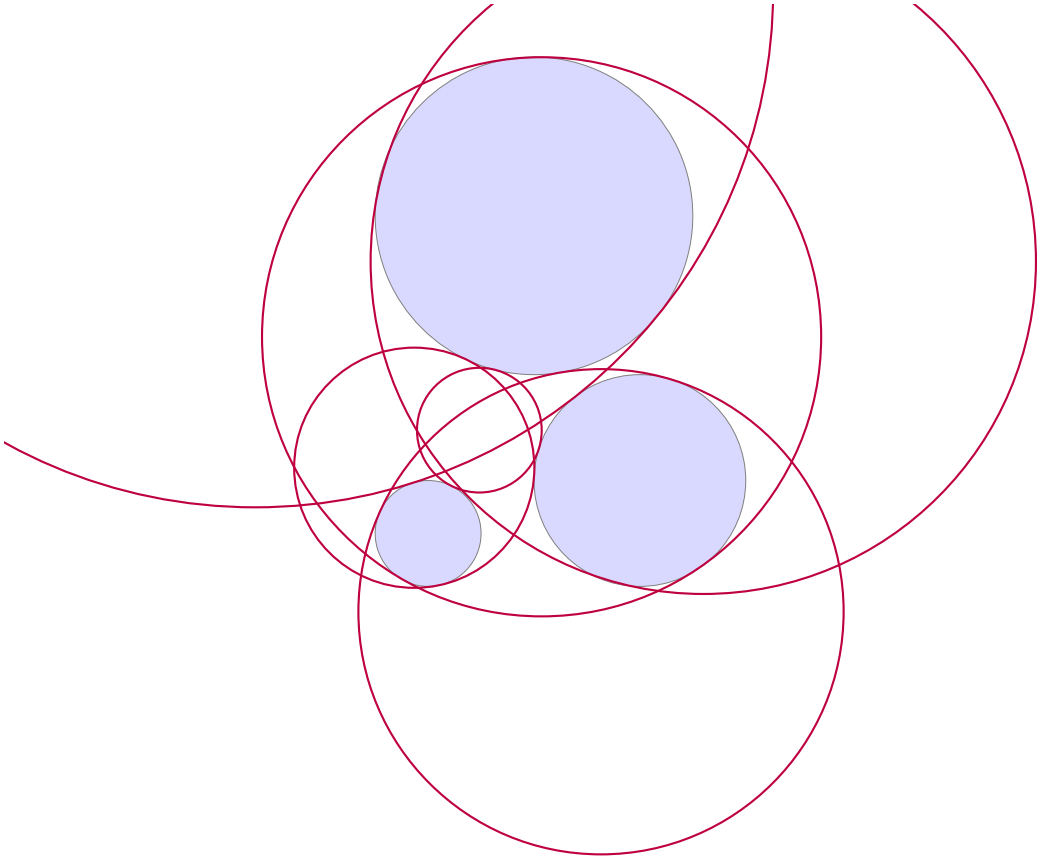
13.9.1. Case: CCC(C2, C3 [, opts])

This method is presented in the document **Euclidean Geometry** presented in altermundus.fr.



```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(6, 0)
z.C = point(2, 5)
z.a = z.A + point(3, 0)
z.b = z.B + point(2, 0)
z.c = z.C + point(1, 0)
C.Aa = circle(z.A, z.a)
C.Bb = circle(z.B, z.b)
C.Cc = circle(z.C, z.c)
PA.center, PA.through, n = C.Aa:CCC(C.Bb, C.Cc)
}
\begin{center}
\begin{tikzpicture}[ scale = .5,gridded]
  \tkzInit[xmin=-8,xmax=12,ymin=-8,ymax=10]
  \tkzClip
  \tkzGetNodes
  \tkzDrawCircles[thick,blue](A,a B,b C,c)
  \tkzDrawCirclesFromPaths[draw,
    purple,thick](PA.center, PA.through)
\end{tikzpicture}
\end{center}
```

13.9.2. Case: CCC_gergonne(C_2, C_3)



```
\directlua{
   z.A = point(0, 0)
   z.a = z.A + point (1, 0)
   z.B = point (4, 1)
   z.b = z.B + point (2, 0)
   z.C = point (2, 6)
   z.c = z.C + point (3, 0)
   C.Aa = circle(z.A, z.a)
   C.Bb = circle(z.B, z.b)
   C.Cc = circle(z.C,z.c)
   PA.center, PA.through = C.Aa:CCC_gergonne(C.Bb, C.Cc)
}
\begin{center}
\begin{tikzpicture}[scale = 0.7]
  \tkzInit[xmin=-8,xmax=12,ymin=-8,ymax=10]
  \tkzClip
  \tkzGetNodes
  \tkzDrawCircles[fill=blue!15](A,a B,b C,c)
  \tkzDrawCirclesFromPaths[draw,purple,thick](PA.center, PA.through)
\end{tikzpicture}
\end{center}
```

13.9.3. Method midcircle

According to Eric Danneels and Floor van Lamoen:

A **midcircle** of two given circles is a circle that maps one to the other via an **inversion**. Midcircles belong to the same pencil of circles as the two originals. The center(s) of the midcircle(s) correspond to the *centers of similitude* of the given circles.

I have adopted the term **midcircle** used by Eric Danneels and Floor van Lamoen, as well as by Eric W. Weisstein, but many other names exist, such as **mid-circle** or **circle of antisimilitude** [Wikipedia]. The latter name can be found in Advanced Euclidean Geometry by Roger A. Johnson (2007). On the website *cut-the-knot.org*, reference is made to the **bisectal circle**, translated from French as **cercle bissecteur** and used by Hadamard, but this refers to the case where the two given circles intersect.
Four cases can be distinguished:

(i) **The two circles intersect**: There are two midcircles, centered at the internal and external centers of similitude;

(ii) **One circle lies entirely inside the other**: There is a unique midcircle centered at the *internal* center of similitude;

(iii) **One circle lies entirely outside the other**: There is a unique midcircle centered at the *external* center of similitude;

(iv) **Tangency or single-point intersection**: These are limiting cases of the configurations above.

Let's examine at each of these cases in more detail:

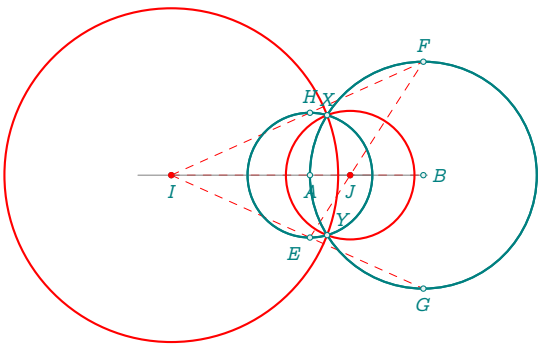
(i) If the two given circles intersect, then there are two circles of inversion through their common points, with centers at the centers of similitudes. The two midcircles bisect their angles and are orthogonal to each other. The centers of the midcircles are the internal center of similitude and the external center of similitude I and J.

- Suppose $(\mathcal{A})$ and $(\mathcal{B})$ are two intersecting circles.
- Their common points define two inversion circles (midcircles), orthogonal to each other.
- The centers of these midcircles are the internal and external centers of similitude, denoted I and J.

To construct these centers:

- Take two diameters EH of $(\mathcal{A})$ and FG of $(\mathcal{B})$, such that the segments EH and FG are parallel.
- The intersection of lines (GE) and (AB) gives point J, the *external* center of similitude.
- The intersection of lines (EF) and (AB) gives point I, the *internal* center of similitude.

The circles $(\mathcal{I})$ and $(\mathcal{J})$, centered respectively at I and J, are orthogonal and form the midcircles of $(\mathcal{A})$ and $(\mathcal{B})$. The division $(A,B;I,J)$ is harmonic.



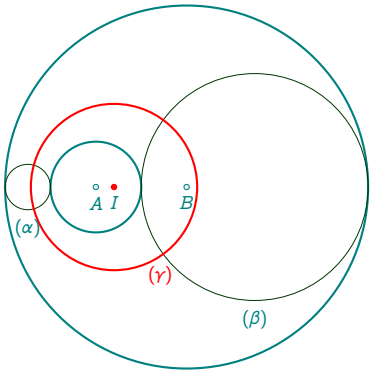

```
\directlua{
 init_elements()
 z.A = point(1, 0)
 z.B = point(3, 0)
 z.O = point(2.1, 0)
 z.P = point(1,0)
 C.AO = circle(z.A, z.O)
 C.BP = circle(z.B, z.P)
 z.E = C.AO.south
 z.H = C.AO.north
 z.F = C.BP.north
 z.G = C.BP.south
 C.IT,C.JV = C.AO:midcircle(C.BP)
 z.I, z.T = C.IT:get()
 z.J, z.V = C.JV:get()
 z.X, z.Y = intersection(C.AO,C.BP)}
```

(ii) **One given circle lies entirely within the other.**

This configuration is slightly more delicate to handle. To construct the midcircle, we proceed as follows:

- First, construct two auxiliary circles (α) and (β), each tangent to both of the given circles — one internally, one externally.
- Then, construct the *radical circle* orthogonal to both (α) and (β).
- The center of this radical circle is the **radical center**, which coincides with the *internal center of similitude* of the original circles.

This radical circle is the unique midcircle in this case: it swaps the two original circles via inversion and lies in the same pencil of circles.




```
\directlua{
   init_elements()
   z.A = point(3, 0)
   z.B = point(5, 0)
   z.O = point(2, 0)
   z.P = point(1, 0)
   L.AB = line(z.A, z.B)
   C.AO = circle(z.A, z.O)
   C.BP = circle(z.B, z.P)
   z.R, z.S = intersection(L.AB,C.BP)
   z.U, z.V = intersection(L.AB,C.AO)
   C.SV = circle:diameter(z.S, z.V)
   C.UR = circle:diameter(z.U, z.R)
   z.x = C.SV.center
   z.y = C.UR.center
   C.IT = C.AO:midcircle(C.BP)
   z.I, z.T = C.IT:get()}
```

(iii) **The two given circles are external to each other.**

In this configuration, there exists a unique midcircle whose center is the **external center of similitude** of the two given circles.

Let I denote this external center of similitude. To construct the corresponding inversion circle (the midcircle), we proceed as follows:

- Construct the external center I based on the line joining the centers of the two given circles and their respective radii.
- Let E and F be the points of tangency (or auxiliary points) such that $IE \cdot IF = IH^2$.
- The point H lies on the desired midcircle, and the circle centered at I and passing through H is the unique **external midcircle**.

This circle performs an inversion that maps one circle onto the other and lies in the same pencil of circles.

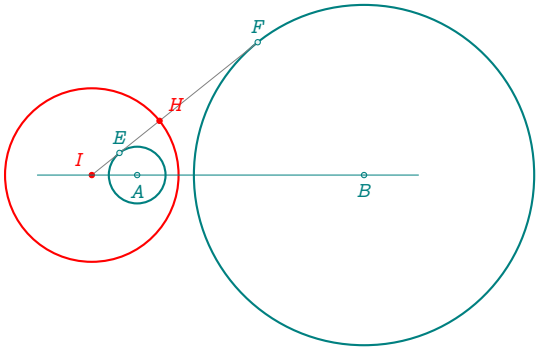
(iv) **The two given circles are external to each other.**

In this configuration, there exists a unique midcircle whose center is the **external center of similitude** of the two given circles.

Let I denote this external center of similitude. To construct the corresponding inversion circle (the midcircle), we proceed as follows:

- Construct the external center I based on the line joining the centers of the two given circles and their respective radii.
- Let E and F be the points of tangency (or auxiliary points) such that $IE \cdot IF = IH^2$.
- The point H lies on the desired midcircle, and the circle centered at I and passing through H is the unique **external midcircle**.

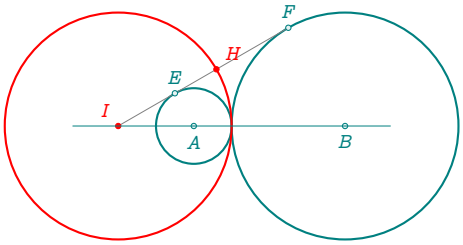
This circle performs an inversion that maps one circle onto the other and lies in the same pencil of circles.



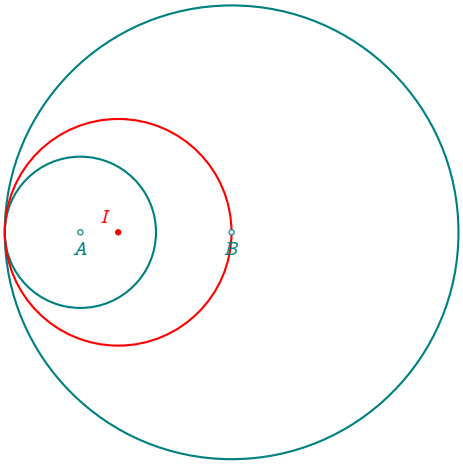

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.a = point(.5, 0)
 z.b = point(1, 0)
 C.Aa = circle (z.A, z.a)
 C.Bb = circle (z.B, z.b)
 L.AB = line(z.A, z.B)
 z.E = C.Aa.north
 z.F = C.Bb.north
 L.EF = line(z.E, z.F)
 C.IT =  C.Aa:midcircle(C.Bb)
 z.I, z.T = C.IT:get()
 L.TF = C.Bb:tangent_from(z.I)
 z.H = intersection(L.TF,C.IT)
 z.E = intersection(L.TF,C.Aa)
 z.F=L.TF.pb}
```

(v) **The two circles are tangent.**

- If circle ($\mathcal{B}$) is *externally tangent* to circle ($\mathcal{A}$), the construction of the midcircle is identical to the case of two disjoint circles. The external center of similitude still exists, and the inversion circle centered at this point transforms one circle into the other.
- If one of the circles lies *inside* the other and they are *internally tangent*, the construction simplifies. The center of the midcircle is the internal center of similitude, which coincides with the point of tangency. The inversion circle is then centered at this point and passes through any auxiliary point satisfying the inversion relation.




```
\directlua{
  init_elements()
  local a,b,c,d
  z.A = point(0, 0)
  z.B = point(4, 0)
  z.a = point(1, 0)
  z.b = point(1, 0)
  C.Aa = circle (z.A, z.a)
  C.Bb = circle (z.B, z.b)
  L.AB = line(z.A, z.B)
  z.E = C.Aa.north
  z.F = C.Bb.north
  L.EF = line(z.E, z.F)
  C.IT =  C.Aa:midcircle(C.Bb)
  z.I, z.T = C.IT:get()
  L.TF = C.Bb:tangent_from(z.I)
  z.H = intersection(L.TF,C.IT)
  z.E = intersection(L.TF,C.Aa)
  z.F=L.TF.pb}
```




```
\directlua{
 init_elements()
 z.A = point(2, 0)
 z.B = point(4, 0)
 z.a = point(1, 0)
 z.b = point(1, 0)
 C.Aa = circle(z.A, z.a)
 C.Bb = circle(z.B, z.b)
 C.IT = C.Aa:midcircle(C.Bb)
 z.I,
 z.T = C.IT:get()}
```

Midcircle between a circle and a line

It is possible to generalize the notion of midcircle to the case of a circle and a line. A midcircle of a circle and a straight line is defined as a circle with respect to which the given circle and line are mutually inverse.
In this situation, there are only three possible cases, but they share several common features. The center of the midcircle lies on the line passing through the center of the given circle and perpendicular to the given line. Moreover, the center is also one of the intersection points of the circle and the line.
Three cases can be distinguished:

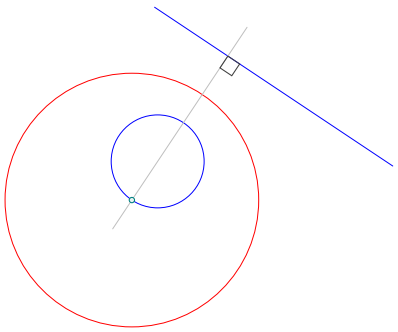
(i) The circle and the line intersect;

(ii) The circle and the line are disjoint;

(iii) The circle and the line are tangent.

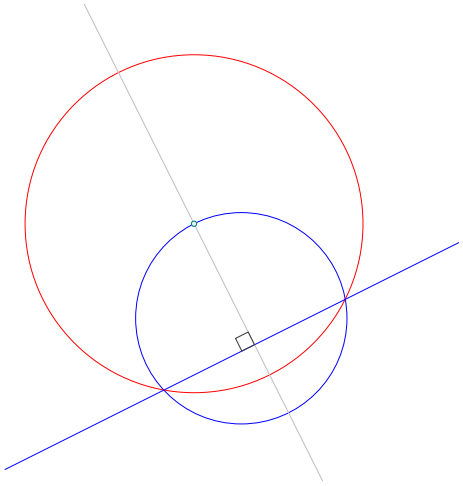
Let's look at each case:
Note: It should be noted that in each case, inversion allows us to verify that the two objects are inverses of each other.

1. The circle and the line are disjoint.



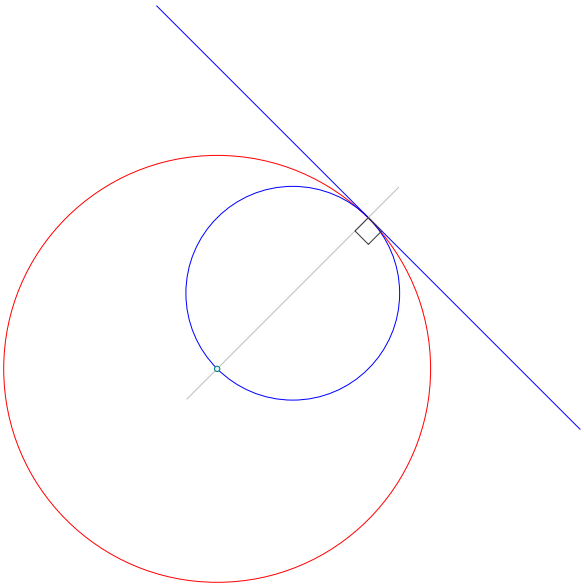
```
\directlua{
init_elements()
z.o = point(-1,1)
z.a = point(1,2)
C.oa = circle(z.o, z.a)
z.c = point(3,2)
z.d = point(0,4)
L.cd = line(z.c, z.d)
C.OH = C.oa:inversion(L.cd)
z.O, z.H = C.OH:get()
L.inv = C.oa:inversion(C.OH)
z.x, z.y = L.inv:get()
C.inv  = C.OH:midcircle(L.cd)
z.w, z.t = C.inv:get()
z.P = L.cd:projection(z.o)
z.T = intersection(C.inv,line(z.o,z.P))}
\begin{center}
\begin{tikzpicture}[scale=.75]
\tkzGetNodes
\tkzDrawCircles[blue](O,H)
\tkzDrawCircles[red](w,t)
\tkzDrawLines[blue](c,d)
\tkzDrawLines[lightgray](o,P)
\tkzDrawPoint(w)
\tkzMarkRightAngle(w,P,c)
\end{tikzpicture}
\end{center}
```

2. The circle and the line intersect



```
\directlua{
init_elements()
z.o = point(0,1)
z.a = point(1,3)
C.oa = circle(z.o, z.a)
z.c = point(-1,2)
z.d = point(1,3)
L.cd = line(z.c, z.d)
C.OH = C.oa:inversion(L.cd)
z.O, z.H = C.OH:get()
L.inv = C.oa:inversion(C.OH)
z.x, z.y = L.inv:get()
C.inv  = C.OH:midcircle(L.cd)
z.w, z.t = C.inv:get()
z.P = L.cd:projection(z.o)
z.T = intersection(C.inv,line(z.o,z.P))}
\begin{center}
\begin{tikzpicture}[scale=.75]
\tkzGetNodes
\tkzDrawCircles[blue](O,H)
\tkzDrawCircles[red](w,t)
\tkzDrawLines[blue,add = 2 and 1](c,d)
\tkzDrawLines[lightgray](o,T)
\tkzDrawPoint(w)
\tkzMarkRightAngle(w,P,c)
\end{tikzpicture}
\end{center}
```

3. The circle and the line are tangent



```
\directlua{
init_elements()
z.o = point(-1,1)
z.a = point(1,3)
C.oa = circle(z.o, z.a)
z.c = point(3,2)
z.d = point(0,4)
L.cd = C.oa:tangent_at(z.a)
C.OH = C.oa:inversion(L.cd)
C.wt = C.OH:midcircle(L.cd)
z.O, z.H = C.OH:get()
z.w, z.t = C.wt:get()
z.c, z.d = L.cd:get()}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircles[red](w,t)
\tkzDrawCircles[blue](O,H)
\tkzDrawLines[blue](c,d)
\tkzDrawLines[lightgray](o,H)
\tkzDrawPoint(w)
\tkzMarkRightAngle(w,a,c)
\end{tikzpicture}
\end{center}
```

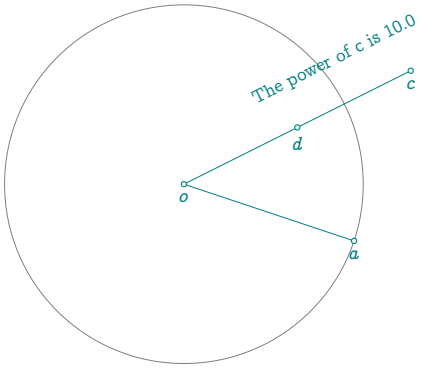
13.10. Transformations: the result is an object

13.10.1. Method inversion(obj):point, line and circle

The **inversion** method can be used on a point, a group of points, a line or a circle. Depending on the type of object, the function determines the correct algorithm to use.

Inversion:point

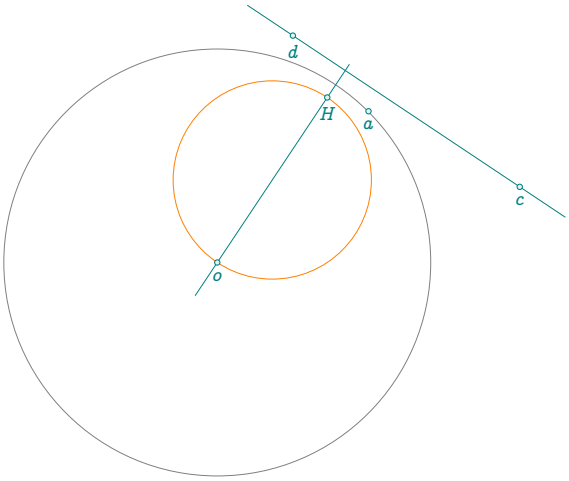
Returns a point.




```
\directlua{
  init_elements()
  z.o = point(-1,2)
  z.a = point(2,1)
  C.oa = circle(z.o, z.a)
  z.c = point(3,4)
  z.d = C.oa:inversion(z.c)
  p = C.oa:power(z.c)}

\begin{center}
\begin{tikzpicture}[scale =.75]
  \tkzGetNodes
  \tkzDrawCircle(o,a)
  \tkzDrawSegments(o,a o,c)
  \tkzDrawPoints(a,o,c,d)
  \tkzLabelPoints(a,o,c,d)
  \tkzLabelSegment[sloped,above=1em](c,d){%
   The power of c is \tkzUseLua{p}}
 \end{tikzpicture}
\end{center}
```

Inversion:line

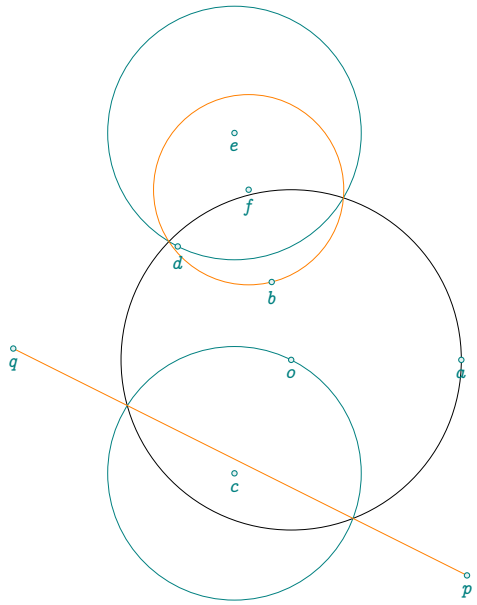
The result is either a straight line or a circle.




```
\directlua{
  init_elements()
  z.o = point(-1,1)
  z.a = point(1,3)
  C.oa = circle(z.o, z.a)
  z.c = point(3,2)
  z.d = point(0,4)
  L.cd = line(z.c, z.d)
  C.OH = C.oa:inversion(L.cd)
  z.O, z.H = C.OH:get()}

\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCircles(o,a)
  \tkzDrawCircles[new](O,H)
  \tkzDrawLines(c,d o,H)
  \tkzDrawPoints(a,o,c,d,H)
  \tkzLabelPoints(a,o,c,d,H)
\end{tikzpicture}
\end{center}
```

Inversion:circle

The result is either a straight line or a circle.




```
\directlua{
  init_elements()
  z.o, z.a = point(-1,3),point(2,3)
  z.c = point(-2,1)
  z.e, z.d = point(-2,7),point(-3,5)
  C.oa = circle(z.o, z.a)
  C.ed = circle(z.e, z.d)
  C.co = circle(z.c, z.o)
  obj = C.oa:inversion(C.co)
  if obj.type == "line" then
     z.p, z.q = obj:get()
  else
      z.f, z.b = obj:get()
  end
  obj = C.oa:inversion(C.ed)
  if obj.type == "line" then
    z.p, z.q = obj:get()
  else
     z.f, z.b = obj:get()
  end
  color = "orange"}
\begin{center}
\begin{tikzpicture}[scale =.75]
  \tkzGetNodes
  \tkzDrawCircles[black](o,a)
  \tkzDrawCircles[teal](c,o e,d)
  \tkzDrawCircles[\tkzUseLua{color}](f,b)
  \tkzDrawSegments[\tkzUseLua{color}](p,q)
  \tkzDrawPoints(a,...,f,o,p,q)
 \tkzLabelPoints(a,...,f,o,p,q)
\end{tikzpicture}
\end{center}
```

13.10.2. Method inversion_neg(obj)

Syntax:
`local Q = C:inversion_neg(obj)`
Purpose:
This method performs the inversion with respect to the circle C but with a *negative power*. For a point $P \neq O$, this means that

$$\overrightarrow{OP'} \cdot \overrightarrow{OP} = -R^2,$$

that is, P' is the inverse of P on the opposite half–line of OP. From a geometric point of view, the image set (line or circle) is the same as with the ordinary inversion, but the points are taken with reversed orientation along that set.
Arguments:

- `obj` – a **point**, a **line** (possibly not passing through O), or a **circle**.

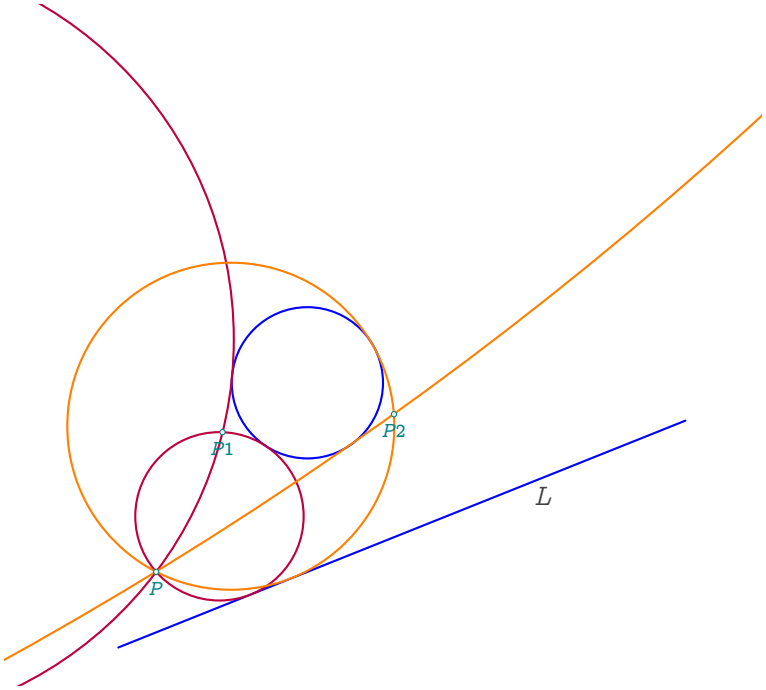
Returns:

- If `obj` is a point $P \neq O$: the point $P' =$ `inversion_neg`(P).
- If `obj` is a line not passing through O: the same circle image as with `C:inversion(line)` (it passes through O), but the orientation of the points on it is reversed.
- If `obj` is a line passing through O: the image is the same line.
- If `obj` is a circle not passing through O: the image is the same circle as with `inversion`, but with reversed orientation.
- If `obj` is a circle passing through O: the image is the same line.

Details:
For a point P,

$$\vec{OP'} = -\frac{R^2}{\|OP\|^2}\vec{OP}.$$

This is equivalent to performing the ordinary inversion followed by a central symmetry with respect to O (taking the antipode).
For lines and circles, `inversion_neg` produces the same geometric image as `inversion`; the negative sign only affects the parametrization along that image.
About the next example:
The aim is to find the circles passing through a point P tangent to a line L and a circle.
In the example given above, the point `P2 = C.inv2:inversion_neg(P)` lies on the required circle because the negative inversion places the image of P on the same geometric locus as the standard inversion, but on the opposite direction relative to O. When combined with `CPP(P,P2)`, this choice selects the correct circle among the possible tangent ones, ensuring that P_2 lies exactly on the desired solution circle.



```
\directlua{
 init_elements()
 z.A = point(0, -5)
 z.B = point(5, -3)
 z.O = point(0, 0)
 z.C = point(2, 0)
 L.AB = line(z.A, z.B)
 C.OC = circle(z.O, z.C)
 L.main = L.AB:orthogonal_from(z.O)
 z.P = point(-4, -5)
 C.inv, C.inv2 = C.OC:midcircle(L.AB)
 z.P1 = C.inv:inversion(z.P)
 local center,
 through = C.OC:CPP(z.P,z.P1)
  z.w1 = center:get(1)
  z.t1 = through:get(1)
  z.w2 = center:get(2)
  z.t2 = through:get(2)
  z.P2 = C.inv2:inversion_neg(z.P)
  PA.c, PA.t = C.OC:CPP(z.P,z.P2)
  tkz.nodes_from_paths(PA.c, PA.t,"w","t",3)
 }
\begin{center}
 \begin{tikzpicture}[scale=.5]
  \tkzInit[xmin=-8,xmax=12,ymin=-8,ymax=10]
  \tkzClip
  \tkzGetNodes
  \tkzDrawLines[thick,blue, add = 1 and 1](A,B)
  \tkzDrawCircle[thick,blue](O,C)
  \tkzDrawCircles[thick,purple](w1,t1 w2,t2)
  \tkzDrawCircles[orange,thick](w3,t3)
  \tkzDrawArc[delta=10,orange,thick](w4,P)(P2)
  \tkzDrawPoints(P,P1,P2)
  \tkzLabelPoints(P,P1,P2)
  \tkzLabelLine[below,pos=1.25](A,B){$L$}
\end{tikzpicture}
 \end{center}
```

13.10.3. Method path(p1, p2, N)

Purpose: The **circle** class includes a method **path** to create a **path** object representing a circular arc between two points on the circle.

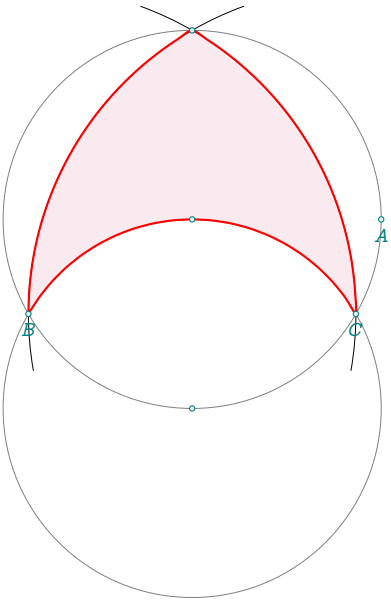
Syntax: This method samples the arc between two points **za** and **zb** lying on the circle, using a specified number of subdivisions.

```
C = circle(z.O, z.A)
PA.arc = C:path(z.B, z.C, 100)
```

The arc begins at **z.B** and ends at **z.C**, and is divided into 100 steps.
You can draw the resulting path in TikZ using:

```
\tkzDrawCoordinates[smooth](PA.arc)
```

Example usage:




```
\directlua{
 z.O = point(0, 0)
 z.A = point(5, 0)
 C.OA = circle(z.O, z.A)
 z.S = C.OA.south
 C.SO = circle(z.S, z.O)
 z.B,z.C = intersection(C.OA, C.SO)
 C.BC =  circle(z.B, z.C)
 L.BC = line(z.B, z.C)
 z.D = intersection(C.OA, C.BC)
 C.CD =  circle(z.C, z.D)
 PA.p1 = C.SO:path(z.C, z.B, 20)
 PA.p2 = C.BC:path(z.C, z.D, 20)
 PA.p3 = C.CD:path(z.D, z.B, 20)
 PA.path = (-PA.p1) + PA.p2 + PA.p3
}
\begin{tikzpicture}[scale = .5]
\tkzGetNodes
\tkzDrawCircles(O,A S,O)
\tkzDrawArc(B,C)(D)
\tkzDrawArc(C,D)(B)
\tkzDrawCoordinates[fill = purple!20,
                    opacity=.4](PA.path)
\tkzDrawCoordinates[smooth,red,
                        thick](PA.path)
\tkzDrawPoints(A,O,B,C,S,D)
\tkzLabelPoints(A,B,C)
\end{tikzpicture}
```

14. Class triangle

The variable `T` holds a table used to store triangle objects. Its use is optional: you are free to choose another variable name, but using `T` is the recommended convention for clarity and consistency across examples and documentation.

If you define your own table, you must initialize it manually. However, if you use the default variable `T`, the **`init_elements()`** function will automatically clear and reset the `T` table when called.

Each triangle object is created using the **`new`** method, which takes three points as input and generates all the associated geometric attributes.

14.1. Creating a triangle

The **`triangle`** class is used to define triangles from three points. It automatically computes a wide range of geometric attributes associated with the triangle.

```
T.ABC = triangle:new(z.A, z.B, z.C)
```

`Short form:`
The short form **`triangle(z.A, z.B, z.C)`** is equivalent and more commonly used:

```
T.ABC = triangle(z.A, z.B, z.C)
```

14.2. Attributes of a triangle

The triangle object is created using the method **`new`**, for example:

`T.ABC = triangle(z.A, z.B, z.C)`

Several attributes are automatically computed and stored:

- `Vertices:` The three vertices are accessible via:

  `T.ABC.pa`, **`T.ABC.pb`**, **`T.ABC.pc`**

- `Sides:` The side lengths are:

$$T.ABC.a = \text{length of } BC$$
$$T.ABC.b = \text{length of } AC$$
$$T.ABC.c = \text{length of } AB$$

- `Area:` The area is computed using Heron's formula:

$$s = \frac{a+b+c}{2}, \quad \text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

  This value is stored in **`T.ABC.area`**.

- `Perimeter:` The perimeter is the sum of side lengths:

$$P = a+b+c$$

  It is stored in **`T.ABC.perimeter`**.

- `Angles:` The internal angles α, β, and γ (at A, B, and C respectively) are computed using the law of cosines:

$$\cos(\alpha) = \frac{b^2+c^2-a^2}{2bc}, \quad \cos(\beta) = \frac{a^2+c^2-b^2}{2ac}, \quad \cos(\gamma) = \frac{a^2+b^2-c^2}{2ab}$$

  They are stored in **`T.ABC.alpha`**, **`T.ABC.beta`**, and **`T.ABC.gamma`**.

These attributes define the essential geometric properties of the triangle and are used in numerous derived methods and constructions.

Table 10: Triangle attributes.

Attributes	Reference
pa	[14.2.1]
pb	idem
pc	idem
type	'triangle'
circumcenter	[14.2.2]
centroid	idem
incenter	idem
orthocenter	idem
eulercenter	idem
spiekercenter	idem; [14.2.2; 3.1.4]
a	[14.2.5]
b	
c	
alpha	[14.2.3]
beta	
gamma	
alpha_	[14.2.4]
beta_	
gamma_	
ab	[14.2.6]
bc	
ca	
semiperimeter	[14.2.5]
area	
orientation	[14.2.7]
cross	[14.2.8]
inradius	[14.2.5]
circumradius	[14.2.5]

14.2.1. Triangle attributes: defining points

Consider a triangle ABC. We want to construct a new triangle whose vertices are the reflections of A, B, and C with respect to the midpoints of their opposite sides.

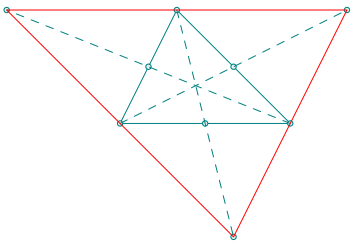
To achieve this, we first need to determine the midpoints of the sides of triangle ABC. The most straightforward approach is to use the method **medial**, which returns the **medial triangle**, i.e., the triangle formed by the midpoints of $[BC]$, $[AC]$, and $[AB]$.

Once this medial triangle is constructed, its vertices can be accessed through standard attributes:

T.medial.A, **T.medial.B**, **T.medial.C**

Alternatively, from the last, the method **get()** retrieves all three vertices at once in a convenient Lua-compatible syntax.

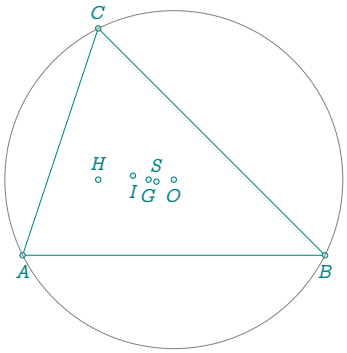
For a practical illustration of this method, see the associated **tikzpicture** example in the documentation.



```
\directlua{ init_elements()
  z.A = point(0, 0)
  z.B=  point(3, 0)
  z.C = point(1, 2)
  T.ABC = triangle(z.A, z.B, z.C)
  T.m = T.ABC:medial()
  z.ma, z.mb, z.mc = T.m.pa, T.m.pb, T.m.pc
  z.Ap = z.ma:symmetry(z.A)
  z.Bp = z.mb:symmetry(z.B)
  z.Cp = z.mc:symmetry(z.C)}
```

14.2.2. Triangle attributes: characteristic points

The **tikzpicture** environment can be viewed in the documentation.



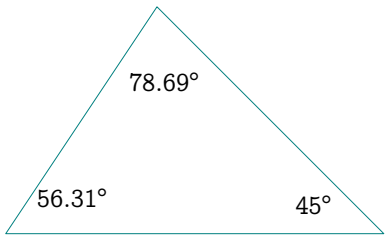

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.C = point(1, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.O = T.ABC.circumcenter
 z.I = T.ABC.incenter
 z.H = T.ABC.orthocenter
 z.G = T.ABC.centroid
 z.S = T.ABC.spiekercenter}
```

14.2.3. Triangle attributes: angles

Numeric measures. The variables **alpha**, **beta**, and **gamma** are *numbers* (measured in radians). They are obtained by evaluating

$$\alpha = |\angle(ZA, ZB, ZC)|, \qquad \beta = |\angle(ZB, ZC, ZA)|, \qquad \gamma = |\angle(ZC, ZA, ZB)|,$$

where **get_angle_** returns an oriented angle (possibly negative, depending on the orientation of the three points). The use of **math.abs** ensures that **alpha**, **beta**, and **gamma** represent the *unsigned* interior angle measures.



The sum of the angles is: 180.0

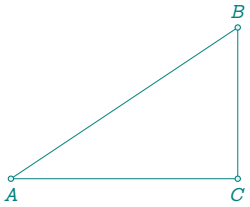
```
\directlua{init_elements()
  z.A = point(0,0)
  z.B = point(5,0)
  z.C = point(2,3)
  T.ABC = triangle(z.A, z.B, z.C)
  S = T.ABC.alpha + T.ABC.beta + T.ABC.gamma}
\def\wangle#1{\tkzPN[2]{%
    \tkzUseLua{math.deg(T.ABC.#1)}}}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygons(A,B,C)
  \tkzLabelAngle(B,A,C){$\wangle{alpha}^\circ$}
  \tkzLabelAngle(C,B,A){$\wangle{beta}^\circ$}
  \tkzLabelAngle(A,C,B){$\wangle{gamma}^\circ$}
  \end{tikzpicture}
\end{center}
The sum of the angles is: \tkzUseLua{math.deg(S)}
```

14.2.4. Triangle attributes: angle objects

`Angle objects:` The variables **alpha_**, **beta_**, and **gamma_** are *objects* of class `angle` created with **angle:new(...)**. They store the defining triple of points and provide methods to retrieve the measure (in radians or degrees) and to support later geometric operations or drawings based on that angle.

`Remark:` The function **get_angle_** returns an *oriented* angle, whose sign depends on the order of the three points. Its value is typically in the interval $(-\pi, \pi]$. Taking the absolute value therefore produces the (non–oriented) interior angle measure.
By contrast, an object of class **angle** may preserve the orientation information, depending on its internal implementation. This distinction allows one to work either with pure numeric measures (for computations) or with full geometric angle objects (for constructions and drawings).
`Example:`

A(value) = 0.58800260354757
A(raw) = -0.58800260354757
A(deg) = 33.69006752598




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.C = point(3, 0)
  z.B = point(3, 2)
  T.ABC = triangle(z.A, z.B, z.C)
  A.A = T.ABC.alpha_
  A.B = T.ABC.beta_
  T.C = T.ABC.gamma_
  tex.print("A(value) = \\", A.A.value)
  tex.print('\\\\')
  tex.print("A(raw) = \\", A.A.raw)
  tex.print('\\\\')
  tex.print("A(deg) = \\", A.A.deg)}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,C)
  \tkzLabelPoints[above](B)
\end{tikzpicture}
\end{center}
```

14.2.5. Triangle attributes: lengths

You can access different lengths, in particular side lengths with:

```
T.ABC = triangle(z.A, z.B, z.C)
p = T.ABC.a + T.ABC.b +T.ABC.c % p perimeter of T
T.ABC.a, T.ABC.b and T.ABC.c are the lengths of the opposite sides to A, B and C.
% Other accessible lengths
s = T.ABC.semiperimeter
ri = T.ABC.inradius
R = T.ABC.circumradius
A = T.ABC.area
```

14.2.6. Triangle attributes: straight lines

Several attributes associated with triangle sides and lines simplify code writing and improve readability.

Consider a triangle ABC. Suppose you want to use the midpoint of the side $[BC]$ (which can also be seen as the base of the median from A). There are multiple ways to construct this midpoint in Lua, but the most concise is:

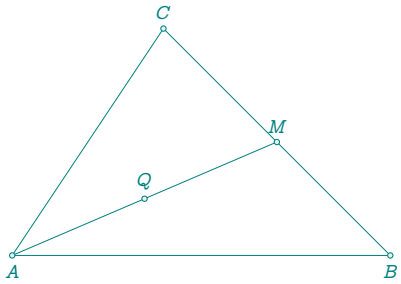
T.bc.mid

Here:

- **T.ABC.bc** is the attribute representing the line (BC),
- It is equivalent to writing **L.BC = line(z.B, z.C)**,
- But with a triangle object already defined as **T.ABC**, you can write simply:

L.BC = T.ABC.bc

This approach saves time and space in scripts involving medians, altitudes, bisectors, or projections based on triangle sides.

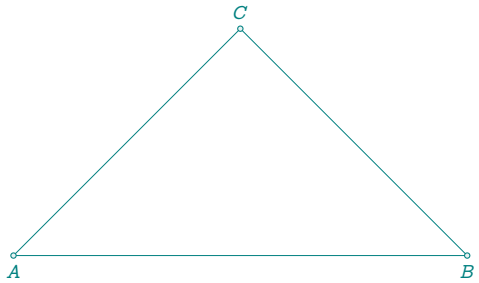



```
\directlua{
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(2, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  z.M = T.ABC.bc.mid
  L.AM = line(z.A, z.M)
  z.Q = L.AM.mid }
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawLine[add = 0 and 1](A,Q)
  \tkzDrawPoints(A,B,C,M,Q)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C,M,Q)
  \end{tikzpicture}
```

14.2.7. Triangle attributes: orientation

The attribute of a triangle indicates the orientation of its vertices. It is positive when the vertices are ordered counter–clockwise, negative when they are ordered clockwise, and zero when the three points are collinear. Geometrically, this value corresponds to the sign of the oriented area of the triangle.

orientation = direct

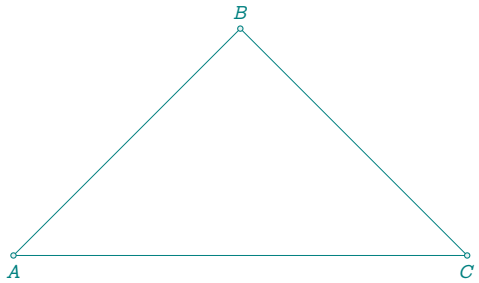



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(2, 0)
  z.C = point(1, 1)
  T.ABC = triangle(z.A, z.B, z.C)
  tex.print("orientation =\\ ", T.ABC.orientation)}
\begin{center}
\begin{tikzpicture}[scale = 3]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C)
\end{tikzpicture}
\end{center}
```

14.2.8. Triangle attributes: cross

In tkz-elements, the operator ^ denotes the two–dimensional cross product of vectors. For three points A, B, and C, the expression
$(B-A)\ \hat{}\ (C-A)$
returns a scalar value. Its sign indicates the relative orientation of the vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$: a positive value corresponds to a counter–clockwise rotation, a negative value to a clockwise rotation, and a zero value indicates collinearity.

cross = -2




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(1, 1)
  z.C = point(2, 0)
  T.ABC = triangle(z.A, z.B, z.C)
  local cross = (z.B - z.A) ^ (z.C - z.A)
  tex.print("cross =\\ ", cross)}
\begin{center}
\begin{tikzpicture}[scale = 3]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,C)
  \tkzLabelPoints[above](B)
\end{tikzpicture}
\end{center}
```

14.3. Methods of the class triangle

Table 11: triangle methods.

Constructor	Reference
new(a, b ,c)	Note[a]; [14.1; 14.3.1]
Methods Returning a Boolean	
in_out(pt)	[14.4.2]
on_triangle(pt)	[14.4.3]
check_equilateral()	[14.4.4]
check_acutangle()	[14.4.5]
Methods Returning a String	
position(pt, EPS)	[14.4.1]
Methods Returning a Real Number	
barycentric_coordinates(pt)	[14.5.1]
trilinear_coordinates(pt)	[14.5.2]
get_angle(arg)	[14.5.3]
trilinear_to_d	14.5.4
Methods Returning a Point	
get(arg)	[14.3.2]
point(r)	[14.6.1]
random(<'inside'>)	[14.6.2]
barycentric(ka,kb,kc)	Note [b]
base(u,v)	[14.6.5]
trilinear(u,v,w)	[14.6.4]
lemoine_point()	[12.9.2]
symmedian_point()	[14.9.9]
lemoine_point()	[14.9.9]
bevan_point()	[14.6.8; 14.8.12]
mittenpunkt_point()	[14.6.12]
gergonne_point()	[14.6.13]
nagel_point()	[14.6.14]
feuerbach_point()	[14.6.15]
spieker_center()	[14.6.17]
projection(p)	[14.6.10]
euler_points()	[14.6.18]
nine_points()	[14.6.19]
taylor_points()	See [14.8.13]
parallelogram()	[24.5.13]
kimberling(n)	See [14.6.6]
isogonal(p)	See [14.6.7]
macbeath_point()(p)	See [14.6.26]
poncelet_point(p)	See [14.6.27]
orthopole(L)	See [14.6.28]
first_fermat_point()	See [14.6.22]
second_fermat_point()	See [14.6.23]
lamoen_points()	See [14.8.15]
soddy_center()	See [14.6.20]
conway_points()	See [14.6.21]
kenmotu_point()	See [14.6.24]
orthic_axis_points()	See [14.9.2; 14.7.7]
isodynamic_points()	[14.6.29]
apollonius_point()	[14.6.31]

a triangle(pt, pt, pt) (short form, recommended)
b The function **barycenter** is used to obtain the barycentre for any number of points

Table 12: triangle methods.

Methods	Reference
Methods Returning a Line	
altitude(arg)	Note [a] [14.7.2]
bisector(arg)	Note [b]; [14.7.3]
bisector_ext(arg)	[14.7.4]
mediator(arg)	[14.7.5]
symmedian_line(arg)	[14.7.1 ; 14.9.9 ; 12.9.2]
euler_line()	Note [c] ; [14.7.8]
antiparallel(pt,n)	[14.7.6;14.6.16]
steiner_line(pt)	[14.7.9]
simson_line(pt)	[14.7.13]
lemoine_axis()	[14.7.10]
brocard_axis()	[14.7.12]
fermat_axis()	
orthic_axis()	See [14.7.7]

a `z.Ha = L.AHa.pb` recovers the common point of the opposite side and altitude. The method `orthic` is usefull. If you don't need to use the triangle object several times, you can obtain a bisector or a altitude with the function `tkz.altitude(z.A, z.B, z.C)` ; [42]

b `_, z.b = L.Bb:get()` recovers the common point of the opposite side and bisector. If you don't need to use the triangle object several times, you can obtain a bisector with the function `tkz.bisector(z.A, z.B, z.C)` [42]

c N center of nine points circle, G centroid, H orthocenter , O circum center

Methods	Comments
Methods Returning a Circle	
euler_circle()	Note [a] [14.8.1]
circum_circle()	[14.8.2]
in_circle()	[14.8.3]
ex_circle(n)	[14.8.4]
first_lemoine_circle()	[14.8.10]
second_lemoine_circle()	14.6.16]
spieker_circle()	[14.8.5]
bevan_circle()	[14.8.12]
cevian_circle()	[14.8.6 ; 14.9.8]
symmedial_circle()	[14.8.7; 14.9.9]
pedal_circle()	[14.8.9]
conway_circle()	[14.8.8]
taylor_circle()	[14.8.13]
kenmotu_circle()	[14.8.18]
c_c(pt)	[14.8.19]
thebault(pt)	[14.8.19]
mixtilinear_incircle(arg)	[14.8.20]
three_tangent_circles	[14.8.21]
adams_circle()	[14.8.14]
lamoen_circle()	See [14.8.15]
soddy_circle()	See [14.8.16]
three_apollonius_circles()	See [14.8.22]
apollonius_circle(side, EPS)	See [14.8.23]
feuerbach_apollonius_k181(side, EPS)	See [14.8.24]
feuerbach_apollonius(side, EPS)	See [14.8.25]
Methods Returning a Triangle	
orthic()	[14.7.2]
medial()	[14.9.1 ; 14.9.9]
incentral()	[14.9.3]
excentral()	[14.9.4; 14.9.7]
extouch()	[14.9.6]
intouch()	[14.9.5; 14.6.13]
contact()	contact = intouch ; [14.6.13]
tangential()	[14.9.14]
anti()	Anticomplementary [14.9.15]
cevian(pt)	[14.9.8]
pedal(pt)	[14.8.9]
symmedial()	[14.9.9]
euler()	[14.9.10; 14.6.18]
lemoine()	[14.9.16]
macbeath()	See [14.9.17]
circumcevian()	See [14.9.13]
Methods Returning a Conic	
kiepert_parabola()	[14.10.1]
kiepert_hyperbola()	[14.10.2]
steiner_inellipse()	[14.10.4]
steiner_circumellipse()	[14.10.4]
euler_ellipse()	[14.10.3]
lemoine_ellipse()	[14.10.5]
brocard_inellipse()	[14.10.6]
macbeath_inellipse()	[14.10.7]
mandart_ellipse()	[14.10.8]
orthic_inellipse()	[14.10.9]
reflection()	[See 14.9.12; 14.7.13; 14.8.17]
Methods Returning a Square	
square_inscribed()	[ex.(14.11.1]

a The midpoint of each side of the triangle, the foot of each altitude, the midpoint of the line segment from each vertex of the triangle to the orthocenter.

14.3.1. Method new(pt, pt, pt)

This method creates a triangle object from three given points, which will serve as its vertices.

It is widely used and appears in most examples. The syntax is:

```
T.ABC = triangle(z.A, z.B, z.C)
```

The resulting object includes many precomputed attributes such as:

- **T.ABC.A**, **T.ABC.B**, **T.ABC.C** — the vertices,
- **T.a**, **T.b**, **T.ABC.c** — the side lengths,
- **T.alpha**, **T.beta**, **T.gamma** — the angles,
- **T.area**, **T.perimeter**,
- as well as key points like the centroid, incenter, orthocenter, circumcenter, etc.

These are available as attributes, not as methods.

```
z.A = point(1, 0)
z.B = point(6, 2)
z.C = point(2, 5)
T.ABC = triangle(z.A, z.B, z.c)
```

14.3.2. Method get(<i>)

This method performs the inverse of triangle creation: it returns the points that define the triangle object.

From the last version, the method also accepts an optional argument **i** to retrieve a specific point:

- **T.ABC:get()** returns all three vertices in order: **pa**, **pb**, **pc**;
- **T.ABC:get(1)** returns only the first point (**pa**),
- **T.ABC:get(2)** returns the second point (**pb**),
- **T.ABC:get(3)** returns the third point (**pc**).

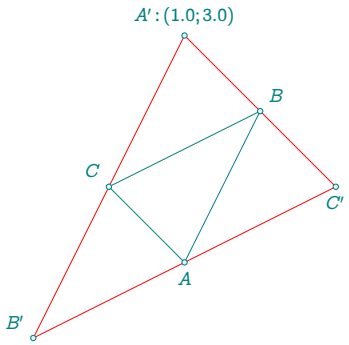
Each of these points is also accessible directly as an attribute:

T.ABC.pa, **T.ABC.pb**, **T.ABC.pc**

For example:

T.ABC:get(2) is equivalent to **T.ABC.pb**

This method belongs to a general mechanism shared by all geometric objects. Its goal is to return the minimal data required to reconstruct the object.




```
\directlua{
  z.A = point(1, 0)
  z.B = point(2, 2)
  z.C = point(0, 1)
  T.ABC = triangle(z.A, z.B, z.C)
  T.med = T.ABC:anticomplementary()
  z.Ap, z.Bp, z.Cp = T.med:get()
  xa,ya = z.Ap:get()}
```

14.4. Returns a boolean value

14.4.1. Method position(pt[, EPS])

This method classifies the position of a point **pt** with respect to the triangle.

Return values:

- **"IN"** — the point lies strictly inside the triangle;
- **"ON"** — the point lies on one of its sides;
- **"OUT"** — the point lies strictly outside.

An optional tolerance parameter **EPS** may be provided. If omitted, the global tolerance **tkz.epsilon** is used.

The classification is based on barycentric coordinates and is numerically robust.

This method provides a unified and explicit classification model, consistent with other geometric objects such as **line:position()** and **circle:position()**.

14.4.2. Method in_out(pt); Deprecated

This method is kept for backward compatibility.
It returns a boolean:

- **true** if the point lies inside the triangle or on its boundary;
- **false** if the point lies strictly outside.

Internally, this method relies on **triangle:position(pt)** and returns:

```
position(pt)  = "OUT"
```

For new developments, the use of **triangle:position(pt)** is recommended.

14.4.3. Method on_triangle(pt); Deprecated

This method is kept for backward compatibility.

Returns:

- **"inside"** if **pt** lies strictly inside the triangle;
- **"on_edge"** if **pt** lies on one of the sides of the triangle;
- **"outside"** otherwise.

14.4.4. Method check_equilateral()

This method checks whether the triangle is equilateral, i.e., whether its three sides are of equal length within a numerical tolerance.

It returns a boolean:

- **true** if the triangle is equilateral,
- **false** otherwise.

This check takes into account floating-point rounding and uses an internal tolerance parameter.

```
if T.ABC:check_equilateral() then
  -- do something
end
```

14.4.5. Method check_acutangle()

Boolean. This method tests whether the triangle is **acutangle**, i.e., if all three of its interior angles are strictly less than 90°.

An *acutangle triangle* is a triangle where:

$$\alpha < 90°, \quad \beta < 90°, \quad \gamma < 90°$$

```
if T.ABC:check_acutangle() then
  -- do something
end
```

14.5. Returns a real number

14.5.1. Method barycentric_coordinates(pt)

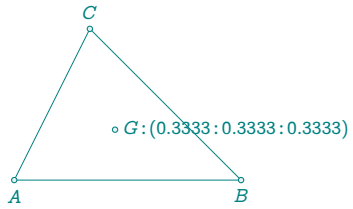
This method returns the **normalized barycentric coordinates** of a point relative to the reference triangle.

```
x, y, z = T: barycentric_coordinates(z.P)
```

The result is a triple **x, y, z** such that:

$$x + y + z = 1$$

If the point lies on a side or at a vertex, one or more coordinates will be zero accordingly. This method is especially useful for checking point location, computing weighted centers, or constructing affine invariants.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(3, 0)
  z.C = point(1, 2)
  T.ABC = triangle(z.A, z.B, z.C)
  z.G = T.ABC.centroid
  xg, yg, zg = T.ABC: barycentric_coordinates(z.G)}
```

14.5.2. Method trilinear_coordinates(pt)

This method returns the **trilinear coordinates** of a point relative to the reference triangle.

Trilinear coordinates represent a point P by its (signed) distances to the sides of the triangle. The triple **x, y, z** corresponds to the directed distances from P to the sides BC, AC, and AB, respectively.

```
x, y, z = T.ABC:trilinear_coordinates(z.P)
```

Unlike barycentric coordinates, trilinear coordinates are homogeneous, i.e., they are defined up to a nonzero scalar multiple:

$$P = x : y : z$$

14.5.3. Method get_angle(arg)

This method returns one of the three internal angles of the triangle.

The argument **arg** identifies the vertex at which the angle is measured. It can be specified in several equivalent ways:

- by a cyclic index **0**, **1**, or **2**,
- by a symbolic identifier (**pa**, **pb**, **pc**),

– or directly by passing one of the triangle's vertices as a point.

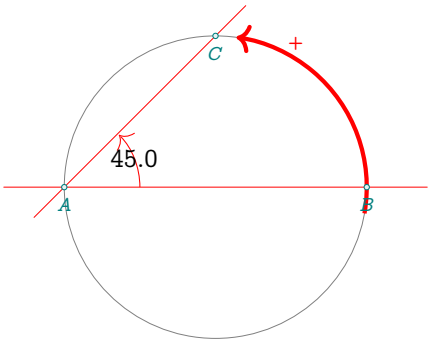
The vertices of the triangle are assumed to be ordered in direct (counter-clockwise) orientation as (A,B,C). Accordingly, the returned angle is determined as follows:

– **arg = 0** or **arg = pa**: returns the angle at vertex A,

– **arg = 1** or **arg = pb**: returns the angle at vertex B,

– **arg = 2** or **arg = pc**: returns the angle at vertex C.

This method should not be confused with the function **get_angle(pta, ptb, ptc)**, which computes the angle formed by three arbitrary points, independently of any triangle object.




```
\directlua{
 z.A = point: new(0, 0)
 z.B = point: new(4, 0)
 z.C = point: new(2, 2)
 T.ABC = triangle(z.A, z.B, z.C)
 tkzAngleA = math.deg(T.ABC:get_angle(z.A))
 z.O = T.ABC.circumcenter
 C.OB = circle(z.O,z.B)
 z.T = C.OB:point(.2)}
\begin{center}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCircle(O,A)
 \tkzDrawArc[red,->,ultra thick](O,B)(T)
 \tkzDrawLines[red](A,B A,C)
 \tkzDrawPoints(A,B,C)
 \tkzLabelPoints(A,B)
 \tkzLabelPoints(C)
 \tkzLabelPoint[right=6pt,red,thick](T){$\mathbf{+}$}
 \tkzMarkAngle[->,red](B,A,C)
 \tkzLabelAngle(B,A,C){\tkzUseLua{tkzAngleA}}
 \end{tikzpicture}
\end{center}
```

14.5.4. Method trilinear_to_d

Purpose: Convert a trilinear triple $(x:y:z)$ into a proportional triple (p,q,r) related to the perpendicular distances from a point to the triangle sides BC, CA, AB. The method uses the current triangle side lengths a,b,c and area Δ.

Interest: Trilinear coordinates are a natural system for locating points defined by geometric relations involving the sides of a triangle (e.g., incenters, excenters, Gergonne or Nagel points, centers of conics, etc.). Many formulas in triangle geometry are expressed in trilinears, but practical geometric constructions often require distances to the sides or to be converted into Cartesian coordinates. The method **trilinear_to_d** provides this essential step:

$$(x:y:z) \quad \longrightarrow \quad (p,q,r) \propto \left(\frac{x}{a}, \frac{y}{b}, \frac{z}{c}\right)$$

This allows the use of trilinear relations directly within the computational framework of tkz-elements.

Syntax: `p, q, r = T:trilinear_to_d(x, y, z)`

Arguments:

– **x**, **y**, **z** — real numbers representing the trilinear coordinates.

Returns:

– **p**, **q**, **r** — real numbers proportional to the distances to the triangle sides.

Definition:

$$p = \Delta\frac{x}{a}, \qquad q = \Delta\frac{y}{b}, \qquad r = \Delta\frac{z}{c},$$

where Δ is the area of the triangle computed with Heron's formula.

Remarks:

- The returned values (p,q,r) are proportional to the distances, not normalized.
- For actual distances (d_a,d_b,d_c) one can use:

$$\lambda = \frac{2\Delta}{ax+by+cz}, \qquad d_a = \lambda x, \quad d_b = \lambda y, \quad d_c = \lambda z.$$

14.6. Returns a point

14.6.1. Method point(r)

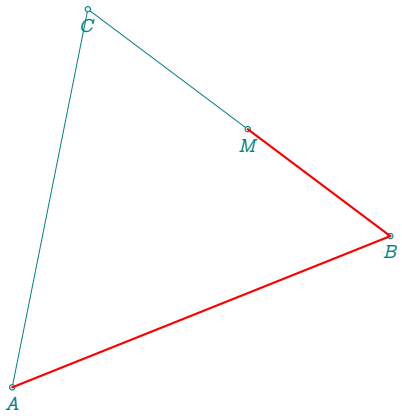
This method is common to most classes. It places a point along the contour of the triangle, proportionally to its perimeter.

The parameter **r** must be a real number between **0** and **1**, representing the fraction of the total perimeter to travel starting from the first vertex (**pa**) along the oriented boundary.

- If **r = 0** or **r = 1**, the method returns the first point (**pa**).
- If **r = 0.5**, the resulting point lies halfway along the triangle's perimeter.

Example: The point M divides the perimeter into two equal arcs:

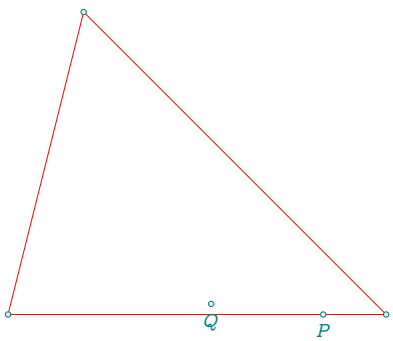
```
z.M = T.ABC:point(0.5)
```




```
\directlua{
  z.A = point(1, 0)
  z.B = point(6, 2)
  z.C = point(2, 5)
  T.ABC = triangle(z.A, z.B, z.C)
  z.M = T.ABC:point(.5)}
\begin{tikzpicture}
  \tkzGetNodes
   \tkzDrawPolygon(A,B,C)
   \tkzDrawPoints(A,B,C,M)
   \tkzLabelPoints(A,B,C,M)
   \tkzDrawSegments[red,thick](A,B B,M)
\end{tikzpicture}
```

14.6.2. Method random(<'inside'>)

This method determines either a random point on one of the sides, or an interior point using the optional **inside** argument.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.Q = T.ABC:random("inside")
  z.P = T.ABC:random() }
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPolygons[red](A,B,C)
 \tkzDrawPoints(A,B,C,P,Q)
 \tkzLabelPoints(P,Q)
\end{tikzpicture}
```

14.6.3. Method barycentric(ka, kb, kc)

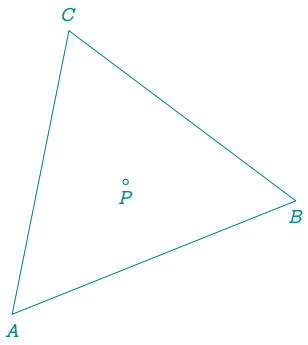
This method, also available under the alias **barycenter**, returns the point with barycentric coordinates $(ka : kb : kc)$ with respect to the triangle.

These coordinates are *homogeneous*, meaning only the ratio between the weights matters. The computed point P satisfies:

$$P = \frac{ka \cdot A + kb \cdot B + kc \cdot C}{ka + kb + kc}$$

where A, B, and C are the triangle's vertices.

This method is widely used internally, notably by the **kimberling()** method to define classical triangle centers such as the centroid, incenter, symmedian point, etc.




```
\directlua{
  z.A = point(1, 0)
  z.B = point(6, 2)
  z.C = point(2, 5)
  T.ABC = triangle(z.A, z.B, z.C)
  z.P = T.ABC:barycentric(1, 1, 1)}
\begin{tikzpicture}[scale=.75]
\tkzGetNodes
\tkzDrawPolygon(A,B,C)
\tkzDrawPoint(P)
\tkzLabelPoints(A,B,P)
\tkzLabelPoints[above](C)
\end{tikzpicture}
```

14.6.4. Method trilinear(x, y, z)

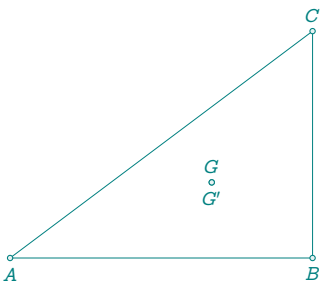
This method determines a point P given its trilinear coordinates relative to a reference triangle ABC.

Trilinear coordinates are defined as an ordered triple (x, y, z) that is proportional to the directed distances from the point P to the sides BC, AC, and AB respectively.

These coordinates are **homogeneous**, meaning they are only defined up to a nonzero scalar multiple. The point P lies at the intersection of the cevians corresponding to these ratios.

Trilinear coordinates were introduced by Plücker in 1835 and remain a fundamental tool in triangle geometry.

Weisstein, Eric W. "Trilinear Coordinates." MathWorld.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.C = point(4, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 a = T.ABC.a
 b = T.ABC.b
 c = T.ABC.c
 z.Gp = T.ABC:trilinear(b * c, a * c, a * b)
 z.G = T.ABC:barycentric(1, 1, 1)}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C,G',G)
  \tkzLabelPoints(A,B,G')
  \tkzLabelPoints[above](C,G)
  \end{tikzpicture}
\end{center}
```

14.6.5. Method base

This method computes a point defined as a linear combination of two sides of the triangle, based at the first vertex.

Given a triangle ABC, the method defines a point D by:

$$\overrightarrow{AD} = \lambda \cdot \overrightarrow{AB} + \mu \cdot \overrightarrow{AC}$$

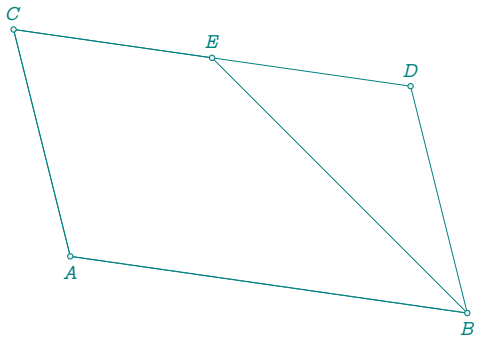
where λ and μ are real coefficients passed as arguments.

Example: If both coefficients are **1**, then:

$$\overrightarrow{AD} = \overrightarrow{AB} + \overrightarrow{AC}$$

This yields a point located "beyond" A in the direction formed by adding the two side vectors.

```
z.D = T.ABC:base(1,1)
```




```
\directlua{
 init_elements()
 z.A = point(1, 1)
 z.B = point(8, 0)
 z.C = point(0, 5)
 z.X = point(2, 2)
 T.ABC = triangle(z.A, z.B, z.C)
 z.D = T.ABC:base(1, 1)
 z.E = T.ABC:base(.5, 1)}
\begin{center}
\begin{tikzpicture}[scale=.75]
     \tkzGetNodes
     \tkzDrawPolygons(A,B,D,C A,B,E,C)
     \tkzDrawPoints(A,B,C,D,E)
     \tkzLabelPoints(A,B)
     \tkzLabelPoints[above](C,D,E)
\end{tikzpicture}
\end{center}
```

14.6.6. Method kimberling(n)

This method returns the triangle center corresponding to the Kimberling number **n**.

The enumeration of triangle centers was established by American mathematician Clark Kimberling in his *Encyclopedia of Triangle Centers*, available online from the University of Evansville:

faculty.evansville.edu/ck6/encyclopedia/ETC.html

Each remarkable triangle center is assigned a unique index $X_{(n)}$. For example, the centroid is $X_{(2)}$, and the orthocenter is $X_{(4)}$.

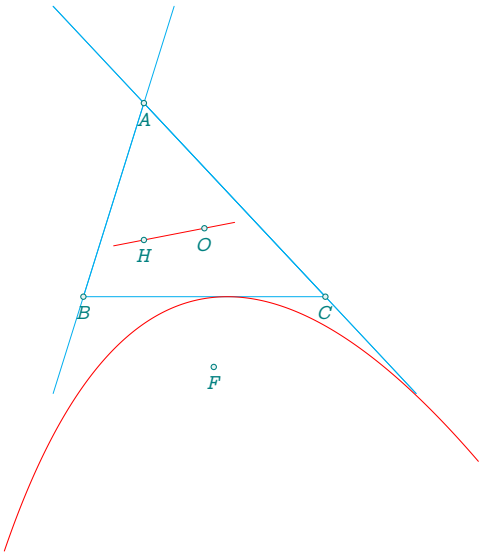
Only a selection of centers is currently implemented in this method. The accessible Kimberling numbers are:

- $X_{(1)}$: incenter
- $X_{(2)}$: centroid
- $X_{(3)}$: circumcenter
- $X_{(4)}$: orthocenter
- $X_{(5)}$: nine-point center
- $X_{(6)}$: symmedian (Lemoine) point
- $X_{(7)}$: Gergonne point
- $X_{(8)}$: Nagel point
- $X_{(9)}$: Mittenpunkt
- $X_{(10)}$: Spieker center
- $X_{(11)}$: Feuerbach point
- $X_{(13)}$: First Fermat point
- $X_{(14)}$: Second Fermat point
- $X_{(19)}$: Clawson point

- $X_{(20)}$: de Longchamps point
- $X_{(55)}$, $X_{(56)}$, $X_{(110)}$, $X_{(111)}$, $X_{(115)}$
- $X_{(175)}$, $X_{(176)}$, $X_{(213)}$, $X_{(371)}$

Example:

```
z.G = T.ABC:kimberling(2)   -> the centroid X(2)
```




```
\directlua{
 init_elements()
 z.B = point(0, 0)
 z.C = point(4, 0)
 z.A = point(1, 3.2)
 T.ABC = triangle(z.A, z.B, z.C)
 z.H = T.ABC:kimberling(4)
 z.O = T.ABC:kimberling(3)
 L.euler = line(z.O, z.H)
 z.F = T.ABC:kimberling(110)
 kiepert = conic(z.F,L.euler,1)
 curve = kiepert:points(-4, 4, 50)
 z.ea, z.eb = L.euler:get() }
\begin{center}
\begin{tikzpicture}[scale=.8]
  \tkzGetNodes
  \tkzDrawLines[cyan,
     add= .5 and .5](A,C A,B A,C)
  \tkzDrawPolygon[cyan](A,B,C)
  \tkzDrawCoordinates[smooth,red](curve)
  \tkzDrawLines[red,add= .5 and .5](ea,eb)
  \tkzDrawPoints(A,B,C,F,O,H)
  \tkzLabelPoints(A,B,C,F,O,H)
\end{tikzpicture}
\end{center}
```

14.6.7. Method isogonal(pt)

This method returns the **isogonal conjugate** of a point **pt** with respect to a triangle ABC.

The isogonal conjugate Y of a point X is defined as the common point of the three lines obtained by reflecting the lines AX, BX, and CX across the respective internal angle bisectors at A, B, and C.

An alternative but equivalent construction — and the one used in this method — consists in:

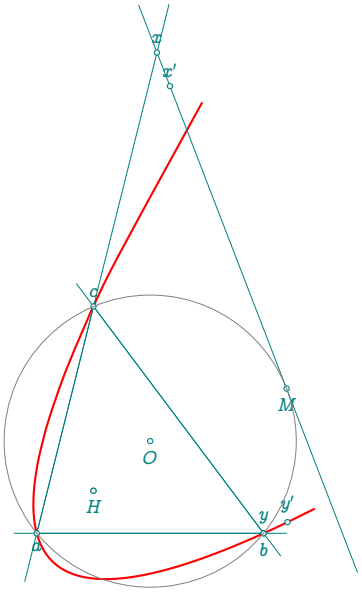
- reflecting point X across each side of the triangle,
- constructing the circle through the three reflected points,
- returning the center of this circle.

Validation: Using this method, you can verify that the isogonal conjugate of the orthocenter is indeed the circumcenter of triangle ABC.

Geometric remark: If a point M lies on the circumcircle, then the isogonal conjugates of the points on the tangent line at M trace a parabola that passes through the three vertices of the triangle. This elegant but subtle configuration highlights the dynamic complexity of isogonal mappings:

- the vertices of the triangle are the conjugates of the tangent's intersection points with the triangle's sides;
- points near M on the tangent map to conjugates that tend toward infinity.

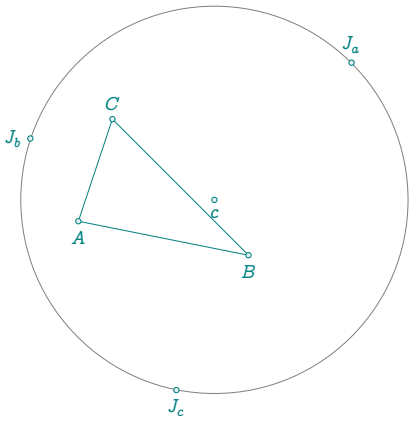
In the accompanying example, x is the intersection of the tangent with side AC, and x' is a nearby point chosen to observe the behavior of its isogonal conjugate y'.




```
\directlua{
  z.a = point(0, 0)
  z.b = point(4, 0)
  z.c = point(1, 4)
  T.abc = triangle(z.a, z.b, z.c)
  z.H = T.abc.orthocenter
  z.O = T.abc:isogonal(z.H)
  z.I = T.abc.incenter
  C.Oa = circle(z.O, z.a)
  z.M = C.Oa:point(0.45)
  Ta = C.Oa:tangent_at(z.M)
  z.u = Ta.pb
  z.v = Ta.pa
  z.x = intersection(Ta,T.abc.ca)
  z.y = T.abc:isogonal(z.x)
  L.Mx = line(z.M,z.x)
  z.xp = L.Mx:point(0.9)
  z.yp = T.abc:isogonal(z.xp)
  PA.points = path()
  for t = 1.5, 50, 1/10 do
    local x = Ta:point(t)
    local y = T.abc:isogonal(x)
    PA.points:add_point(y)
  end
  for t = -55, 0.2, 1/10 do
    local x = Ta:point(t)
    local y = T.abc:isogonal(x)
   PA.points:add_point(y)  end}
\begin{center}
  \begin{tikzpicture}[scale=.75]
     \tkzGetNodes
     \tkzDrawPolygon(a,b,c)
     \tkzDrawCoordinates[smooth,red,thick](PA.points)
     \tkzDrawLines[add =.1 and .1](x,v a,b b,c a,x)
     \tkzDrawPoints(a,b,c,H,M,x,y,H,O,x',y')
     \tkzLabelPoints(a,b,M,H,O)
     \tkzLabelPoints[above](c,x,y,x',y')
     \tkzDrawCircles(O,a)
  \end{tikzpicture}
\end{center}
```

14.6.8. Method bevan_point()

The Bevan point of a triangle is the circumcenter of the excentral triangle.




```
\directlua{
 init_elements()
 z.A = point(1, 1)
 z.B = point(6, 0)
 z.C = point(2, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 T.exc = T.ABC:excentral()
 z.J_a, z.J_b, z.J_c = T.exc:get()
 z.c = T.ABC:bevan_point()}
\begin{tikzpicture}[scale=.45]
  \tkzGetNodes
  \tkzDrawPolygons(A,B,C)
  \tkzDrawCircle(c,J_a)
  \tkzDrawPoints(A,B,C,c,J_a,J_b,J_c)
  \tkzLabelPoints(A,B,c,J_c)
  \tkzLabelPoints[above](C,J_a)
  \tkzLabelPoints[left](J_b)
\end{tikzpicture}
```

14.6.9. Method excenter(pt)

Since the argument is one of the triangle's vertices, this method returns the center of the corresponding exinscribed circle.

14.6.10. Method projection(pt)

This method returns the three orthogonal projections of a point onto the sides of the triangle.

Given a triangle ABC and a point P, the method computes the feet of the perpendiculars dropped from P to each of the sides $[BC]$, $[AC]$, and $[AB]$.

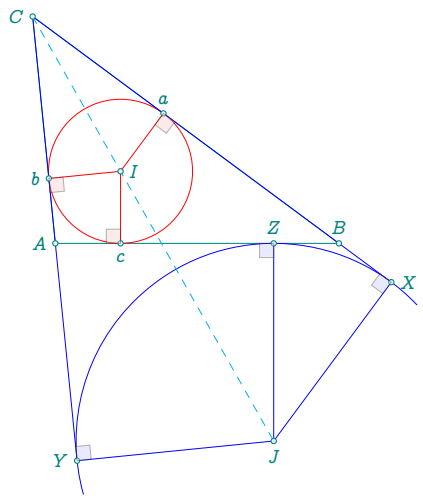
The result consists of three points, which can be retrieved as follows:

```
z.D, z.E, z.F = T.ABC:projection(z.P)
```

where:

- **z.D** is the projection onto BC,
- **z.E** onto AC,
- **z.F** onto AB.

This method is useful, for example, in constructing the pedal triangle of a point with respect to a triangle.




```
\directlua{
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(-.4 , 4)
T.ABC = triangle(z.A, z.B, z.C)
z.I = T.ABC.incenter
z.a,
z.b,
z.c = T.ABC:projection(z.I)
z.J = T.ABC:excenter(z.C)
z.X,
z.Y,
z.Z = T.ABC:projection(z.J)}
\begin{center}
  \begin{tikzpicture}[scale=.75]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawArc[blue](J,X)(Y)
  \tkzDrawCircle[red](I,a)
  \tkzDrawSegments[blue](J,X J,Y J,Z C,Y C,X)
  \tkzDrawSegments[red](I,a I,b I,c)
  \tkzDrawSegments[cyan,dashed](C,J)
  \tkzDrawPoints(A,B,C,I,a,b,c,J,X,Y,Z)
  \tkzLabelPoints(J,c)
  \tkzLabelPoints[right](I,X)
  \tkzLabelPoints[above](B,Z,a)
  \tkzLabelPoints[left](A,Y,b,C)
  \tkzMarkRightAngles[fill=blue!20,
     opacity=.4](A,Z,J A,Y,J J,X,B)
     \tkzMarkRightAngles[fill=red!20,
        opacity=.4](A,b,I A,c,I I,a,B)
  \end{tikzpicture}
  \end{center}
```

14.6.11. Method parallelogram()

Complete a triangle as a parallelogram. If **z.D = T.ABC:parallelogram()** then $ABCD$ is a parallelogram.

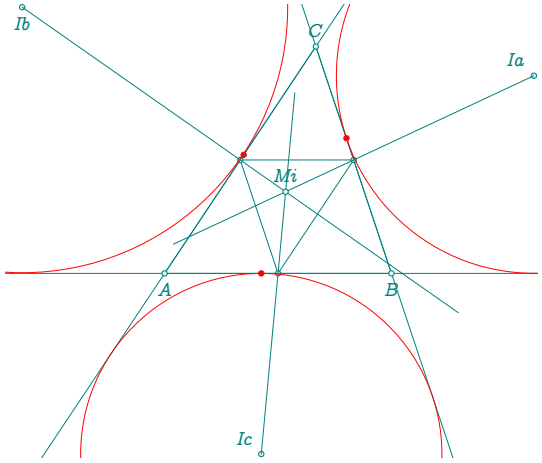
14.6.12. Method mittenpunkt

This method returns the **Mittenpunkt** of the triangle, also known as the *middlespoint*.

The Mittenpunkt is defined as the *symmedian point of the excentral triangle*, i.e., the point of intersection of the lines joining each excenter to the midpoint of the corresponding side of the original triangle.

It is a notable triangle center, designated as $X_{(9)}$ in Kimberling's classification.

Weisstein, Eric W. "Mittenpunkt." MathWorld.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(4, 6)
 T.ABC = triangle(z.A, z.B, z.C)
 z.Ma,
 z.Mb,
 z.Mc = T.ABC:medial():get()
 z.Ia, z.Ib, z.Ic = T.ABC:excentral():get()
 z.Mi = T.ABC:mittenpunkt_point()
 T.int = T.ABC:extouch()
 z.Ta, z.Tb,
 z.Tc = T.int:get()}
```

14.6.13. Method gergonne_point()

This method returns the **Gergonne point** of the triangle, denoted $X_{(7)}$ in Kimberling's classification.

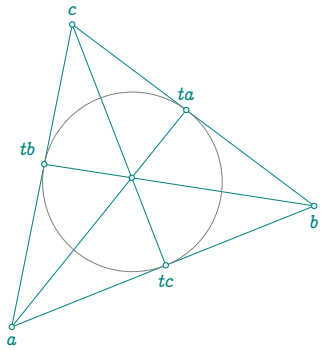
The Gergonne point is the common point of the lines connecting each vertex of the triangle to the point of contact of the incircle with the opposite side.

In this example, the method is often combined with:

- `intouch` – to get the contact triangle (also called the intouch triangle),
- `contact` – to retrieve the three contact points individually.

These contact points are the feet of the perpendiculars from the incenter to each side, i.e., the tangency points of the incircle with the triangle's sides.

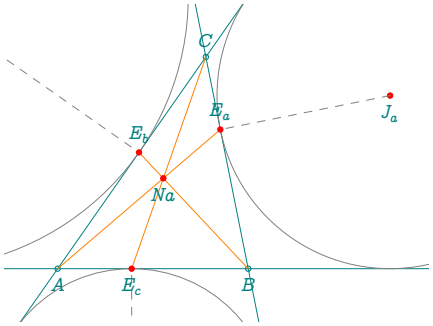
Weisstein, Eric W. "Gergonne Point." MathWorld.




```
\directlua{
  init_elements()
  z.a = point(1,0)
  z.b = point(6,2)
  z.c = point(2,5)
  T.abc = triangle(z.a, z.b, z.c)
  z.g = T.abc:gergonne_point()
  z.i = T.abc.incenter
  z.ta, z.tb, z.tc = T.abc:intouch():get()}
```

14.6.14. Method Nagel_point

Let E_a be the point at which the J_a-excircle meets the side (BC) of a triangle ABC, and define E_b and E_c similarly. Then the lines A,E_a, B,E_b and C,E_c concur in the Nagel point Na.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(3.6, 0)
  z.C = point(2.8, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.Na = T.ABC:nagel_point()
  z.J_a, z.J_b,
  z.J_c = T.ABC:excentral():get()
  z.E_a, z.E_b,
  z.E_c = T.ABC:extouch():get()}
```

14.6.15. Method feuerbach_point()

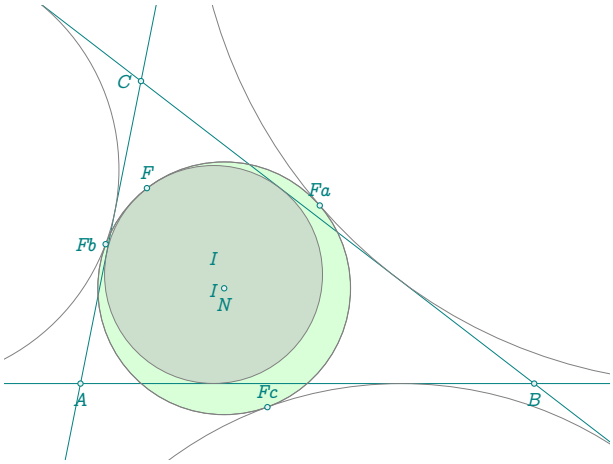
This method returns the **Feuerbach point** of the triangle, designated as $X_{(11)}$ in Kimberling's classification.

The Feuerbach point F is the unique point where the triangle's *incircle* and *nine-point circle* are tangent to each other.

In addition, the three *excircles* of the triangle are each tangent to the nine-point circle. These three points of tangency define what is called the **Feuerbach triangle**, whose vertices are often denoted F_a, F_b, and F_c.

This construction reveals a remarkable configuration of internal and external circle tangency with the nine-point circle.

Weisstein, Eric W. "Feuerbach Point." MathWorld.




```
\directlua{
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(0.8, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 z.N = T.ABC.eulercenter
 z.Fa, z.Fb,
 z.Fc = T.ABC:feuerbach():get()
 z.F =  T.ABC:feuerbach_point()
 z.Ja, z.Jb,
 z.Jc = T.ABC:excentral():get()
 z.I = T.ABC.incenter}
```

14.6.16. Method symmedian_point()

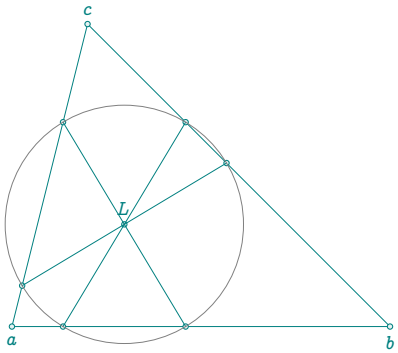
This method returns the **symmedian point** K of the triangle, also known as the *Lemoine point* (in English and French literature) or the *Grebe point* (in German).

The symmedian point is the point of concurrence of the triangle's three **symmedians**, which are the isogonal conjugates of the medians. In other words, K is the **isogonal conjugate** of the centroid G.

You can also use the aliases **lemoine_point** or **grebe_point** to call this method.

A beautiful geometric property of the Lemoine point is the following: *The antiparallels to the triangle's sides passing through the symmedian point intersect the sides in six points that all lie on a same circle* — this is known as the **first Lemoine circle**.

Weisstein, Eric W. "Symmedian Point." MathWorld.




```
\directlua{
 init_elements()
 z.a = point(0, 0)
 z.b = point(5, 0)
 z.c = point(1, 4)
 T.abc = triangle(z.a, z.b, z.c)
 z.L = T.abc:lemoine_point()
 L.anti = T.abc:antiparallel(z.L, 0)
 z.x_0, z.x_1 =L.anti:get()
 L.anti = T.abc:antiparallel(z.L, 1)
 z.y_0, z.y_1 = L.anti:get()
 L.anti = T.abc:antiparallel(z.L, 2)
 z.z_0, z.z_1 = L.anti:get()}
```

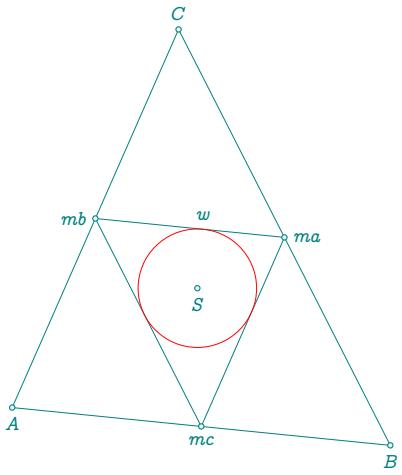
14.6.17. Method spieker_center

This method returns the **Spieker center** of the triangle, denoted $X_{(10)}$ in Kimberling's classification.

The Spieker center S_p is the **incenter of the medial triangle** of the reference triangle ABC. It is also the center of the *Spieker circle*, which is the incircle of that medial triangle.

An important property is that the Spieker center is also the **center of the radical circle** of the triangle's three *excircles*. This makes it a key point in both classical triangle geometry and circle configurations.

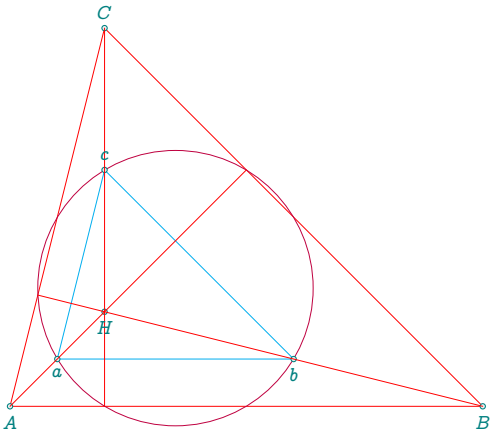
Weisstein, Eric W. "Spieker Circle." MathWorld.




```
\directlua{
 z.A = point (0, 0)
 z.B = point (5, -0.5)
 z.C = point (2.2, 5)
 T.ABC = triangle(z.A, z.B, z.C)
 z.S = T.ABC:spieker_center()
 T.m = T.ABC:medial()
 z.ma, z.mb, z.mc = T.m:get()
 z.w = T.m.ab:projection(z.S)}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPolygons(A,B,C ma,mb,mc)
 \tkzDrawCircles[red](S,w)
 \tkzDrawPoints(A,B,C,S,ma,mb,mc)
 \tkzLabelPoints(A,B,S,mc)
 \tkzLabelPoints[above](C,w)
 \tkzLabelPoints[right](ma)
 \tkzLabelPoints[left](mb)
\end{tikzpicture}
```

14.6.18. Method euler_points

The points a, b and c are the Euler points. They are the midpoints of the segments AH, BH and CH.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.N = T.ABC.eulercenter
  z.a,
  z.b,
  z.c = T.ABC:euler():get()
  z.H = T.ABC.orthocenter
  T.orthic = T.ABC:orthic()
  z.Ha,
  z.Hb,
  z.Hc = T.orthic:get()}
```

14.6.19. Method nine_points

This method returns the **nine classical points** that lie on the Euler circle (also called the nine-point circle) of a triangle.

The returned points, in order, are:

- the three midpoints of the sides of the triangle,
- the three feet of the altitudes,

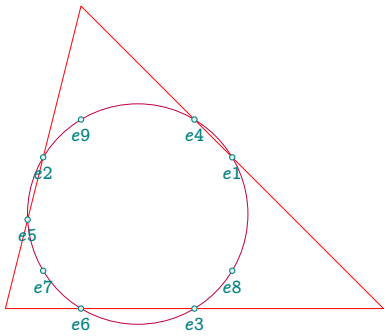
– the three Euler points (i.e., the midpoints between each vertex and the orthocenter).

These nine points lie on the same circle, whose center is the midpoint of the segment joining the orthocenter to the circumcenter.

In the next example, we also compute the centroid (center of gravity) in two different ways:

– using the **trilinear** method with coordinates (1 : 1 : 1),

– using the **barycentric** method with weights (1, 1, 1).




```
\directlua{
  init_elements()
  z.A = point(0 ,0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.N = T.ABC.eulercenter
  z.e1,
  z.e2,
  z.e3,
  z.e4,
  z.e5,
  z.e6,
  z.e7,
  z.e8,
  z.e9 = T.ABC:nine_points()}
```

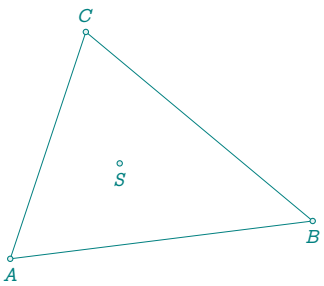
14.6.20. Method soddy_center

This method returns the **Soddy center** of the triangle.

Given three mutually tangent circles (typically the triangle's three *inner Apollonius circles* centered at each vertex), there exist exactly two non-intersecting circles that are tangent to all three. These are called the *inner* and *outer Soddy circles*, and their respective centers are called the **inner** and **outer Soddy centers**.

By default, the method returns the *inner* Soddy center. A keyword option such as **"outer"** may be used to retrieve the second one, if implemented.

See section 14.8.1 for details about the associated Soddy circles.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0.5)
  z.C = point(1, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  z.S = T.ABC:soddy_center()}
```

14.6.21. Method conway_points()

This method returns the six points defined in **Conway's circle theorem**.

The Conway circle theorem states the following:

> If the sides meeting at each vertex of a triangle are extended by the length of the opposite side, the six endpoints of the resulting three segments all lie on a common circle. This circle is called the **Conway circle** of the triangle, and its center is the **incenter** of the triangle.

The six returned points can then be used to draw this Conway circle and to verify its geometric properties.

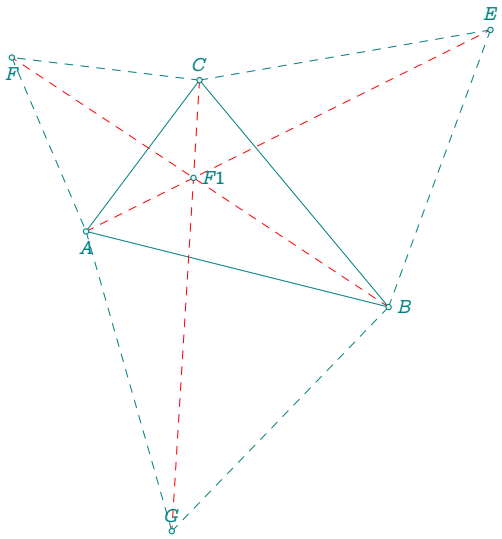
See the corresponding figure in section 14.8.8 for a complete illustration.

Wikipedia – Conway circle theorem

```
C.conway = T.ABC:conway_circle()
z.w,z.t  = C.conway:get() %   % z.w = T.ABC : conway_center ()
z.t1, z.t2, z.t3, z.t4, z.t5, z.t6 = T.ABC: onway_points()
```

14.6.22. Method first_fermat_point()

In a given triangle ABC with all angles less than 120 degrees (2pi/3), the first Fermat point is the point which minimizes the sum of distances from A, B, and C.




```
\directlua{
 z.A = point(1, 2)
 z.B = point(5, 1)
 z.C = point(2.5, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 z.F1 = T.ABC:first_fermat_point()
 _,_,z.E = T.ABC.bc:equilateral("swap"):get()
 _,_,z.F = T.ABC.ca:equilateral("swap"):get()
 _,_,z.G = T.ABC.ab:equilateral("swap"):get()}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPolygon(A,B,C)
 \tkzDrawSegments[dashed](B,E E,C C,F F,A)
 \tkzDrawSegments[dashed](A,G G,B)
 \tkzDrawSegments[dashed,red](A,E B,F C,G)
 \tkzDrawPoints(A,B,C,F1,E,F,G)
 \tkzLabelPoints[right](B,F1)
 \tkzLabelPoints[above](C,E,G)
 \tkzLabelPoints(A,F)
\end{tikzpicture}
```

14.6.23. Method second_fermat_point()

See [14.7.11]]

14.6.24. Method kenmotu_point()

This method returns the **Kenmotu point** of the triangle, also known as the *congruent squares point*. It is listed as $X_{(371)}$ in Kimberling's classification.

The Kenmotu point is the unique point where three equal squares, each inscribed in the triangle and touching two sides, intersect at a single common point. Each square is constructed so that it fits snugly between two adjacent triangle sides.

This triangle center is remarkable for its connection to equal-area geometric constructions and is defined purely by square congruence and tangency conditions.

See the illustration in section 14.8.18 for a visual representation of the configuration.

Weisstein, Eric W. "Kenmotu Point." MathWorld

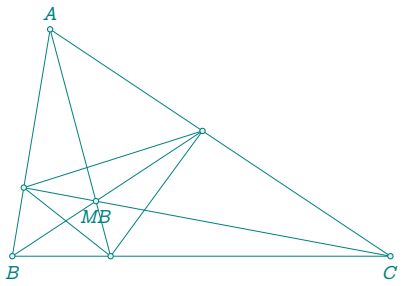
14.6.25. Method adams_points()

Given a triangle ABC, construct the **contact** triangle $T_AT_BT_C$. Now extend lines parallel to the sides of the contact triangle from the **Gergonne** point. These intersect the triangle ABC in the six points P,Q,R,S,T, and U. **C. Adams** proved in 1843 that these points are concyclic in a circle now known as the Adams' circle.

Weisstein, Eric W. "Adams' Circle." From MathWorld–A Wolfram Web Resource.

14.6.26. Method macbeath_point

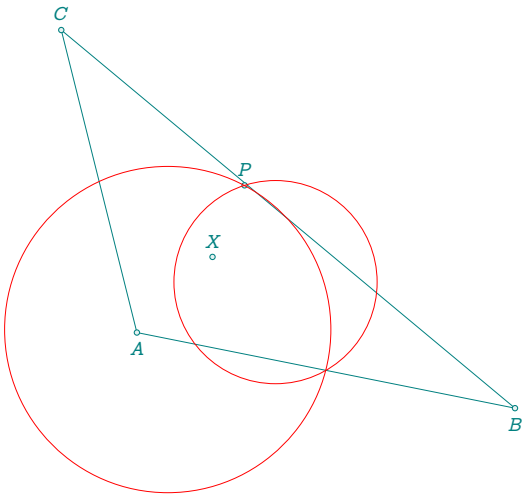
The MacBeath point is the center of the cercle kimberling(264).




```
\directlua{
init_elements()
z.A = point(.5, 3)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A, z.B, z.C)
z.MB = T.ABC:macbeath_point()
z.Xa, z.Xb,
z.Xc = T.ABC:macbeath():get()}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygons(A,B,C Xa,Xb,Xc)
\tkzDrawSegments(A,Xa B,Xb C,Xc)
\tkzDrawPoints(A,B,C,MB,Xa,Xb,Xc)
\tkzLabelPoints(B,C,MB)
\tkzLabelPoints[above](A)
\end{tikzpicture}
```

14.6.27. Method poncelet_point(p)

If the three vertices and the p point do not form a orthocentric system et no three of them are collinear. The nine-point circles of the four triangles obtained with three of the four points have one thing in common, called **Poncelet** point.



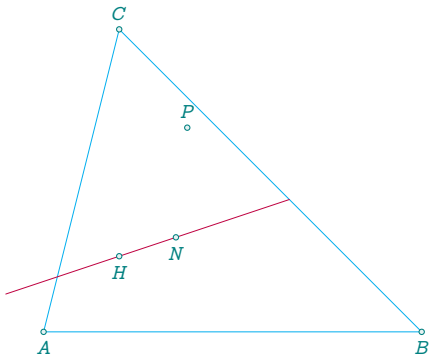

```
\directlua{
 init_elements()
 z.A = point(1, 1)
 z.B = point(6, 0)
 z.C = point(0, 5)
 z.X = point(2, 2)
 T.ABC = triangle(z.A, z.B, z.C)
 z.P = T.ABC:poncelet_point(z.X)
 z.I = T.ABC.eulercenter
 z.Ma, z.Mb, z.Mc = T.ABC:medial():get()
 T.ABX = triangle(z.A, z.B, z.X)
 z.I1 = T.ABX.eulercenter}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygons(A,B,C)
  \tkzDrawPoints(A,B,C,X,P)
  \tkzDrawCircles[red](I,Ma I1,Mc)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C,X,P)
\end{tikzpicture}
\end{center}
```

14.6.28. Method orthopole

For the definition and some properties, please refer to:

Weisstein, Eric W. "Orthopole." From MathWorld–A Wolfram Web Resource.

See the following example for the link between orthopole and Simson line: (An example x can be found in the document **Euclidean Geometry** presented in altermundus.fr.)




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(5, 0)
 z.C = point(1, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 z.N = T.ABC.eulercenter
 z.H = T.ABC.orthocenter
 L.NH = line(z.N, z.H)
 z.P = T.ABC:orthopole(L.NH)
}
\begin{center}
 \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygons[cyan](A,B,C)
  \tkzDrawLine[purple,add = 2 and 2](N,H)
  \tkzDrawPoints(A,B,C,N,H,P)
  \tkzLabelPoints(A,B,H,N)
  \tkzLabelPoints[above](C,P)
 \end{tikzpicture}
\end{center}
```

14.6.29. Method isodynamic_points()

This method computes the two isodynamic points of a non-degenerate triangle.

Geometric background: Given a triangle ABC, consider the three Apollonius circles associated with its sides:

$$\mathcal{A}_a, \quad \mathcal{A}_b, \quad \mathcal{A}_c,$$

each defined as the locus of points M satisfying a ratio of distances corresponding to the opposite sides.
A classical and fundamental property states that:

> *The three Apollonius circles intersect in exactly two points. These intersections are the first and second isodynamic points of the triangle.*

The first isodynamic point is symmetric to the second by reflection in the circumcenter. Both points are isogonal conjugates of the Fermat points.

Description: The method **isodynamic_points()** returns the two common intersection points of the Apollonius circles.

Return values:

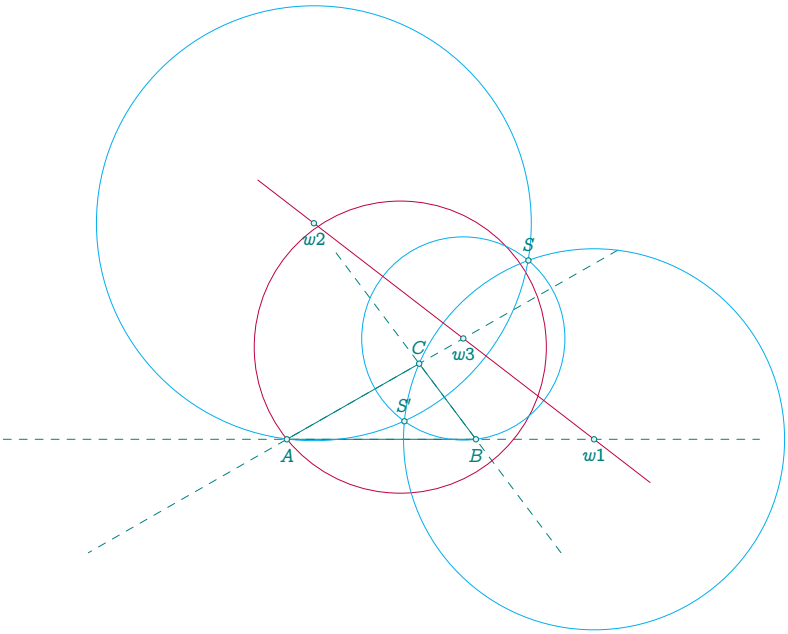
- **S1** — the first isodynamic point,
- **S2** — the second isodynamic point.

Example:

```
S1, S2 = T.ABC:isodynamic_points()
```

Each point can then be used like any other **point** object in subsequent constructions.

Application:




```
\directlua{%
z.A = point (0, 0)
z.B = point (5, 0)
z.C = point (3.5, 2)
T.ABC = triangle (z.A,  z.B, z.C)
C.ab, C.bc,
C.ca = T.ABC:three_apollonius_circles()
z.w1, z.t1 = C.ab:get()
z.w2, z.t2 = C.bc:get()
z.w3, z.t3 = C.ca:get()
z.S, z.Sp = T.ABC:isodynamic_points()
}
\begin{center}
\begin{tikzpicture}[scale=.5]
 \tkzGetNodes
 \tkzDrawLines[add = 1.5 and 1.5,
               dashed](A,B B,C C,A)
 \tkzDrawPolygons(A,B,C)
 \tkzDrawCircle[purple](O,A)
 \tkzDrawCircles[cyan](w1,t1 w2,t2 w3,t3)
 \tkzDrawLines[purple](w1,w2)
 \tkzDrawPoints(A,B,C,w1,w2,w3,S,S')
 \tkzLabelPoints(A,B,w1,w2,w3)
  \tkzLabelPoints[above](C,S,S')
\end{tikzpicture}
\end{center}
```

14.6.30. Method apollonius_points(side)

This method returns the two Apollonius division points associated with a specified side of the triangle.

Description. Given a triangle ABC and a chosen side:

- **"ab"** — side AB, ratio defined by the opposite side $c = AB$,
- **"bc"** — side BC, ratio defined by the opposite side $a = BC$,

– **"ca"** — side CA, ratio defined by the opposite side $b = CA$,

the method returns the two points that divide the chosen side internally and externally according to the ratio of the adjacent sides.

Return values: Two points:

– **P_int** — internal division point on the chosen side,

– **P_ext** — external division point on the chosen side.

Example:

```
Pin, Pex = T.ABC:apollonius_points("ab")
```

The returned objects are standard **point** instances usable in further constructions.
Application: See [14.8.23]

14.6.31. Method apollonius_point()

This method should not be confused with the previous one.
Consider the excircles J_A, J_B, and J_C of a triangle, and the external Apollonius circle tangent externally to all three. Denote the contact point by xa, xb and xc, etc. Then the lines Axa, Bxb, and Cxc concur in a point known as the Apollonius point. This point is Kimberling center $X_{(}181)$ (Kimberling 1998, p. 102).

(Reference: Weisstein, Eric W. "Apollonius Point." MathWorld)

14.7. Returns a line

14.7.1. Method symmedian_line(n)

This method returns one of the **symmedians** of the triangle.

The lines AL_a, BL_b, and CL_c, which are the isogonal conjugates of the medians AM_a, BM_b, and CM_c, are called the triangle's **symmedians**. These lines concur at a point L, known as the **Lemoine point** or **symmedian point**. It is the isogonal conjugate of the centroid G.

The triangle $L_aL_bL_c$, formed by the intersections of the symmedians with the sides of the triangle ABC, is called the **symmedial triangle**. The circle circumscribed about this triangle is known as the **symmedial circle**.

To obtain:

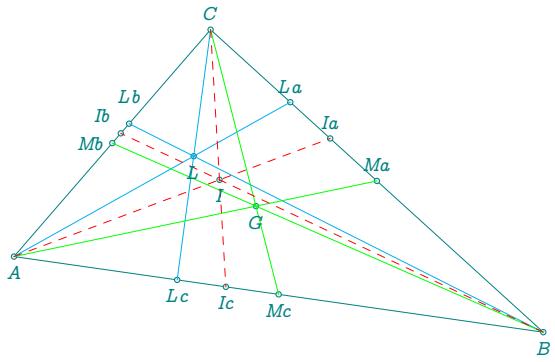
– all three symmedians, use the method **symmedian()**;

– the point of concurrency, use **symmedian_point()** or **lemoine_point()**;

– only one symmedian, use **symmedian_line(n)** with $n = 0$, 1, or 2, corresponding respectively to the vertices A, B, or C.

In the following example, the point L is the Lemoine point, and the triangle $L_aL_bL_c$ is the symmedial triangle.

Weisstein, Eric W. "Symmedian Point." MathWorld.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(2.4, 1.8)
 T.ABC = triangle(z.A, z.B, z.C)
 z.L = T.ABC:lemoine_point()
 T.SY = T.ABC:symmedian()
 T.med = T.ABC:medial()
 z.Ka, z.Kb, z.Kc = T.SY:get()
 z.Ma, z.Mb, z.Mc = T.med:get()
 L.Kb = T.ABC:symmedian_line(1)
 _, z.Kb = L.Kb:get()
 z.G = T.ABC.centroid
 z.Ia, z.Ib,
 z.Ic = T.ABC:incentral():get()
 z.I = T.ABC.incenter}
```

14.7.2. Method altitude(arg)

This method returns one of the **altitudes** of a triangle.

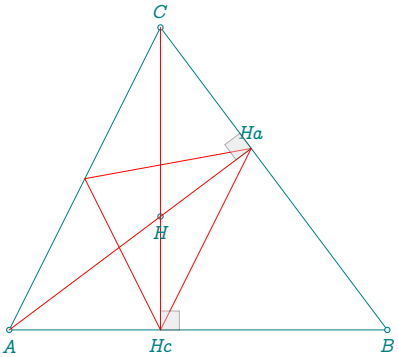
There are multiple ways to access altitudes:

- `orthic` returns the **orthic triangle**, whose vertices are the feet of the three altitudes.
- `altitude` returns a single altitude line, based on an optional argument.

The optional argument **arg** determines from which vertex the altitude is dropped. It may be:

- **nil** or **0** (default): returns the altitude from vertex A;
- **1**: returns the altitude from vertex B;
- **2**: returns the altitude from vertex C;
- a point equal to **z.A**, **z.B**, or **z.C** (the vertex of origin).

This interface matches that of other methods such as `bisector` and `symmedian_line`.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.H = T.ABC.orthocenter
  L.HA = T.ABC:altitude()
  L.HC = T.ABC:altitude(z.C)
  z.Hc = L.HC.pb
  z.Ha = L.HA.pb
  z.a, z.b,
  z.c = T.ABC:orthic():get()}
```

14.7.3. Method bisector(arg)

This method returns one of the three **internal angle bisectors** of a triangle.

In many cases, it may be sufficient to compute the incenter and connect it to the triangle's vertices, since all bisectors pass through the incenter. For example:

```
z.I = T.ABC.incenter
```

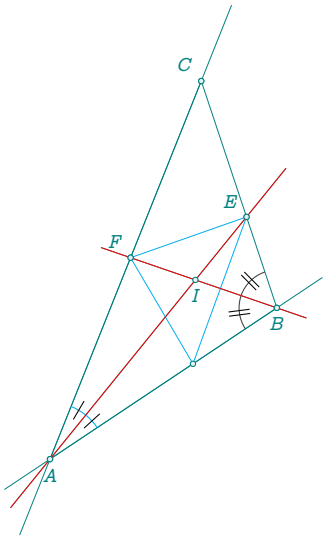
However, the method **bisector(arg)** explicitly computes the intersection of the bisector with the opposite side. This is useful for constructing the *incentral triangle* or other derived figures.

If all three bisectors are needed, use the method **incentral**, which returns the **incentral triangle**, whose vertices are the feet of the internal bisectors.

The optional argument **arg** controls from which vertex the bisector originates. It can be:

- **nil** or **0** (default): bisector from vertex A;
- **1**: bisector from vertex B;
- **2**: bisector from vertex C;
- a point equal to one of the triangle's vertices, such as **z.A**, **z.B**, or **z.C**.

This behavior is consistent with other methods like **altitude**, **symmedian_line**, and **mediator**.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(3, 2)
  z.C = point(2, 5)
  T.ABC = triangle(z.A, z.B, z.C)
  L.AE = T.ABC:bisector()
  z.E  = L.AE.pb
  z.F  = T.ABC:bisector(z.B).pb
  z.a, z.b, z.c = T.ABC:incentral():get()
  z.I = T.ABC.incenter}
```

14.7.4. Method bisector_ext(arg)

This method returns an **external angle bisector** of the triangle, from a specified vertex.

Its interface is identical to **bisector**, accepting either an index **n** or a vertex point (**z.A**, **z.B**, **z.C**). See Section 14.7.3 for usage details.

14.7.5. Method mediator(...)

This method returns the perpendicular bisector of one side of a triangle.
Syntax:

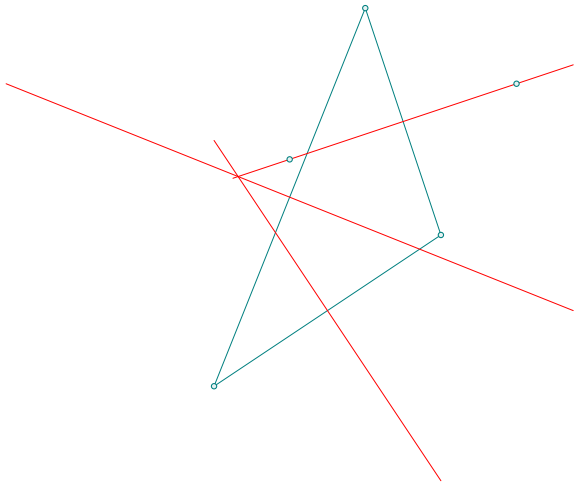
```
L = triangle:mediator()
L = triangle:mediator(n) n = 0, 1 or 2
L = triangle:mediator(pt) pt vertex of the triangle
```

Arguments:

- If no argument or `n = 0` is given, the bisector of side **AB** is returned.
- If `n = 1`, the bisector of side **BC** is returned.
- If `n = 2`, the bisector of side **CA** is returned.
- If a point (**pa**, **pb**, **pc**) is passed, the method returns the bisector of the side opposite to that vertex.

Return value:
Returns a **line** object representing the perpendicular bisector of the selected side.
Examples.

```
T = triangle(z.A, z.B, z.C)
L1 = T:mediator()        -- bisector of AB
L2 = T:mediator(1)       -- bisector of BC
L3 = T:mediator(z.C)     -- bisector of AB (since C is opposite AB)
```



```
\directlua{
  z.A = point(0, 0)
  z.B = point(3, 2)
  z.C = point(2, 5)
  T.ABC = triangle(z.A, z.B, z.C)
  L.MA  = T.ABC:mediator(z.A)
  L.MB  = T.ABC:mediator(z.B)
  L.MC  = T.ABC:mediator(z.C)
  z.ma1, z.ma2 = L.MA:get()
  z.mb1, z.mb2 = L.MB:get()
  z.mc1, z.mc2 = L.MC:get()}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawLines[red,
  add = .25 and .25](ma1,ma2 mb1,mb2
        mc1,mc2)
  \tkzDrawPoints(A,B,C,ma1,ma2)
\end{tikzpicture}
\end{center}
```

See also. `mediator`, `altitude`, `bisector`

14.7.6. Method antiparallel(arg)

This method returns an **antiparallel line** associated with a triangle and one of its angles.
Two lines, such as (PQ) and (BC), are said to be **antiparallel** with respect to an angle (e.g., $\angle A$) if they form equal angles (in opposite directions) with the bisector of that angle.
Useful properties of antiparallel lines:

- If (PQ) and (BC) are antiparallel with respect to $\angle A$, then the four points P, Q, B, and C are **concyclic**.
- Antiparallelism is symmetric: if (PQ) is antiparallel to (BC), then (BC) is antiparallel to (PQ).

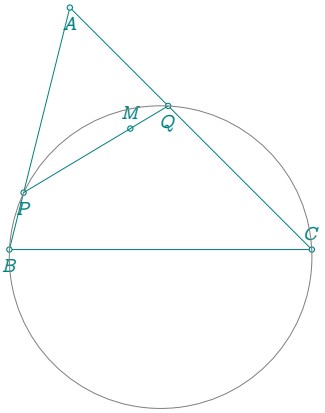
This method takes two arguments:

- **pt** (mandatory): a point through which the antiparallel line will pass;
- **arg** (optional): specifies the vertex of the triangle that defines the reference angle.

The second argument **arg** can be:

- **nil** or **0** (default): antiparallel to BC w.r.t. angle $\hat{A}$;
- **1**: antiparallel to AC w.r.t. angle $\hat{B}$;
- **2**: antiparallel to AB w.r.t. angle $\hat{C}$;
- a point equal to one of the triangle's vertices: **z.A**, **z.B**, or **z.C**.

This method is used, for example, in the construction of the symmedian point (see Section 14.6.16).




```
\directlua{
  init_elements()
  z.B = point(0, 0)
  z.C = point(5, 0)
  z.A = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.M = point(2, 2)
  L.anti = T.ABC:antiparallel(z.M, z.A)
  z.P, z.Q = L.anti:get()
  T.PQ = triangle(z.P, z.Q, z.B)
  z.W = T.PQ.circumcenter}
```

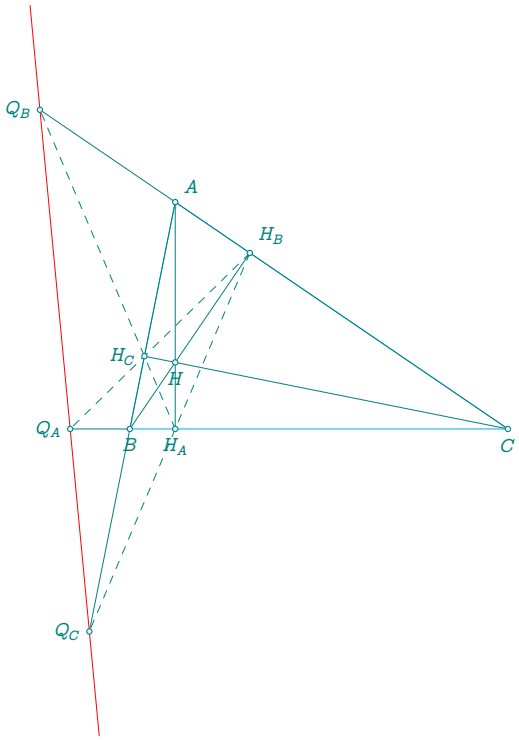
14.7.7. Method orthic_axis() and orthic_axis_points()

Let $H_AH_BH_C$ be the **orthic triangle** of a reference triangle ABC (i.e., the triangle formed by the feet of the altitudes from each vertex).

Each side of triangle ABC intersects each side of triangle $H_AH_BH_C$, and the three points of intersection all lie on a common straight line. This line is known as the **orthic axis** of the triangle.

The method `orthic_axis()` returns this line as an object of class **`line`**. The method `orthic_axis_points()` returns the three intersection points between corresponding sides of the reference triangle and the orthic triangle. These are the points O_A, O_B, and O_C aligned on the orthic axis.

This remarkable line is a classical geometric construction, often overlooked despite its elegance.




```
\directlua{
  z.B = point(0, 0)
  z.C = point(5, 0)
  z.A = point(.6, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  z.H = T.ABC.orthocenter
  z.H_A, z.H_B,
  z.H_C  = T.ABC:orthic():get()
  L.orthic  = T.ABC:orthic_axis()
  z.Q_A, z.Q_B,
  z.Q_C = T.ABC:orthic_axis_points()}
\begin{center}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPolygons[cyan](A,B,C)
 \tkzDrawLines[red](Q_C,Q_B)
 \tkzDrawSegments(A,Q_C B,Q_A A,H_A)
 \tkzDrawSegments(B,H_B C,H_C C,Q_B)
 \tkzDrawSegments[dashed](H_B,Q_C H_B,Q_A)
 \tkzDrawSegments[dashed](H_A,Q_B)
 \tkzDrawPoints(A,B,C,H_A,H_B,H_C)
 \tkzDrawPoints(H,Q_A,Q_B,Q_C)
 \tkzLabelPoints(H)
 \tkzLabelPoints(B,C,H_A)
 \tkzLabelPoints[above right](A,H_B)
 \tkzLabelPoints[left](H_C,Q_A,Q_B,Q_C)
\end{tikzpicture}
\end{center}
```

14.7.8. Methods euler_line() and orthic_axis()

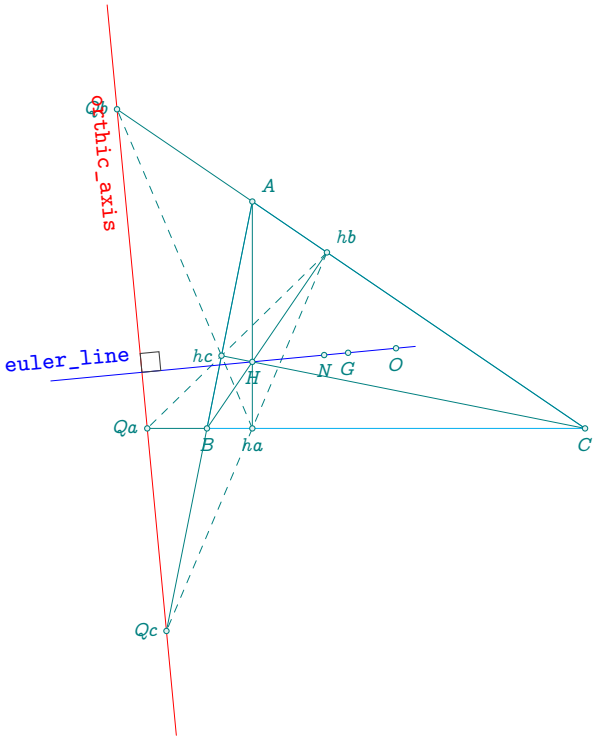
These two methods return two fundamental lines associated with triangle geometry: the **Euler line** and the **orthic axis**.

Euler line — The line on which lie several remarkable triangle centers:

- the orthocenter H,
- the centroid G,
- the circumcenter O,
- the nine-point center N,
- and many others.

Orthic axis — Let h_a, h_b, and h_c be the vertices of the orthic triangle of triangle ABC. Each side of triangle ABC intersects each side of the orthic triangle. The three points of intersection lie on a line called the **orthic axis**.

- This line is *perpendicular* to the Euler line.
- It is returned by the method `orthic_axis()` as a **`line`** object.
- To obtain the three intersection points explicitly, use `orthic_axis_points()`.




```
\directlua{
   init_elements()
    z.B = point(0 ,0)
    z.C = point(5, 0)
    z.A = point(.6, 3)
    T.ABC = triangle(z.A, z.B, z.C)
    z.N = T.ABC.eulercenter
    z.H = T.ABC.orthocenter
    z.O = T.ABC.circumcenter
    z.G = T.ABC.centroid
    z.ha, z.hb, z.hc = T.ABC:orthic():get()
    L.orthic = T.ABC:orthic_axis()
    z.Qa, z.Qb,
    z.Qc = T.ABC:orthic_axis_points()
    L.euler = T.ABC:euler_line()
    z.ea, z.eb = L.euler:get()
    z.K = L.orthic:projection(z.N)}
```

14.7.9. Method `steiner_line(pt)`

Let ABC be a triangle with orthocenter H, and let M be a point on the circumcircle of triangle ABC.

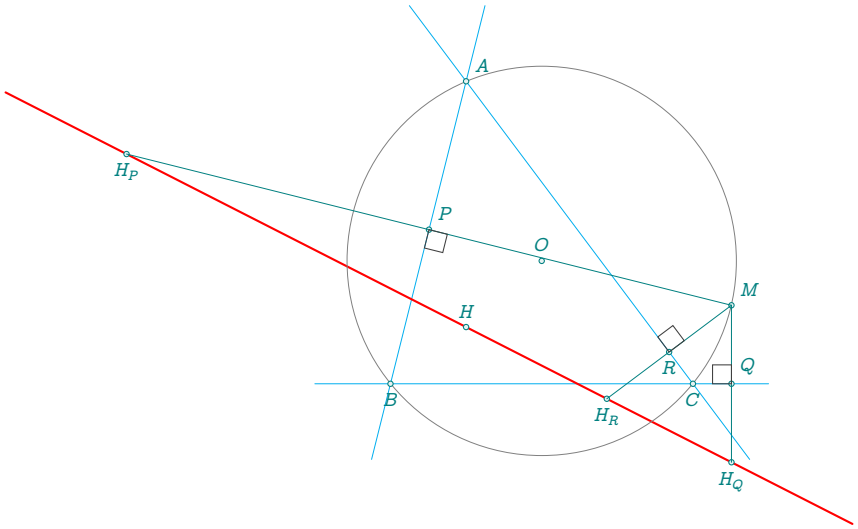
Define the following:

- H_A, H_B, and H_C are the reflections of point M across the sides BC, AC, and AB respectively.
- The points H_A, H_B, H_C, and the orthocenter H are collinear.

The line passing through these four points is called the **Steiner line** of point M with respect to triangle ABC.

The method `steiner_line(pt)` returns this line as a **`line`** object, provided that the point **`pt`** lies on the circumcircle of the triangle. The Steiner line always passes through the orthocenter H.

A necessary and sufficient condition for the points H_A, H_B, and H_C to be collinear (and for the Steiner line to exist) is that M lies on the circumcircle of triangle ABC.




```
\directlua{
 z.B = point(0, 0)
 z.C = point(4, 0)
 z.A = point(1, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 C.ABC = T.ABC:circum_circle()
 z.O = T.ABC.circumcenter
 z.H = T.ABC.orthocenter
 z.M = C.ABC:point(.65)
 z.H_P = T.ABC.ab:reflection(z.M)
 z.H_Q = T.ABC.bc:reflection(z.M)
 z.H_R = T.ABC.ca:reflection(z.M)
 L.steiner = T.ABC:steiner_line(z.M)
 z.P = T.ABC.ab:projection(z.M)
 z.Q = T.ABC.bc:projection(z.M)
 z.R = T.ABC.ca:projection(z.M)}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[cyan,add= .25 and .25](A,C A,B B,C)
  \tkzDrawLines[red,thick](H_P,H_Q)
  \tkzDrawCircles(O,A)
  \tkzDrawPoints(A,B,C,M,H_P,H_Q,H_R,O,H,P,Q,R)
  \tkzLabelPoints(B,C,H_P,H_Q,H_R,R)
  \tkzLabelPoints[above](O,H)
  \tkzLabelPoints[above right](P,Q,A,M)
  \tkzDrawSegments(M,H_P M,H_Q M,H_R)
  \tkzMarkRightAngles(B,P,M M,R,A M,Q,B)
  \end{tikzpicture}
\end{center}
```

14.7.10. Method lemoine_axis()

This method returns the **Lemoine axis** of the triangle.

The Lemoine axis is a classical and remarkable line associated with triangle geometry. It is defined in several equivalent ways:

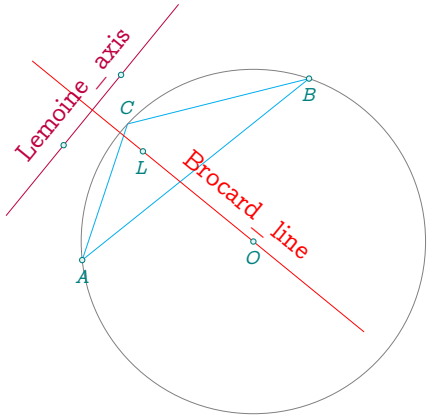
– It is the **polar** of the **Lemoine point** (or symmedian point) with respect to the circumcircle of the triangle.

– It passes through the points of intersection of the **tangents** to the circumcircle at each vertex and the **opposite sides**. These tangents are antiparallel to the opposite sides.

– It also contains the **centers of the three Apollonius circles** corresponding to the ordered triplets:

$$(A,B,\frac{CA}{CB}),\quad (B,C,\frac{AB}{AC}),\quad (C,A,\frac{BC}{BA})$$

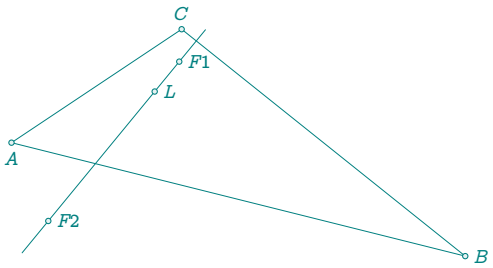
The method `lemoine_axis()` returns the line as an object of type **`line`**.




```
\directlua{
  z.A = point(0, -2)
  z.B = point(5, 2)
  z.C = point(1, 1)
  T.ABC = triangle(z.A, z.B, z.C)
  z.L = T.ABC:lemoine_point()
  z.O = T.ABC.circumcenter
  L.L = T.ABC:lemoine_axis()
  z.la,
  z.lb = L.L:get()
  L.B = T.ABC:brocard_axis()
  z.ba,
  z.bb = L.B:get()}
```

14.7.11. Method fermat_axis

The Fermat axis is the line connecting the first and second Fermat points.




```
\directlua{
 z.A = point(1, 2)
 z.B = point(5, 1)
 z.C = point(2.5, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.F1 = T.ABC:first_fermat_point()
 z.F2 = T.ABC:second_fermat_point()
 L.F = T.ABC:fermat_axis()
 z.a, z.b = L.F:get()
 z.L = T.ABC:lemoine_point()}
```

14.7.12. Method brocard_axis()

The **Brocard axis** is the straight line passing through the **symmedian point** K and the **circumcenter** O of a triangle. The segment $[KO]$ is referred to as the *Brocard diameter* (see Kimberling, 1998, p. 150).

This line has several remarkable properties:

– It is perpendicular to the Lemoine axis (see Section 14.7.10).

– It is the isogonal conjugate of the Kiepert hyperbola.

The method `brocard_axis()` returns this line as an object of type **`line`**.

Weisstein, Eric W. "Brocard Axis." MathWorld

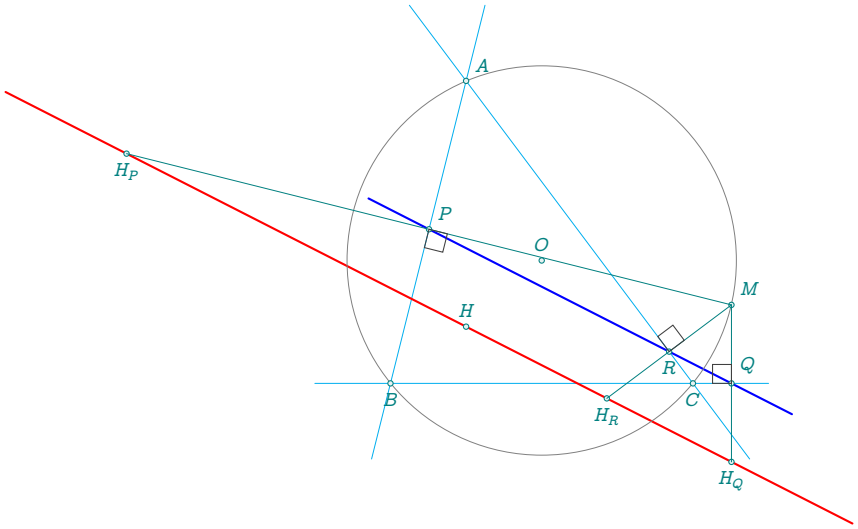
14.7.13. Method simson_line(pt)

The **Simson line** of a point M with respect to triangle ABC is the line that passes through the feet of the perpendiculars dropped from M to the sides (or their extensions) of the triangle.

Let P, Q, and R be the feet of the perpendiculars from M to sides BC, CA, and AB respectively. If M lies on the **circumcircle** of triangle ABC, then the points P, Q, and R are **collinear**. The line through them is called the *Simson line* of M.

The method `simson_line(pt)` returns this line (as a **`line`** object), assuming the point **`pt`** lies on the circumcircle of the triangle.

Jackson, Frank and Weisstein, Eric W. "Simson Line." MathWorld




```
\directlua{
 z.B = point(0, 0)
 z.C = point(4, 0)
 z.A = point(1, 4)
 T.ABC = triangle(z.A, z.B, z.C)
 C.ABC = T.ABC:circum_circle()
 z.O = T.ABC.circumcenter
 z.H = T.ABC.orthocenter
 z.M = C.ABC:point(.65)
 z.H_P = T.ABC.ab:reflection(z.M)
 z.H_Q = T.ABC.bc:reflection(z.M)
 z.H_R = T.ABC.ca:reflection(z.M)
 L.steiner = T.ABC:steiner_line(z.M)
 z.P = T.ABC.ab:projection(z.M)
 z.Q = T.ABC.bc:projection(z.M)
 z.R = T.ABC.ca:projection(z.M)
 L.simson = T.ABC:simson_line(z.M)
 z.sa, z.sb = L.simson:get()}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[cyan,add= .25 and .25](A,C A,B B,C)
  \tkzDrawLines[red,thick](H_P,H_Q)
  \tkzDrawLines[blue,thick](sa,sb)
  \tkzDrawCircles(O,A)
  \tkzDrawPoints(A,B,C,M,H_P,H_Q,H_R,O,H,P,Q,R)
  \tkzLabelPoints(B,C,H_P,H_Q,H_R,R)
  \tkzLabelPoints[above](O,H)
  \tkzLabelPoints[above right](P,Q,A,M)
  \tkzDrawSegments(M,H_P M,H_Q M,H_R)
  \tkzMarkRightAngles(B,P,M M,R,A M,Q,B)
  \end{tikzpicture}
\end{center}
```

14.8. Returns a circle

14.8.1. Method euler_circle()

The **nine-point circle**, also called the **Euler circle** or **Feuerbach circle**, is a fundamental circle in triangle geometry.

It passes through the following nine notable points of any triangle ABC:

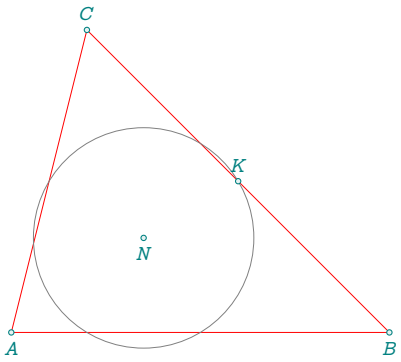
- The feet H_A, H_B, H_C of the altitudes from each vertex to the opposite side;
- The midpoints M_A, M_B, M_C of the sides of the triangle;
- The midpoints E_A, E_B, E_C of the segments joining each vertex to the orthocenter H (called the *Euler points*).

This circle was described by `Euler` in 1765 and further studied by `Feuerbach`, who proved that it is tangent to the triangle's incircle and excircles.

There are two ways to define this circle in the package:

- By computing its center using the attribute **`T.eulercenter`**, then using a midpoint to define the radius;
- Or directly, using the method **`euler_circle()`**, which returns a **`circle`** object representing the nine-point circle.

Weisstein, Eric W. "Nine-Point Circle." MathWorld




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  C.euler = T.ABC:euler_circle()
  z.N, z.K = C.euler:get()}
\begin{center}
  \begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawPolygons[red](A,B,C)
   \tkzDrawCircle(N,K)
   \tkzDrawPoints(A,B,C,N,K)
   \tkzLabelPoints(A,B,N)
   \tkzLabelPoints[above](C,K)
  \end{tikzpicture}
\end{center}
```

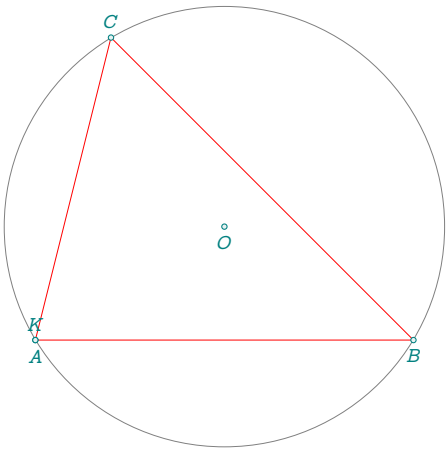
14.8.2. Method circum_circle()

The **circumscribed circle** (or **circumcircle**) of a triangle is the unique circle that passes through its three vertices.

There are two ways to obtain it:

- If you only need the *center*, use the attribute **`T.circumcenter`**.
- If you need the *entire circle object* (center and a point on the circle), use the method **`circum_circle()`**, which returns a **`circle`**.

This method is useful if the circle needs to be reused in further constructions, such as drawing or testing tangency.




```
\directlua{
init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  C.circum = T.ABC:circum_circle()
  z.O,
  z.K = C.circum:get()}
\begin{center}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPolygons[red](A,B,C)
 \tkzDrawCircle(O,K)
 \tkzDrawPoints(A,B,C,O,K)
 \tkzLabelPoints(A,B,O)
 \tkzLabelPoints[above](C,K)
\end{tikzpicture}
\end{center}
```

14.8.3. Method in_circle()

The **incircle** of a triangle is the unique circle that is *tangent to all three sides* of the triangle and lies entirely within it. This circle is also known as the *inscribed circle*.

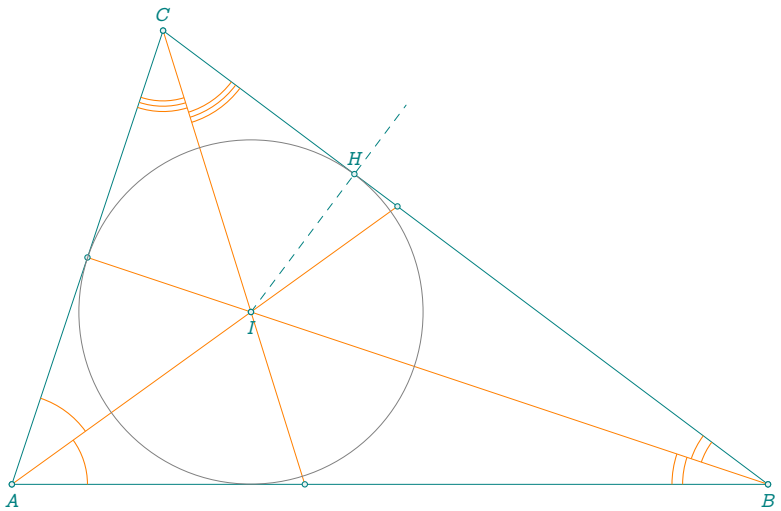
Its properties are as follows:

- The **center** I of the incircle, called the *incenter*, is the point of intersection of the three internal **angle bisectors**.
- The **radius** of the circle is called the *inradius*.
- The points of tangency M_A, M_B, and M_C of the incircle with the triangle's sides form the so-called **contact triangle**.

Geometrically, the contact triangle can also be seen as the **pedal triangle** of the incenter (see Section 14.8.9) or as the **tangential triangle** (see Section 14.9.1).

The method `in_circle()` returns a **`circle`** object representing the incircle.

Weisstein, Eric W. "Incircle." MathWorld



```
\directlua{
init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  z.E = T.ABC:bisector().pb
  z.F = T.ABC:bisector(1).pb
  z.G = T.ABC:bisector(2).pb
  C.IH = T.ABC:in_circle()
  z.I, z.H = C.IH:get()}
```

```
\begin{tikzpicture}%
 [new/.style ={color = orange },
  one/.style = { new,/tkzmkangle/size=.5 },
  two/.style = { new,/tkzmkangle/size=.6 },
  l/.style = { /tkzmkangle/arc=l },
  ll/.style = { /tkzmkangle/arc=ll },
  lll/.style = { /tkzmkangle/arc=lll }]
\tkzGetNodes
\tkzDrawPolygon(A,B,C)
\tkzDrawSegments[new](A,E B,F C,G)
\tkzDrawSegments[dashed,add=0 and .5](I,H)
\tkzDrawPoints(A,B,C,E,F,G,I)
\tkzDrawCircle(I,H)
\tkzDrawPoints(I,A,B,C,H)
\begin{scope}[one]
  \tkzMarkAngles[l](B,A,E)
  \tkzMarkAngles[ll](C,B,F)
  \tkzMarkAngles[lll](A,C,G)
\end{scope}
\begin{scope}[two]
  \tkzMarkAngles[l](E,A,C)
  \tkzMarkAngles[ll](F,B,A)
  \tkzMarkAngles[lll](G,C,B)
\end{scope}
\tkzLabelPoints(A,B,I)
\tkzLabelPoints[above](C,H)
\end{tikzpicture}
```

14.8.4. Method ex_circle(arg)

An **excircle** (or *escribed circle*) of a triangle is a circle that lies outside the triangle and is tangent to one side and to the extensions of the two others.

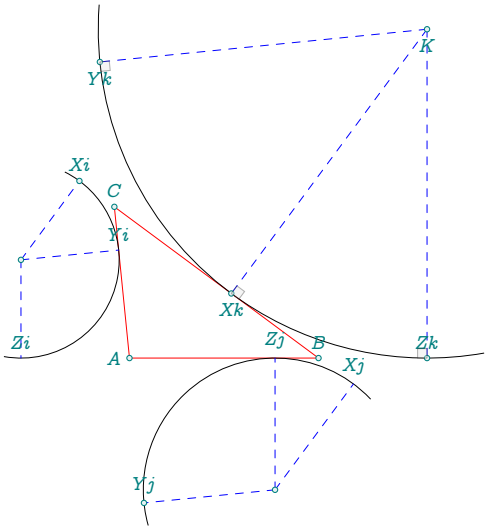
For each vertex of the triangle, there exists one such circle:

- The excircle opposite vertex A is tangent to side BC and the extensions of sides AB and AC;
- The corresponding **excenter** lies at the intersection of two *external* angle bisectors and one *internal* bisector.

The method **ex_circle(arg)** returns the excircle associated with the first vertex of the list obtained by performing a cyclic permutation of (A,B,C) arg times.

- arg = 0 (or nil) corresponds to vertex A,
- arg = 1 corresponds to vertex B,
- arg = 2 corresponds to vertex C.

The argument arg may also be the vertex itself (e.g., **z.A**, **z.B**, or **z.C**).




```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(-.4, 4)
T.ABC = triangle(z.A, z.B, z.C)
z.I,_ = T.ABC:ex_circle():get()
z.J,_ = T.ABC:ex_circle(1):get()
z.K,_ = T.ABC:ex_circle(2):get()
z.Xk,
z.Yk,
z.Zk = T.ABC:projection(z.K)
z.Xi,
z.Yi,
z.Zi = T.ABC:projection(z.I)
z.Xj,
z.Yj,
z.Zj = T.ABC:projection(z.J)}
```

14.8.5. Method spieker_circle()

In triangle geometry, the **Spieker circle** is defined as the **incircle of the medial triangle** of a reference triangle ABC.

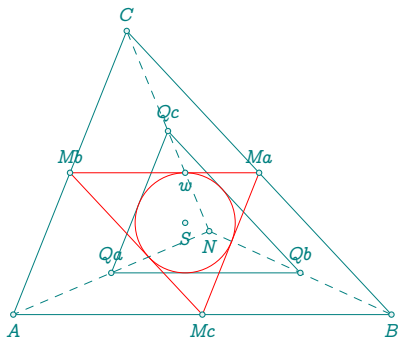
Its center is known as the **Spieker center**, which is also the center of the *radical circle* of the three *excircles* of triangle ABC.

To construct the Spieker circle:

- First form the *medial triangle* whose vertices are the midpoints of the sides of ABC;
- Then construct the incircle of this medial triangle.

The method `spieker_circle()` returns a **`circle`** object representing this circle.

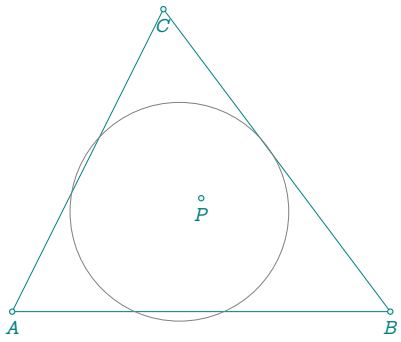
Weisstein, Eric W. "Spieker Circle." MathWorld




```
\directlua{
   init_elements()
   z.A = point(1, 1)
   z.B = point(5, 1)
   z.C = point(2.2, 4)
   T.ABC = triangle(z.A, z.B, z.C)
   C.first_lemoine = T.ABC:spieker_circle()
   z.S, z.w = C.first_lemoine:get()
   z.Ma, z.Mb, z.Mc = T.ABC:medial():get()
   z.N = T.ABC:nagel_point()
   z.Qa = tkz.midpoint(z.A, z.N)
   z.Qb = tkz.midpoint(z.B, z.N)
   z.Qc = tkz.midpoint(z.C, z.N)}
```

14.8.6. Method cevian_circle(pt)

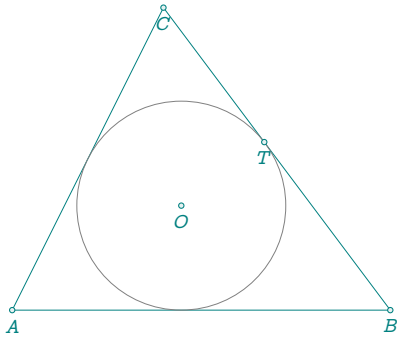
See [14.9.8]
The **Cevian circle** of a point P with respect to triangle ABC is the circle passing through the three points where the cevians AP, BP, and CP intersect the opposite sides (or their extensions). It is a special case of a pedal circle and is closely related to triangle center constructions.
The method `cevian_circle(pt)` returns the circle passing through these three intersection points.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.P = point(2.5, 1.5)
  C.cev = T.ABC:cevian_circle(z.P)
  z.w,z.t = C.cev:get()}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircle(w,t)
  \tkzDrawPoints(A,B,C,P)
  \tkzLabelPoints(A,B,C,P)
\end{tikzpicture}
```

14.8.7. Method symmedial_circle()

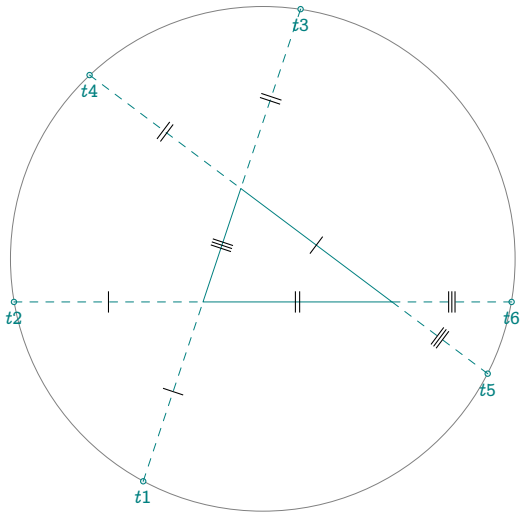
The symmedial circle is the circumcircle of the symmedial triangle.




```
\directlua{
  init_elements()
  z.A = point(0,0)
  z.B = point(5,0)
  z.C = point(2,4)
  T.ABC = triangle(z.A,z.B,z.C)
  C.sym = T.ABC:symmedial_circle()
  z.O,z.T = C.sym:get()}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircle(O,T)
  \tkzDrawPoints(A,B,C,O,T)
  \tkzLabelPoints(A,B,C,O,T)
\end{tikzpicture}
```

14.8.8. Methods conway_points() and conway_circle()

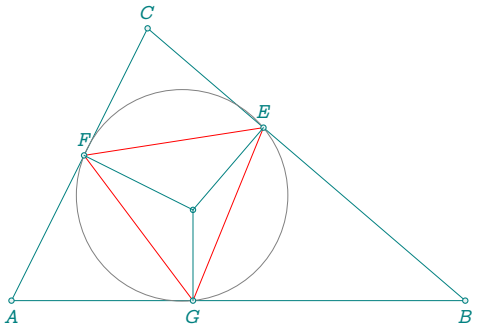
In plane geometry, Conway's circle theorem states that when the sides meeting at each vertex of a triangle are extended by the length of the opposite side, the six endpoints of the three resulting line segments lie on a circle whose centre is the centre of incenter of the triangle.
The method **conway_points()** creates the six points as defined by Conway's theorem and constructs the circle that passes through them.




```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.C = point(5, 0)
 z.B = point(1, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 C.conway = T.ABC:conway_circle()
 z.w, z.t = C.conway:get()
 z.t1, z.t2, z.t3,
 z.t4, z.t5,
  z.t6= T.ABC:conway_points()}
\begin{tikzpicture}[ scale = .5]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircles(w,t)
  \tkzDrawPoints(t1,t2,t3,t4,t5,t6)
  \tkzLabelPoints(t1,t2,t3,t4,t5,t6)
  \tkzDrawSegments[dashed](t1,A t2,A t3,B)
    \tkzDrawSegments[dashed](t4,B t5,C t6,C)
  \tkzMarkSegments(B,C t1,A t2,A)
  \tkzMarkSegments[mark=||](A,C t3,B t4,B)
  \tkzMarkSegments[mark=|||](A,B t5,C t6,C)
\end{tikzpicture}
```

14.8.9. Methods pedal() and pedal_circle()

Given a point P, the pedal triangle of P is the triangle whose polygon vertices are the feet of the perpendiculars from P to the side lines.

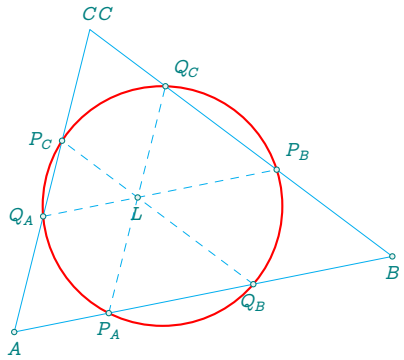



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1.5, 3)
  z.O = point(2, 1)
  T.ABC = triangle(z.A, z.B, z.C)
  T.pedal = T.ABC:pedal(z.O)
  z.E, z.F, z.G = T.pedal:get()
  C.pedal = T.ABC:pedal_circle(z.O)
  z.w = C.pedal.center
  z.T = C.pedal.through}
```

14.8.10. Method first_lemoine_circle()

This method constructs the *first Lemoine circle* of a triangle.
Through the **symmedian point** L (also called the **Lemoine point**) of triangle ABC, draw three lines parallel to each side of the triangle:
The lines $(P_AQ_A) \parallel (BC)$, $(P_BQ_B) \parallel (AC)$, and $(P_CQ_C) \parallel (AB)$ pass through the Lemoine point L. These lines intersect the sides of the triangle in six points that lie on a common circle: the *first Lemoine circle*.
Each of these lines intersects the triangle's sides in two points. The six points thus obtained lie on a same circle, known as the *first Lemoine circle*. Its center is L, and it is a special case of a circle associated with symmedians.

Weisstein, Eric W. "First Lemoine Circle." From MathWorld—A Wolfram Web Resource.

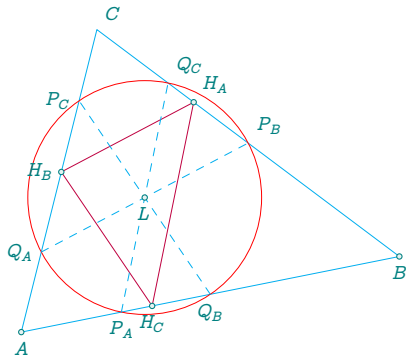



```
\directlua{
  z.A  = point(0, 0)
  z.B  = point(5, 1)
  z.C  = point(1, 4)
  T.ABC  = triangle(z.A, z.B, z.C)
  z.L = T.ABC:lemoine_point()
  C.flemoine  = T.ABC:first_lemoine_circle()
  z.w, z.t = C.flemoine:get()
  z.Q_A, z.Q_B, z.Q_C,
  z.P_A, z.P_B,
  z.P_C = T.ABC:first_lemoine_points()}
```

14.8.11. Method second_lemoine_circle()

This method constructs the *second Lemoine circle* of a triangle.
Through the **symmedian point** L of triangle ABC, draw lines parallel to the sides of the *orthic triangle*. These lines intersect the sides of triangle ABC in six points. All six points lie on the same circle, called the *second Lemoine circle*, whose center is also the point L.

Weisstein, Eric W. "Second Lemoine Circle." From MathWorld—A Wolfram Web Resource.

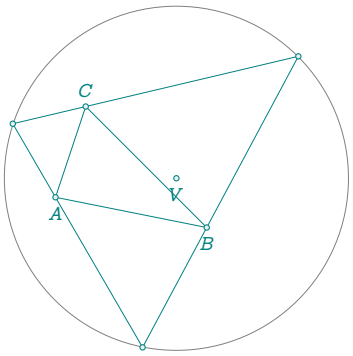



```
\directlua{
  init_elements()
  z.A  = point(0, 0)
  z.B  = point(5, 1)
  z.C  = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.L = T.ABC:lemoine_point()
  C.slemoine  = T.ABC:second_lemoine_circle()
  z.w, z.t = C.slemoine:get()
  T.orthic = T.ABC:orthic()
  z.H_A, z.H_B, z.H_C = T.orthic:get()
  z.Q_A, z.Q_B, z.Q_C,
  z.P_A, z.P_B, z.P_C = T.ABC:second_lemoine_points()}
```

14.8.12. Method bevan_circle()

This method constructs the *Bevan circle*, also known as the *excentral circle*.
The Bevan circle is the *circumcircle* of the excentral triangle of the reference triangle ABC, i.e., the circle passing through the three *excenters* of ABC. Its center is known as the *Bevan point*, which is therefore the **circumcenter** of the excentral triangle.

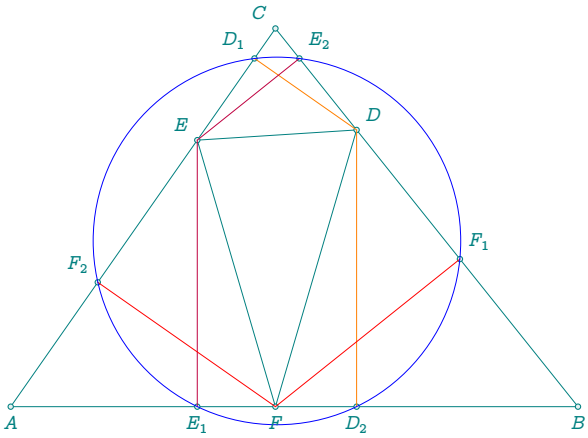
Weisstein, Eric W. "Bevan Circle." From MathWorld—A Wolfram Web Resource.




```
\directlua{
    init_elements()
    z.A = point(1, 1)
    z.B = point(6, 0)
    z.C = point(2, 4)
    T.ABC = triangle(z.A, z.B, z.C)
    z.Ea,z.Eb,z.Ec = T.ABC:excentral():get()
    C.bevan = T.ABC:bevan_circle()
    z.V, z.t = C.bevan:get()}
\begin{center}
      \begin{tikzpicture}[scale =.4]
      \tkzGetNodes
      \tkzDrawPolygons(A,B,C Ea,Eb,Ec)
      \tkzDrawCircle(V,t)
      \tkzDrawPoints(A,B,C,V,Ea,Eb,Ec)
      \tkzLabelPoints(A,B,V)
      \tkzLabelPoints[above](C)
  \end{tikzpicture}
    \end{center}
```

14.8.13. Method taylor_circle()

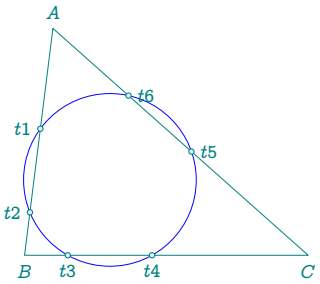
The six projections of the feet of the heights of a triangle onto the adjacent sides are cocylic.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(6, 0)
  z.C = point(2.8, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  T.DEF = T.ABC:orthic()
  z.D, z.E, z.F = T.DEF:get()
  z.D_1,
  z.D_2,
  z.E_1,
  z.E_2,
  z.F_1,
  z.F_2 = T.ABC:taylor_points()
  C.taylor = T.ABC:taylor_circle()
  z.w, z.t = C.taylor:get()}
  \begin{center}
\begin{tikzpicture}[scale = 1.25]
  \tkzGetNodes
  \tkzDrawPolygons(A,B,C D,E,F)
  \tkzDrawPoints(A,B,...,F)
  \tkzDrawPoints(D_1,D_2,E_1,E_2,F_1,F_2)
  \tkzDrawSegments[orange](D,D_1 D,D_2)
  \tkzDrawSegments[purple](E,E_1 E,E_2)
  \tkzDrawSegments[red](F,F_1 F,F_2)
  \tkzDrawCircles[blue](w,t)
  \tkzLabelPoints(A,B,F,E_1,D_2)
  \tkzLabelPoints[above left](F_2,E,C,D_1)
  \tkzLabelPoints[above right](E_2,D,F_1)
\end{tikzpicture}
\end{center}
```

14.8.14. Method adams_circle()

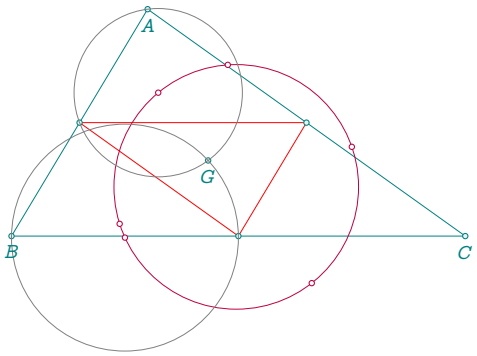
See [14.6.25]




```
\directlua{
  z.A = point(0.5, 4)
  z.B = point(0, 0)
  z.C = point(5, 0)
  T.ABC = triangle(z.A, z.B, z.C)
  C.a  = T.ABC:adams_circle()
  z.w, z.t = C.a:get()
  z.t1, z.t2, z.t3,
  z.t4, z.t5,
   z.t6 = T.ABC:adams_points()}
\begin{center}
   \begin{tikzpicture}[scale = .75]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircles[blue](w,t)
  \tkzDrawPoints(t1,t2,t3,t4,t5,t6)
  \tkzLabelPoints(t3,t4)
  \tkzLabelPoints(B,C)
  \tkzLabelPoints[above](A)
  \tkzLabelPoints[left](t1,t2)
  \tkzLabelPoints[right](t5,t6)
\end{tikzpicture}
\end{center}
```

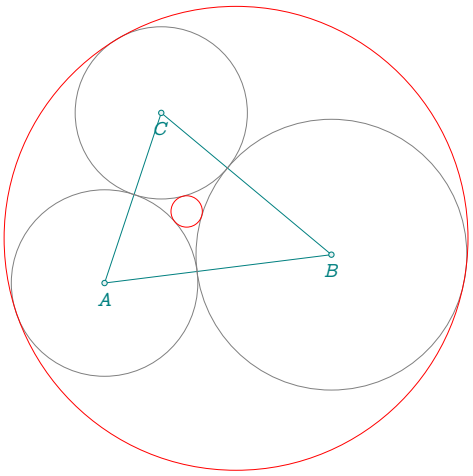
14.8.15. Method lamoen_circle

This method returns the six *Lamoen points* associated with a triangle.
By dividing a triangle using its three medians, six smaller triangles are formed. Remarkably, the *circumcenters* of these six triangles are *concyclic*, meaning they lie on a common circle. This circle is known as the *van Lamoen circle*.
The six circumcenters themselves are called the *Lamoen points*. See an example on the document **Euclidean Geometry** presented in altermundus.fr.




```
\directlua{
 init_elements()
 z.A = point(1.2, 2)
 z.B = point(0, 0)
 z.C = point(4, 0)
 T.ABC = triangle(z.A, z.B, z.C)
 T.med = T.ABC:medial()
 z.G = T.ABC.centroid
 z.ma,z.mb,z.mc = T.med:get()
 z.Oab, z.Oac,
 z.Oba, z.Obc,
 z.Oca, z.Ocb = T.ABC:lamoen_points()
 C.lamoen = T.ABC:lamoen_circle()
 z.w = C.lamoen.center}
\begin{center}
 \begin{tikzpicture}[scale = 1.5]
 \tkzGetNodes
 \tkzDrawPolygon(A,B,C)
 \tkzDrawPolygon[red](ma,mb,mc)
 \tkzDrawPoints(A,B,C,ma,mb,mc,G)
 \tkzDrawCircle[purple](w,Oab)
 \tkzDrawCircles(Oab,A Oac,B)
 \tkzDrawPoints[size=2,purple,
  fill=white](Oab,Oac,Oba,Obc,Oca,Ocb)
 \tkzLabelPoints(A,B,C,G)
\end{tikzpicture}
 \end{center}
```

14.8.16. Method soddy_circle()

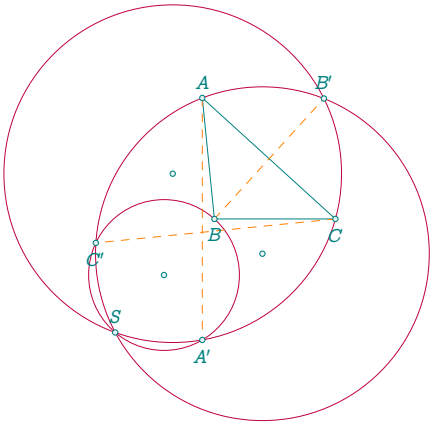



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0.5)
  z.C = point(1, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  local ra = (T.ABC.b + T.ABC.c -T.ABC.a)/2
  local rb = (T.ABC.c + T.ABC.a -T.ABC.b)/2
  local rc = (T.ABC.b + T.ABC.a -T.ABC.c)/2
  C.a = circle(through(z.A, ra))
  C.b = circle(through(z.B, rb))
  C.c = circle(through(z.C, rc))
  z.ta = C.a.through
  z.tb = C.b.through
  z.tc = C.c.through
  C.i = T.ABC:soddy_circle()
  z.w, z.t = C.i:get()
  C.o = T.ABC:soddy_circle("outer")
  z.wp,z.tp = C.o:get()}
\begin{tikzpicture}[scale=.75]
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircles(A,ta B,tb C,tc)
  \tkzDrawCircles[red](w,t w',t')
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B,C)
\end{tikzpicture}
```

14.8.17. Method yiu_circles()

The *Yiu A-circle* of a reference triangle ABC is the circle passing through vertex A and the reflections of vertices B and C across the opposite sides AC and AB, respectively.
The *Yiu B- and C-circles* are defined analogously. These three circles intersect at a unique point — their common *radical center*. However, they do not admit a common radical circle.

Weisstein, Eric W. "Yiu Circles." From MathWorld—A Wolfram Web Resource.

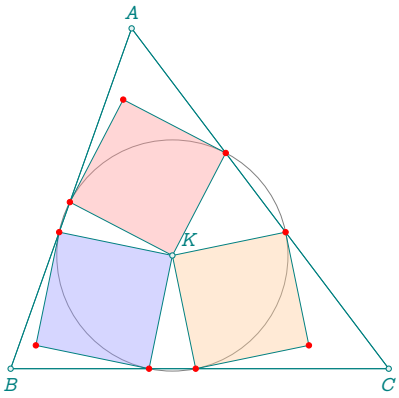



```
\directlua{
z.A = point(-.2, 2)
z.B = point(0, 0)
z.C = point(2, 0)
T.ABC = triangle(z.A,z.B,z.C)
z.Ap,
z.Bp,
z.Cp = T.ABC:reflection():get()
z.O_A, z.O_B, z.O_C = T.ABC:yiu_centers()
C.A,
C.B,
C.C = T.ABC:yiu_circles()
x,y = intersection(C.A,C.B)
if C.C:in_out(x) then  z.S = x
  else  z.S = y end}
\begin{center}
   \begin{tikzpicture}[scale=.8]
\tkzGetNodes
\tkzInit[xmin=-4,xmax=4,ymin=-4,ymax=4]
\tkzClip
\tkzDrawPolygon(A,B,C)
\tkzDrawCircles[purple](O_A,A O_B,B O_C,C)
\tkzDrawSegments[orange,
                 dashed](A,A' B,B' C,C')
\tkzDrawPoints(A,B,C,A',B',C',O_A,O_B,O_C,S)
\tkzLabelPoints(B,C,A',C')
\tkzLabelPoints[above](A,B',S)
\end{tikzpicture}
   \end{center}
```

14.8.18. Method kenmotu_circle()

The *Kenmotu circle* is the circle passing through the six contact points of the three congruent squares constructed on the sides of a triangle. These squares are used in the construction of the *Kenmotu point*, also known as the *congruent squares point* (see Kimberling center $X_{(371)}$).
The Kenmotu circle contains all six points of tangency between the triangle sides and the inscribed squares.

Weisstein, Eric W. "Kenmotu Circle." From MathWorld—A Wolfram Web Resource.




```
\directlua{
init_elements()
z.A = point(1.6, 4.5)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A, z.B, z.C)
local ken_circle = T.ABC:kenmotu_circle()
z.K, z.T = ken_circle.center, ken_circle.through
z.p1,z.p2,z.p3,z.p4,z.p5,z.p6 = T.ABC:kenmotu_points()
z.p7 = z.p1 + z.p6 - z.K
z.p8 = z.p2 + z.p3 - z.K
z.p9 = z.p4 + z.p5 - z.K}
\begin{center}
\begin{tikzpicture}[scale = 1]
\tkzGetNodes
\tkzDrawCircles(K,T)
\tkzFillPolygon[opacity=.4,red!40](K,p1,p7,p6)
\tkzFillPolygon[opacity=.4,blue!40](K,p2,p8,p3)
\tkzFillPolygon[opacity=.4,orange!40](K,p4,p9,p5)
\tkzDrawPolygons(A,B,C)
\tkzDrawPolygons(K,p1,p7,p6 K,p2,p8,p3 K,p4,p9,p5)
\tkzDrawPolygons(A,B,C)
\tkzDrawPoints(A,B,C,K)
\tkzLabelPoints(B,C)
\tkzDrawPoints[red](p1,p2,p3,p4,p5,p6,p7,p8,p9)
\tkzLabelPoints[above](A)
\tkzLabelPoints[above right](K)
\end{tikzpicture}
\end{center}
```

14.8.19. Method thebault or c_c

The c_c (or thebault) method constructs a circle tangent to two sides of a triangle at a chosen vertex, and tangent to a given circle passing through the two remaining vertices.

Name and purpose:
Circle tangent to the two sides issuing from a vertex p and tangent to the circle centered at C passing through the other two vertices of the triangle. Alias: **triangle.thebault = triangle.c_c**.
Given a non-degenerate triangle ABC and a chosen vertex $p \in \{A,B,C\}$, this method constructs the unique circle $\mathcal{T}$ that

- is tangent to the two sides adjacent to p (i.e. the lines through p that contain the two incident edges of the triangle), and
- is tangent to the circle with center C and radius equal to the distance from C to each of the two vertices *other than* p.

Signature:

```
circle triangle:c_c(p, C)
```

Parameters:

p The chosen vertex of the triangle (**triangle.pa**, **.pb** or **.pc**). The method internally detects which vertex you passed.

C The center of the reference circle through the other two vertices (i.e. if $p = A$, then CB = CC = C\!B and the circle $\mathcal{C}(C,\text{CB})$ passes through B and C). In practice, C must lie on the perpendicular bisector of the segment joining those two vertices.

Return value:
A **circle** object $\mathcal{T}$ tangent to the two sides adjacent to p and tangent to the circle $\mathcal{C}(C,\cdot)$ through the other two vertices.
Preconditions & diagnostics :

– The triangle must be non-degenerate (three non-collinear points).

– The point **p** must coincide with one of the triangle's vertices; otherwise the internal index resolution will be inconsistent.

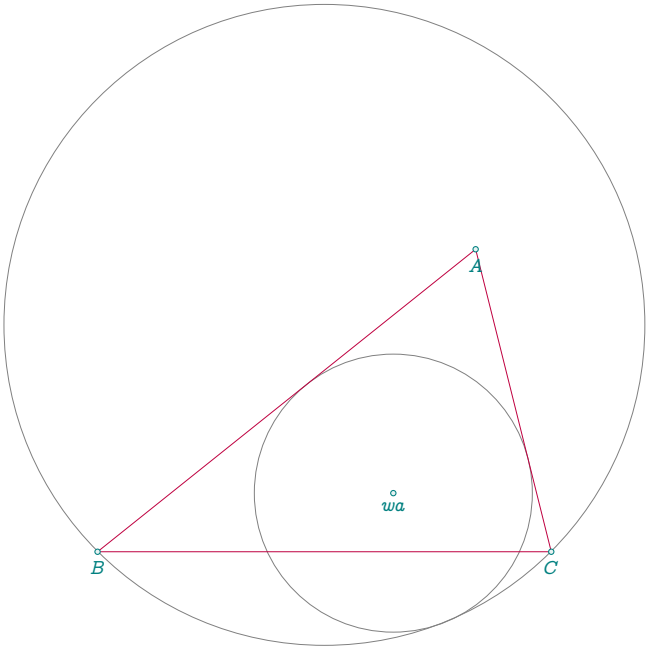
– The point **C** must be equidistant from the two vertices other than p (i.e. it lies on their perpendicular bisector). If not, the "reference circle" is ill-defined for this problem and the construction may fail or produce an irrelevant circle.

Notes:

– This configuration is a classical variant often linked to Thébault-type constructions; for convenience an alias **triangle.thebault** is provided.

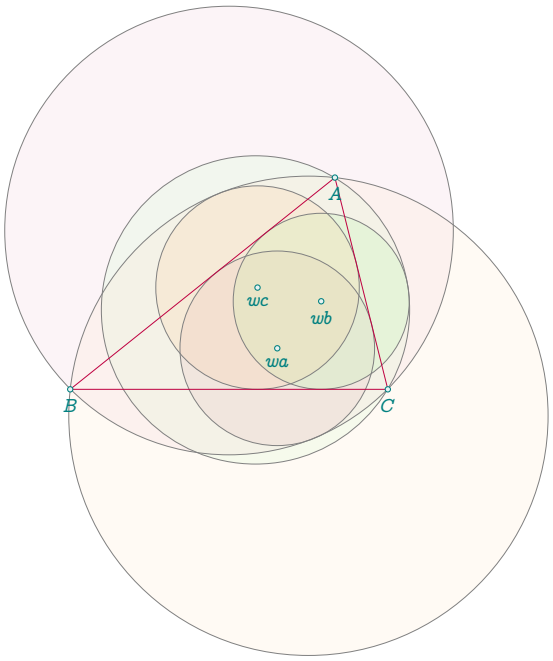
– The circle returned is tangent to the *lines* supporting the two sides adjacent to p. If you need the segment contact points explicitly, intersect the result with those lines and/or use orthogonal projections.

Example usage:




```
\directlua{
 init_elements()
 z.A = point(2, 4)
 z.B = point(-3, 0)
 z.C = point(3, 0)
 T.ABC = triangle(z.A, z.B, z.C)
 L.med = T.ABC:mediator(z.A)
 z.Oa = L.med:point(1)
 C.main = circle(z.Oa, z.C)
 z.wa,
 z.Ea = T.ABC:thebault(z.A, C.main):get()
}
\begin{center}
\begin{tikzpicture}%[ scale = 1]
  \tkzGetNodes
  \tkzDrawCircles(Oa,C)
  \tkzDrawCircles(wa,Ea)
  \tkzDrawPolygon[color = purple](A,B,C)
  \tkzDrawPoints(A,B,C,wa)
  \tkzLabelPoints(A,B,C,wa)
\end{tikzpicture}
\end{center}
```

Application:



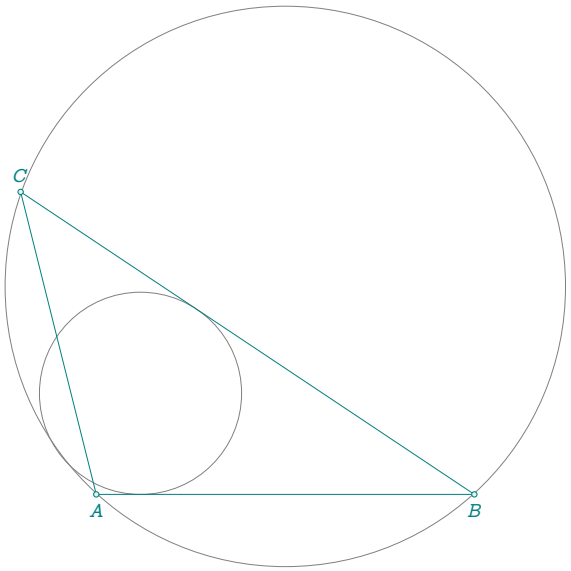

```
\directlua{
 init_elements()
 z.A = point(2, 4)
 z.B = point(-3, 0)
 z.C = point(3, 0)
 T.ABC = triangle(z.A, z.B, z.C)
 L.med = T.ABC:mediator(z.A)
 z.Oa = L.med:point(1)
 C.main = circle(z.Oa, z.C)
 z.wa,
 z.Ea = T.ABC:thebault(z.A, C.main):get()
 L.med = T.ABC:mediator(z.B)
 z.Ob = L.med:point(1)
 C.main = circle(z.Ob, z.C)
 z.wb,
 z.Eb = T.ABC:thebault(z.B, C.main):get()
 L.med = T.ABC:mediator(z.C)
 z.Oc = L.med:point(1)
 C.main = circle(z.Oc, z.A)
 z.wc,
 z.Ec = T.ABC:thebault(z.C, C.main):get()
 }
\begin{center}
 \begin{tikzpicture}[scale = .7]
  \tkzGetNodes
  \tkzFillCircles[purple!20,opacity=.2](Oa,C)
  \tkzFillCircles[purple!40,opacity=.2](wa,Ea)
  \tkzFillCircles[green!20,opacity=.2](Ob,C)
  \tkzFillCircles[green!40,opacity=.2](wb,Eb)
  \tkzFillCircles[orange!20,opacity=.2](Oc,A)
  \tkzFillCircles[orange!40,opacity=.2](wc,Ec)
  \tkzDrawCircles(Oa,C Ob,C Oc,A)
  \tkzDrawCircles(wa,Ea wb,Eb wc,Ec)
  \tkzDrawPolygon[color = purple](A,B,C)
  \tkzDrawPoints(A,B,C,wa,wb,wc)
  \tkzLabelPoints(A,B,C,wa,wb,wc)
\end{tikzpicture}
 \end{center}
```

14.8.20. Method mixtilinear_incircle(arg)

A circle that in internally tangent to two sides of a triangle and to the circumcircle is called a mixtilinear incircle. There are three mixtilinear incircles, one corresponding to each angle of the triangle.

MathWorld — van Lamoen, Floor. "Mixtilinear Incircles.

The argument is either one of the vertices of the triangle or an integer between 0 and 2.




```
\directlua{
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(-1, 4)
T.ABC = triangle(z.A, z.B, z.C)
C.mix = T.ABC:mixtilinear_incircle(z.B)
z.w, z.t = C.mix:get()
z.O = T.ABC.circumcenter}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircles(w,t O,A)
\tkzDrawPolygon(A,B,C)
\tkzDrawPoints(A,B,C)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C)
\end{tikzpicture}
```

14.8.21. Method three_tangent_circles()

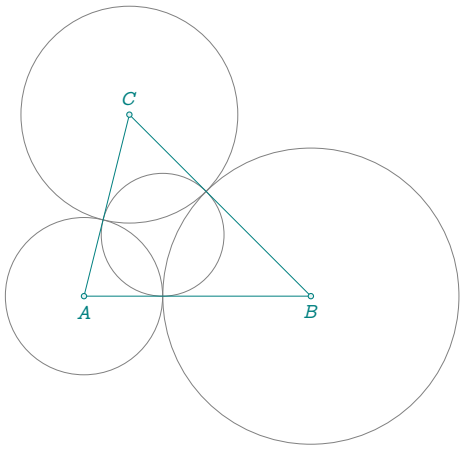
Any three points can serve as the centers of three mutually tangent circles. If we connect these centers, we obtain a triangle. The angle bisectors of this triangle meet at a single point called the *incenter*. From this point, perpendiculars dropped to the three sides determine the points of tangency with an inscribed circle (the *incircle*).

This incircle touches each side at exactly one point, and those points also serve as the common tangency points with the three outer circles. Each of these outer circles is defined by one vertex of the triangle (its center) and the tangency point opposite to it.

The method **three_tangent_circles()** implements this geometric construction as follows:

- it computes the incenter of the triangle,
- finds the perpendicular projections of this incenter onto the three sides,
- and constructs the three outer circles, each passing through one vertex and tangent to the incircle.

Return value:
Three circle objects tangent to one another and to the triangle's sides.
Example usage:




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  C.A,
  C.B,
  C.C = T.ABC:three_tangent_circles()
  z.ta = C.A.through
  z.tb = C.B.through
  z.tc = C.C.through
  C.ins = T.ABC:in_circle()
  z.I, z.T = C.ins:get()}
\begin{center}
\begin{tikzpicture}[scale=.6]
\tkzGetNodes
\tkzDrawCircles(A,ta B,tb C,tc I,T)
\tkzDrawPolygon(A,B,C)
\tkzDrawPoints(A,B,C)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C)
\end{tikzpicture}
\end{center}
```

14.8.22. Method three_apollonius_circles()

This method constructs the three Apollonius circles of a non-degenerate triangle.

Geometric background: Given a triangle ABC with side lengths $a = BC$, $b = CA$, and $c = AB$, one can associate to each vertex an Apollonius circle, defined as the locus of points M for which the ratio of distances to the two opposite vertices is fixed (in terms of the side lengths of the triangle).
These three Apollonius circles have the remarkable property that they intersect in exactly two common points, which are the isodynamic points of the triangle (see Method **isodynamic_points()**).

Description. The method **three_apollonius_circles()** builds the three Apollonius circles associated with the triangle and returns them as circle objects.

Return values:

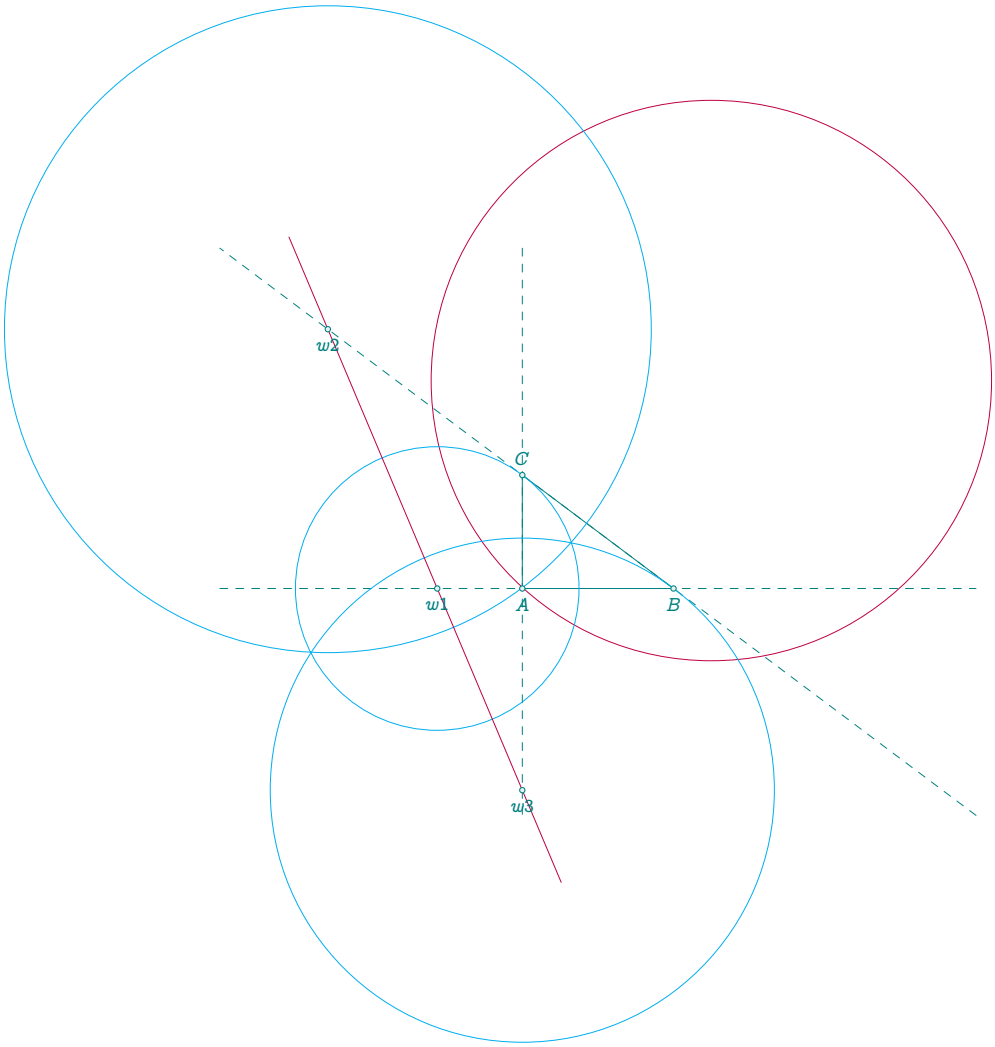
- **Ca** — Apollonius circle associated with side $a = BC$,
- **Cb** — Apollonius circle associated with side $b = CA$,
- **Cc** — Apollonius circle associated with side $c = AB$.

Example:

```
Ca, Cb, Cc = T.ABC:three_apollonius_circles()
```

Each of **Ca**, **Cb**, and **Cc** is a **circle** object that can be used in further constructions (for instance, to compute the isodynamic points from their intersections).

Application:


w2
C
w1
A
B
w3

```
\directlua{%
z.A = point (0, 0)
z.B = point (4, 0)
z.C = point (0, 3)
T.ABC = triangle (z.A,  z.B, z.C)
C.ab, C.bc,
C.ca = T.ABC:three_apollonius_circles()
z.w1, z.t1 = C.ab:get()
z.w2, z.t2 = C.bc:get()
z.w3, z.t3 = C.ca:get()}
\begin{center}
\begin{tikzpicture}[scale=.5]
 \tkzGetNodes
 \tkzDrawLines[add = 2 and 2,
         dashed](A,B B,C C,A)
 \tkzDrawPolygons(A,B,C)
 \tkzDrawCircle[purple](O,A)
 \tkzDrawCircles[cyan](w1,t1 w2,t2 w3,t3)
 \tkzDrawLines[purple](w3,w2)
 \tkzDrawPoints(A,B,C,w1,w2,w3)
 \tkzLabelPoints(A,B,w1,w2,w3)
  \tkzLabelPoints[above](C)
\end{tikzpicture}
\end{center}
```

14.8.23. Method apollonius_circle(side, EPS)

This method returns one of the three Apollonius circles of the triangle.

Description: Given a triangle ABC, the method **apollonius_circle(side)** constructs the Apollonius circle associated with the specified side:

- **"ab"** $\rightarrow$ Apollonius circle opposite vertex C,
- **"bc"** $\rightarrow$ Apollonius circle opposite vertex A,
- **"ca"** $\rightarrow$ Apollonius circle opposite vertex B.

The optional parameter **EPS** controls numerical tolerance for the underlying intersection or ratio calculations. If omitted, it defaults to **tkz.epsilon**.

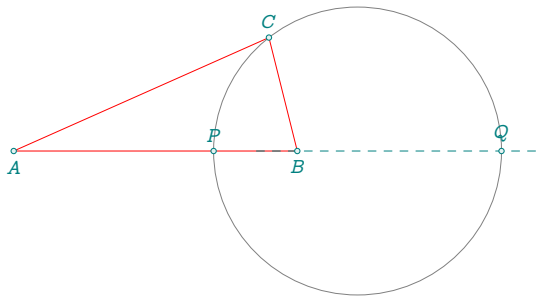
Return value: A single **circle** object corresponding to the chosen side.

Example:

```
Cc = T.ABC:apollonius_circle("ab")
```

This circle can be used independently or as part of the construction of the three Apollonius circles or the isodynamic points.

Application:




```
\directlua{
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(4.5, 2)
  T.ABC  = triangle: new(z.A, z.B, z.C)
  C.apo_ab  = T.ABC:apollonius_circle("ab")
  z.N,z.K =  C.apo_ab:get()
  z.P,
  z.Q = T.ABC:apollonius_points("ab")}
\begin{center}
\begin{tikzpicture}[scale=.75]
 \tkzGetNodes
 \tkzDrawPolygons[red](A,B,C)
 \tkzDrawCircle(N,K)
 \tkzDrawLine[dashed](B,Q)
 \tkzDrawPoints(A,B,C,P,Q)
 \tkzLabelPoints(A,B)
 \tkzLabelPoints[above](C,P,Q)
\end{tikzpicture}
\end{center}
```

14.8.24. Method feuerbach_apollonius_k181(<EPS<)

This method constructs a circle through three points obtained from simple line–line intersections involving:

- the three Feuerbach points E_a, E_b, E_c,
- the Apollonius point K_{181} of the triangle,
- the Spieker center S.

`Construction idea:` For each Feuerbach point E_a, consider the lines (A, K_{181}) and (S, E_a). Their intersection defines a point x_a. Repeating the process with B, E_b and C, E_c gives three points x_a, x_b, x_c which always lie on a unique circle.
This circle is the *Apollonius–K181 Feuerbach circle.*
`Usage:`

```
local C = T.ABC:feuerbach_apollonius_k181()
```

`Return:` A **circle** object: the circumcircle of the three intersection points x_a, x_b, x_c.

14.8.25. Method feuerbach_apollonius(EPS)

This method constructs a remarkable Apollonius-type circle associated with the triangle. The construction is based on three Feuerbach points and an inversion in a circle orthogonal to the excircle touching point.
`Definition:` Let E_a, E_b, E_c be the three Feuerbach contact points of the triangle, and let J_a be the excenter opposite to vertex A. Let $C_{J_aE_a}$ be the circle with center J_a passing through E_a, and let S be the Spieker center. The circle $C_{J_aE_a}$ admits a unique orthogonal circle through S. Inverting the Euler circle in this orthogonal circle maps the three Feuerbach points to points x_a, x_b, x_c which lie on a common circle. This is the *Feuerbach–Apollonius circle.*

Usage:

```
local C = T.ABC:feuerbach_apollonius()
```

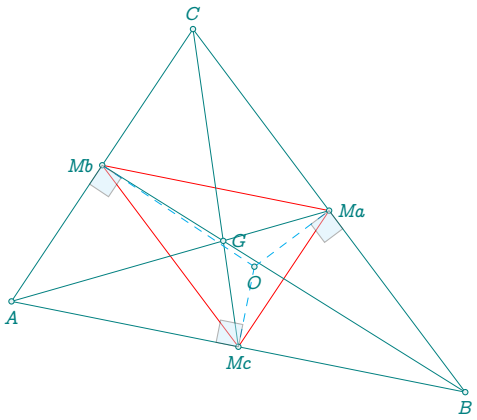
`Return:` A **circle** object: the circle passing through the three inverted Feuerbach points.

14.9. Returns a triangle

14.9.1. Method medial()

The *medial triangle* of a triangle ABC is the triangle formed by connecting the midpoints M_a, M_b, and M_c of the sides BC, AC, and AB respectively. This triangle is similar to the reference triangle and shares the same centroid.
The medial triangle is sometimes also called the *auxiliary triangle* Dixon 1991. It appears frequently in classical geometry and serves as a useful tool for constructions involving symmetry, similarity, and triangle centers.

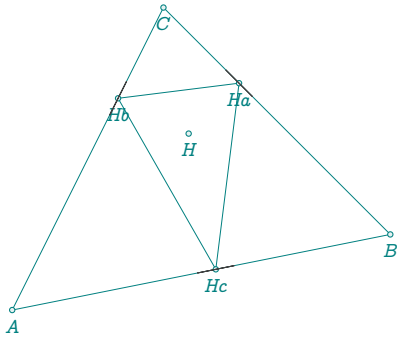
Weisstein, Eric W. "Medial Triangle." From MathWorld—A Wolfram Web Resource.




```
\directlua{
  init_elements()
  z.A = point(0, 1)
  z.B = point(5, 0)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  T.med = T.ABC:medial()
  z.Ma, z.Mb, z.Mc= T.med:get()
  z.G = T.ABC.centroid
  z.O = T.ABC.circumcenter}
```

14.9.2. Method orthic()

Given a triangle ABC, the triangle $H_AH_BH_C$ formed by connecting the feet of the perpendiculars dropped from each vertex to the opposite side is called the *orthic triangle*, or sometimes the *altitude triangle*.
The points H_A, H_B, and H_C are the feet of the altitudes from A, B, and C, respectively. The three altitudes (AH_A), (BH_B), and (CH_C) are concurrent at a single point: the *orthocenter* H of the triangle ABC.
This method returns the triangle $H_AH_BH_C$ as a new triangle object.



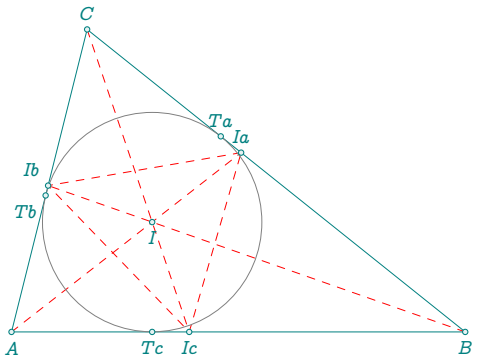

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 1)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  T.H = T.ABC:orthic()
  z.Ha, z.Hb, z.Hc = T.H:get()
  z.H = T.ABC.orthocenter
}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPolygon(Ha,Hb,Hc)
  \tkzDrawPoints(A,B,C,Ha,Hb,Hc,H)
  \tkzLabelPoints(A,B,C,Ha,Hb,Hc,H)
  \tkzMarkRightAngle(A,Hc,B)
  \tkzMarkRightAngle(B,Ha,C)
  \tkzMarkRightAngle(C,Hb,A)
\end{tikzpicture}
```

14.9.3. Method incentral()

The *incentral triangle* $I_aI_bI_c$ is the Cevian triangle of the reference triangle ABC with respect to its incenter I. That is, the vertices of the incentral triangle are the points where the internal angle bisectors of triangle ABC intersect the opposite sides.

This triangle is useful in many triangle center constructions, especially in connection with the incircle and the Gergonne point.

Weisstein, Eric W. "Incentral Triangle." From MathWorld–A Wolfram Web Resource.

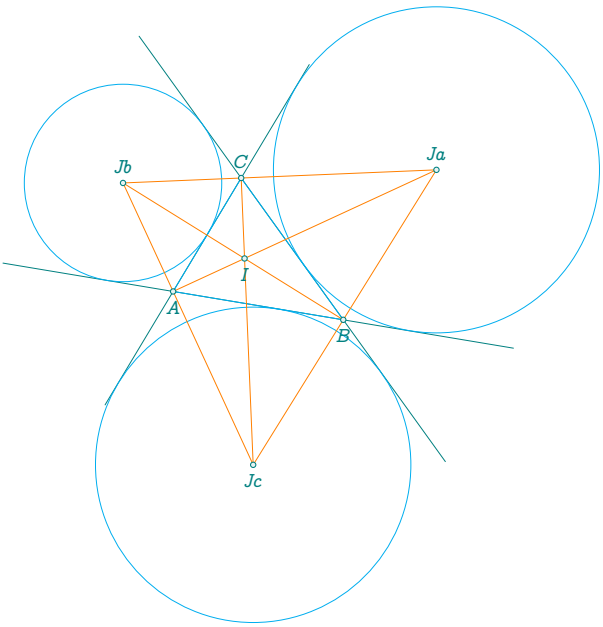



```
\directlua{
   init_elements()
   z.A = point(0, 0)
   z.B = point(6, 0)
   z.C = point(1, 4)
   T.ABC = triangle(z.A, z.B, z.C)
   z.I = T.ABC.incenter
   z.Ia, z.Ib,
   z.Ic = T.ABC:incentral():get()
   z.Ta, z.Tb,
   z.Tc = T.ABC:intouch():get()}
```

14.9.4. Method excentral()

The *excentral triangle*, also called the *tritangent triangle*, of a reference triangle ABC is the triangle $J_AJ_BJ_C$ whose vertices are the excenters of ABC. Each excenter is the intersection point of two external angle bisectors and one internal angle bisector of the triangle.
The excentral triangle plays an important role in triangle geometry, particularly in constructions involving the excircles, the Bevan circle, and certain triangle centers.

Weisstein, Eric W. "Excentral Triangle." From MathWorld–A Wolfram Web Resource.

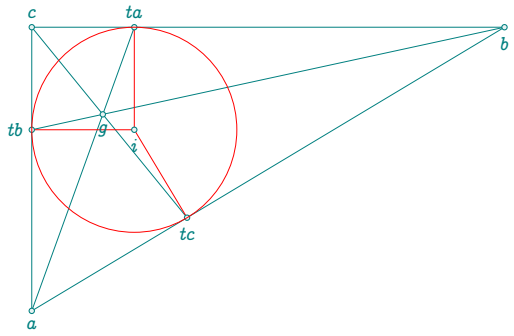



```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(3, -0.5)
 z.C = point(1.2, 2)
 T.ABC = triangle(z.A, z.B, z.C)
 T.exc = T.ABC:excentral()
 z.I = T.ABC.incenter
 z.Ja, z.Jb, z.Jc = T.exc:get()
 z.Xa, _, _ = T.ABC:projection(z.Ja)
 z.Xb, _, _ = T.ABC:projection(z.Jb)
 z.Xc, _, _ = T.ABC:projection(z.Jc)}
\begin{tikzpicture}[scale=.75]
  \tkzGetNodes
  \tkzDrawLines[add=1 and 1](A,B B,C C,A)
  \tkzDrawPolygon[cyan](A,B,C)
  \tkzDrawCircles[cyan](Ja,Xa Jb,Xb Jc,Xc)
  \tkzDrawPolygon[orange](Ja,Jb,Jc)
  \tkzDrawSegments[orange](A,Ja B,Jb C,Jc)
  \tkzDrawPoints(A,B,C,Ja,Jb,Jc,I)
  \tkzLabelPoints(A,B,Jc,I)
  \tkzLabelPoints[above](C,Ja,Jb)
\end{tikzpicture}
```

14.9.5. Method intouch()

The *contact triangle* of a triangle ABC, also known as the *intouch triangle*, is the triangle $t_at_bt_c$ formed by the points of tangency of the incircle of ABC with its three sides. These points are where the incircle touches BC, AC, and AB respectively.
The intouch triangle is useful in studying properties related to the incircle and its associated centers such as the Gergonne point.

Weisstein, Eric W. "Contact Triangle." From MathWorld–A Wolfram Web Resource.



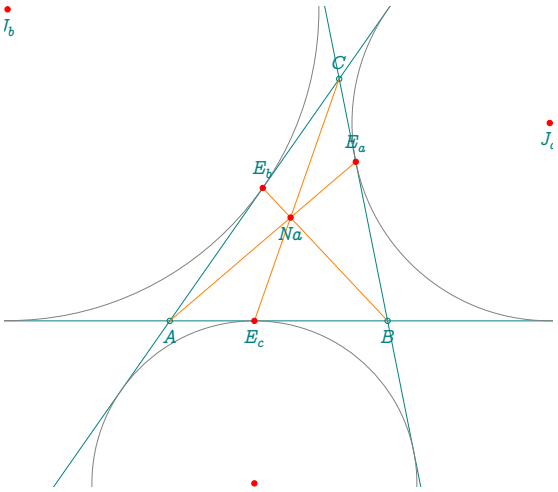

```
\directlua{
 init_elements()
 z.a = point(1, 0)
 z.b = point(6, 3)
 z.c = point(1, 3)
 T.abc = triangle(z.a, z.b, z.c)
 z.g = T.abc:gergonne_point ()
 z.i = T.abc.incenter
 z.ta, z.tb,
 z.tc = T.abc:intouch():get()}
```

14.9.6. Method extouch()

The *extouch triangle* of a triangle ABC is the triangle formed by the points of tangency of the triangle with its three *excircles*. Each excircle is tangent to one side of the triangle and the extensions of the other two. The points of tangency lie respectively on the sides BC, AC, and AB.
The extouch triangle is the Cevian triangle of the *Nagel point*, which is the point of concurrency of the segments joining the triangle's vertices to the points where the opposite excircles touch the triangle.

Weisstein, Eric W. "Extouch Triangle." From MathWorld–A Wolfram Web Resource.

```
z.E_a,z.E_b,z.E_c = T.ABC:cevian(z.Na):get()
```

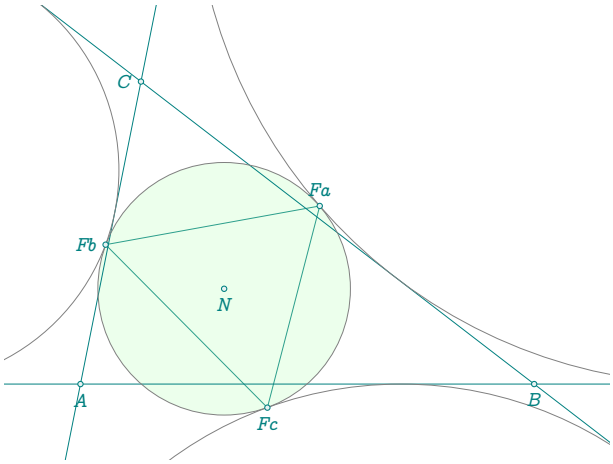



```
\directlua{
  z.A = point(0, 0)
  z.B = point(3.6, 0)
  z.C = point(2.8, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.Na = T.ABC:nagel_point()
  z.J_a,z.J_b,
  z.J_c = T.ABC:excentral():get()
  z.E_a,z.E_b,
  z.E_c = T.ABC:extouch():get()}
```

14.9.7. Method feuerbach()

The *Feuerbach triangle* of a reference triangle ABC is the triangle formed by the three points of tangency between the *nine-point circle* and the three *excircles* of ABC.
By *Feuerbach's theorem*, each excircle is tangent to the nine-point circle, and these three tangency points define the Feuerbach triangle.
The circumcenter of the Feuerbach triangle coincides with the nine-point center of triangle ABC.

Weisstein, Eric W. "Feuerbach Triangle." From MathWorld–A Wolfram Web Resource.

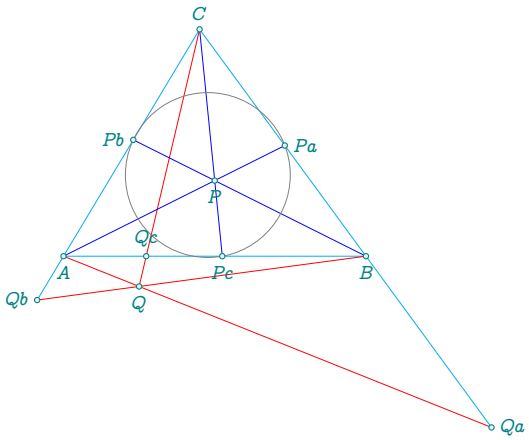



```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(0.8, 4)
 T.ABC = triangle(z.A, z.B, z.C)
  z.N = T.ABC.eulercenter
 z.Fa, z.Fb,
 z.Fc = T.ABC:feuerbach():get()
 z.Ja, z.Jb,
 z.Jc = T.ABC:excentral():get()}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzInit[xmin=-1,xmax=7,ymin=-1,ymax=5]
  \tkzClip
  \tkzDrawPoints(Ja,Jb,Jc)
  \tkzDrawPolygons(Fa,Fb,Fc)
  \tkzFillCircles[green!30,
                  opacity=.25](N,Fa)
  \tkzDrawLines[add=3 and 3](A,B A,C B,C)
  \tkzDrawCircles(Ja,Fa Jb,Fb Jc,Fc N,Fa)
  \tkzDrawPoints(A,B,C,Fa,Fb,Fc,N)
  \tkzLabelPoints(N,A,B,Fc)
  \tkzLabelPoints[above](Fa)
  \tkzLabelPoints[left](Fb,C)
\end{tikzpicture}
```

14.9.8. Method cevian()

A *Cevian* is a line segment that joins a vertex of a triangle to a point on the opposite side (or its extension). The condition for three Cevians to concur is given by *Ceva's Theorem*.
Given a point P in the interior of triangle ABC, the Cevians from each vertex through P intersect the opposite sides at points P_a, P_b, and P_c. The triangle $P_aP_bP_c$ is called the *Cevian triangle* of ABC with respect to P. The circumcircle of the triangle $P_aP_bP_c$ is known as the *Cevian circle*.

Weisstein, Eric W. "Cevian Triangle." From MathWorld–A Wolfram Web Resource.



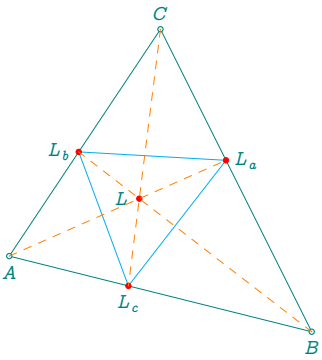

```
\directlua{
init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.C = point(1.8, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.Q = point(1, -0.4)
 z.P = point(2, 1)
 T.cevian = T.ABC:cevian(z.Q)
 z.Qa, z.Qb, z.Qc = T.cevian:get()
 T.cevian = T.ABC:cevian(z.P)
 z.Pa, z.Pb, z.Pc = T.cevian:get()
 C.cev = T.ABC:cevian_circle(z.P)
 z.w = C.cev.center}
```

14.9.9. Method symmedian()

The lines AL_a, BL_b, and CL_c which are isogonal to the triangle medians AM_a, BM_b, and CM_c of a triangle are called the triangle's **symmedian**. The symmedians are concurrent in a point L called the **Lemoine** point or the **symmedian** point which is the isogonal conjugate of the triangle centroid G.
The **symmedial** or **symmedian** triangle L_aL_bL-c is the triangle whose vertices are the intersection points of the symmedians with the reference triangle ABC.

The following example groups several concepts around the symmedian. As a reminder, a symmedian of a triangle is the reflection of the median with respect to the angle bisector.

Weisstein, Eric W. "Symmedian Point." From MathWorld–A Wolfram Web Resource.

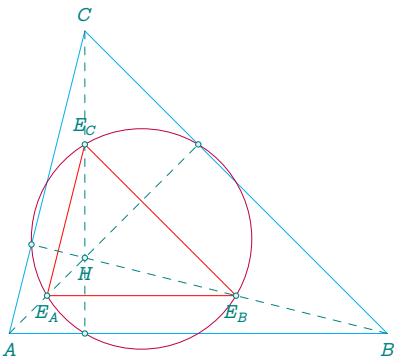



```
\directlua{
  init_elements()
  z.A = point(1, 2)
  z.B = point(5, 1)
  z.C = point(3, 5)
  T.ABC = triangle(z.A, z.B, z.C)
  T.SY = T.ABC:symmedian ()
  z.L_a,z.L_b,z.L_c = T.SY:get()
  z.L = T.ABC:lemoine_point()}
```

14.9.10. Method euler()

The *Euler triangle* of a triangle ABC is the triangle $E_AE_BE_C$ whose vertices are the midpoints of the segments joining the orthocenter H to each of the respective vertices. These points, called the *Euler points*, lie on the **nine-point** circle and are among the nine classical points of triangle geometry.

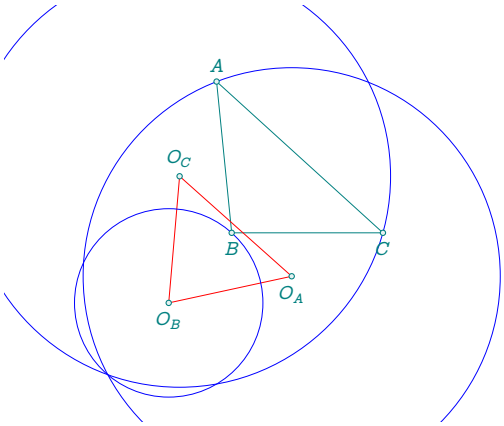
Weisstein, Eric W. "Euler Triangle." From MathWorld–A Wolfram Web Resource.




```
\directlua{
  init_elements()
  z.A  = point(0, 0)
  z.B  = point(5, 0)
  z.C  = point(1, 4)
  T.ABC  = triangle(z.A, z.B, z.C)
  z.N  = T.ABC.eulercenter
  z.H  = T.ABC.orthocenter
  T.euler = T.ABC:euler()
  z.E_A,
  z.E_B,
  z.E_C = T.euler:get ()
  z.H_A,
  z.H_B,
  z.H_C = T.ABC:orthic():get ()}
```

14.9.11. Method yiu()

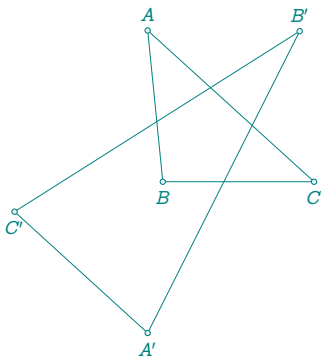
The Yiu triangle $O_AO_BO_C$ is the triangle formed by the centers of the Yiu circles. See [14.8.17]




```
\directlua{
  z.A = point(-.2, 2)
  z.B = point(0, 0)
  z.C = point(2, 0)
  T.ABC = triangle(z.A, z.B, z.C)
  T.Yiu = T.ABC:yiu()
  z.O_A, z.O_B,
  z.O_C = T.Yiu:get()}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzInit[xmin=-3,xmax=4,ymin=-2.5,ymax=3]
  \tkzClip
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPolygon[red](O_A,O_B,O_C)
  \tkzDrawCircles[blue](O_A,A O_B,B O_C,C)
  \tkzDrawPoints(A,B,C,O_A,O_B,O_C)
  \tkzLabelPoints(B,C,O_A,O_B)
  \tkzLabelPoints[above](A,O_C)
\end{tikzpicture}
```

14.9.12. Method reflection()

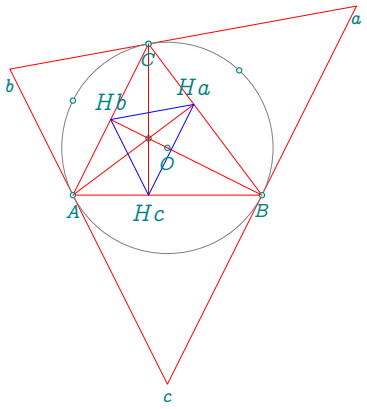
This method returns a triangle whose vertices are the reflections of the original triangle's vertices with respect to the opposite sides. That is, each vertex is reflected across the side opposite to it.




```
\directlua{%
z.A = point(-.2, 2)
z.B = point(0, 0)
z.C = point(2, 0)
T.ABC = triangle(z.A,z.B,z.C)
z.Ap,
z.Bp,
z.Cp = T.ABC:reflection():get()
}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygons(A,B,C A',B',C')
\tkzDrawPoints(A,B,C,A',B',C')
\tkzLabelPoints(B,C,A',C')
\tkzLabelPoints[above](A,B')
\end{tikzpicture}
```

14.9.13. Method circumcevian(pt)

Given a triangle ABC and a point P not located at a vertex, the *circumcevian triangle* is defined as follows: let A' be the second point of intersection (other than A) between the line (AP) and the circumcircle of ABC; define B' and C' analogously for lines (BP) and (CP). The triangle $A'B'C'$ is then called the *circumcevian triangle* of ABC with respect to point P.

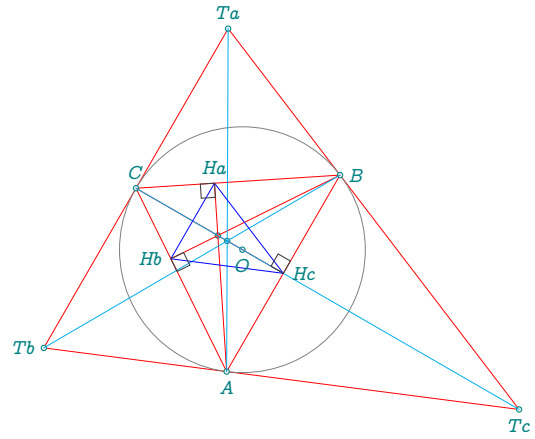



```
  \directlua{
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.H = T.ABC.orthocenter
  z.O = T.ABC.circumcenter
  z.Ha, z.Hb, z.Hc = T.ABC:orthic():get()
  z.a, z.b, z.c = T.ABC:tangential():get()
  z.p, z.q, z.r = T.ABC:circumcevian(z.H):get()
  }
\begin{center}
  \begin{tikzpicture}[  scale    = .5]
     \tkzGetNodes
     \tkzDrawPolygons[red](A,B,C a,b,c)
     \tkzDrawCircle(O,A)
     \tkzDrawPoints(A,B,C,O,H,p,q,r)
     \tkzDrawSegments[red](C,Hc B,Hb A,Ha)
     \tkzDrawPolygons[blue](Ha,Hb,Hc)
     \tkzLabelPoints(A,B,C,O,a,b,c)
     \tkzLabelPoints[font=\small](Hc)
     \tkzLabelPoints[font=\small,above](Ha,Hb)
    \end{tikzpicture}
\end{center}
```

14.9.14. Method tangential()

The tangential triangle is the triangle $TaTbTc$ formed by the lines tangent to the circumcircle of a given triangle DeltaABC at its vertices. It is therefore antipedal triangle of ABC with respect to the circumcenter O. It is also anticevian triangle of ABC with the symmedian point K as the anticevian point(Kimberling 1998, p. 156). Furthermore, the symmedian point K of ABC is the Gergonne point of $TaTbTc$.
The sides of an orthic triangle are parallel to the tangents to the circumcircle at the vertices(Johnson 1929, p. 172). This is equivalent to the statement that each line from a triangle's circumcenter to a vertex is always perpendicular to the corresponding side of the orthic triangle(Honsberger 1995, p. 22), and to the fact that the orthic and tangential triangles are homothetic.

Weisstein, Eric W. "Tangential Triangle." From MathWorld–A Wolfram Web Resource.




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 0)
  z.C = point(2, 3)
  T.ABC triangle(z.A, z.B, z.C)
  z.H = T.ABC.orthocenter
  z.O = T.ABC.circumcenter
  z.L = T.ABC:symmedian_point()
  T.orthic = T.ABC:orthic()
  z.Ha,
  z.Hb,
  z.Hc = T.orthic:get()
  z.Ta,
  z.Tb,
  z.Tc = T.ABC:tangential():get()}
```

14.9.15. Method anti()

The *anticomplementary triangle* of a reference triangle ABC is the triangle $TaTbTc$ whose medial triangle is ABC. It is also known as the *anticevian triangle* of the centroid G (see Kimberling 1998, p. 156). This triangle

is obtained by extending each side of the reference triangle beyond its vertices such that the new triangle is homothetic to the reference triangle with ratio -2 and center G.

Weisstein, Eric W. "Anticomplementary Triangle." From MathWorld–A Wolfram Web Resource.

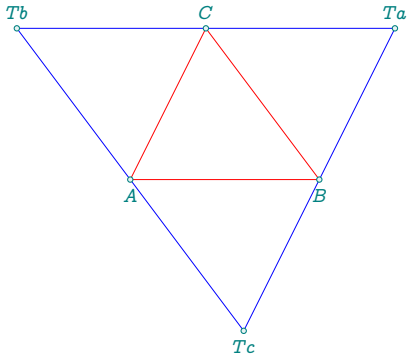
This method constructs a new triangle whose sides are parallel to the sides of the original triangle and which pass through its vertices.

This triangle can be obtained using any of the following method names: **anti**, **anticomplementary**, or **similar**.

The anticomplementary triangle is the triangle $TaTbTc$ which has a given triangle ABC as its medial triangle. It is therefore the anticevian triangle with respect to the triangle centroid G(Kimberling 1998, p. 156).

Weisstein, Eric W. "Anticomplementary Triangle." From MathWorld–A Wolfram Web Resource.

This method creates a new triangle whose sides are parallel to the sides of the original triangle and pass through its vertices. You have several method names for obtaining this triangle: either **anticomplementary** or **similar**.

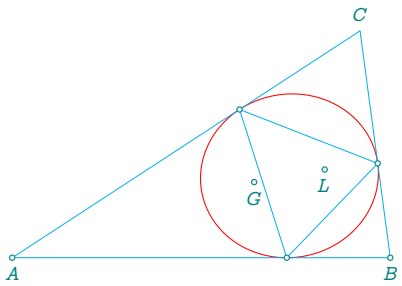



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(2, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  T.similar = T.ABC:anti()
  z.Ta,
  z.Tb,
  z.Tc = T.similar:get()}
```

14.9.16. Method lemoine()

The *Lemoine triangle* of a reference triangle ABC is the Cevian triangle associated with the Kimberling center $X_{(598)}$. Its vertices are the points of tangency between the *Lemoine inellipse* and the sides of triangle ABC.

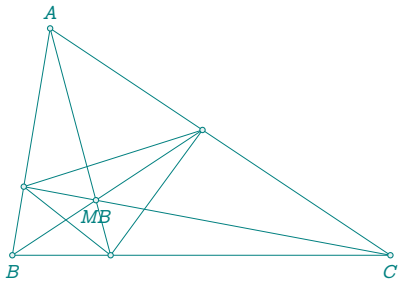
This triangle provides an elegant construction linked to the geometry of the Lemoine point and highlights interesting affine properties. The method returns the triangle formed by these three tangency points.




```
\directlua{
 z.A = point(0, 0)
 z.B = point(5, 0)
 z.C = point(4.6, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.L = T.ABC:lemoine_point()
 z.G = T.ABC.centroid
 EL = T.ABC:lemoine_inellipse()
 curve = EL:points(0, 1, 100)
 z.Xa, z.Xb,
z.Xc = T.ABC:lemoine():get()}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth,red](curve)
 \tkzDrawPolygons[cyan](A,B,C Xa,Xb,Xc)
 \tkzDrawPoints(A,B,L,Xa,Xb,Xc,G)
 \tkzLabelPoints(A,B,L,G)
 \tkzLabelPoints[above](C)
\end{tikzpicture}
```

14.9.17. Method macbeath()

The MacBeath triangle, is the triangle whose vertices are the contact points of the MacBeath inconic with the reference triangle. MB is the MacBeath point




```
\directlua{
init_elements()
z.A = point(.5, 3)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A, z.B, z.C)
z.MB = T.ABC:kimberling(264)
z.Xa, z.Xb,
z.Xc = T.ABC:macbeath():get()}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygons(A,B,C Xa,Xb,Xc)
\tkzDrawSegments(A,Xa B,Xb C,Xc)
\tkzDrawPoints(A,B,C,MB,Xa,Xb,Xc)
\tkzLabelPoints(B,C,MB)
\tkzLabelPoints[above](A)
\end{tikzpicture}
```

14.10. Returns a conic

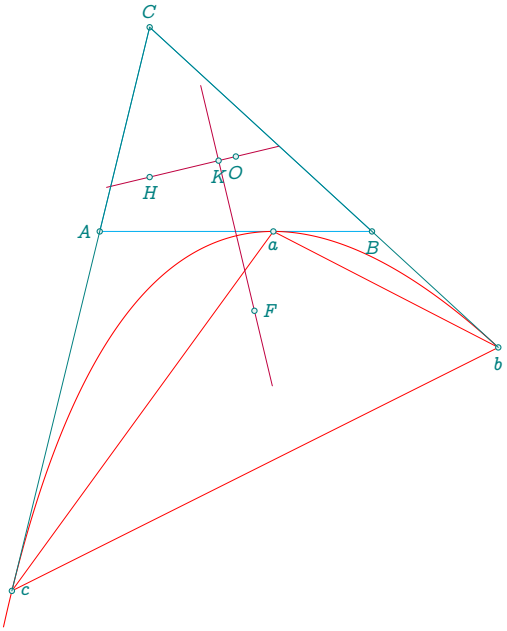
As the code is relatively large, I've avoided including the tkz-euclide part in most of the examples. You can find the complete code in the source of this document in the file **TKZdoc-elements-triangle.tex**.
The most important point is to trace the conics, which is done with the help of a tool:

```
\tkzDrawCoordinates[smooth,cyan](curve)
```

14.10.1. Method kiepert_parabola

Among all parabolas tangent to the sides of a triangle, the *Kiepert parabola* is distinguished by its geometric properties. Its directrix coincides with the Euler line of the triangle, and it is also tangent to the *Lemoine axis*.

This parabola encapsulates rich affine and projective structures and arises in various advanced triangle geometry contexts.



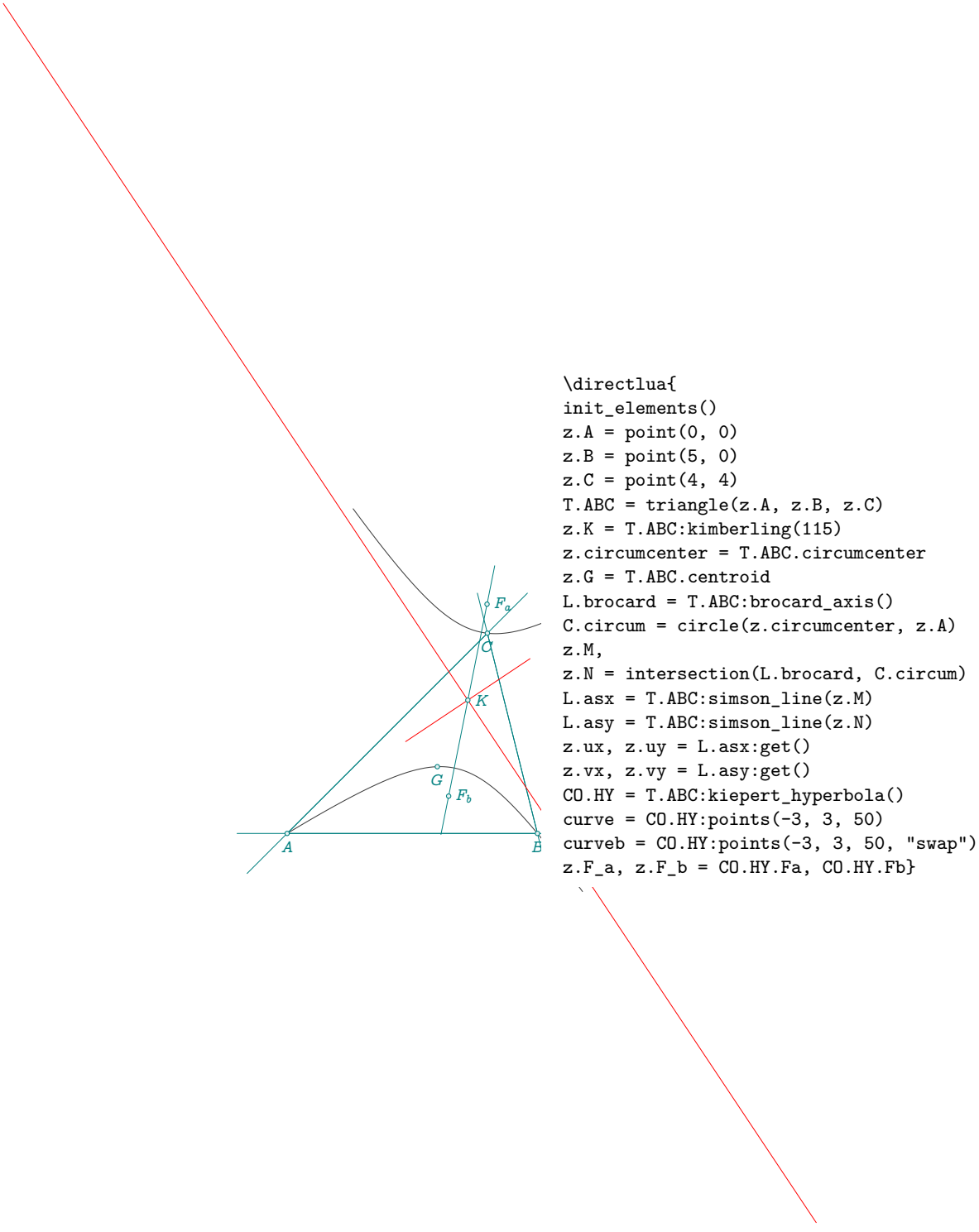

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(1.1, 4.5)
 T.ABC = triangle(z.A, z.B, z.C)
 z.H = T.ABC.orthocenter
 z.O = T.ABC.circumcenter
 kiepert = T.ABC:kiepert_parabola()
 curve = kiepert:points(-5, 7, 50)
 z.F = kiepert.Fa
 z.S = kiepert.vertex
 z.K = kiepert.K
 z.a = intersection(kiepert, T.ABC.ab)
 z.b = intersection(kiepert, T.ABC.bc)
 z.c = intersection(kiepert, T.ABC.ca)}
```

14.10.2. Method kiepert_hyperbola()

The *Kiepert hyperbola* is a remarkable triangle conic that arises from Lemoine's problem and its extension involving isosceles triangles constructed externally on the sides of a given triangle.

This hyperbola possesses several notable geometric properties: it passes through the three vertices of the triangle, its centroid, and its orthocenter.

Weisstein, Eric W. "Kiepert Hyperbola." From MathWorld–A Wolfram Web Resource.



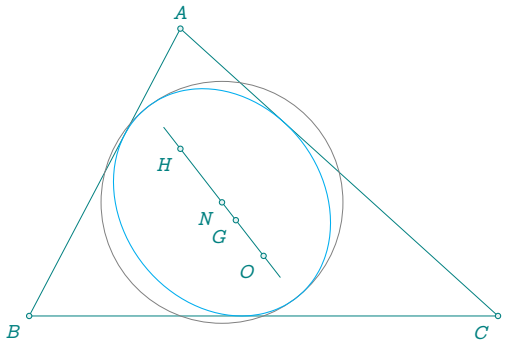

```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(4, 4)
T.ABC = triangle(z.A, z.B, z.C)
z.K = T.ABC:kimberling(115)
z.circumcenter = T.ABC.circumcenter
z.G = T.ABC.centroid
L.brocard = T.ABC:brocard_axis()
C.circum = circle(z.circumcenter, z.A)
z.M,
z.N = intersection(L.brocard, C.circum)
L.asx = T.ABC:simson_line(z.M)
L.asy = T.ABC:simson_line(z.N)
z.ux, z.uy = L.asx:get()
z.vx, z.vy = L.asy:get()
CO.HY = T.ABC:kiepert_hyperbola()
curve = CO.HY:points(-3, 3, 50)
curveb = CO.HY:points(-3, 3, 50, "swap")
z.F_a, z.F_b = CO.HY.Fa, CO.HY.Fb}
```

14.10.3. Method euler_ellipse()

The *Euler ellipse* is a conic section associated with a triangle. It is tangent to the three sides of the triangle and has two notable triangle centers as its foci: the orthocenter and the circumcenter.

This ellipse can be seen as a generalization of the Euler circle (nine-point circle), which is obtained when the conic becomes a circle. The method **euler_ellipse()** computes this conic based on the triangle geometry.

An example combining both the Euler circle and the Euler ellipse is provided in the documentation to illustrate their relationship.




```
\directlua{
   init_elements()
   z.A = point(2, 3.8)
   z.B = point(0, 0)
   z.C = point(6.2, 0)
   L.AB = line(z.A, z.B)
   T.ABC = triangle(z.A, z.B, z.C)
   z.K = tkz.midpoint(z.B, z.C)
   EL.euler = T.ABC:euler_ellipse()
   curve = EL.euler:points(0, 1, 50)
   z.N = T.ABC.eulercenter
   C.euler = circle(z.N, z.K)
   z.O = T.ABC.circumcenter
   z.G = T.ABC.centroid
   z.H = T.ABC.orthocenter}
```

14.10.4. Methods steiner_inellipse() and steiner_circumellipse()

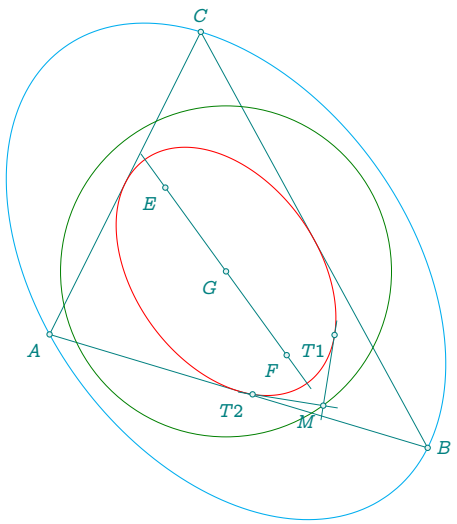
These methods return the inner and outer Steiner ellipses associated with a triangle.

The *Steiner inellipse* is the unique ellipse inscribed in a triangle and tangent to the midpoints of its sides. Its center is the centroid of the triangle. It minimizes the sum of the squared distances from any point on the ellipse to the triangle's sides.

The *Steiner circumellipse*, or outer ellipse, is the unique ellipse circumscribed about the triangle, passing through the three vertices and centered at the centroid. It minimizes the sum of the squared distances from the vertices to the ellipse.

These constructions are valid for acute triangles. The orthoptic circle, i.e., the locus of points from which chords of the ellipse are seen under right angles, is also shown in the associated example.

Weisstein, Eric W. *"Steiner Inellipse"*,
Weisstein, Eric W. *"Steiner Circumellipse"*



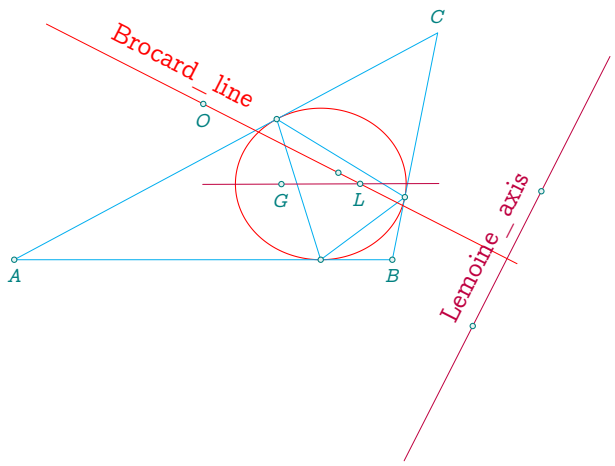

```
\directlua{
 init_elements()
 z.A = point(1, 4)
 z.B = point(11, 1)
 z.C = point(5, 12)
 T.ABC = triangle(z.A, z.B, z.C)
 CO.EL_a = T.ABC:steiner_inellipse()
 curve_a = CO.EL_a:points(0,1,100)
 z.G = CO.EL_a.center
 ang = math.deg(CO.EL_a.slope)
 z.F = CO.EL_a.Fa
 z.E = CO.EL_a.Fb
 C = CO.EL_a:orthoptic()
 z.w = C.center
 z.o = C.through
 CO.EL_b = T.ABC:steiner_circumellipse()
 curve_b = CO.EL_b:points(0, 1, 100)
 z.M = C:point(0)
 L.T1,
 L.T2 = CO.EL_a:tangent_from(z.M)
 z.T1 = L.T1.pb
 z.T2 = L.T2.pb}
```

14.10.5. Method lemoine_inellipse

The *Lemoine inellipse* is the unique inconic (an inscribed conic) of a triangle that has the centroid G and the symmedian point K as its foci.

This ellipse is always an actual ellipse and reflects key symmetries of the triangle. It is intimately connected with the triangle's geometry and often appears in advanced triangle center constructions.

Weisstein, Eric W. *"Lemoine Ellipse."* From MathWorld



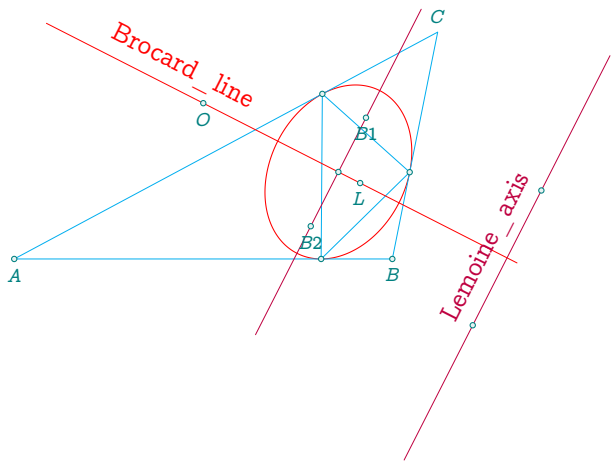

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(5, 0)
 z.C = point(5.6, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.L = T.ABC:lemoine_point()
 z.O = T.ABC.circumcenter
 z.G = T.ABC.centroid
 L.L = T.ABC:lemoine_axis()
 z.la, z.lb = L.L:get()
 L.B = T.ABC:brocard_axis()
 z.ba, z.bb = L.B:get()
 z.W = T.ABC:kimberling(39)
 CO.EL = T.ABC:lemoine_inellipse()
 curve = CO.EL:points(0, 1, 100)
 z.Ka, z.Kb, z.Kc = T.ABC:symmedian():get()
 z.Xa, z.Xb, z.Xc = T.ABC:lemoine():get()}
```

14.10.6. Method brocard_inellipse

The *Brocard inellipse* is the unique inconic of a triangle whose foci are the Brocard points and whose center is the Brocard midpoint.

This ellipse is tangent to the sides of the reference triangle, and the triangle formed by its points of tangency is the *symmedial triangle* of the reference triangle.

Weisstein, Eric W. *"Brocard Ellipse."* From MathWorld



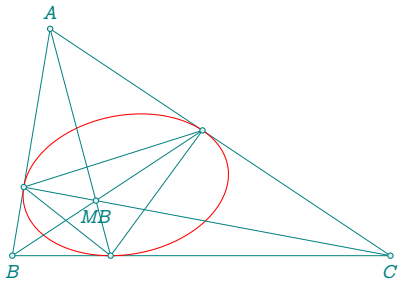

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(5, 0)
  z.C = point(5.6, 3)
  T.ABC = triangle(z.A, z.B, z.C)
  z.L = T.ABC:lemoine_point()
  z.O = T.ABC.circumcenter
  L.L = T.ABC:lemoine_axis()
  z.la, z.lb = L.L:get()
  L.B = T.ABC:brocard_axis()
  z.ba, z.bb = L.B:get()
  z.B1 = T.ABC:first_brocard_point()
  z.B2 = T.ABC:second_brocard_point()
  z.W = T.ABC:kimberling(39)
  CO.ELB = T.ABC:brocard_inellipse()
  curve = CO.ELB:points(0, 1, 100)
  z.Ka, z.Kb, z.Kc = T.ABC:symmedian():get()
  z.Xa, z.Xb, z.Xc = T.ABC:lemoine():get()}
```

14.10.7. Method macbeath_inellipse

The *MacBeath inconic* of a triangle is an ellipse tangent to the sides of the reference triangle.

Its foci are the circumcenter O and the orthocenter H, which implies that its center is the nine-point center N. The triangle formed by the points of tangency of the MacBeath ellipse with the triangle's sides is called the *MacBeath triangle*.

Weisstein, Eric W. *"MacBeath Inconic."* From MathWorld

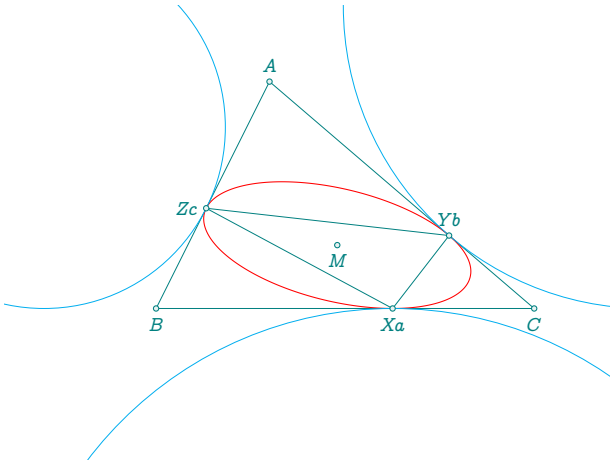



```
\directlua{
init_elements()
z.A = point(.5, 3)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A, z.B, z.C)
z.MB = T.ABC:kimberling(264)
z.Xa, z.Xb, z.Xc = T.ABC:macbeath():get()
CO = T.ABC:macbeath_inellipse()
curve = CO:points(0, 1, 100)}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygons(A,B,C Xa,Xb,Xc)
\tkzDrawCoordinates[smooth,red](curve)
\tkzDrawSegments(A,Xa B,Xb C,Xc)
\tkzDrawPoints(A,B,C,MB,Xa,Xb,Xc)
\tkzLabelPoints(B,C,MB)
\tkzLabelPoints[above](A)
\end{tikzpicture}
```

14.10.8. Method mandart_inellipse

The *Mandart ellipse* is the inconic that touches the sides of the reference triangle at the vertices of the *extouch triangle*, which is also its polar triangle.

Its center is the *mittenpunkt* M of the triangle.

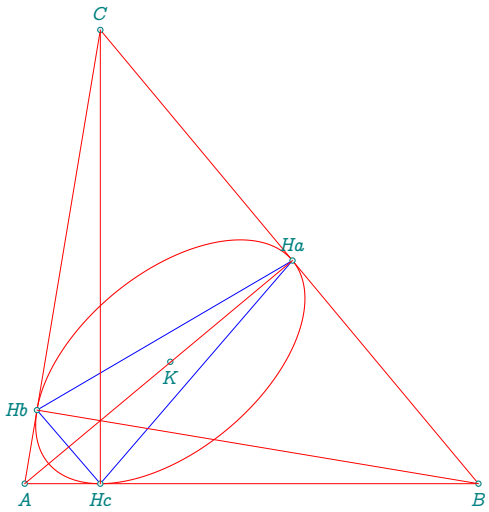



```
\directlua{
init_elements()
z.A = point(1.5, 3)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A, z.B, z.C)
z.M = T.ABC:mittenpunkt_point()
T.exc = T.ABC:excentral()
z.Ja, z.Jb, z.Jc = T.exc:get()
z.Xa, _,_ = T.ABC:projection(z.Ja)
z.Xb, z.Yb,_ = T.ABC:projection(z.Jb)
z.Xc,_, z.Zc = T.ABC:projection(z.Jc)
CO = T.ABC:mandart_inellipse()
curve = CO:points(0, 1, 100)}
\begin{tikzpicture}
\tkzGetNodes
\tkzInit[xmin=-2,xmax=6,ymin=-2,ymax = 4]
\tkzClip
\tkzDrawPolygons[](A,B,C Xa,Yb,Zc)
\tkzDrawCoordinates[smooth,red](curve)
\tkzDrawCircles[cyan](Ja,Xa Jb,Xb Jc,Xc)
\tkzDrawPoints(A,B,C,Xa,M,Zc,Yb)
\tkzLabelPoints(B,C,Xa,M)
\tkzLabelPoints[above](A,Yb)
\tkzLabelPoints[left](Zc)
\end{tikzpicture}
```

14.10.9. Method orthic_inellipse

The *orthic inellipse* is an inconic defined for acute triangles. It is tangent to the sides of the triangle at the vertices of the *orthic triangle*, which also serves as its polar triangle.

Its center is the *symmedian point K* of the reference triangle.




```
\directlua{
   z.A = point(0, 0)
   z.B = point(6, 0)
   z.C = point(1, 6)
   T.ABC = triangle(z.A, z.B, z.C)
   z.K = T.ABC:symmedian_point()
   z.Ha, z.Hb, z.Hc = T.ABC:orthic():get()
   CO =T.ABC:orthic_inellipse()
   curve = CO:points(0, 1, 100)}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawCoordinates[smooth,red](curve)
   \tkzDrawPolygons[red](A,B,C)
   \tkzDrawPolygons[blue](Ha,Hb,Hc)
   \tkzDrawPoints(A,B,C,K,Ha,Hb,Hc)
   \tkzDrawSegments[red](C,Hc B,Hb A,Ha)
   \tkzLabelPoints(A,B,K)
   \tkzLabelPoints(Hc)
   \tkzLabelPoints[left](Hb)
   \tkzLabelPoints[above](Ha,C)
  \end{tikzpicture}
```

14.11. The result is a square

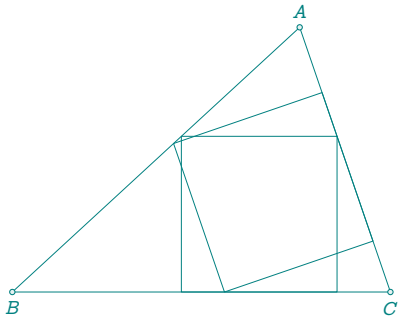
14.11.1. Method square_inscribed(n)

This method constructs a square inscribed in the triangle.

The optional argument **n** (an integer) determines which side of the triangle serves as the base of the square. The vertices of the triangle are cyclically ordered.

- If **n = 0** or omitted, the base is the segment (pa, pb);
- If **n = 1**, the base is (pb, pc);
- If **n = 2**, the base is (pc, pa).

The constructed square lies entirely within the triangle and shares one side with the chosen base.



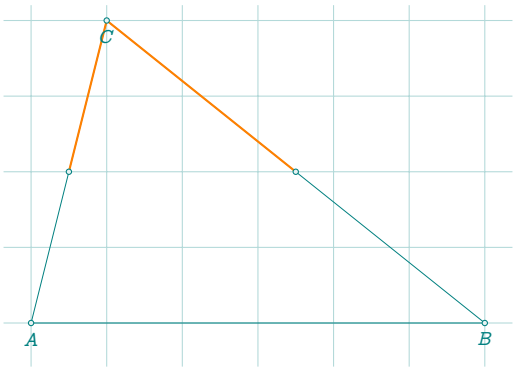

```
\directlua{
z.A = point(3.8, 3.5)
z.B = point(0, 0)
z.C = point(5, 0)
T.ABC = triangle(z.A,z.B,z.C)
S = T.ABC:square_inscribed(1)
z.b1,z.b2,z.b3,z.b4 = S:get()
S = T.ABC:square_inscribed(0)
z.a1,z.a2,z.a3,z.a4 = S:get()}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygons(A,B,C a1,a2,a3,a4)
\tkzDrawPolygons(b1,b2,b3,b4)
\tkzDrawPoints(A,B,C)
\tkzLabelPoints(B,C)
\tkzLabelPoints[above](A)
\end{tikzpicture}
```

14.11.2. Method path()

This method builds a **path** object that follows the boundary of a triangle from point p_1 to point p_2, walking in counterclockwise (direct) orientation along the triangle's perimeter.
Syntax: PA.edge = T.ABC:path(zA, zC)
Arguments:

- **p1**, **p2** — two points located on the triangle's sides

Details:
The method checks that both points lie on the triangle's boundary using the segment method **in_out_segment**. If they belong to the same side, the resulting path is direct. Otherwise, the method follows the triangle in counterclockwise order, adding intermediate vertices as needed.
This function is useful for tracing polygonal arcs or combining triangle edges with other curves to build filled shapes.



15. Class occs

15.1. Description

The occs class models an *orthonormal Cartesian coordinate system* (OCCS). Its primary use is to transition from the default coordinate system to one aligned with geometric features of the figure (e.g., a directrix, a major axis, or the vertex of a conic).
All calculations in this package are performed in a fixed OCCS. For convenience and clarity, this class allows defining and transforming to a new OCCS using a direction (a line) and a new origin point.

The typical use case is to simplify the coordinates of geometric objects such as conics, by aligning the axes with their natural directions.

15.2. Creating an occs

An OCCS (orthonormal Cartesian coordinate system) is defined by:

- a line, which sets the direction of the new *ordinate* axis (the y-axis), and
- a point, which becomes the new origin.

The abscissa axis (the x-axis) is automatically computed to be orthogonal to the ordinate axis, ensuring a right-handed coordinate system.

There are two equivalent ways to create an OCCS:

```
O.S = occs(L.axis, z.S)                    (short form, recommended)
O.S = occs:new(L.axis, z.S)                (explicit constructor)
```

The result is stored in the table `O` under the key `"S"`, representing the OCCS centered at point `z.S`.

This object can then be used to:

- compute coordinates of points in the new system,
- project points onto the new axes, or
- reconstruct geometric elements aligned with the new frame.

The new abscissa axis is automatically computed to ensure orthonormality.

15.3. Attributes of an occs

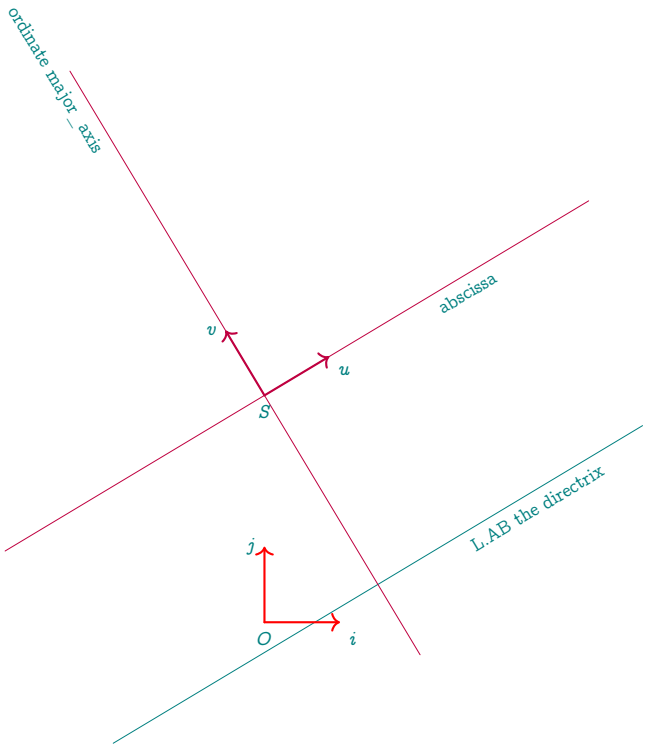
Table 13: Attributes of an occs object.

Attribute	Description
`type`	Object type name, always `"occs"`
`origin`	The new origin point
`x`	Point that defines the new x-axis (abscissa)
`y`	Point that defines the new y-axis (ordinate)
`abscissa`	Line representing the abscissa axis
`ordinate`	Line representing the ordinate axis

15.3.1. Example: attributes of occs

A few words about the arguments of a new OCCS. The most important is the line that defines the direction of the new ordinate axis. For a parabola, this direction should point from the directrix to the focus. For a hyperbola or an ellipse, it should go from the center to the principal focus. The orientation is determined by the order of the two points used to construct the line: it runs from the first to the second.
If this line is constructed as an orthogonal to another line (as in the example below), then its orientation depends on the orientation of the original line.

```
 \directlua{
 init_elements()
 z.O = point(0, 0)
 z.i = point(1, 0)
 z.j = point(0, 1)
 z.A = point(-1, -1)
 z.B = point(4, 2)
 L.AB = line(z.A, z.B)
 z.S = point(0, 3)
 L.axis = L.AB:ortho_from(z.S)
 % new occs
 O.S = occs(L.axis, z.S)
 z.u = O.S.x
 z.v = O.S.y
 z.ax = O.S.abscissa.pa
 z.bx = O.S.ordinate.pa
 z.ay = O.S.abscissa.pb
 z.by = O.S.ordinate.pb}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawLines(A,B)
 \tkzDrawLines[purple,add=4 and 4](ax,ay bx,by)
 \tkzDrawSegments[->,red,thick](O,i O,j)
 \tkzDrawSegments[->,purple,thick](S,u S,v)
 \tkzLabelSegment[below,sloped,pos=.9](A,B){L.AB the directrix}
 \tkzLabelSegment[below,sloped,pos=3](ax,ay){abscissa}
 \tkzLabelSegment[below,sloped,pos=5](bx,by){ordinate major\_axis}
 \tkzLabelPoints(O,S)
 \tkzLabelPoints[left](j,v)
 \tkzLabelPoints[below right](i,u)
\end{tikzpicture}
```



15.4. Methods of the class occs

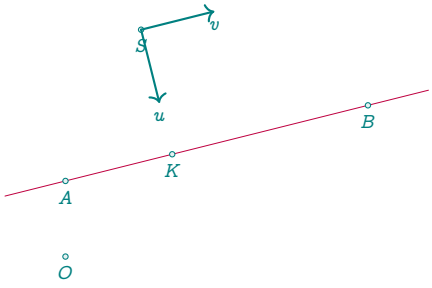
This class provides the following methods:

Table 14: occs methods.

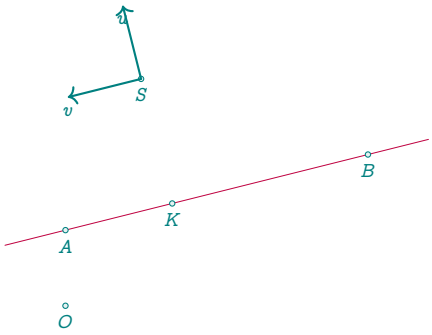
Methods	Reference
Constructor	
occs(dir, origin)	Note[8]; [15.5; 15.3.1]
Methods Returning a Real Number	
coordinates(pt)	[15.6]

15.5. Method occs(L, pt)

The general creation of an orthonormal coordinate system attached to a line has already been presented [see 15.2].
Here we emphasize the important role of the two points defining the line L. If these points are swapped, the resulting coordinate system is not the same: the orientation of the axes is reversed. In every case, the system is oriented in the forward direction, following the order of the defining points.




```
\directlua{%
z.O = point(0, 0)
z.A = point(0, 1)
z.B = point(4, 2)
z.S = point(1, 3)
L.AB = line(z.A, z.B)
z.K = L.AB:projection(z.S)
O.sys = occs(L.AB, z.S)
z.u, z.v = O.sys.x, O.sys.y}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[purple](A,B)
  \tkzDrawPoints(A,B,S,O,K)
  \tkzLabelPoints(A,B,S,O,K,u,v)
  \tkzDrawSegments[thick,->](S,u S,v)
\end{tikzpicture}
```



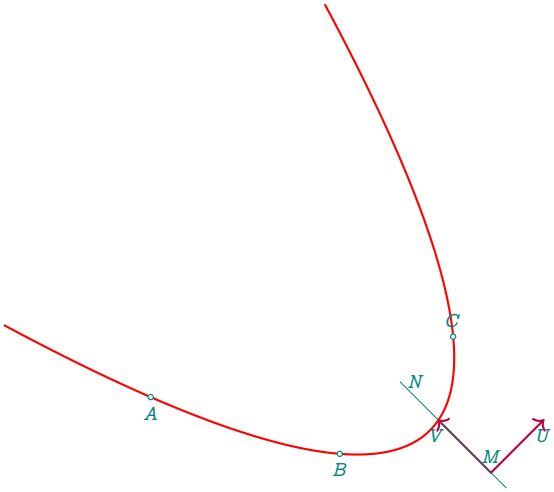

```
\directlua{%
z.O = point(0, 0)
z.A = point(0, 1)
z.B = point(4, 2)
z.S = point(1, 3)
L.AB = line(z.B, z.A)
z.K = L.AB:projection(z.S)
O.sys = occs(L.AB, z.S)
z.u, z.v = O.sys.x, O.sys.y}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[purple](A,B)
  \tkzDrawPoints(A,B,S,O,K)
  \tkzLabelPoints(A,B,S,O,K,u,v)
  \tkzDrawSegments[thick,->](S,u S,v)
\end{tikzpicture}
```

15.6. Method coordinates(pt)

We want to construct the parabola passing through three given points, knowing that the axis of symmetry of the parabola is parallel to a given line (MN).

1. First, we must check that the three points A,B,C are not collinear, and that no two of them lie on the same line perpendicular to (MN).

2. Next, we define a reference system $(M, \overrightarrow{MU}, \overrightarrow{MV})$ attached to the line (MN).
3. In this new system, there exists a unique parabola through the three points. The function **parabola(pt, pt, pt)** [see 29.18] computes the coefficients of the quadratic function $y = Ax^2 + Bx + C$.
4. To apply this function, we need the coordinates of A, B, C in the new system. This is exactly the role of the method **coordinates(pt)**.



15.7. Example: Using occs with a parabola

Let us consider a practical example. Suppose we want to compute the intersection points between a parabola and a line. The parabola is defined by its directrix and focus. In a coordinate system centered at the vertex S of the parabola, with the x-axis parallel to the directrix and passing through S, the equation of the parabola becomes

$$y = \frac{x^2}{2p}$$

where p is the *latus rectum*, i.e., the distance from the focus to the directrix.

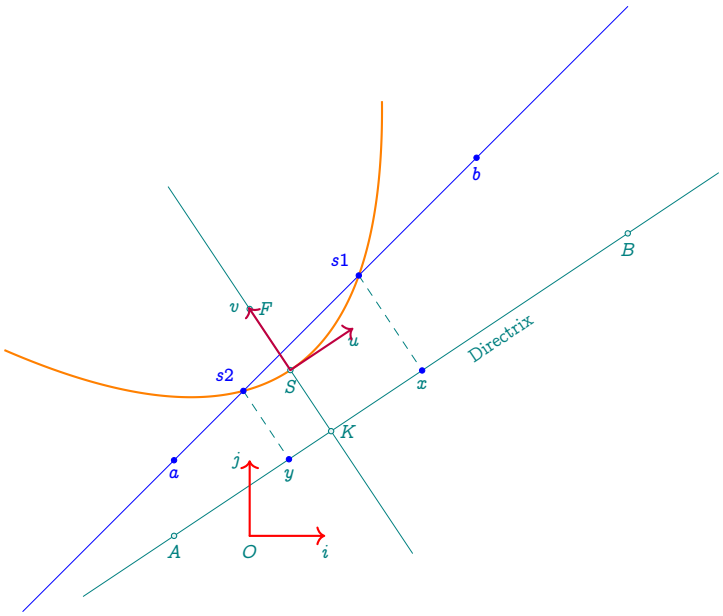
To determine the intersection points, we first express the equation of the given line in the new coordinate system. Then, we solve the equation system using $y = \frac{x^2}{2p}$. Internally, this is handled using two utility functions.

If solutions exist, the result consists of two values r_1 and r_2, which are the abscissas of the intersection points in the new OCCS. Once the corresponding ordinates are computed, we can either:

- express the coordinates in Ti*k*Z directly using a projection onto the new OCCS, or
- geometrically reconstruct the points: for each r, find the corresponding point x on the new x-axis (**OCCS.abscissa**), and use it to locate the point on the parabola.

```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.i = point(1, 0)
 z.j = point(0, 1)
 z.A = point(-1, 0)
 z.B = point(5, 4)
 L.dir = line(z.A, z.B)
 z.F = point(0, 3)
 CO.parabola = conic(z.F, L.dir, 1)
 PA.curve = CO.parabola:points(-3, 3, 50)
 local p = CO.parabola.p
 z.P = L.dir:report(p, z.K)
```

```
 z.X = PA:point(p)
 z.H = L.dir:projection(z.X)
 z.K = CO.parabola.K
 z.S = CO.parabola.vertex
 L.KF = CO.parabola.major_axis
  % new occs
 O.SKF = occs(L.KF, z.S)
 z.u = O.SKF.x
 z.v = O.SKF.y
  % line a,b
 z.a = point(-1, 1)
 z.b = point(3, 5)
 L.ab = line(z.a, z.b)
  % % coordinates in the new occs
 X, Y = O.SKF:coordinates(z.F)
 Xa, Ya = O.SKF:coordinates(z.a)
 Xb, Yb = O.SKF:coordinates(z.b)
  % solve in the new occs
 local r, s = tkz.line_coefficients(Xa, Ya, Xb, Yb)
 r1, r2 = solve_para_line(p, r, s)
 z.x = O.SKF.abscissa:report(r1, z.K)
 z.y = O.SKF.abscissa:report(r2, z.K)
 L1 = L.dir:ortho_from(z.x)
 L2 = L.dir:ortho_from(z.y)
 z.s1 = intersection(L.ab, L1)
 z.s2 = intersection(L.ab, L2)}

\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth,orange,thick](PA.curve)
 \tkzDrawLines(A,B)
 \tkzDrawLines[add = 1 and 1](K,F)
 \tkzDrawSegments[add = .5 and .5,blue](a,b)
 \tkzDrawSegments[dashed](s1,x s2,y)
 \tkzDrawPoints(A,B,F,K,S)
 \tkzDrawPoints[blue,size=2](a,b)
 \tkzDrawPoints[blue,size=2](s1,s2,x,y)
 \tkzLabelPoints[blue](a,b)
 \tkzLabelPoints[blue,above left](s1,s2)
 \tkzLabelPoints(O,i,u,S,A,B,x,y)
 \tkzLabelPoints[left](j,v)
 \tkzLabelPoints[right](F,K)
 \tkzDrawSegments[->,red,thick](O,i O,j)
 \tkzDrawSegments[->,purple,thick](S,u S,v)
 \tkzLabelSegment[below,sloped,pos=.7](A,B){Directrix}
\end{tikzpicture}
```

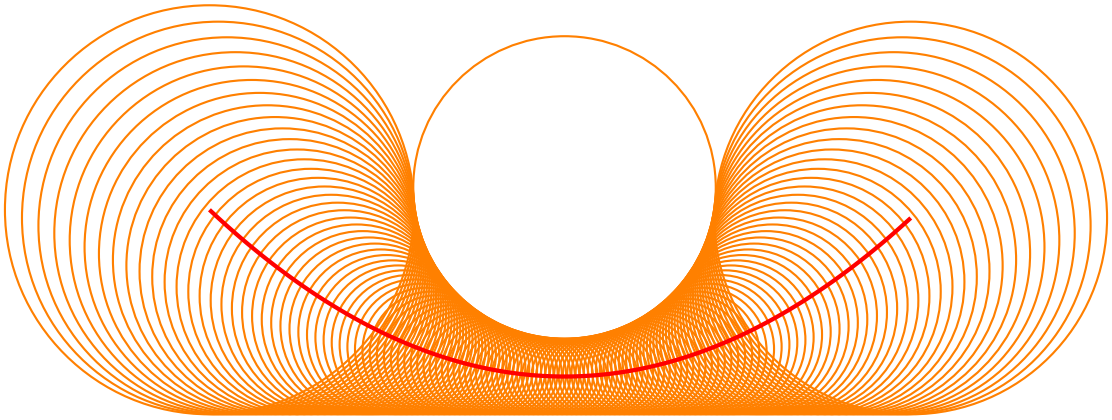

b
s1
B
v
F
u
Directrix
s2
S
x
K
a
j
y
A
O
i

16. Class conic

The variable CO holds a table used to store conics. It is optional, and you are free to choose the variable name. However, using CO is a recommended convention for clarity and consistency. If you use a custom variable (e.g., **Conics**), you must initialize it manually. The **init_elements()** function reinitializes the CO table if used.

16.1. Preamble

To illustrate some of the methods for working with conics, consider the construction of a parabola defined as the locus of all centers of circles tangent to a fixed circle and a line.
In the example below, the table **points** contains the coordinates of these centers. Since Ti*k*Z only requires a list of coordinate pairs enclosed in parentheses, the table is converted accordingly.
The table that defines the circles is slightly more elaborate. It stores, for each circle, both its center and its point of tangency with the given curve or line. Each entry is a sequence of four coordinates. These sequences are then concatenated into a string using a comma (",") as a separator.
These coordinates are finally read by **\foreach**, using the `expand list` option.



Here's the code. Two paths(tables) are created, one containing the points of the parabola, the other the points that define the tangent circles. The parabola is obtained using Ti*k*Z's ability to draw a curve from a list of coordinates.

```
\directlua{
z.O = point(0, 0)
z.K = point(0, 1)
z.P = point(0, 6)
z.M = point(0, 2)
z.I = point(1, 0)
C.PM = circle (z.P,z.M)
PA.center = path()
PA.through = path()
 for t = -0.24, 0.24, 0.004 do
   if (t > - 0.002 and t < 0.002) then else
       local a = C.PM:point(t)
       L.OI = line (z.O, z.I)
       L.PA = line (z.P, a)
       local c = intersection (L.OI, L.PA)
       L.tgt = C.PM:tangent_at (a)
       local x = intersection(L.tgt, L.OI)
       local o = tkz.bisector(x, a, c).pb
       PA.center:add_point(o)
       PA.through:add_point(a)
    end
  end}
```

```
\begin{tikzpicture}[scale =.5]
  \tkzGetNodes
  \tkzDrawCircles[thick, orange](P,M K,M)
  \tkzDrawCirclesFromPaths[draw,orange,thick](PA.center,PA.through)
  \tkzDrawCoordinates[smooth,red,ultra thick](PA.center)
\end{tikzpicture}
```

This class replaces the one dedicated to ellipses. From now on, you can work with parabolas, hyperbolas and ellipses. The circle is not part of this class. As you'll see from the examples, ellipses used to be built by Ti*k*Z, now conics are obtained by point-by-point construction. A cloud of points is sent to Ti*k*Z, which simply connects them.

```
plot[<local options>]coordinates{<coordinate 1><coordinate 2>…<coordinate n>}
```

is used by the macro **\tkzDrawCoordinates**. One advantage of this method is that you can easily draw only part of a conic.

16.2. Creating a conic

The **conic** class unifies the construction of all classical conic sections: ellipses, parabolas, and hyperbolas. It supersedes the previously available **ellipse** class, which is now deprecated.

The most natural and flexible construction method is based on the classical definition using:

- a focus point,
- a directrix (line),
- and an eccentricity value $e > 0$.

```
C0 = conic(z.F, L.dir, e)
```

Depending on the value of the eccentricity:

- if $e = 1$, the result is a parabola;
- if $e < 1$, an ellipse is constructed;
- if $e > 1$, the result is a hyperbola.

`Short form:`
The class also supports a short form:

```
C0 = conic(z.F, L.dir, e)  -- equivalent to conic:new(...)
```

`Other constructions:`
Additional creation methods are also available:

- Bifocal construction using two foci and a constant sum or difference;
- Construction from a center, vertex, and covertex;
- Construction from a general quadratic equation (conic coefficients).

These alternative methods are described in the following sections.

16.3. Attributes of a conic

Parabolas, hyperbolas, and ellipses share many attributes, though some are specific to ellipses and hyperbolas. Originally, ellipses were defined using three points: the center, a vertex, and a co-vertex. From now on, conics are preferably defined by a focus, a directrix, and an eccentricity. The old method remains available, and some tools allow conversion between both definitions.

The key attributes for this new definition are:

- **Fa**: the focus,
- **directrix**: the directrix line,
- **e**: the eccentricity.

A conic is the locus of all points P such that the ratio of the distance to the focus F over the distance to the directrix L is constant and equal to e:

$$\frac{PF}{PL} = e$$

Depending on the value of e:

- If $0 < e < 1$, the conic is an *ellipse*.
- If $e = 1$, it is a *parabola*.
- If $e > 1$, it is a *hyperbola*.

Table 15: Conic attributes.

Attributes	Definition	Reference
Fa	main foyer of the conic	
directrix	directrix of the conic	
major_axis	Axis through focal points	
minor_axis	Axis through focal points	
e	eccentricity of the conic	
type	The type is 'conic'	
subtype	'parabola', 'hyperbola' or 'ellipse'	
a	Only for hyperbola and ellipse	
b	Only for hyperbola and ellipse	
c	Only for hyperbola and ellipse	
p	semi latus rectum	
slope	Slope of the line passes through the foci	
K	Projection of the focus onto the directrix	
Fb	Second focus for hyperbola and ellipse	
vertex	main vertex	
covertex		
Rx	Radius from center to vertex	
Ry	Radius from center to covertex	

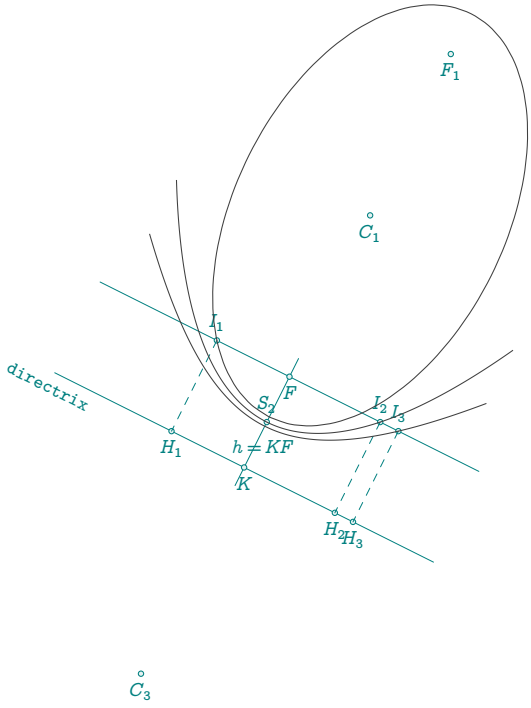
16.3.1. About attributes of conic

The figure below and the associated table show common attributes and differences according to exentricity values.

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, -2)
 L.dir = line(z.A, z.B)
 z.F = point(2, 2)
 CO.EL = conic(z.F, L.dir, 0.8)
```

```
 CO.PA = conic(z.F, L.dir, 1)
 CO.HY = conic(z.F, L.dir, 1.2)
 PA.EL = CO.EL:points(0, 1, 40)
 PA.PA = CO.PA:points(-5, 5, 40)
 PA.HY = CO.HY:points(-5, 5, 40)
 z.K = CO.EL.K
 z.u, z.v = CO.EL.major_axis:get()
 z.x = L.dir:report(-4, z.K)
 z.y = L.dir:report(4, z.K)
 z.r = (z.F - z.K):orthogonal(-4):at(z.F)
 z.s = (z.F - z.K):orthogonal(4):at(z.F)
 L.rs = line(z.r, z.s)
 z.I_1 = intersection(L.rs, CO.EL)
 z.I_2 = intersection(L.rs, CO.PA)
 z.I_3, _ = intersection(L.rs, CO.HY)
 z.H_1 = CO.EL.directrix:projection(z.I_1)
 z.H_2 = CO.PA.directrix:projection(z.I_2)
 z.H_3 = CO.HY.directrix:projection(z.I_3)
 z.S_2 = CO.PA.vertex
 z.F_1 = CO.EL.Fb
 z.C_1 = CO.EL.center
 z.C_3 = CO.HY.center}

\begin{tikzpicture}[scale=.8]
\tkzGetNodes
 \tkzDrawLines(x,y u,v r,s)
 \tkzDrawPoints(F,K,I_1,I_2,I_3,S_2)
 \tkzDrawPoints(H_1,H_2,H_3,F_1,C_1,C_3)
 \tkzLabelPoints(F,K,H_1,H_2,H_3,F_1,C_1,C_3)
 \tkzDrawSegments[dashed](I_1,H_1 I_2,H_2)
 \tkzDrawSegments[dashed](I_3,H_3)
 \tkzLabelPoints[above](I_1,I_2,I_3,S_2)
 \tkzDrawCoordinates[smooth](PA.EL)
 \tkzDrawCoordinates[smooth](PA.PA)
 \tkzDrawCoordinates[smooth](PA.HY)
 \tkzLabelSegment[pos=.4](K,F){$h = KF$}
 \tkzLabelSegment[sloped,pos=-.2](x,y){%
    \texttt{directrix}}
\end{tikzpicture}
```

The focus F, the line **directrix** and the value of $h = KF$ are attributes common to all three conics. These conics differ in their eccentricity e, here 0.8 for the ellipse, 1 for the parabola and 1.2 for the hyperbola. The **semi latus rectum** p is equal to $e * h$ and differs depending on the conic. It is represented by FI_1, FI_2 and FI_3. By definition, $e = \dfrac{p}{h}$



16.3.2. Attributes of a parabola

Let

```
CO.PA = conic(z.F, L.AB, 1)
```

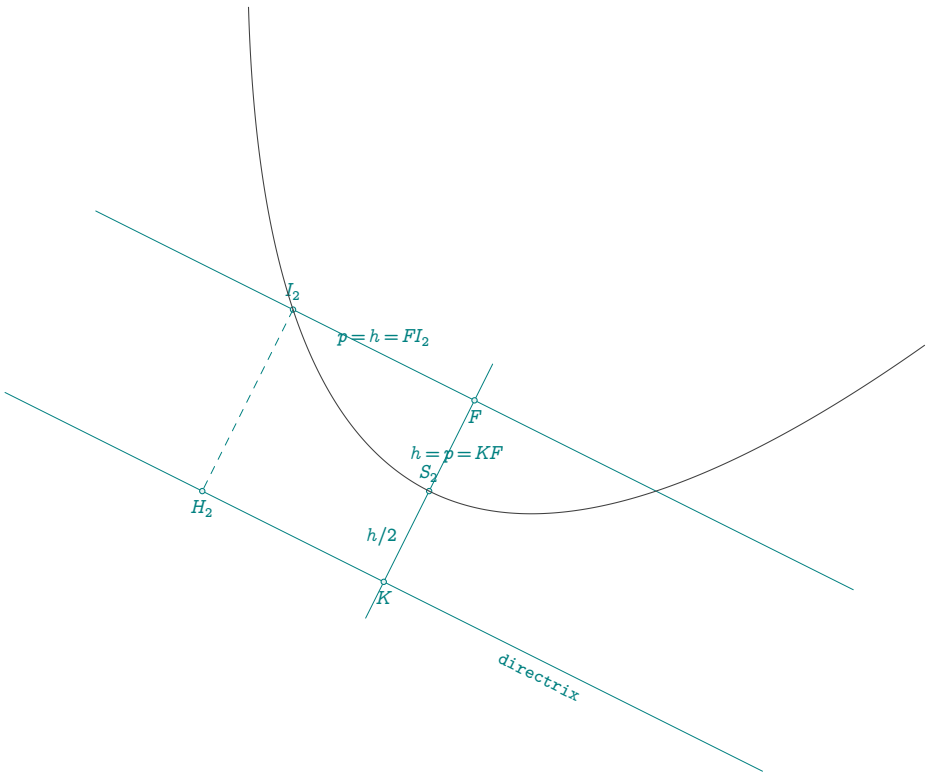
The focus is F, accessed via **CO.PA.Fa**. Since the eccentricity is 1, the conic is a parabola. Unlike ellipses and hyperbolas, a parabola has only one focus and no center. Its definition line, the directrix, is stored in **CO.PA.directrix**, and its type is identified by the attribute **CO.PA.subtype = 'parabola'**.

The projection of F onto the directrix is the point K, obtained with **CO.PA.K**. The semi-latus rectum p is given by $e \cdot h$, and here $p = h$. The vertex of the parabola, the midpoint of the segment $[KF]$, is stored as **CO.PA.vertex**. Unlike ellipses, the parabola does not have a **covertex**.

In a coordinate system centered at the vertex S and aligned with the axis of the parabola, the equation becomes $y = \dfrac{x^2}{2p}$. The usual parameters a, b, and c for conics are not defined here, as they relate to centered conics only.

Two attributes are common to all conics:

- **CO.PA.major_axis**, the main axis from K to F;
- **CO.PA.slope**, the angle between this axis and the horizontal.



16.3.3. Attributes of a hyperbola

Let

```
CO.HY = conic(z.F, L.AB, 1.2)
```

In this case, the eccentricity is greater than 1, so the conic is a hyperbola. As with all conics, the focus is given by **CO.HY.Fa**, the directrix by **CO.HY.directrix**, and the subtype is identified by **CO.HY.subtype = 'hyperbola'**.

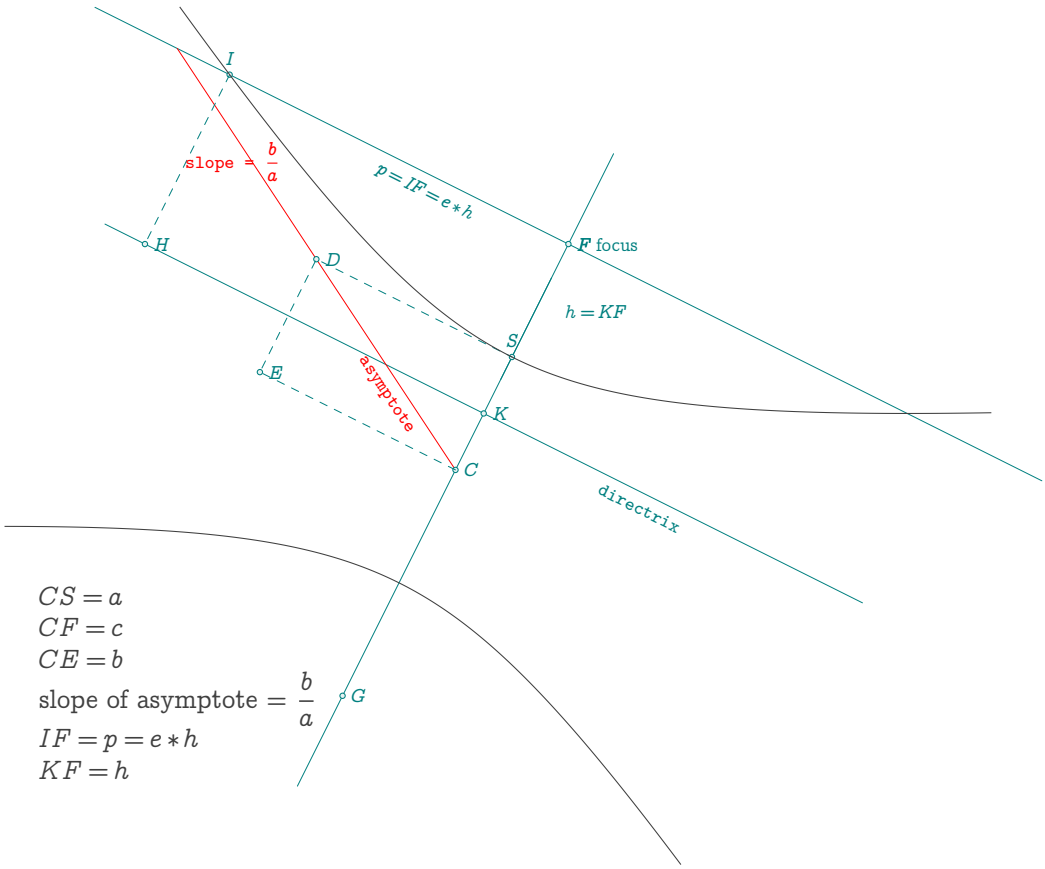
Two specific attributes of hyperbolas are:

- the second focus, **CO.HY.Fb**,
- the center of the hyperbola, **CO.HY.center**.

Several key segments characterize the hyperbola:

- $a = \overline{CS}$ is the distance from the center to a vertex, accessed with **CO.HY.a**,
- $c = \overline{CF}$ is the distance from the center to a focus, accessed with **CO.HY.c**.

The main axis (**CO.HY.major_axis**) and its slope (**CO.HY.slope**) are also available, as with all conics.



16.3.4. Attributes of an ellipse

An ellipse shares many attributes with the hyperbola. It is created using the same method, for example:

```
CO.EL = conic(z.F, L.AB, 0.8)
```

As usual, the focus is given by **CO.EL.Fa**, the directrix by **CO.EL.directrix**, and the subtype is identified by **CO.EL.subtype = 'ellipse'**. Since the eccentricity is less than 1, the conic is an ellipse.

Specific attributes include:

- the second focus: **CO.EL.Fb**,
- the center of the ellipse: **CO.EL.center**.

Distances used to characterize the ellipse:

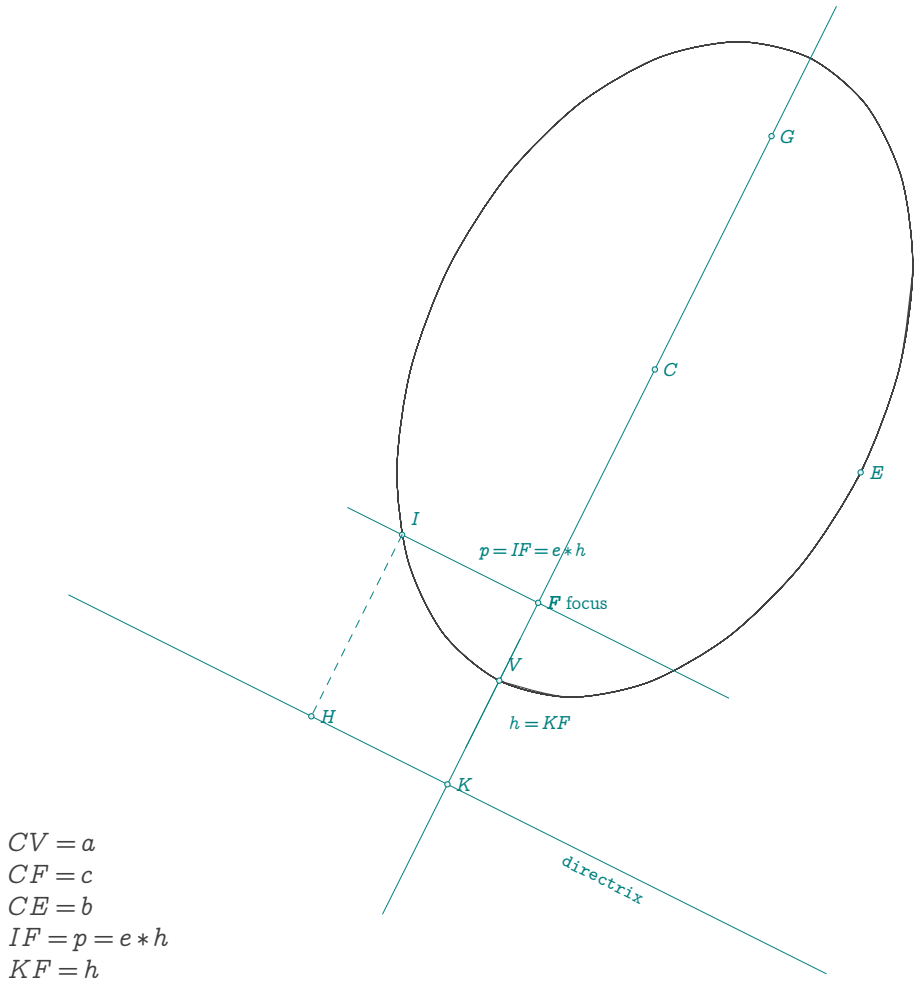
- $a = \overline{CS}$: distance from the center to a vertex, accessed via **CO.EL.a**,
- $b = \overline{CM}$: semi-minor axis (perpendicular to the major axis), via **CO.EL.b**,
- $c = \overline{CF}$: distance from the center to a focus, via **CO.EL.c**.

The relationship between these distances for an ellipse is:

$$c = \sqrt{a^2 - b^2}$$

as opposed to the hyperbola, where $c = \sqrt{a^2 + b^2}$.

As for all conics, the major axis is available via **CO.EL.major_axis**, and its orientation via **CO.EL.slope**.



16.4. Point-by-point conic construction

Let's take a closer look at the methods used to construct conics. The general approach is to generate a table of points —with a nearly constant number— to define the curve's path. This table is then passed to Ti*k*Z to produce the plot.

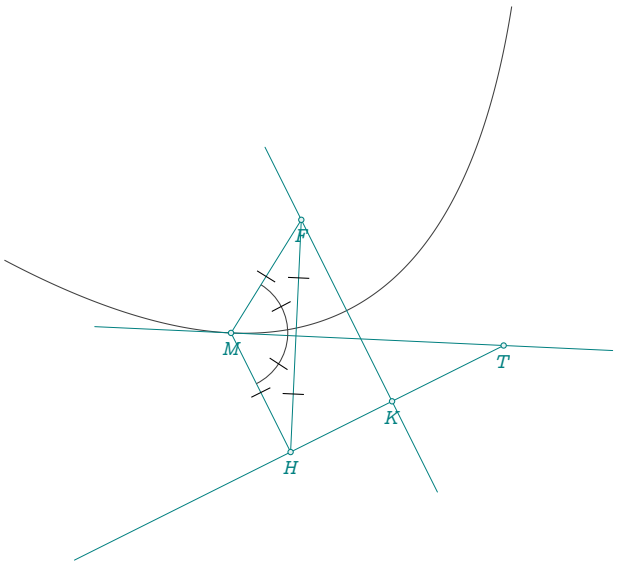
16.4.1. Parabola construction

The construction method is based on the following geometric property: if a point M lies on a parabola, then the bisector of the segment joining the focus F and the projection H of M onto the directrix is also the angle bisector of $\widehat{HFT}$ and corresponds to the tangent to the parabola at M.
This principle is used to determine a discrete set of points forming the parabola.
The result is stored in a table called **curve**, which contains the coordinates of the points on the conic. The directrix is assumed to coincide with the x-axis. You must specify:

- the starting abscissa,
- the ending abscissa,
- and the number of points to compute.

The name **curve** is only a suggestion—you may use any name you like for the table.

```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(4, 3)
 z.F = point(2, 6)
 L.AB = line(z.A, z.B)
 CO.PA = conic(z.F, L.AB, 1)
 z.K = CO.PA.K
 z.M = CO.PA:point(-2)
 z.H = CO.PA.directrix:projection(z.M)
 L.FH = line(z.F, z.H)
 L.med = L.FH:mediator()
 L.orth = CO.PA.directrix:ortho_from(z.H)
 z.T = intersection(L.AB, L.med)
 PA.curve = CO.PA:points(-5, 5, 50)
 z.m = tkz.midpoint(z.H, z.F)}

\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.curve)
 \tkzDrawLines[add = .5 and .5](A,B M,T K,F)
 \tkzDrawSegments(M,H H,F F,M)
 \tkzDrawPoints(F,K,T,M,H)
 \tkzLabelPoints(F,K,T,M,H)
 \tkzMarkAngles[mark=|](H,M,T T,M,F)
 \tkzMarkSegments[mark=|](H,M M,F)
 \tkzMarkSegments[mark=|](H,m m,F)
\end{tikzpicture}
```



16.4.2. Hyperbola construction

Constructing a hyperbola is slightly more involved.
Let K be the projection of the focus F onto the directrix. The distance FK is denoted h, and is used to compute the center C of the hyperbola. Let e be the eccentricity. The distance from the focus to the center is given by:

$$c = \frac{e^2 h}{e^2 - 1}$$

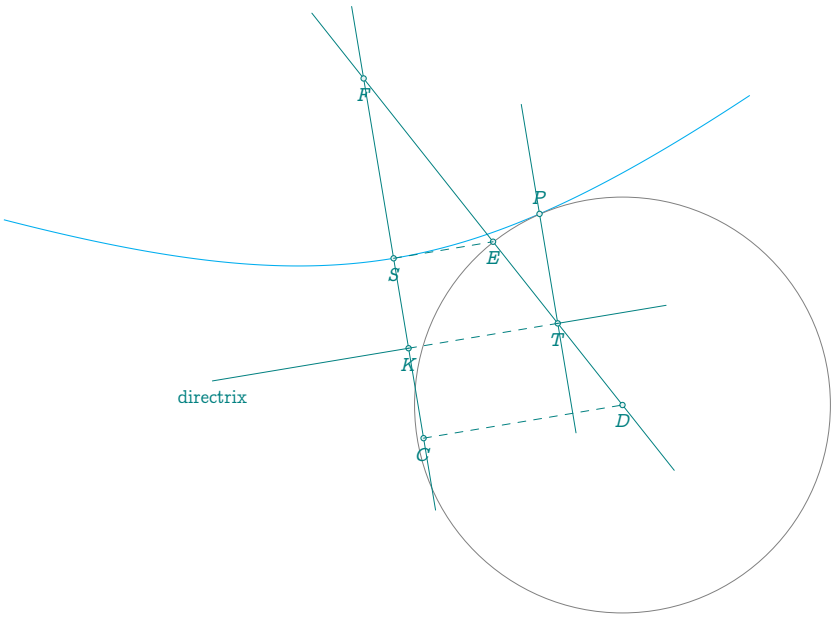
From this, the distance between the center and a vertex of the hyperbola is:

$$a = \frac{eh}{e^2 - 1} \quad \text{or equivalently} \quad a = \frac{c}{e}$$

To construct a point on the hyperbola:

- Select a point T on the directrix.
- Construct the line (FT) through the focus and T.
- This line intersects the perpendiculars at K and C (to the line (FK)) at points E and D respectively.
- The circle centered at D and passing through E intersects the perpendicular to the directrix at T in a point P.

This point P lies on the hyperbola.




```
\directlua{
 init_elements()
 z.A = point(-2, -1)
 z.B = point(4, 0)
 L.AB = line(z.A, z.B)
 z.F = point(0, 3)
 CO.HY = conic(z.F, L.AB, 2)
 PA.curve = CO.HY:points(-5, 5, 50)
 z.K = CO.HY.K
 z.S = CO.HY.vertex
 z.O = CO.HY.center
 z.X = CO.HY:point(2)
 z.T = CO.HY.directrix:report(2, CO.HY.K)
 LT = CO.HY.major_axis:ll_from(z.T)
 z.u, z.v = LT:get()
 LC = CO.HY.minor_axis
 LS = LC:ll_from(HY.vertex)
 z.D = intersection_ll_(LC.pa, LC.pb, CO.HY.Fa, z.T)
 z.E = intersection_ll_(LS.pa, LS.pb, CO.HY.Fa, z.T)
 z.P, z.Q = intersection_lc_(LT.pa, LT.pb, z.D, z.E)
 z.C = CO.HY.center}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates[smooth,cyan](PA.curve)
  \tkzDrawCircle(D,E)
  \tkzDrawLines(F,C F,D)
  \tkzDrawLines[add = 1 and 1](T,P)
  \tkzDrawPoints(C,F,K,S,T,P,D,E)
  \tkzLabelPoints(C,F,K,S,T,D,E)
  \tkzLabelPoint[below,sloped](A){directrix}
  \tkzLabelPoints[above](P)
  \tkzDrawSegments(A,K T,B)
  \tkzDrawSegments[dashed](S,E K,T C,D)
  \end{tikzpicture}
```

16.4.3. Ellipse construction

The ellipse can be constructed point by point using an affinity. This transformation is applied to the principal circle, using the focal axis as the direction of the affinity and a ratio equal to b/a, where a and b are the semi-major and semi-minor axes of the ellipse.
Let Q be a point on the principal circle, and H its projection onto the focal axis. Consider the segment $OA = a$ on the focal axis, and the segment $OB = b$ perpendicular to it. Draw the line (AB) and then a line parallel to it passing through Q. This line intersects the focal axis at point T.
By construction, the similarity ratio is:

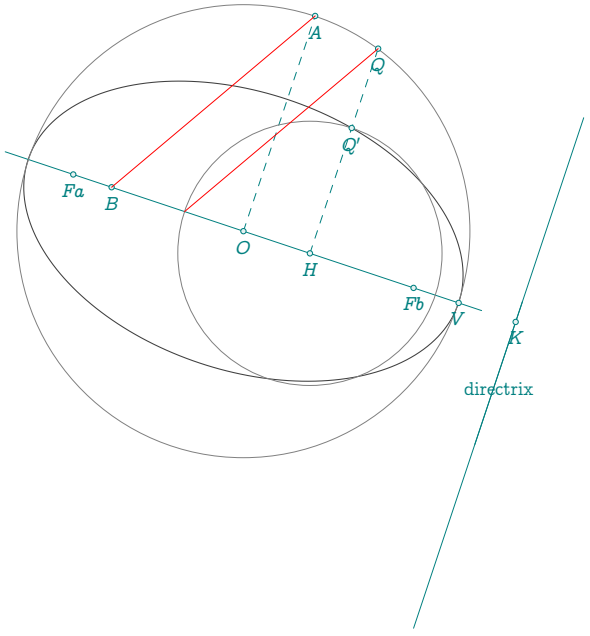
$$\frac{b}{a} = \frac{HT}{HQ}$$

Now, if we reflect Q over T, we obtain a new point Q' such that $HT = HQ'$. Hence:

$$\frac{HQ'}{HQ} = \frac{b}{a}$$

The point Q' lies on the ellipse. Repeating this construction for various points Q on the principal circle gives the ellipse as the image of the circle under the affinity.

```
\directlua{
 init_elements()
 z.Fb = point(3, 0)
 z.Fa = point(-3, 2)
 local c = tkz.length(z.Fa, z.Fb) / 2
 local a = 4
 local b = math.sqrt(a ^ 2 - c ^ 2)
 local e = c / a
 L.focal = line(z.Fa, z.Fb)
 z.O = L.focal.mid
 L.OFb = line(z.O, z.Fb)
 z.K = L.OFb:report(a ^ 2 / c)
 z.Ko = ortho_from_(z.K, z.K, z.Fb)
 L.dir = line(z.K, z.Ko)
 CO.EL = conic(z.Fb, L.dir, e)
 PA.curve = CO.EL:points(0, 1, 100)
 z.V = CO.EL.vertex
 local C = circle(z.O, CO.EL.vertex)
 z.A = C:point(0.25)
 z.B = L.focal:report(-CO.EL.b, z.O)
 z.Q = C:point(0.2)
 z.H = L.focal:projection(z.Q)
 z.Qp = L.focal:affinity(L.focal:
   ortho_from(z.O), b / a, z.Q)
 z.T = intersection_ll_(z.Q,
   ll_from_(z.Q, z.A, z.B), z.Fb, z.Fa)}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCoordinates[smooth](PA.curve)
\tkzDrawLines(Fa,Fb K,Ko)
\tkzDrawLines[add = 2 and 2](K,Ko)
\tkzDrawSegments[dashed](H,Q O,A)
\tkzDrawCircles(O,Q  H,T)
\tkzDrawPoints(Fa,Fb,Q,Q',H,V,A,B,O,K)
\tkzDrawSegments[red](A,B Q,T)
\tkzLabelPoints(Fa,Fb,Q,Q',H,V,A,B,O,K)
\tkzLabelSegment(K,Ko){directrix}
\end{tikzpicture}
```



16.5. Methods of the class conic

The methods previously designed for the (now obsolete) **ellipse** class have been generalized to the **conic** class.

The most natural creation method is now the one based on a focus, a directrix, and the eccentricity.

```
CO = conic(z.F,L.dir,e)
```

Depending on the latter, it is easy to distinguish between parabolas, hyperbolas, and ellipses. The bifocal definition of hyperbolas and ellipses is also available, as well as the one based on three points: the center, a vertex, and a covertex.

Table 16: Conic methods.

Methods	Reference
Constructor	
new (pt, L , e)	CO = conic: new (focus, directrix,eccentricity)
Methods Returning a Point	
get(i) point(t)	[16.5.1] [16.5.2]
Methods Returning a String	
position(pt)	[16.5.7]
Methods Returning a Boolean	
in_out(pt)	[16.5.8]
Methods Returning a Line	
tangent_at(pt) tangent_from(pt) asymptotes()	[16.5.4] [16.5.5] [16.5.23]
orthoptic_curve()	[16.5.9]
Methods Returning a Path	
path(pt, pt, nb, swap) points(ta,tb,nb,<'sawp'>)	[16.5.10] See [16.5.2]

Table 17: Conic functions.

Functions	Reference
PA_dir(pt,pt,pt)	[16.5.13]
PA_focus(L,pt,pt)	[16.5.14]
HY_bifocal(pt,pt,pt or r)	[16.5.15]
EL_bifocal(pt,pt,pt or r)	[16.5.16]
EL_points(L,pt,pt)	[16.5.17]
search_ellipse(s1, s2, s3, s4, s5)	[16.5.19]
test_ellipse(pt,t)	[16.5.21]
search_center_ellipse(t)	[16.5.22]
ellipse_axes_angle(t)	[16.5.20]

16.5.1. Method get()

This method retrieves the main characteristic points of a conic. The points returned depend on the subtype of the conic (ellipse, hyperbola, or parabola), but the calling convention is uniform.

- For a *parabola*, the defining points are the vertex and the focus.
- For an *ellipse* or a *hyperbola*, the defining points are the center and the two foci.

- **C:get()** returns all defining points of the conic.
- **C:get(1)** returns the first defining point.

- `C:get(2)` returns the second defining point.
- `C:get(3)` returns the third defining point, if it exists.

More precisely:

- For a parabola, `C:get()` returns the vertex and the focus.
- For an ellipse or a hyperbola, `C:get()` returns the center and the two foci.

This method provides a unified way to access the essential points of a conic, independently of its construction (focus–directrix definition, axis-based construction, etc.). It is particularly useful for composing constructions, exporting points, or defining derived geometric objects.

16.5.2. Method points

This method generates a set of points lying on the conic. These points can then be used with tkz-euclide, which, via TikZ's `draw[options] plot coordinates`, will render the curve.
The method requires three arguments: the minimum value of the parameter t, the maximum value, and the number of intermediate points between these two bounds.

```
C0 = conic(z.F, L.dir, e)
curve = C0:points(ta, tb, nb)
```

Once the list of points is created, it can be plotted using the macro `\tkzDrawCoordinates`:

```
\tkzDrawCoordinates[smooth,red](curve)
```

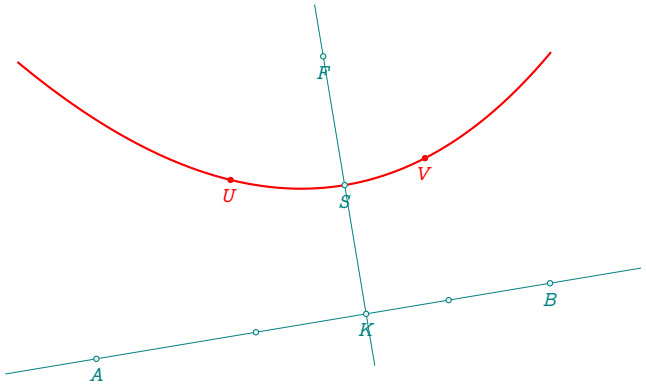
Examples with the three types of conics

1. With parabola

   t is the abscissa of a point on the parabola, in an orthonormal frame of reference with origin K and based on the directrix line and focal axis (major_axis).

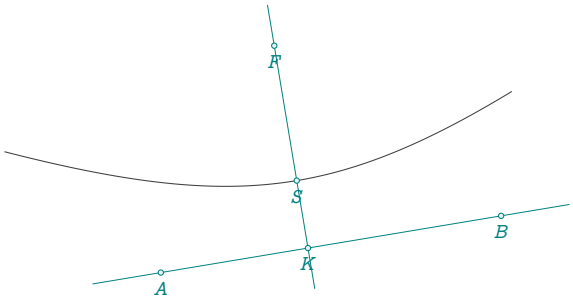
```
\directlua{
 init_elements()
 z.A = point(-2, -1)
 z.B = point(4, 0)
 z.F = point(1, 3)
 L.dir = line(z.A, z.B)
 C0.PA = conic(z.F, L.dir, 1)
 PA.curve = C0.PA:points(-4, 3, 50)
 z.K = C0.PA.K
 z.S = C0.PA.vertex
 L.AF = line(z.A, z.F)
 L.BF = line(z.B, z.F)
 z.U = intersection(C0.PA, L.AF)
 z.V = intersection(C0.PA, L.BF)
 part = C0.PA:points(-4, 3, 50)
 z.HU = L.dir:projection(z.U)
 z.HV = L.dir:projection(z.V)
 local ta = tkz.length(z.HU, z.K)
 local tb = tkz.length(z.HV, z.K)
 PA.curvered = C0.PA:points(-ta, tb, 20)}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.curvered)
```

```
\tkzDrawCoordinates[smooth,red,
                    thick](PA.curvered)
\tkzDrawLines(A,B K,F)
\tkzDrawPoints(A,B,F,K,S,HU,HV)
\tkzDrawPoints[red](U,V)
\tkzLabelPoints[red](U,V)
\tkzLabelPoints(A,B,F,K,S)
\end{tikzpicture}
```



2. With hyperbola

   As with the parabola, the parameter t denotes the abscissa of a point on the hyperbola. The directrix is assumed to be the x-axis. To generate the second branch of the hyperbola, simply include the argument **"swap"**.



```
\directlua{
 init_elements()
 z.A = point(-2, -1)
 z.B = point(4, 0)
 L.AB = line(z.A, z.B)
 z.F = point(0, 3)
 CO.HY = conic(z.F, L.AB, 2)
 curve = CO.HY:points(-5, 4, 50)
 curveb = CO.HY:points(-5, 4, 50, "swap")
 z.K =  CO.HY.K
 z.S =  CO.HY.vertex
 z.O =  CO.HY.center}
 \begin{tikzpicture}
 \tkzGetNodes
```

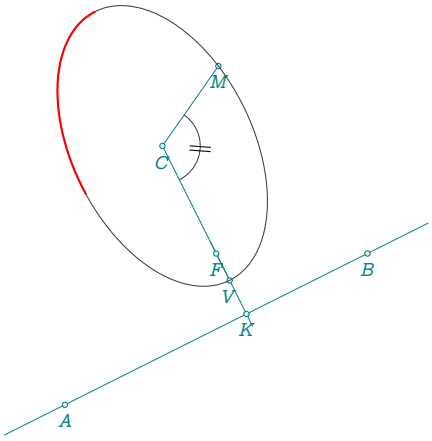
```
\tkzDrawCoordinates[smooth,cyan](curve)
\tkzDrawCoordinates[smooth,cyan](curveb)
\tkzDrawLines(A,B F,K)
\tkzDrawPoints(A,B,F,K,S)
\tkzLabelPoints(A,B,F,K,S)
\end{tikzpicture}
```

3. With ellipse

   This case differs slightly: the parameter t is a real number between 0 and 1, representing a fraction of the angle $\widehat{MCV}$ measured in radians, where C is the center of the ellipse, V a vertex, and M a point on the ellipse. Thus, $t = 0$ and $t = 1$ correspond to the same vertex, $t = 0.5$ gives the opposite vertex, and $t = 0.25$ corresponds to a covertex.

   The next example shows how to draw only a portion of the ellipse.

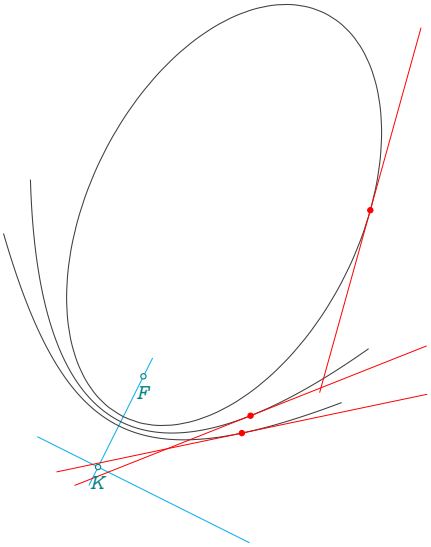
```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(4, 2)
  L.dir = line(z.A, z.B)
  z.F = point(2, 2)
  CO.EL = conic(z.F, L.dir, 0.8)
  PA.curve = CO.EL:points(0, 1, 50)
  PA.part = CO.EL:points(0.5, 0.75, 50)
  z.K = CO.EL.K
  z.C = CO.EL.center
  z.V = CO.EL.vertex
  z.M = CO.EL:point(0.3)}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawLines(A,B K,F)
\tkzDrawSegments(C,V C,M)
\tkzDrawPoints(A,B,C,F,K,M,V)
\tkzLabelPoints(A,B,C,F,K,M,V)
\tkzDrawCoordinates[smooth](curve)
\tkzDrawCoordinates[smooth,
  red,thick](part)
\tkzMarkAngles[mark=||,size=.5](V,C,M)
\end{tikzpicture}
```



16.5.3. Method point(r)

This method defines a point on the conic. Unlike the **points** method, it takes only a single argument—the abscissa of the point. See also: Sections 16.5.4, 16.4.1, and 16.4.2 for related uses.

16.5.4. Method tangent_at

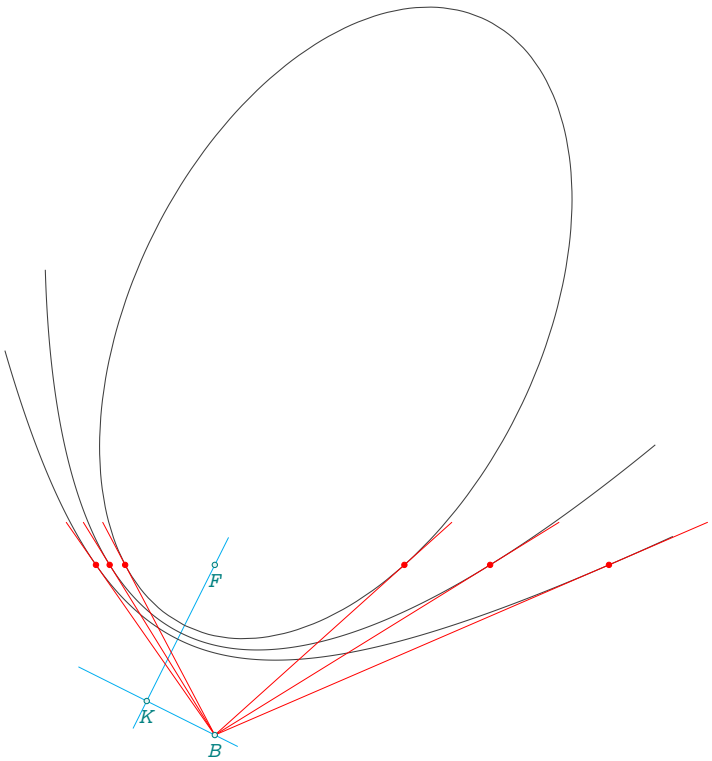



```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, -2)
 L.dir = line(z.A, z.B)
 z.F = point(2, 2)
 CO.EL = conic(z.F, L.dir, 0.8)
 CO.PA = conic(z.F, L.dir, 1)
 CO.HY = conic(z.F, L.dir, 1.2)
 PA.EL = CO.EL:points(0, 1, 50)
 PA.PA = CO.PA:points(-5, 5, 50)
 PA.HY = CO.HY:points(-5, 5, 50)
 z.X_1 = CO.EL:point(0.3)
 z.X_2 = CO.PA:point(3)
 z.X_3 = CO.HY:point(3)
 T1 = CO.EL:tangent_at(z.X_1)
 T2 = CO.PA:tangent_at(z.X_2)
 T3 = CO.HY:tangent_at(z.X_3)
 z.u1, z.v1 = T1:get()
 z.u2, z.v2 = T2:get()
 z.u3, z.v3 = T3:get()
 z.K = CO.PA.K}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawLines[cyan](A,B K,F)
 \tkzDrawCoordinates[smooth](PA.EL)
 \tkzDrawCoordinates[smooth](PA.PA)
 \tkzDrawCoordinates[smooth](PA.HY)
 \tkzDrawLines[add = 2 and 2,red](u1,v1)
 \tkzDrawLines[add = 2 and 2,red](u2,v2)
 \tkzDrawLines[add = 2 and 2,red](u3,v3)
 \tkzDrawPoints[red](X_1,X_2,X_3)
 \tkzDrawPoints(K,F)
 \tkzLabelPoints(K,F)
\end{tikzpicture}
```

16.5.5. Method tangent_from

This method computes the two tangents drawn from a given point to the conic. It returns the contact points of the tangents on the curve as two distinct points. This is useful for illustrating geometric constructions involving external tangents to ellipses, parabolas, or hyperbolas.

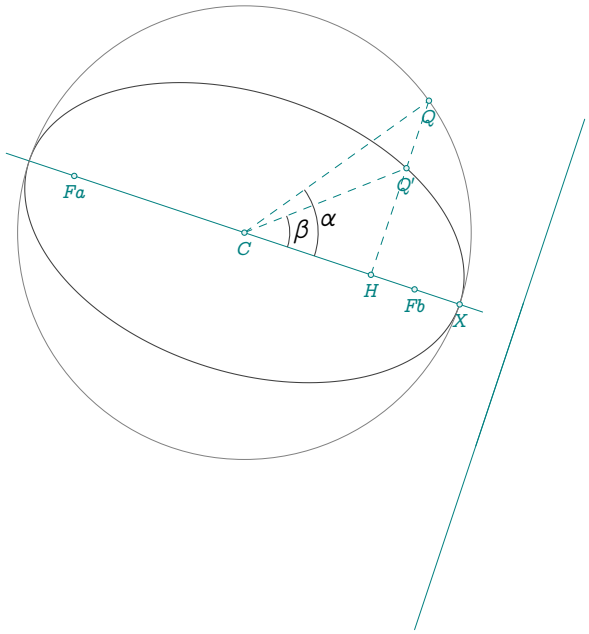
```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(2, -1)
L.dir = line(z.A, z.B)
z.F = point(2, 2)
CO.EL = conic(z.F, L.dir, 0.8)
CO.PA = conic(z.F, L.dir, 1)
CO.HY = conic(z.F, L.dir, 1.2)
PA.EL = CO.EL:points(0, 1, 50)
PA.PA = CO.PA:points(-5, 6, 50)
PA.HY = CO.HY:points(-5, 7, 50)
R1, R2 = CO.EL:tangent_from(z.B)
S1, S2 = CO.PA:tangent_from(z.B)
T1, T2 = CO.HY:tangent_from(z.B)
z.u1, z.v1 = R1:get()
z.u2, z.v2 = R2:get()
z.r1, z.s1 = S1:get()
z.r2, z.s2 = S2:get()
z.x1, z.y1 = T1:get()
z.x2, z.y2 = T2:get()
z.K = CO.PA.K}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines[cyan](A,B K,F)
  \tkzDrawCoordinates[smooth](PA.EL)
  \tkzDrawCoordinates[smooth](PA.PA)
  \tkzDrawCoordinates[smooth](PA.HY)
  \tkzDrawLines[add = 0 and .25,red](B,v1 B,v2 B,s1 B,s2 B,y1 B,y2)
  \tkzDrawPoints[red](v1,v2,s1,s2,y1,y2)
  \tkzDrawPoints(K,F,B)
  \tkzLabelPoints(K,F,B)
\end{tikzpicture}
```



16.5.6. Method point

This method is similar to the **point** method found in other classes, with specific differences depending on the conic type. The parameter t typically represents a fraction of a given distance and usually lies between 0 and 1. However, this interpretation does not apply to parabolas and hyperbolas: in these cases, t corresponds to the abscissa on the directrix of a point on the curve.
For ellipses, t has the same interpretation as in the case of circles: it defines a point on the associated principal circle. The corresponding point on the ellipse is then obtained by an affine transformation with ratio $\frac{b}{a}$, where a and b are the semi-major and semi-minor axes, respectively.
See also Section 16.5.4 for related information.
A few remarks on the following example: it illustrates how an ellipse can be constructed from its principal circle via an affinity. For instance, the segment $HQ' = \frac{b}{a}HQ$, and the angle relation $\tan(\beta) = \frac{b}{a}\tan(\alpha)$ also holds.

```
\directlua{
 init_elements()
 z.Fb = point(3, 0)
 z.Fa = point(-3, 2)
 local c = tkz.length(z.Fa, z.Fb) / 2
 local a = 4
 local b = math.sqrt(a ^ 2 - c ^ 2)
 local e = c / a
 L.FaFb = line(z.Fa, z.Fb)
 z.C = L.FaFb.mid
 L.CFb = line(z.C, z.Fb)
 z.K = L.CFb:report(a ^ 2 / c)
 z.Ko = ortho_from_(z.K, z.K, z.Fb)
 L.dir = line(z.K, z.Ko)
 CO.EL = conic(z.Fb, L.dir, e)
 PA.curve = CO.EL:points(0, 1, 100)
 z.X = CO.EL.vertex
 C.X = circle(z.C, z.X)
 z.Q = C.X:point(0.15)
 z.H = L.FaFb:projection(z.Q)
 z.Qp = L.FaFb:affinity(L.FaFb:
 ortho_from(z.C), b / a, z.Q)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates[smooth](PA.curve)
  \tkzDrawLines(Fa,Fb K,Ko)
  \tkzDrawLines[add = 2 and 2](K,Ko)
  \tkzDrawCircles(C,Q)
  \tkzDrawSegments[dashed](C,Q C,Q' H,Q)
  \tkzDrawPoints(Fa,Fb,C,X,Q,Q',H)
  \tkzLabelPoints(Fa,Fb,C,X,Q,Q',H)
  \tkzLabelAngle(Fb,C,Q'){$\beta$}
  \tkzMarkAngle[size=.8](Fb,C,Q')
  \tkzLabelAngle[pos=1.5](Fb,C,Q){$\alpha$}
  \tkzMarkAngle[size=1.3](Fb,C,Q)
\end{tikzpicture}
```



16.5.7. Method position(pt[,EPS])

This method determines the relative position of a point with respect to a conic.

It takes a point **pt** and an optional tolerance **EPS**. If omitted, the global tolerance **tkz.epsilon** is used.

The method returns one of the following strings:

- **"IN"** — the point lies in the interior region;
- **"ON"** — the point lies on the conic (within tolerance);
- **"OUT"** — the point lies in the exterior region.

The behaviour depends on the conic type:

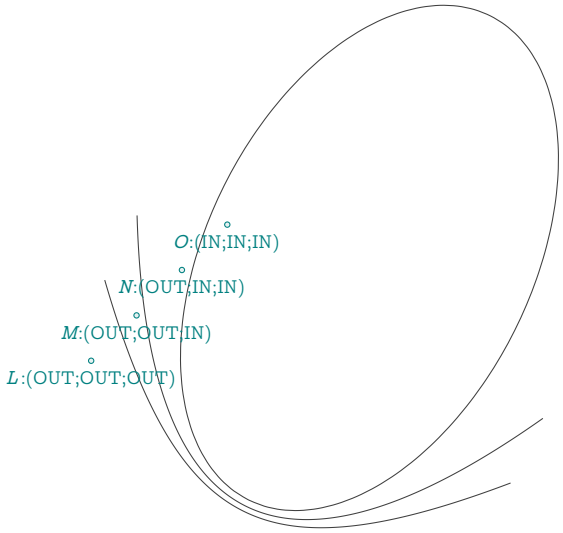
- Ellipse ($e < 1$) — interior region is bounded;
- Parabola ($e = 1$) — interior is the convex side;
- Hyperbola ($e > 1$) — interior corresponds to the region between the two branches.

`Compatibility note:` If a boolean test is required (for example in legacy code), one may use:

```
CO:position(P)  = "OUT"
```

which is true for both "IN" and "ON".

```
Example:
\directlua{
 init_elements()
 z.A = point(0,0)
 z.B = point(4,-2)
 L.dir = line(z.A,z.B)
 z.F = point(2,2)

 CO.EL = conic(z.F, L.dir, 0.8)
 CO.PA = conic(z.F, L.dir, 1.0)
 CO.HY = conic(z.F, L.dir, 1.2)

 PA.EL = CO.EL:points(0,1,60)
 PA.PA = CO.PA:points(-5,5,60)
 PA.HY = CO.HY:points(-5,5,60)

 z.L = point(-2,4)
 z.M = point(-1,5)
 z.N = point(0,6)
 z.O = point(1,7)

 Lel = CO.EL:position(z.L)
 Lpa = CO.PA:position(z.L)
 Lhy = CO.HY:position(z.L)

 Mel = CO.EL:position(z.M)
 Mpa = CO.PA:position(z.M)
 Mhy = CO.HY:position(z.M)

 Nel = CO.EL:position(z.N)
 Npa = CO.PA:position(z.N)
 Nhy = CO.HY:position(z.N)

 Oel = CO.EL:position(z.O)
 Opa = CO.PA:position(z.O)
 Ohy = CO.HY:position(z.O)
}
```



16.5.8. Method in_out; Deprecated

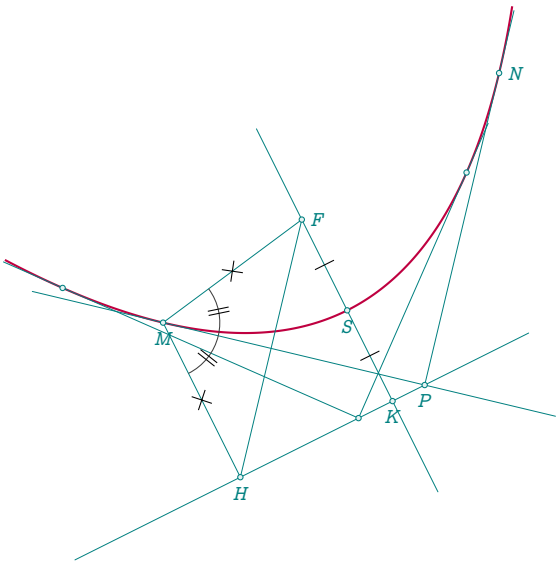
This method is kept for backward compatibility.

16.5.9. Method orthoptic

In curve geometry, the orthoptic of a conic is the set of points from which two tangents to the curve intersect at a right angle. For a parabola, the orthoptic is simply the directrix. For ellipses and hyperbolas, the orthoptic is a circle—but in the case of the hyperbola, this holds only if the eccentricity lies between 1 and $\sqrt{2}$. [9]

9 When the eccentricity equals $\sqrt{2}$, the hyperbola is rectangular (equilateral). In an orthonormal coordinate system, its asymptotes then have equations $y = x$ and $y = -x$.

```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(4, 3)
 z.F = point(2, 6)
 L.AB = line(z.A, z.B)
 CO.PA = conic(z.F, L.AB, 1)
 PA.curve = CO.PA:points(-5, 5, 50)
 z.K = CO.PA.K
 z.S = CO.PA.vertex
 z.M = CO.PA:point(-3)
 z.H = CO.PA.directrix:projection(z.M)
 L.FH = line(z.F, z.H)
 L.med = L.FH:mediator()
 z.P = intersection(L.AB, L.med)
 z.N = CO.PA:tangent_from(z.P).pb
 D = CO.PA:orthoptic()
 z.v = D:point(0.75)
 T1, T2 = CO.PA:tangent_from(z.v)
 z.t1 = T1.pb
 z.t2 = T2.pb}
  \begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates[smooth,
            thick,purple](PA.curve)
  \tkzDrawLines[add = 0 and .2](v,t1
       v,t2 P,N)
  \tkzDrawLines[add = .5 and .5](A,B
        M,P K,F)
  \tkzDrawSegments(M,H H,F F,M)
  \tkzDrawPoints(F,K,P,M,H,v,t1,t2,S,N)
  \tkzLabelPoints(K,P,M,H,S)
  \tkzLabelPoints[right](F,N)
  \tkzMarkAngles[mark=||](H,M,P P,M,F)
  \tkzMarkSegments[mark=x](H,M M,F)
  \tkzMarkSegments[mark=|](F,S K,S)
  \end{tikzpicture}
```



16.5.10. Method path(pt, pt, nb, mode, dir)

The **conic** class provides a method **path** that creates a **path** object representing a portion of the conic between two given points (an arc of conic).

Syntax: PA.arc = CO.myconic:path(zA, zB, 40, "swap")
Arguments:

- **za**, **zb** — two points lying on the conic.
- **nb** — number of interpolation steps (default: 20).
- **mode** — optional string:
  - omitted or **"direct"** (default): traces the shortest arc (for ellipses).
  - **"swap"**: selects the complementary arc (other part of the ellipse).
- **dir** — optional orientation for ambiguous cases

  In most cases, the points **za** and **zb** determine a unique arc of the conic. However, when the two points are *opposite* with respect to the center (for instance, on an ellipse), two different arcs of equal length exist. In such cases, the optional argument can be used to select the desired direction:
  - **"ccw"** (*counterclockwise*) — the arc is traced in the direct, or positive, direction.

- **"cw"** (*clockwise*) — the arc is traced in the opposite, or negative, direction.

In general use, this parameter can be omitted.

Details:
Internally, the method computes the conic parameter t at **za** and **zb** using **get_t_from_point(z)**. For ellipses, this parameter is normalized in [0,1) (one turn). For parabolas and hyperbolas, it is not periodic.
A linear interpolation of t produces intermediate points on the conic:

$$t_i = t_a + \frac{i}{nb}(t_b - t_a)$$

For ellipses specifically:

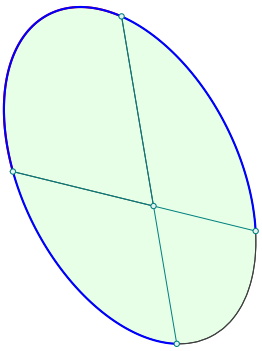
- the angle/parameter is taken modulo 1;
- in **"direct"** mode, the interpolation follows the shortest arc between the two points;
- in **"swap"** mode, the complementary arc is used (the same endpoints, but the other part of the ellipse);
- if the points are exactly opposite ($\Delta t = \pm 0.5$), the option **dir = "cw"** or **"ccw"** can be used to choose the direction.

```
-- Example: parabola
C = conic(z.F, L.directrix, 1)
PA.arc = C:path(z.A, z.B, 100)
```

```
-- Example: ellipse, complementary arc
E = conic(z.F1, z.F2, 0.6) -- an ellipse
PA.arc1 = E:path(z.A, z.B, 80, "direct") -- shortest arc
PA.arc2 = E:path(z.A, z.B, 80, "swap")   -- other arc
```

To draw the resulting path with TikZ:

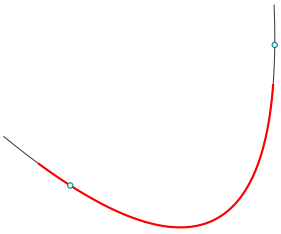
```
\tkzDrawCoordinates[smooth]
```



```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(4, 2)
L.dir = line(z.A, z.B)
z.F = point(2, 2)
CO.EL = conic(z.F, L.dir, 0.8)
PA.curve = CO.EL:points(0, 1, 50)
z.M = CO.EL:point(0.95)
z.N = CO.EL:point(0.15)
z.P = CO.EL:point(0.4)
z.Q = CO.EL:point(0.7)
PA.parta = CO.EL:path(z.M, z.N, 100, "swap")
PA.partb = CO.EL:path(z.P, z.Q, 100)
L.PM = line(z.P, z.M)
L.QN = line(z.Q, z.N)
z.I = intersection(L.PM, L.QN)
L.NI = line(z.N, z.I)
L.IM = line(z.I, z.M)
L.IP = line(z.I, z.P)
L.QI = line(z.Q, z.I)
PA.zone1 = CO.EL:path(z.M, z.N, 50, "swap")
   + L.NI:path(2) + L.IM:path(2)
PA.zone2 = CO.EL:path(z.P, z.Q, 50, "swap")
   + L.QI:path(2) + L.IP:path(2)
}
\begin{tikzpicture}[ scale = 1.2]
\tkzGetNodes
\tkzDrawCoordinates[fill = green!10](PA.zone1)
\tkzDrawCoordinates[fill = green!10](PA.zone2)
\tkzDrawCoordinates[smooth](PA.curve)
\tkzDrawCoordinates[smooth,red,thick](PA.partb)
\tkzDrawCoordinates[smooth,blue,thick](PA.parta)
\tkzDrawSegments(P,M Q,N)
\tkzDrawPoints(M,N,P,Q,I)
\end{tikzpicture}
```

Example with parabola

This example shows the case of a parabola. It would be the same for a hyperbola (the ”swap” option is not yet authorized).



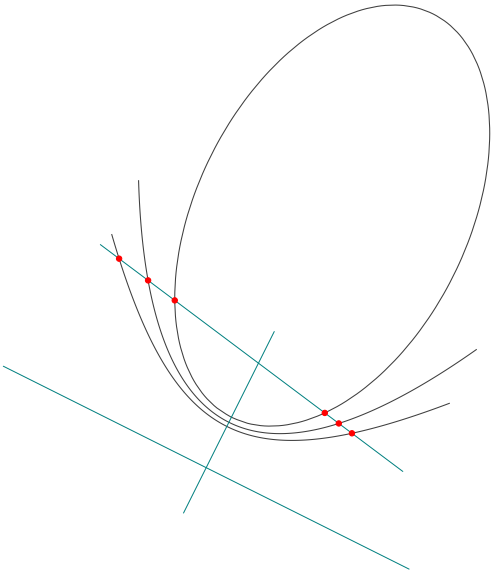
```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 2)
 L.dir = line(z.A, z.B)
 z.F = point(2, 2)
 CO.EL = conic(z.F, L.dir, 1)
 PA.curve = CO.EL:points(-2, 2, 50)
 z.P = CO.EL:point(-1.75)
 z.Q = CO.EL:point(1.5)
 PA.part = CO.EL:path(z.P, z.Q, 100)
}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.curve)
 \tkzDrawCoordinates[smooth,red,thick](PA.part)
 \tkzDrawPoints(P,Q)
\end{tikzpicture}
```

16.5.11. Intersection: Line and Conic

Additional details can be found in Section 27, in particular Subsection 27.5.

The following example illustrates how to compute the intersection between a straight line and each of the three types of conics. As with other intersection methods, there is no need to specify the type of conic explicitly—the package will determine the appropriate class automatically. Currently, intersections are only supported between straight lines and conics.

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, -2)
 L.dir = line(z.A, z.B)
 z.F = point(2, 2)
 CO.EL = conic(z.F, L.dir, 0.8)
 CO.PA = conic(z.F, L.dir, 1)
 CO.HY = conic(z.F, L.dir, 1.2)
 PA.EL = CO.EL:points(0, 1, 50)
 PA.PA = CO.PA:points(-5, 5, 50)
 PA.HY = CO.HY:points(-5, 5, 50)
 z.K = CO.EL.K
 z.u, z.v = CO.EL.major_axis:get()
 z.x = L.dir:report(-3, z.K)
 z.y = L.dir:report(3, z.K)
 z.r = point(0, 4)
 z.s = point(4, 1)
 L.rs = line(z.r, z.s)
 z.u_1, z.u_2 = intersection(L.rs, CO.EL)
 z.v_1, z.v_2 = intersection(L.rs, CO.PA)
 z.w_1, z.w_2 = intersection(L.rs, CO.HY)}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.EL)
 \tkzDrawCoordinates[smooth](PA.PA)
 \tkzDrawCoordinates[smooth](PA.HY)
 \tkzDrawLines[add =.5 and .5](r,s u,v x,y)
 \tkzDrawPoints[red](u_1,u_2,v_2,v_1,w_1,w_2)
\end{tikzpicture}
```



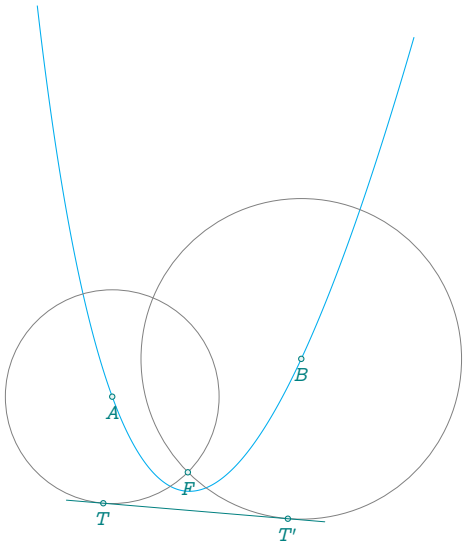
16.5.12. Useful Tools

This section presents utility functions for retrieving key geometric elements of a conic: its focus, directrix, and eccentricity.

16.5.13. Function PA_dir

This function computes the directrix of a parabola, given its focus and two points on the curve. The method involves constructing circles centered at the two given points and passing through the focus. The common tangents to these two circles correspond to the possible directrices of the parabola—two solutions exist.
To obtain the second solution, simply reverse the order of the return values: replace `_, L.dir` with `L.dir, _`.

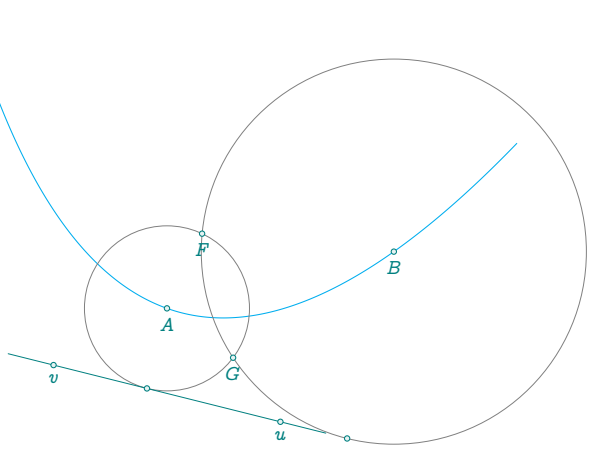
```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(5, 2)
 z.F = point(2, -1)
 _, L.dir = PA_dir(z.F, z.A, z.B)
 CO.PA = conic(z.F, L.dir, 1)
 PA.curve = CO.PA:points(-5, 5, 50)
 z.T, z.Tp = L.dir:get()}
\begin{tikzpicture}[scale = .5]
 \tkzGetNodes
 \tkzDrawCoordinates[smooth,
    cyan](PA.curve)
 \tkzDrawCircles(A,F B,F)
 \tkzDrawPoints(A,B,F,T,T')
 \tkzDrawLine(T,T')
 \tkzLabelPoints(A,B,F,T,T')
\end{tikzpicture}
```



16.5.14. Function PA_focus

This function computes the focus of a parabola, given its directrix and two points on the curve.
The method is based on constructing two circles, each centered at one of the given points and tangent to the directrix. If such a construction is possible, the focus corresponds to one of the intersection points of these two circles.

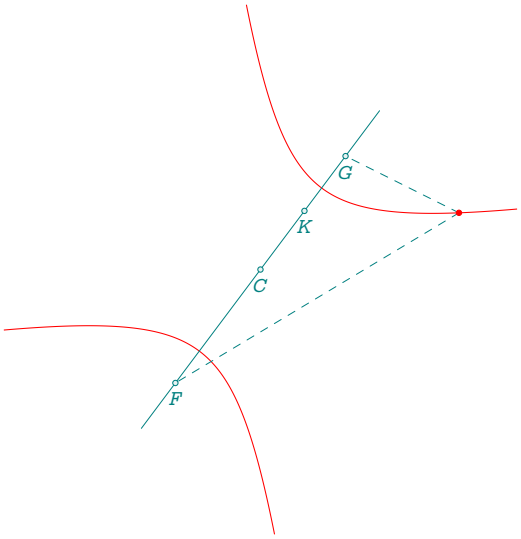
```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(4, 2)
 z.u = point(2, -1)
 z.v = point(-2, 0)
 L.dir = line(z.u, z.v)
 z.hA = L.dir:projection(z.A)
 z.hB = L.dir:projection(z.B)
 z.F, z.G = PA_focus(L.dir, z.A, z.B)
 CO.PA = conic(z.F, L.dir, 1)
 PA.curve = CO.PA:points(-5, 5, 50)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates[smooth,cyan](PA.curve)
  \tkzDrawCircles(A,hA B,hB)
  \tkzDrawLines(u,v)
  \tkzDrawPoints(A,B,u,v,hA,hB,F,G)
  \tkzLabelPoints(A,B,F,G,u,v)
\end{tikzpicture}
```



16.5.15. Function HY_bifocal

For the hyperbola, this is currently the only available tool, and it relies on the bifocal definition. The inputs are the two foci, along with either the semi-major axis a (i.e., the distance from the center to a vertex), or a point lying on the hyperbola.
The method proceeds by applying standard formulas that characterize a hyperbola to compute its main geometric attributes.

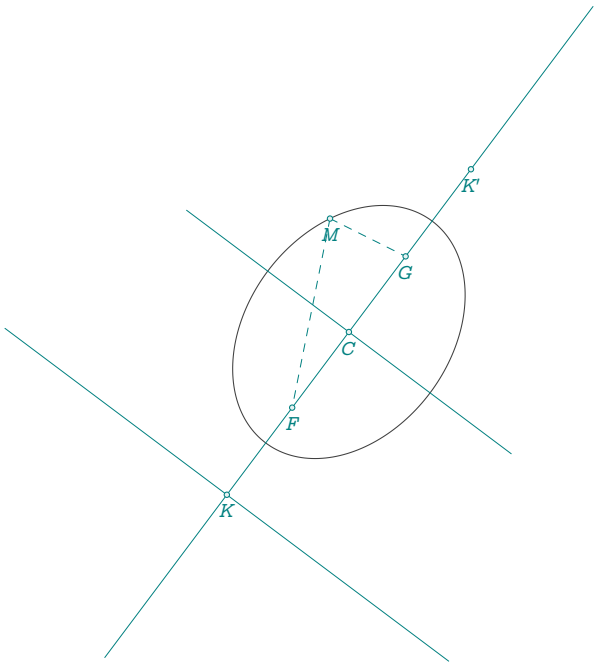
```
\directlua{
  init_elements()
  z.F = point(1, -1)
  z.G = point(4, 3)
  z.M = point(6, 2)
  z.C = tkz.midpoint(z.F,z.G)
  CO.HY = conic(HY_bifocal(z.G, z.F, z.M))
  PA.curve = CO.HY:points(-3, 3, 50)
  z.K = CO.HY.K
  PA.curveb = CO.HY:points(-3, 3, 50, "swap")}
\begin{tikzpicture}[scale = .5]
 \tkzGetNodes
 \tkzDrawCoordinates[smooth,red](PA.curve)
 \tkzDrawCoordinates[smooth,red](PA.curveb)
 \tkzDrawSegments[dashed](M,F M,G)
 \tkzDrawLine(F,G)
 \tkzDrawPoints[red](M)
 \tkzDrawPoints(C,F,G,K)
 \tkzLabelPoints(C,F,G,K)
\end{tikzpicture}
```



16.5.16. Function EL_bifocal

For the ellipse, two methods are available. The first one, **EL_bifocal**, follows the same principle as for the hyperbola: it uses the bifocal definition of the ellipse.
This function takes as input the two foci and either the semi-major axis a or a point on the ellipse. The main attributes of the ellipse are then computed using the standard bifocal relationships.

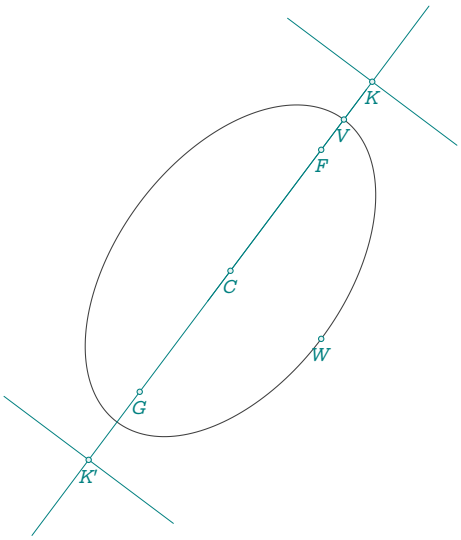
```
\directlua{
 init_elements()
 z.F = point(1, -1)
 z.G = point(4, 3)
 z.M = point(2, 4)
 z.C = tkz.midpoint(z.F, z.G)
 local a = (tkz.length(z.F, z.M) + tkz.length(z.G, z.M)) / 2
 CO.EL = conic(EL_bifocal(z.F, z.G, z.M))
 PA.curve = CO.EL:points(0, 1, 100)
 L.dir = CO.EL.directrix
 z.K = CO.EL.K
 z.Kp = z.C:symmetry(z.K)
 z.u, z.v = CO.EL.minor_axis:get()
 z.r, z.s = CO.EL.directrix:get()}
\begin{tikzpicture}[scale = .5]
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.curve)
 \tkzDrawLines[add = .5 and .5](K,K' u,v r,s)
 \tkzDrawSegments[dashed](M,F M,G)
 \tkzDrawPoints(C,F,K,K',G,M)
 \tkzLabelPoints(C,F,K,K',G,M)
\end{tikzpicture}
```



16.5.17. Function EL_points(pt, pt, pt)

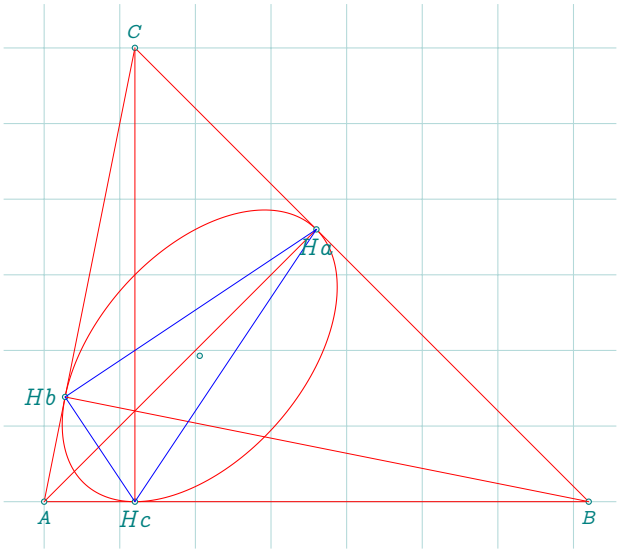
The second method corresponds to the classical approach based on the center, a vertex, and a covertex of the ellipse. The function **EL_points** takes these three points as arguments and computes all necessary attributes of the ellipse.
This approach replaces older constructions that manually derived parameters step by step—those lines have now been condensed into this single function.

```
\directlua{
 init_elements()
 z.C = point(1, -1)
 z.V = point(4, 3)
 z.W = (z.C - z.V):orthogonal(3):at(z.C)
 local a = tkz.length(z.C, z.V)
 local b = tkz.length(z.C, z.W)
 local c = math.sqrt(a ^ 2 - b ^ 2)
 local e = c / a
 axis = line(z.C, z.V)
   % foci
 z.F = axis:report(c, z.C)
 z.G = z.C:symmetry(z.F)
   % directrix
 z.K = axis:report(b ^ 2 / c, z.F)
 z.Kp = axis:report(-b ^ 2 / c, z.G)
   % major_axis
 z.u = (z.C - z.K):orthogonal(2):at(z.K)
 z.v = (z.C - z.K):orthogonal(-2):at(z.K)
 L.dir = line(z.u, z.v)
   % axis:ortho_from (z.K)
 z.r = (z.C - z.Kp):orthogonal(2):at(z.Kp)
 z.s = (z.C - z.Kp):orthogonal(-2):at(z.Kp)
   % CO      = conic(z.F,L.dir,e)
 CO.EL = conic(EL_points(z.C, z.V, z.W))
 PA.curve = CO.EL:points(0, 1, 100)}
\begin{tikzpicture}[gridded]
\tkzGetNodes
\tkzDrawCoordinates[smooth](PA.curve)
\tkzDrawLines(u,v r,s K,K')
\tkzDrawLine(C,V)
\tkzDrawPoints(V,W,C,F,K,K',G)
\tkzLabelPoints(V,W,C,F,K,K',G)
\end{tikzpicture}
```



16.5.18. Function EL_radii(pt, ra, rb, slope)

This function defines an ellipse from its center (**pt**), its two radii (ra and rb), and the slope of its principal axis. Unlike the previous method, where the inclination was implicitly determined from the positions of the center and a vertex, here the slope must be explicitly provided as an argument.
This approach is useful when the ellipse is defined by its geometric parameters rather than specific points on the curve.




```
\directlua{
 z.A = point(0, 0)
 z.B = point(6, 0)
 z.C = point(1, 5)
 T.ABC = triangle(z.A, z.B, z.C)
 z.H = T.ABC.orthocenter
 z.O = T.ABC.circumcenter
 T.orthic = T.ABC:orthic()
 z.K = T.ABC:symmedian_point()
 z.Ha, z.Hb, z.Hc = T.orthic:get()
 z.a, z.b, z.c = T.ABC:tangential():get()
 z.p, z.q, z.r = T.ABC:circumcevian(z.H):get()
 z.Sa, z.Sb = z.K:symmetry(z.Ha,z.Hb)
 local coefficients = search_ellipse("Ha", "Hb", "Hc", "Sa", "Sb")
 local center, ra, rb, angle = ellipse_axes_angle(coefficients)
 CO.EL = conic(EL_radii(center, ra, rb, angle))
 % or CO.EL = conic(EL_radii(ellipse_axes_angle(coefficients)))
 PA.curve = CO.EL:points(0, 1, 100)}

\begin{center}
  \begin{tikzpicture}[gridded,scale =1.20]
    \tkzGetNodes
    \tkzDrawCoordinates[smooth,red](PA.curve)
    \tkzDrawPolygons[red](A,B,C)
    \tkzDrawPoints(A,B,C,K,Ha,Hb,Hc)
    \tkzDrawSegments[red](C,Hc B,Hb A,Ha)
    \tkzDrawPolygons[blue](Ha,Hb,Hc)
    \tkzLabelPoints(A,B)
    \tkzLabelPoints[above](C)
    \tkzLabelPoints[font=\small,left](Hb)
    \tkzLabelPoints[font=\small](Hc)
    \tkzLabelPoints[font=\small](Ha)
  \end{tikzpicture}
\end{center}
```

16.5.19. Function search_ellipse(s1, s2, s3, s4, s5)

This function, which will eventually be renamed **five_points**, is currently limited to ellipses.

Given five points lying on an ellipse, it computes the coefficients of the general conic equation

$$f(x,y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F,$$

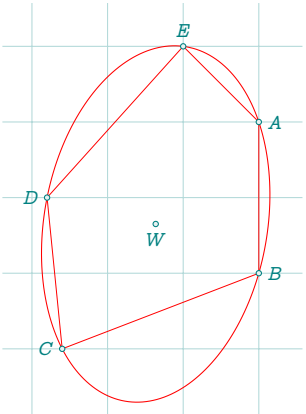
by solving a linear system using Gauss–Jordan elimination. The sixth coefficient F is fixed to 1, reducing the number of unknowns to five.
The arguments are the *names* of the points (as strings), not the points themselves. The method builds a linear system where each point satisfies the equation above, then solves it to find the values of A, B, C, D, E.
The results are stored in a table, indexed by the point names. You can retrieve the coefficients manually, as shown in the example, but functions using this output can also directly access the table, making it easier to work with.

A = 4.047619047619
B = -5.25
C = 3.6904761904762
D = -4.1190476190476
E = -6.5
F = 1

```
\directlua{
 z.A = point(4, 3)
 z.D = point(0.2, 2)
 z.C = point(0.4, 0)
 z.B = point(3, 1.5)
 z.E = point(2, 4)
 local coefficients = search_ellipse("A", "B", "C", "D", "E")
 local A, B, C, D, E, F =
     coefficients.A, coefficients.B,
     coefficients.C, coefficients.D,
     coefficients.E, coefficients.F

 tex.print(" A = "..A)
 tex.print('\\\\')
 tex.print("B = "..B)
 tex.print('\\\\')
 tex.print("C = "..C)
 tex.print('\\\\')
 tex.print("D = "..D)
 tex.print('\\\\')
 tex.print("E = "..E)
 tex.print('\\\\')
 tex.print("F = "..F)}
```

16.5.20. Function ellipse_axes_angle(t)

This function complements the previous one. Once the general equation of the ellipse has been determined, it becomes necessary to extract key geometric characteristics—such as the center, the radii, and the inclination of the principal axis—in order to work with the **ellipse** object.
To retrieve these attributes, simply pass the table of coefficients (as returned by **search_ellipse**) to this function. It will compute and return all the necessary parameters for further use.




```
\directlua{
 init_elements()
 z.A = point(3, 3)
 z.D = point(0.2, 2)
 z.C = point(0.4, 0)
 z.B = point(3, 1)
 z.E = point(2, 4)
 local coefficients = search_ellipse("A", "B", "C", "D", "E")
 local center, ra, rb,
 angle = ellipse_axes_angle(coefficients)
 z.W = center
 CO.EL = conic(EL_radii(center, ra, rb, angle))
 PA.curve = CO.EL:points(0, 1, 100)}
```

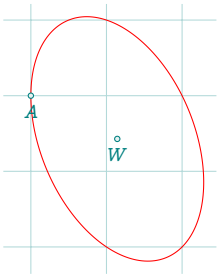
16.5.21. Function test_ellipse(pt, t)

This function checks whether a given point **pt** lies on the ellipse whose general equation is defined by the coefficients stored in table **t**.
It evaluates the equation

$$f(x,y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F$$

at the coordinates of the point, using the values from the table **t**.
This is particularly useful for verifying whether a point satisfies the equation of a previously computed ellipse. For example, the function can be used to test a point against the ellipse defined by the equation

$$2x^2 + xy + y^2 - 5x - 2y + 1 = 0.$$

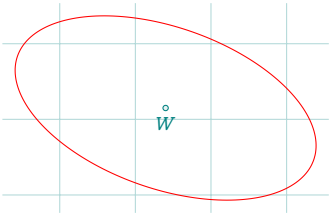


A on the ellipse ?yes

```
\directlua{
 init_elements()
 coefficients = {A = 2, B = 1, C = 1, D = -5, E = -
2, F = 1}
 z.A = point(0, 1)
 if test_ellipse(z.A,coefficients)==0 then
   answer = "yes" end
 tex.print("A on the ellipse ?"..answer)
 local center, ra, rb,
 angle = ellipse_axes_angle(coefficients)
 z.W = center
 CO.EL = conic(EL_radii(center, ra, rb, angle))
 PA.curve = CO.EL:points(0, 1, 100)}
```

16.5.22. Function search_center_ellipse(t)

This function computes the center of an ellipse from its general equation. The argument **t** is a table containing the coefficients A, B, C, D, E, F of the conic equation. The center is obtained by solving the system of equations corresponding to the partial derivatives $\partial f/\partial x = 0$ and $\partial f/\partial y = 0$.




```
\directlua{
 coefficients = {A = .75, B = 1, C = 2, D = -5, E = -4, F = 4}
 local center = search_center_ellipse(coefficients)
 z.W = center
 local c, ra, rb,
 angle = ellipse_axes_angle(coefficients)
 CO.EL = conic(EL_radii(center, ra, rb, angle))
 cPA.urve = CO.EL:points(0, 1, 100)}
```

16.5.23. Method asymptotes()

This method computes the asymptotes of a conic when the conic is a *hyperbola*. For other types of conics (ellipse or parabola), asymptotes are not defined and an error is raised.

`Description:` This method computes the asymptotes of a conic when the conic is a *hyperbola*. For other types of conics (ellipse or parabola), asymptotes are not defined and an error is raised.

Let $\mathcal{H}$ be a hyperbola with center O, semi-major axis a and semi-minor axis b. The asymptotes of $\mathcal{H}$ are the two straight lines passing through the center O and having slopes $\pm\frac{b}{a}$ with respect to the principal axis.
The method constructs these asymptotes geometrically:

- a point P is computed on the transverse axis of the hyperbola,
- two directions orthogonal to the principal axis and scaled by b are constructed at P,
- the corresponding points are symmetrized with respect to the center in order to define two straight lines.

Syntax:

```
L.a1, L.a2 = C:asymptotes()
```

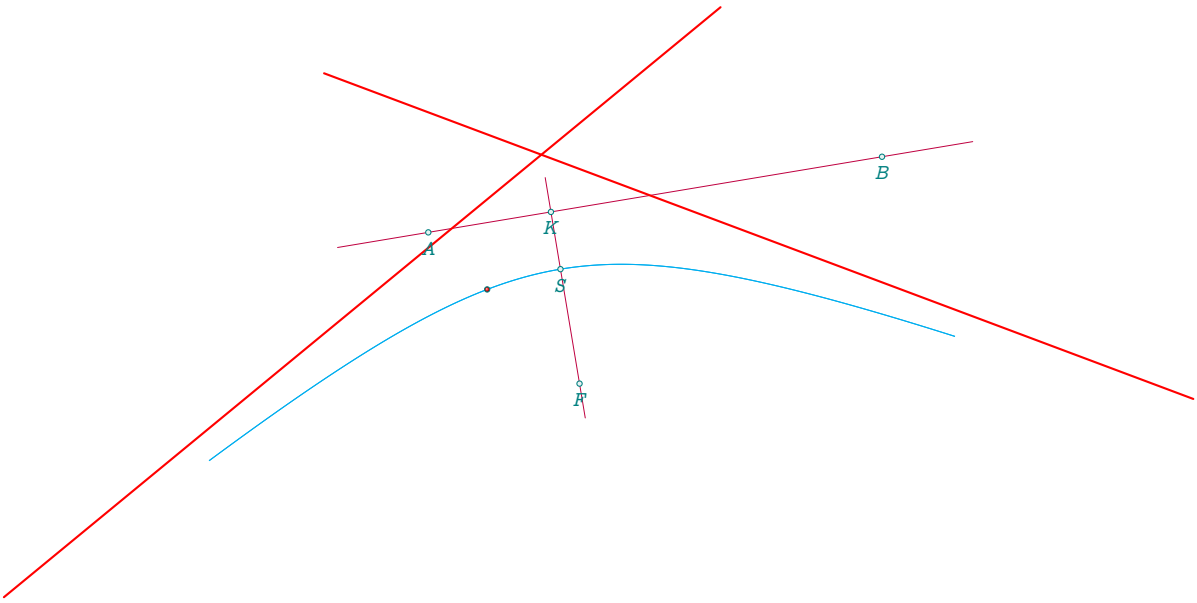
Parameters: None.

Return values:
If the conic is an hyperbola, the method returns two objects of type line:

- L.a1: first asymptote,
- L.a2: second asymptote.

If the conic is not an hyperbola, an error is raised.

Example:




```
\directlua{
  init_elements()
  z.A = point(-2, -1)
  z.B = point(4, 0)
  L.AB = line(z.A, z.B)
  z.F = point(0, -3)
  CO.HY  = conic(z.F, L.AB, 2)
  curve  = CO.HY:points(-5, 5, 50)
  curveb = CO.HY:points(-5, 5, 50, swap)
  z.K = CO.HY.K
  z.S = CO.HY.vertex
  z.O = CO.HY.center
  z.X = CO.HY:point(1)
  L.A1, L.A2 = CO.HY:asymptotes()
  z.u1, z.v1 = L.A1:get()
  z.u2, z.v2 = L.A2:get()
}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCoordinates[smooth,cyan](curve)
\tkzDrawCoordinates[smooth,cyan](curveb)
\tkzDrawPoint[mark=ball,ball color=red](X)
\tkzDrawLines[purple](A,B F,K)
\tkzDrawLines[red,thick,add=0 and 1](u1,v1 u2,v2)
\tkzDrawPoints(A,B,F,K,S)
\tkzLabelPoints(A,B,F,K,S)
\end{tikzpicture}
```

Remark: The asymptotes are returned as infinite straight lines. They are independent of the chosen parametrization of the hyperbola and depend only on its geometric definition.

17. Class quadrilateral

The variable Q holds a table used to store quadrilaterals. It is optional, and you are free to choose the variable name. However, using **Q** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., Quad), you must initialize it manually. The **init_elements()** function reinitializes the **Q** table if used.

17.1. Creating a quadrilateral

The **quadrilateral** class requires four points. The order defines the sides of the quadrilateral.

The object is usually stored in Q, which is the recommended variable name.

```
Q.ABCD = quadrilateral:new(z.A, z.B, z.C, z.D)
```

Short form.

A shorthand constructor is also available:

```
Q.ABCD = quadrilateral(z.A, z.B, z.C, z.D)
```

17.2. Quadrilateral Attributes

Points are created in the direct direction. A test is performed to check whether the points form a rectangle, otherwise compilation is blocked.

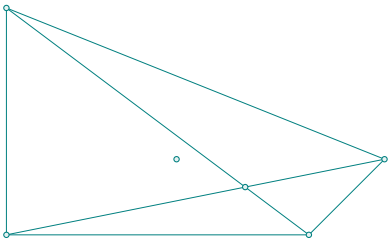
Creation: Q.new = rectangle:new(z.A,z.B,z.C,z.D)

Table 18: rectangle attributes.

Attributes	Application	
pa	z.A = Q.new.pa	
pb	z.B = Q.new.pb	
pc	z.C = Q.new.pc	
pd	z.D = Q.new.pd	
type	Q.new.type= 'quadrilateral'	
center	z.I = Q.new.center	intersection of diagonals
g	z.G = Q.new.g	barycenter
a	AB = Q.new.a	barycenter
b	BC = Q.new.b	barycenter
c	CD = Q.new.c	barycenter
d	DA = Q.new.d	barycenter
ab	Q.new.ab	line passing through two vertices
ac	Q.new.ca	idem.
ad	Q.new.ad	idem.
bc	Q.new.bc	idem.
bd	Q.new.bd	idem.
cd	Q.new.cd	idem.

17.2.1. Quadrilateral attributes

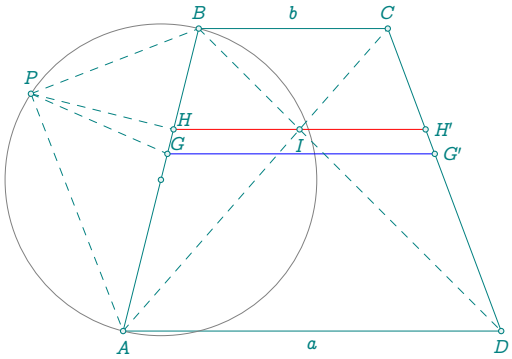
```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(4, 0)
z.C = point(5, 1)
z.D = point(0, 3)
Q.ABCD = quadrilateral(z.A, z.B, z.C, z.D)
z.I = Q.ABCD.i
z.G = Q.ABCD.g}

\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawSegments(A,C B,D)
\tkzDrawPoints(A,B,C,D,I,G)
\end{tikzpicture}
```



17.2.2. Quadrilateral examples

Advanced Euclidean Geometry 2013 Supplement June 26

Construction of geometric mean:



Proof. A trapezoid with parallel sides AD and BC is given.
Let H be the endpoint of the parallel through I to the side AD. $HH' = \dfrac{2ab}{a+b}$.
PH is perpendicular to AB
(PG) is the bisector of $\widehat{APB}$

$$GG' = \frac{a\cdot\sqrt{b}+b\cdot\sqrt{a}}{\sqrt{a}+\sqrt{b}} = \frac{\sqrt{ab}(\sqrt{a}+\sqrt{b})}{\sqrt{a}+\sqrt{b}} = \sqrt{ab}$$

□

Code:

```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(1, 4)
z.C = point(3.5, 4)
z.D = point(5, 0 )
Q.ABCD = quadrilateral(z.A, z.B, z.C, z.D)
z.I = Q.ABCD.center
L.ll = Q.ABCD.da:ll_from (z.I)
z.H = intersection(L.ll, Q.ABCD.ab)
z.Hp = intersection(L.ll, Q.ABCD.cd)
```

```
z.M =  Q.ABCD.ab.mid
C.MA = circle(z.M,z.A)
L.perp = Q.ABCD.ab:ortho_from(z.H)
_,z.P = intersection(C.MA, L.perp)
T.PAB = triangle(z.P, z.A, z.B)
L.bis =  T.PAB:bisector()
z.G = L.bis.pb
L.llg = Q.ABCD.da:ll_from(z.G)
L.llm = Q.ABCD.da:ll_from(z.M)
z.Gp = intersection(L.llg, Q.ABCD.cd)}
```

17.3. Quadrilateral methods

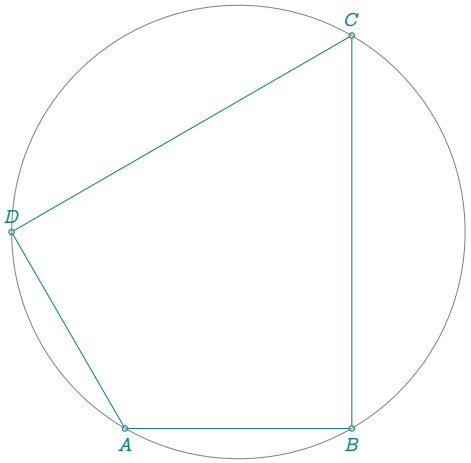
Table 19: Quadrilateral methods.

Methods	Reference
new()	Note[10]; [17.1]
is_cyclic ()	[17.3.1]
is_convex ()	[17.3.2]
poncelet_point()	[17.3.3]

17.3.1. Method is_cyclic()

Inscribed quadrilateral

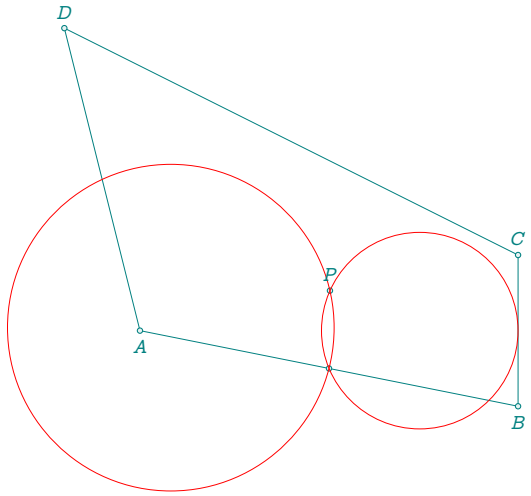
```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.D = point:polar(4, tkz.tau / 3)
 L.DB = line(z.D, z.B)
 T.equ = L.DB:equilateral()
 z.C = T.equ.pc
 Q.new = quadrilateral(z.A, z.B, z.C, z.D)
 bool = Q.new:is_cyclic()
 if bool == true then
  C.cir = triangle(z.A, z.B, z.C):
   circum_circle()
  z.O = C.cir.center
 end}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C,D)
  \tkzDrawPoints(A,B,C,D)
  \tkzDrawCircle(O,A)
  \ifthenelse{\equal{\tkzUseLua{bool}}{
     true}}{\tkzDrawCircle(O,A)}{}
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C,D)
\end{tikzpicture}
```



17.3.2. Method is_convex()

17.3.3. Method poncelet_point

See [14.6.27] for the definition.




```
\directlua{
 init_elements()
 z.A = point(1, 1)
 z.B = point(6, 0)
 z.D = point(0, 5)
 z.C = point(6, 2)
 Q.ABCD = quadrilateral(z.A, z.B, z.C, z.D)
 z.P = Q.ABCD:poncelet_point()
 T.ABC = triangle(z.A,z.B,z.C)
 z.I = T.ABC.eulercenter
 z.Mc = tkz.midpoint(z.A, z.B)
 T.ABD = triangle(z.A, z.B, z.D)
 z.I1 = T.ABD.eulercenter}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygons(A,B,C,D)
  \tkzDrawPoints(A,B,C,D,P,Mc)
  \tkzDrawCircles[red](I,Mc I1,Mc)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C,D,P)
\end{tikzpicture}
\end{center}
```

18. Class square

The variable S holds a table used to store squares. It is optional, and you are free to choose the variable name. However, using **S** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., Squares), you must initialize it manually. The **init_elements()** function reinitializes the **S** table if used.

18.1. Creating a square

The **square** class constructs a square from two adjacent vertices. The order of the points defines the orientation.

The result is stored in S.

```
S.ABCD = square:new(z.A, z.B)
```

Short form.
The short form **square(z.A, z.B)** is equivalent:

```
S.ABCD = square(z.A, z.B)
```

18.2. Square attributes

Points are created in the direct direction. A test is performed to check whether the points form a square. Otherwise, compilation is blocked."

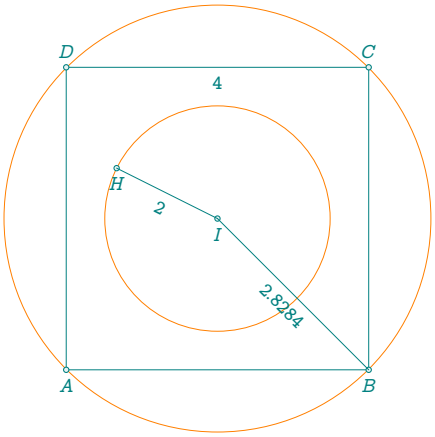
```
Creation  S.AB = square(z.A,z.B,z.C,z.D)
```

Table 20: Square attributes.

Attributes	Application	
pa	z.A = S.AB.pa	
pb	z.B = S.AB.pb	
pc	z.C = S.AB.pc	
pd	z.D = S.AB.pd	
type	S.AB.type= 'square'	
side	s = S.AB.center	s = length of side
center	z.I = S.AB.center	center of the square
circumradius	S.AB.circumradius	radius of the circumscribed circle
inradius	S.AB.inxradius	radius of the inscribed circle (apothem)
apothem_foot	S.AB.proj	projection of the center on one side
ab	S.AB.ab	line passing through two vertices
ac	S.AB.ca	idem.
ad	S.AB.ad	idem.
bc	S.AB.bc	idem.
bd	S.AB.bd	idem.
cd	S.AB.cd	idem.

18.2.1. Example: square attributes

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.C = point(4, 4)
 z.D = point(0, 4)
 S.new = square(z.A, z.B, z.C, z.D)
 z.I = S.new.center
 z.H = S.new.proj}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCircles[orange](I,A I,H)
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D,H,I)
\tkzLabelPoints(A,B,H,I)
\tkzLabelPoints[above](C,D)
\tkzDrawSegments(I,B I,H)
\tkzLabelSegment[sloped](I,B){%
  \pmpn{\tkzUseLua{S.new.circumradius}}}
\tkzLabelSegment[sloped](I,H){%
  \pmpn{\tkzUseLua{S.new.inradius}}}
\tkzLabelSegment[sloped](D,C){%
  \pmpn{\tkzUseLua{S.new.side}}}
\end{tikzpicture}
```



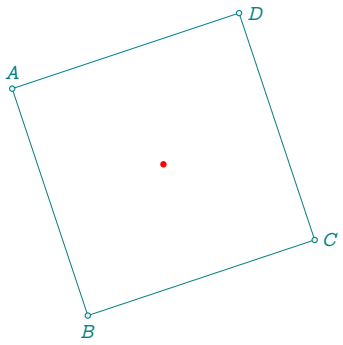
18.3. Square Methods and Functions

Table 21: Square methods

Methods	Reference
new(za,zb,zc,zd)	Note[11];[18.3.1]
square.by_rotation (zi,za)	[18.3.1]
square.from_side(za,zb,"swap")	18.3.2

18.3.1. Function square.by_rotation(pt,pt)

I square center; *A* first vertex

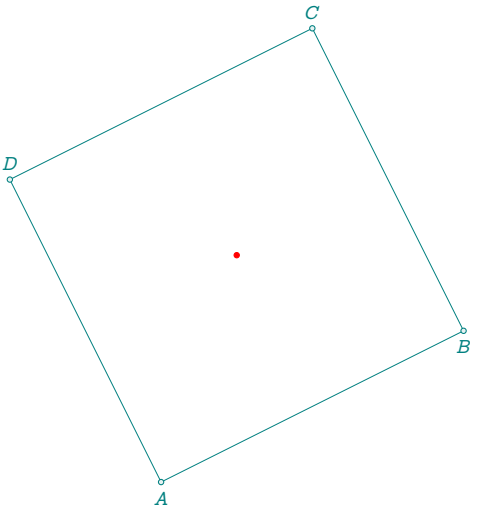



```
\directlua{
 z.A  = point(0, 0)
 z.I  = point(2, -1)
 S    = square.by_rotation(z.I, z.A)
 z.B  = S.pb
 z.C  = S.pc
 z.D  = S.pd
 z.I  = S.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(B)
\tkzLabelPoints[above](A)
\tkzLabelPoints[right](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```

18.3.2. Method square.from_side(za,zb)

With the option **"swap"** then the square is defined in counterclockwise. The result can also be obtained from a line [12.12.1].

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(2, 1)
 S.side = square.from_side(z.A, z.B)
 z.B = S.side.pb
 z.C = S.side.pc
 z.D = S.side.pd
 z.I = S.side.center}
\begin{tikzpicture}[scale = 2]
   \tkzGetNodes
   \tkzDrawPolygon(A,B,C,D)
   \tkzDrawPoints(A,B,C,D)
   \tkzLabelPoints(A,B)
   \tkzLabelPoints[above](C,D)
   \tkzDrawPoints[red](I)
\end{tikzpicture}
```



19. Class rectangle

The variable **R** holds a table used to store triangles. It is optional, and you are free to choose the variable name. However, using **R** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., rectangles), you must initialize it manually. The **init_elements()** function reinitializes the **R** table if used.

19.1. Rectangle attributes

Points are created in the direct direction. A test is performed to check whether the points form a rectangle, otherwise compilation is blocked.

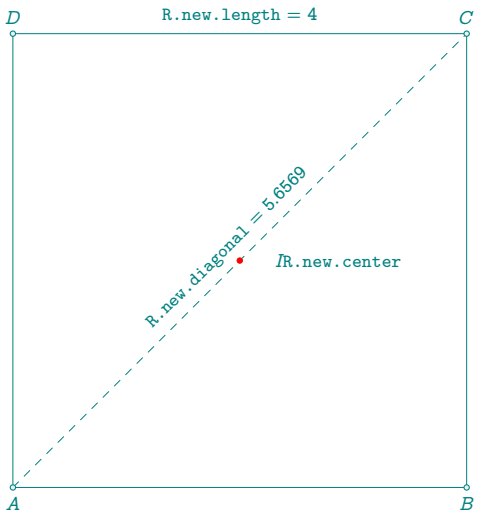
Creation `R.ABCD = rectangle:new(z.A, z.B, z.C, z.D)`

Table 22: rectangle attributes.

Attributes	Application	
pa	z.A = R.ABCD.pa	
pb	z.B = R.ABCD.pb	
pc	z.C = R.ABCD.pc	
pd	z.D = R.ABCD.pd	
type	R.ABCD.type= 'rectangle'	
center	z.I = R.ABCD.center	center of the rectangle
length	R.ABCD.length	the length
width	R.ABCD.width	the width
diagonal	R.ABCD.diagonal	diagonal length
ab	R.ABCD.ab	line passing through two vertices
ac	R.ABCD.ca	idem.
ad	R.ABCD.ad	idem.
bc	R.ABCD.bc	idem.
bd	R.ABCD.bd	idem.
cd	R.ABCD.cd	idem.

19.1.1. Example

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.C = point(4, 4)
 z.D = point(0, 4)
 R.new = rectangle(z.A, z.B, z.C, z.D)
 z.I = R.new.center}

\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```



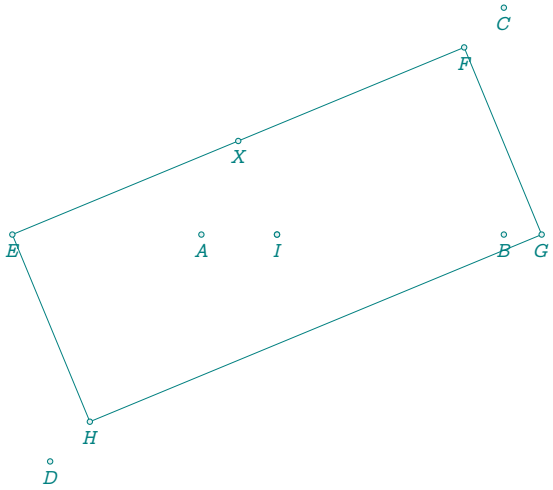
19.2. Rectangle methods

Table 23: Rectangle methods.

Methods	Reference
new(za ,zb, zc, zd)	Note[12];[19.2.1]
angle (zi, za, angle)	[19.2.2]
gold (za, zb)	[19.2.5]
diagonal (za, zc)	[19.2.4]
side (za, zb, d)	[19.2.3]
get_lengths ()	[19.2.6]

19.2.1. Method new(pt,pt,pt,pt)

This function creates a square using four points. No test is performed, and verification is left to the user.

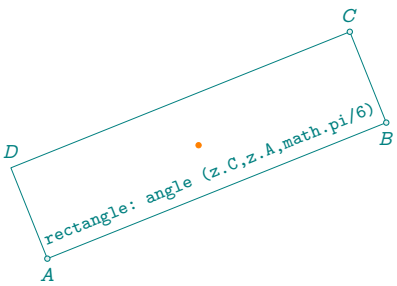



```
\directlua{
z.A = point(0, 0)
z.B = point(4, 0)
z.C = point(4, 3)
z.D = point(-2, -3)
L.AB = line(z.A,z.B)
L.CD = line(z.C,z.D)
z.I = intersection(L.AB, L.CD)
C.I = circle(through(z.I, 3.5))
z.G,z.E = intersection(L.AB, C.I)
z.F,z.H = intersection(L.CD, C.I)
R.I = rectangle(z.E, z.F, z.G, z.H)
z.X = R.I.ab:projection(z.I)}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(E,F,G,H)
\tkzDrawPoints(A,B,C,D,I,E,F,G,H,X)
\tkzLabelPoints(A,B,C,D,I,E,F,G,H,X)
\tkzDrawPoints(I)
\end{tikzpicture}
```

19.2.2. Method angle(pt,pt,an)

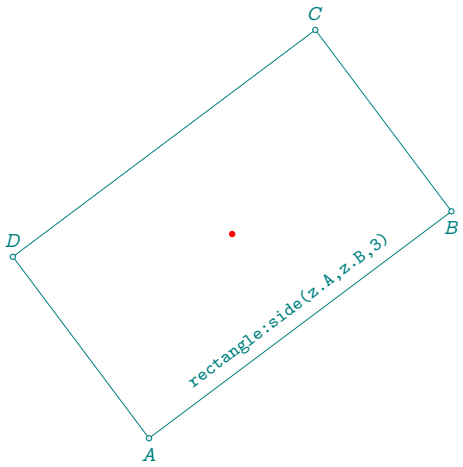
R.ang = rectangle:angle(z.I,z.A) ; z.A vertex ; ang angle between 2 vertices

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 0)
 z.I = point(4, 3)
 P.ABCD = rectangle:angle(z.I, z.A,
          math.pi / 6)
 z.B = P.ABCD.pb
 z.C = P.ABCD.pc
 z.D = P.ABCD.pd}
\begin{tikzpicture}[scale   = .5]
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C)
\tkzLabelPoints(A,B,C,D)
\tkzDrawPoints[new](I)
\end{tikzpicture}
```



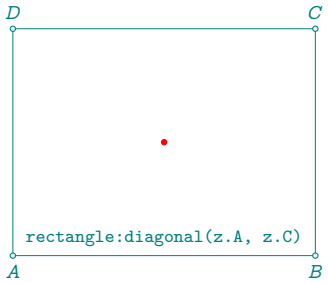
19.2.3. Method side(pt,pt,d)

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 3)
 R.side = rectangle:side(z.A, z.B, 3)
 z.C = R.side.pc
 z.D = R.side.pd
 z.I = R.side.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```



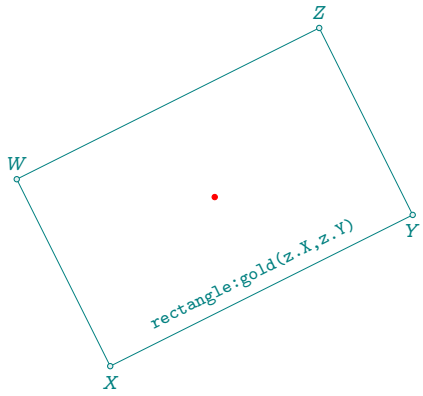
19.2.4. Method diagonal(pt,pt)

```
\directlua{
init_elements()
z.A = point(0, 0)
z.C = point(4, 3)
R.diag = rectangle:diagonal(z.A, z.C)
z.B = R.diag.pb
z.D = R.diag.pd
z.I = R.diag.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\tkzLabelSegment[sloped,above](A,B){%
    |rectangle:diagonal(z.A,z.C)|}
\end{tikzpicture}
```

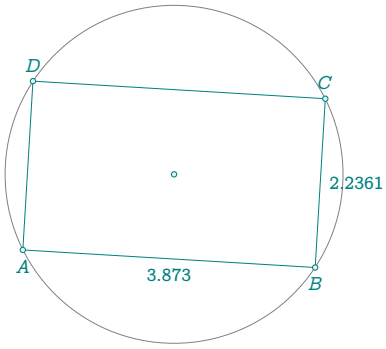


19.2.5. Method gold(pt,pt)

```
\directlua{
 init_elements()
 z.X = point(0, 0)
 z.Y = point(4, 2)
 R.gold = rectangle:gold(z.X, z.Y)
 z.Z = R.gold.pc
 z.W = R.gold.pd
 z.I = R.gold.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(X,Y,Z,W)
\tkzDrawPoints(X,Y,Z,W)
\tkzLabelPoints(X,Y)
\tkzLabelPoints[above](Z,W)
\tkzDrawPoints[red](I)
\tkzLabelSegment[sloped,above](X,Y){%
|rectangle:gold(z.X,z.Y)|}
\end{tikzpicture}
```



19.2.6. Method get_lengths()




```
\directlua{
 init_elements()
 z.I = point(2, 1)
 z.A = point(0, 0)
 R.ABCD = rectangle:angle(z.I, z.A, math.pi / 3)
 z.B = R.ABCD.pb
 z.C = R.ABCD.pc
 z.D = R.ABCD.pd
 tkzx,tkzy = R.ABCD:get_lengths()}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawCircle(I,A)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[](I)
\tkzLabelSegment(A,B){%
    $\pmpn{\tkzUseLua{tkzx}}$}
\tkzLabelSegment[right](B,C){%
    $\pmpn{\tkzUseLua{tkzy}}$}
\end{tikzpicture}
```

20. Class parallelogram

The variable P holds a table used to store parallelograms. It is optional, and you are free to choose the variable name. However, using **P** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., parall), you must initialize it manually. The **init_elements()** function reinitializes the **P** table if used.

20.1. Creating a parallelogram

The **parallelogram** class creates a parallelogram using three points. The fourth vertex is computed automatically.

The resulting object is stored in P. You are free to use another name, but P is preferred for consistency.

```
P.ABCD = parallelogram:new(z.A, z.B, z.C)
```

Short form.
You may also use the short form:

```
P.ABCD = parallelogram(z.A, z.B, z.C)
```

20.2. Parallelogram attributes

Points are created in the direct direction. A test is performed to check whether the points form a parallelogram, otherwise compilation is blocked.

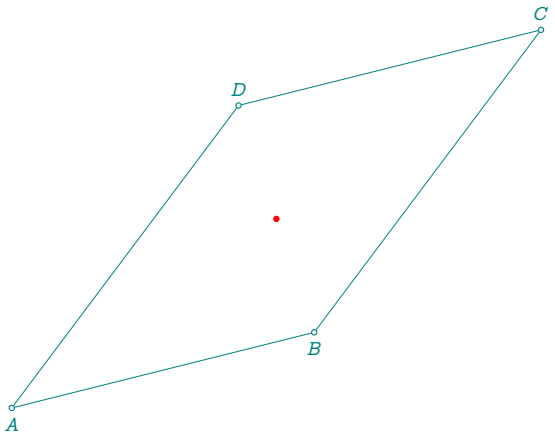
```
Creation  P.new = parallelogram(z.A,z.B,z.C,z.D)
```

Table 24: Parallelogram attributes.

Attributes	Application	
pa	z.A = P.new.pa	
pb	z.B = P.new.pb	
pc	z.C = P.new.pc	
pd	z.D = P.new.pd	
type	P.new.type= 'parallelogram'	
center	z.I = P.new.center	intersection of diagonals
ab	P.new.ab	line passing through two vertices
ac	P.new.ca	idem.
ad	P.new.ad	idem.
bc	P.new.bc	idem.
bd	P.new.bd	idem.
cd	P.new.cd	idem.

20.2.1. Example: attributes

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, 1)
 z.C = point(7, 5)
 z.D = point(3, 4)
 P.new = parallelogram(z.A, z.B, z.C, z.D)
 z.B = P.new.pb
 z.C = P.new.pc
 z.D = P.new.pd
 z.I = P.new.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```

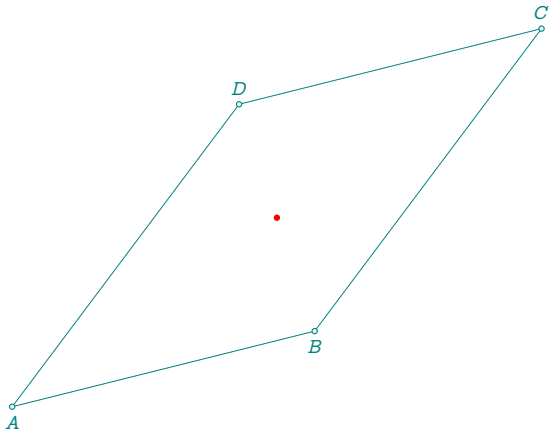


20.3. Parallelogram functions

Table 25: Parallelogram functions.

Functions	Reference
new (za, zb, zc, zd)	20.2.1
parallelogram.fourth (za,zb,zc)	20.3.2

20.3.1. Method new(pt,pt,pt,pt)

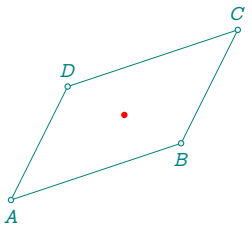



```
\directlua{
 z.A = point(0, 0)
 z.B = point(4, 1)
 z.C = point(7, 5)
 z.D = point(3, 4)
 P.ABCD = parallelogram(z.A, z.B, z.C, z.D)
 z.B = P.ABCD.pb
 z.C = P.ABCD.pc
 z.D = P.ABCD.pd
 z.I = P.ABCD.center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```

20.3.2. Method fourth(pt,pt,pt)

completes a triangle by parallelogram (See next example)

```
\directlua{
init_elements()
z.A = point(0, 0)
z.B = point(3, 1)
z.C = point(4, 3)
P.four= parallelogram.fourth(z.A, z.B, z.C)
z.D = P.four.pd
z.I = P.four.center}
\begin{tikzpicture}[ scale = .75]
\tkzGetNodes
\tkzDrawPolygon(A,B,C,D)
\tkzDrawPoints(A,B,C,D)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C,D)
\tkzDrawPoints[red](I)
\end{tikzpicture}
```



21. Class regular polygon

The variable RP holds a table used to store regular polygons. It is optional, and you are free to choose the variable name. However, using **RP** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., REP), you must initialize it manually. The **init_elements()** function reinitializes the **RP** table if used.

21.1. Creating a regular polygon

The **regular_polygon** class builds a regular polygon from its center, a first vertex, and the number of sides.

The result is usually stored in P, a variable used to group polygonal objects.

```
P.hex = regular_polygon:new(z.O, z.A, 6)
```

Short form.
Use the short form for brevity:

```
P.hex = regular_polygon(z.O, z.A, 6)
```

21.2. Regular_polygon attributes

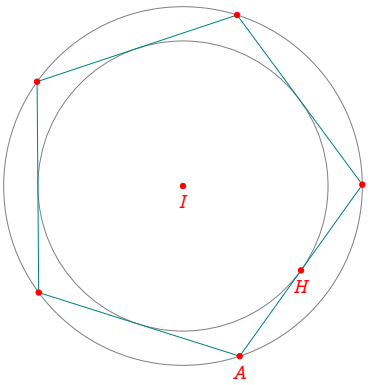
```
Creation  RP.IA = regular_polygon(z.I,z.A,6)
```

Table 26: Regular_polygon attributes.

Attributes	Application
center	z.I = RP.IA.center
vertices	array containing all vertex affixes
through	first vertex
circle	defines the circle with center I passing through A
type	RP.IA.type= 'regular_polygon'
side	s = RP.IA.side ; s = length of side
circumradius	S.AB.circumradius ; radius of the circumscribed circle
inradius	S.AB.inxradius ; radius of the inscribed circle
apothem	RP.IA.apothem ; projection of the center on one side
angle	RP.IA.angle ; angle formed by the center and 2 consecutive vertices

21.2.1. Pentagon

```
\directlua{
init_elements()
z.O = point(0, 0)
z.I = point(1, 3)
z.A = point(2, 0)
RP.five  = regular_polygon(z.I, z.A, 5)
RP.five:name("P_")
C.ins  = circle:radius(z.I,
             RP.five.inradius)
z.H = RP.five.apothem
}
\begin{tikzpicture}[scale = .75]
\def\nb{\tkzUseLua{RP.five.nb}}
\tkzGetNodes
\tkzDrawCircles(I,A I,H)
\tkzDrawPolygon(P_1,P_...,P_\nb)
\tkzDrawPoints[red](P_1,P_...,P_\nb,H,I)
\tkzLabelPoints[red](I,A,H)
\end{tikzpicture}
```

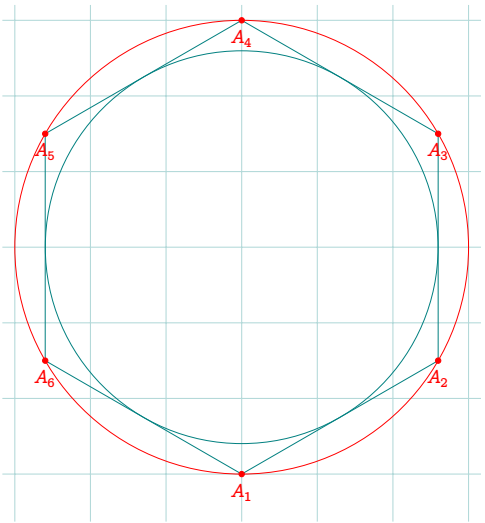


21.3. Regular_polygon methods

Table 27: regular_polygon methods.

Constructor	
new(O,A,n)	[21.3.1]
Methods Returning a Circle	
incircle ()	[21.3.2]
Methods Returning a Point	
name (string)	[21.3.3]

21.3.1. Method new(pt, pt, n)




```
\directlua{
  z.A = point(0, -4)
  z.O = point(0, 0)
  RP.six = regular_polygon(z.O, z.A, 6)
  RP.six:name("A_")
  z.i, z.p = RP.six:incircle():get()}

\begin{tikzpicture}[gridded,scale=.75]
  \tkzGetNodes
  \tkzDrawCircles[red](O,A)
  \tkzDrawCircles[teal](i,p)
  \tkzDrawPolygon(A_1,A_...,A_6)
  \tkzDrawPoints[red](A_1,A_...,A_6)
  \tkzLabelPoints[red](A_1,A_...,A_6)
\end{tikzpicture}
```

21.3.2. Method incirle()

See previous example [21.3.1]

21.3.3. Method name(s)

See [21.3.1]

Part II.

Algebra and Tools

22. Class vector

A **vector** object represents an oriented segment from a tail point A to a head point B. Internally, it is built from two points and , interpreted as complex numbers. The vector stores both its geometric endpoints and its analytic data (components, length, direction).
The variable V holds a table used to store vectors. It is optional, and you are free to choose the variable name. However, using **V** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., Vectors), you must initialize it manually. The **init_elements()** function reinitializes the **V** table if used.

In fact, they are more a class of oriented segments than vectors in the strict mathematical sense.
A vector is defined by giving two points (i.e. two affixes). `V.AB = vector(z.A, z.B)` creates the vector $(\overrightarrow{AB})$, i.e. the oriented segment with origin A representing a vector. A few rudimentary operations are defined, such as sum, subtraction and multiplication by a scalar.

22.1. Creating a vector

The **vector** class represents a vector between two points or a free vector with given coordinates.

The result is usually stored in V.

Short form:

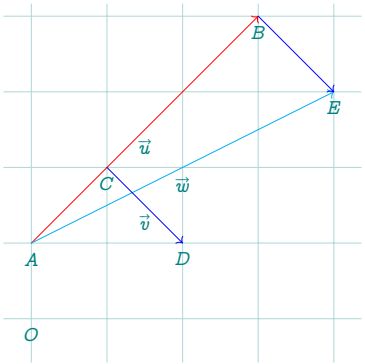
```
V.v1 = vector:new(z.A, z.B) -- from A to B
The short form is equivalent:
V.v1 = vector(z.A, z.B)
```

22.2. Attributes of a vector

Table 28: Vector attributes.

Attributes	Reference
tail	
head	[22.2.1]
type	[22.2.2]
slope	[22.2.3]
z	[22.2.7]
dx	[22.2.4]
dy	[22.2.4]
norm	[22.2.5]
mtx	[22.2.6]

22.2.1. Attribute head




```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = point(0, 1)
 z.B = point(3, 4)
 z.C = point(1, 2)
 z.D = point(2, 1)
 V.u = vector(z.A, z.B)
 V.v = vector(z.C, z.D)
 V.w = V.u + V.v
 z.E = V.w.head}
```

22.2.2. Attributes type

With previous data:

```
V.u.type = 'vector'
```

22.2.3. Attribute slope

The attribute gives the slope of the line supporting a vector representative.

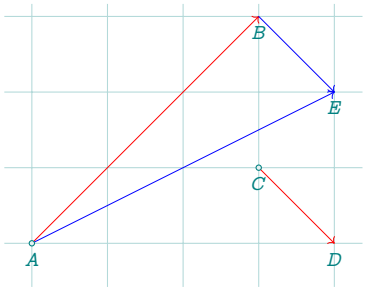
36.870

```
\directlua{
   z.A = point(-1, 0)
   z.B = point (3, 3)
   V.u = vector(z.A, z.B)
   local d = math.deg(V.u.slope)
   tex.print(utils.format_number(d,3))}
```

22.2.4. Attributes dx, dy

These attributes give the coordinates of the vectors in the reference base.

(dx,dy) =(4,2)




```
\directlua{
  z.A = point(0, 1)
  z.B = point(3, 4)
  z.C = point(3, 2)
  z.D = point(4, 1)
  V.u = vector(z.A, z.B)
  V.v = vector(z.C, z.D)
  V.w = V.u + V.v
  z.E = V.w.head
  local pc = string.char(37)
  local format = "("..pc.."g,"..pc.."g)"
  tex.print("(dx,dy) =
     ",string.format(format, V.w.dx, V.w.dy))
  }
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints(A,B,C,D,E)
  \tkzDrawSegments[->,red](A,B C,D)
  \tkzDrawSegments[->,blue](A,E B,E)
  \tkzDrawPoints(A,C)
\end{tikzpicture}
  \end{center}
```

22.2.5. Attribute norm

This attribute gives the length of the segment between the two ends.

norm =5.0

```
\directlua{
   z.A = point(-1, 0)
   z.B = point(3, 3)
   V.u = vector(z.A, z.B)
   V.d = V.u.norm
   tex.print("norm = ",V.d)}
```

22.2.6. Attribute mtx

This involves associating a matrix in the form of a **column vector** with the vector under consideration.

$$\begin{bmatrix} 1 \\ 2+i \end{bmatrix}\begin{bmatrix} 1 & 2+i \end{bmatrix}$$

```
\directlua{
   z.O = point(1, 0)
   z.I = point(2, 1)
   V.u = vector(z.O, z.I)
   V.u.mtx:print()
   V.v = V.u.mtx:transpose()
   V.v:print()}
```

22.2.7. Attribute z

This attribute is very useful for working with the `^` and `..` metamethods.

determinant(u,v) = 7
dot product(u,v) = 1

```
\directlua{
 z.A = point(1, 1)
 z.B = point(2, -1)
 z.C = point(0, 1)
 z.D = point(3, 2)
 V.u = vector(z.A, z.B)
 V.v = vector(z.C, z.D)
 n = V.u.z ^ V.v.z
 m = V.u.z .. V.v.z
 tex.print("determinant(u,v) = "..tostring(n))
 tex.print('\\\\')
 tex.print("dot product(u,v) = "..tostring(m))
}
```

22.3. Metamethods overview of the class vector

Table 29: Methods of the class vector.

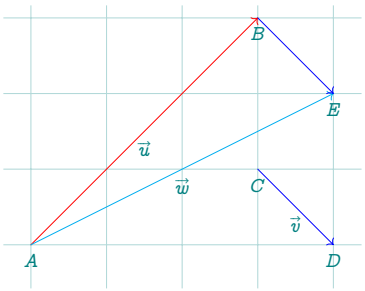
Metamethods	Reference
add(u,v)	[22.4.1]
sub(u,v)	[22.4.2]
unm(u)	[22.4.4]
mul(k,u)	[22.4.3]
pow(k,u)	[22.4.5]
concat(k,u)	[22.4.6]

22.4. Example of metamethods

22.4.1. Method add

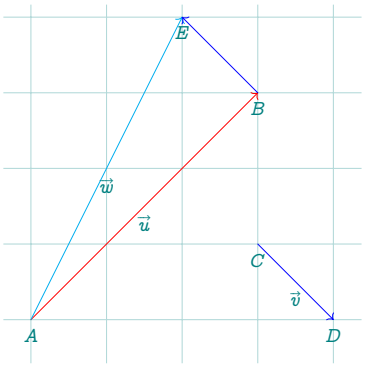
The sum is defined as follows:
Let V.AB + V.CD result in a vector V.AE defined as follows
If $\overrightarrow{CD} = \overrightarrow{BE}$ then $\overrightarrow{AB}+\overrightarrow{CD} = \overrightarrow{AB}+\overrightarrow{BE} = \overrightarrow{AE}$

```
z.A  = point(0, 1)
z.B  = point(3, 4)
z.C  = point(3, 2)
z.D  = point(4, 1)
V.AB = vector(z.A, z.B)
V.CD = vector(z.C, z.D)
V.AE = V.AB + V.CD  % possible V.AB:add(V.CD)
z.E  = V.AE.head % we recover the final point (head)
```




```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(3, 4)
 z.C = point(3, 2)
 z.D = point(4, 1)
 V.u = vector(z.A, z.B)
 V.v = vector(z.C, z.D)
 V.w = V.u + V.v % or V.u:add(V.v)
 z.E = V.w.head}
\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints(A,B,C,D,E)
  \tkzDrawSegments[->,red](A,B)
  \tkzDrawSegments[->,cyan](A,E)
  \tkzDrawSegments[->,blue](C,D B,E)
  \tkzLabelSegment(A,B){$\overrightarrow{u}$}
  \tkzLabelSegment(C,D){$\overrightarrow{v}$}
  \tkzLabelSegment(A,E){$\overrightarrow{w}$}
\end{tikzpicture}
```

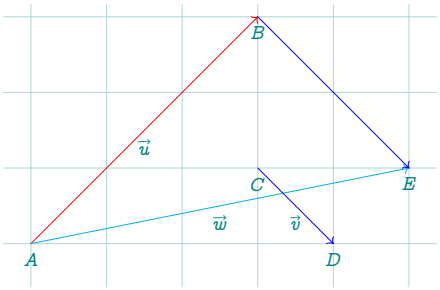
22.4.2. Method sub




```
\directlua{
 z.A = point(0, 1)
 z.B = point(3, 4)
 z.C = point(3, 2)
 z.D = point(4, 1)
 V.u = vector(z.A, z.B)
 V.v = vector(z.C, z.D)
 V.w = V.u - V.v
 z.E = V.w.head}

\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints(A,B,C,D,E)
  \tkzDrawSegments[->,red](A,B)
    \tkzDrawSegments[->,cyan](A,E)
   \tkzDrawSegments[->,blue](C,D B,E)
  \tkzLabelSegment(A,B){$\overrightarrow{u}$}
  \tkzLabelSegment(C,D){$\overrightarrow{v}$}
  \tkzLabelSegment(A,E){$\overrightarrow{w}$}
\end{tikzpicture}
```

22.4.3. Method mul

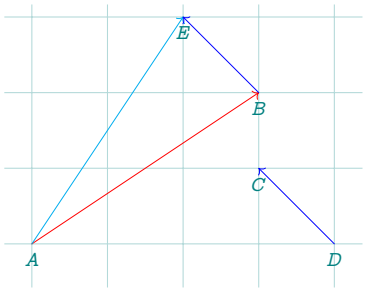
This is, of course, multiplication by a scalar.




```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(3, 4)
 z.C = point(3, 2)
 z.D = point(4, 1)
 V.u = vector(z.A, z.B)
 V.v = vector(z.C, z.D)
 V.w = V.u + 2 * V.v
 z.E = V.w.head}

\begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints(A,B,C,D,E)
  \tkzDrawSegments[->,red](A,B)
    \tkzDrawSegments[->,cyan](A,E)
   \tkzDrawSegments[->,blue](C,D B,E)
  \tkzLabelSegment(A,B){$\overrightarrow{u}$}
  \tkzLabelSegment(C,D){$\overrightarrow{v}$}
  \tkzLabelSegment(A,E){$\overrightarrow{w}$}
\end{tikzpicture}
```

22.4.4. Method unm

Cette méthode vous permet d'écrire **V.w = -V.v**




```
\directlua{
  z.A = point(0, 1)
  z.B = point(3, 3)
  z.C = point(3, 2)
  z.D = point(4, 1)
  V.u = vector(z.A, z.B)
  V.v = vector(z.C, z.D)
  V.v = -V.v
  V.w = V.u + V.v
  z.E = V.w.head}
\begin{center}
  \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzLabelPoints(A,B,C,D,E)
  \tkzDrawSegments[->,red](A,B)
  \tkzDrawSegments[->,cyan](A,E)
  \tkzDrawSegments[->,blue](D,C B,E)
\end{tikzpicture}
\end{center}
```

22.4.5. Method ^

Instead of the power, which wouldn't make sense here, we're talking about the determinant of two vectors. Note: the reference frame used is orthonormal and direct.

```
V.u = vector(z.A, z.B) with z.A = point(xa, ya) and z.B = point(xb, yb)
V.v = vector(z.C, z.D) with z.C = point(xc, yc) and z.D = point(xd, yd)            then
V.u ^ V.v = xa * yb -  xb * ya
```

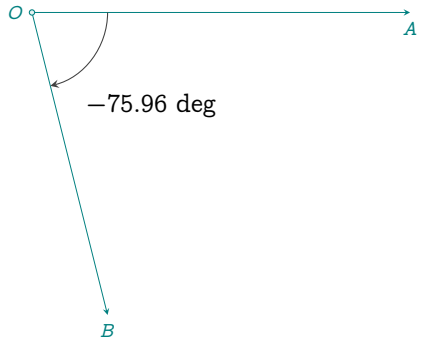
remark: `u ^ v = u.norm * v.norm * sin(u,v)`

22.4.6. Method ..

Instead of the concatenation, which wouldn't make sense here, we're talking about the dot product of two vectors. Note: the reference frame used is orthonormal and direct.

```
V.u = vector(z.A, z.B) with z.A = point(xa, ya) and z.B = point(xb, yb)
V.v = vector(z.C, z.D) with z.C = point(xc, yc) and z.D = point(xd, yd)            then
V.u .. V.v = xa * xb +  ya * yb
```

remark: `V.u .. V.v = V.u.norm * V.v.norm * cos(V.u,V.v)`
We're going to use the two last methods . We can determine the cosine and sine of the angle using the dot product and determinant expressed in the direct orthonormal frame used by the package.




```
\directlua{
  z.O = point(0, 0)
  z.A = point(5, 0)
  z.B = point(1, -4)
  V.u = vector(z.O, z.A)
  V.v = vector(z.O, z.B)
  local dp = V.u .. V.v % dot product
  local d = V.u ^ V.v % determinant
  % costheta = dp / (u.norm * v.norm)
  % sintheta = d / (u.norm * v.norm)
  an = math.atan(d / dp)}
```

The code required to display the angle measurement is as follows:

```
\tkzLabelAngle[pos=2](B,O,A){$\directlua{
local format = string.char(37) .. ".2f"
tex.print(string.format( format , math.deg(an)))}$ deg}
```

The main methods provided by the **vector** class are summarised below.

Table 30: Methods of the class vector.

Methods	Reference
Constructor	
new(pt, pt)	Note[13]; 22.1
Methods Returning a Boolean	
is_zero([EPS]) is_parallel(v,[EPS]) is_orthogonal(v,[EPS])	
Methods Returning a Point	
get()	
Methods Returning a Vector add(v) scale(d) dot(v) cross(v) normalize() angle_to(v) at (pt) rotate() orthogonal([side],[length])	 [22.6.1] [22.6.2]

22.5. Returns a boolean

22.5.1. Predicates: is_zero, is_parallel, is_orthogonal

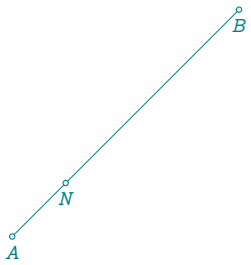
```
if V:is_zero() then ... end
if V:is_parallel(W) then ... end
if V:is_orthogonal(W) then ... end
```

These methods test basic geometric relations between vectors, using an optional tolerance , which defaults to . They are convenient when deciding whether two directions should be treated as parallel or orthogonal in numerical computations.

22.6. Returns a vector

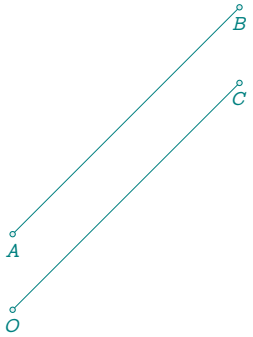
22.6.1. Method normalize()

This method produces a normalized vector that is collinear with the initial vector.




```
\directlua{
init_elements()
z.A = point(0, 1)
z.B = point(3, 4)
V.AB = vector(z.A, z.B)
V.AN = V.AB:normalize()
z.N = V.AN.head}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawSegments(A,B)
\tkzDrawPoints(A,B,N)
\tkzLabelPoints(A,B,N)
\end{tikzpicture}
```

22.6.2. Method at()



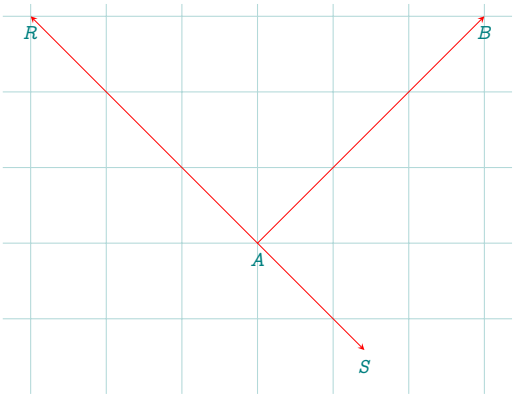

```
\directlua{
init_elements()
z.A = point(0, 1)
z.B = point(3, 4)
z.O = point(0, 0)
V.AB = vector(z.A, z.B)
V.OC = V.AB:at(z.O)
z.C = V.OC.head}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawSegments(A,B O,C)
\tkzDrawPoints(A,B,O,C)
\tkzLabelPoints(A,B,O,C)
\end{tikzpicture}
```

22.6.3. Method orthogonal([side],[length])

```
Vperp  = V:orthogonal()           -- default: "ccw"
VperpC = V:orthogonal("cw")       -- clockwise orthogonal
Vlen   = V:orthogonal("ccw", 2.0) -- orthogonal of prescribed length
```

The **orthogonal** method constructs an orthogonal vector to **V**, with the same tail:

- the optional argument may be **"ccw"** (counter-clockwise, the default) or **"cw"** (clockwise),
- the optional argument may be used to prescribe the norm of the resulting vector; if omitted, the length is determined by a rotation of the current vector.




```
\directlua{
 init_elements()
 z.A = point(0, 1)
 z.B = point(3, 4)
 V.AB = vector(z.A, z.B)
 V.AR = V.AB:orthogonal(2 * math.sqrt(2))
 z.R = V.AR.head
 V.AS = V.AB:orthogonal("cw",2)
 z.S = V.AS.head}
\begin{center}
 \begin{tikzpicture}[gridded]
  \tkzGetNodes
  \tkzDrawSegments[>=stealth,->,
    red](A,B A,R A,S)
  \tkzLabelPoints(A,B,R,S)
\end{tikzpicture}
\end{center}
```

23. Class matrix

The variable M holds a table used to store matrices. It is optional, and you are free to choose the variable name. However, using **M** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., Matrices), you must initialize it manually.
The **init_elements()** function reinitializes the **M** table if used.

The **matrix** class is currently experimental, and its attribute and method names have not yet been finalized, indicating that this class is still evolving. Certain connections have been made with other classes, such as the **point** class. Additionally, a new attribute, **mtx**, has been included, associating a column matrix with the point, where the elements correspond to the point's coordinates in the original base. Similarly, an attribute has been added to the **vector** class, where **mtx** represents a column matrix consisting of the two affixes that compose the vector.
This **matrix** class has been created to avoid the need for an external library, and has been adapted to plane transformations. It allows you to use complex numbers.
☞ To display matrices, you'll need to load the amsmath package.
☞ While some methods are valid for any matrix size, the majority are reserved for square matrices of order 2 and 3.

23.1. Matrix creation

The creation of a matrix is the result of numerous possibilities. Let's take a look at the different cases
The first one is to use an array of arrays, that is, a table wherein each element is another table. For instance, you can create a matrix of zeros with dimensions N by M with the following code:

- The first method is: [23.5.2]. This function is the most important, as it's the one that creates an object. The other functions create specific objects and always use this function.

  The matrix class represents a 2×2 matrix defined by four values.

  ```
  M = matrix:new(1, 0, 0, 1) -- identity matrix
  ```

  Short form:

  A more concise form is available:

  ```
  M = matrix(1, 0, 0, 1)
  ```

  ```
   M.new = matrix({ { a, b }, { c, d } })
  ```
  a, b, c, et d being real or complex numbers.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
\directlua{
  init_elements()
  local a, b, c, d = 1, 2, 3, 4
  M.new = matrix({ { a, b }, { c, d } })
  tex.print('M = ') M.new:print()}
```

- With the function **create**, you get a matrix whose coefficients are all zero, with a number of columns and rows of your choice. [23.5.5]

  ```
  M.cr = matrix.create(4,5)
  ```

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.cr = matrix.create(4, 5)
  tex.print('M = ') M.cr:print()}
```

– The identity matrix of size n is the $n \times n$ square matrix with ones on the main diagonal and zeros elsewhere. See [23.5.7]

```
M.I = matrix.identity(3)
```

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.I = matrix.identity(3)
  tex.print('$I_3 = $') M.I:print()}
```

– It is also possible to obtain a square matrix with: [23.5.6]

```
M.sq = matrix.square (2,a,b,c,d)
```

$$\mathrm{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
\directlua{
  init_elements()
   local a, b, c, d = 1, 2, 3, 4
  M.sq = matrix.square(2, a, b, c, d)
  tex.print('M = ') M.sq:print()}
```

– In the case of a column vector: [23.5.3]

```
M.V = matrix.vector(1, 2, 3) also possible M.V = matrix.column(1, 2, 3)
```

$$\mathrm{V} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.V = matrix.vector(1, 2, 3)
  tex.print('V = ') M.V:print()}
```

– In the case of a row vector: [23.5.4]

```
M.V = matrix.row_vector(1, 2, 3)
```

$$\mathrm{V} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.V = matrix.row_vector(1, 2, 3)
  tex.print('V = ') M.V:print()}
```

– Matrix associated with a point

```
M.p = matrix({ { p.re }, { p.im } })
```

– Matrix associated with a vector

It's a column matrix made up of the affixes of the two points defining the vector.

```
local M.v = matrix{ { za }, { zb } }
```

$$\begin{bmatrix} 1+2i \\ 3+4i \end{bmatrix}$$

```
\directlua{
z.A = point(1, 2)
z.B = point(3, 4)
V.u = vector(z.A, z.B)
V.u.mtx:print()}
```

– Homogeneous transformation matrix [23.5.18]

The objective is to generate a matrix with homogeneous coordinates capable of transforming a coordinate system through rotation, translation, and scaling. To achieve this, it is necessary to define both the rotation angle, the coordinates of the new origin ans the scaling factors.

$$H = \begin{bmatrix} 1 & -0.87 & 1 \\ 0.87 & 0.50 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.h = matrix.htm(math.pi / 3, 1, 2, 2, 1)
  tex.print('H = ') M.h:print()}
```

23.2. Method print()

This method (See 23.5.1) is necessary to control the results, so here are a few explanations on how to use it. It can be used on real or complex matrices, square or not. A few options allow you to format the results. You need to load the amsmath package to use the "print" method. Without this package, it is possible to display the contents of the matrix without formatting with `print_array (M)`

$$\begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix}$$

```
\directlua{
  init_elements()
  M.new = matrix { { 1, -1}, { 2, 0 } }
  M.new:print()}
```

23.3. Attibutes of a matrix

Table 31: Matrix attributes.

Attributes	Reference
set	[23.3.2]
rows	[23.3.3]
cols	[23.3.3]
type	
det	[23.3.4]

23.3.1. Attribute type

```
M.new = matrix{ { 1, 1}, { 0, 2} }  A = { { 1, 1 }, { 0, 2 } }
```

M is a matrix (and therefore a table) whereas A is a table. Thus **M.type** gives **'matrix'** and **A.type = nil**. **type(A) or type(M) = table**.

23.3.2. Attribute set

A simple array such as {{1,2},{2,-1}} is often considered a **matrix**. In tkz-elements, we'll consider `M.new` defined by
`matrix({ { 1, 1 }, { 0, 2 } })`
as a matrix and `M.new.set` as an array (`M.new.set = { { 1, 1 }, {0, 2 } }`).
You can access a particular element of the matrix, for example: `M.new.set[2][1]` gives 0.
`\tkzUseLua{M.new.set[2][1]}` is the expression that displays 2.

23.3.3. Attributes rows and cols

The number of rows is accessed with `M.n.rows` and the number of columns with `M.n.cols`, here's an example:

```
\directlua{
 init_elements()
 M.n = matrix({ { 1, 2, 3 }, { 4, 5, 6 } })
 M.n:print()
 tex.print("Rows:  "..M.n.rows)
 tex.print("Cols:  "..M.n.cols)}
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$ Rows: 2 Cols: 3

23.3.4. Attributes det

Give the determinant of the matrix if it is square, otherwise it is **nil**. The coefficients of the matrix can be complex numbers.

```
\directlua{
  init_elements()
  M.s = matrix.square(3, 1, 1, 0, 2, -1, -2, 1, -1, 2)
  M.s:print()
  tex.print ('\\\\')
  tex.print ("Its determinant is:  " .. M.s.det)
  }
```

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & -1 & -2 \\ 1 & -1 & 2 \end{bmatrix}$$

Its determinant is: -10.0
Its determinant is: -4.00i

23.4. Metamethods for the matrices

Conditions on matrices must be valid for certain operations to be possible.

Table 32: Matrix metamethods.

Metamethods	Refrence
`add(M1,M2)`	See [23.4.1]
`sub(M1,M2)`	See [23.4.1]
`unm(M`	`- M`
`mul(M1,M2)`	[23.4.2]
`pow(M,n)`	[23.4.2]
`tostring(M,n)`	displays the matrix
`eq(M1,M2)`	true or false

23.4.1. Addition and subtraction of matrices

To simplify the entries, I've used a few functions to simplify the displays.

```
\directlua{
  init_elements()
  M.A = matrix({ { 1, 2 }, { 2 , -1 } })
  M.B = matrix({ { -1, 0}, { 1, 3 } })
  S = M.A + M.B
  D = M.A - M.B
  dsp(M.A,'A')
  nl() nl()
  dsp(M.B,'B')
  nl() nl()
  dsp(M.S,'S') sym(" = ")
  dsp(M.A) sym(' + ') dsp(M.B)
  nl() nl()
  dsp(M.D,'D') sym(" = ")
  dsp(M.A) sym(' - ') dsp(M.B)
}
```

$$A = \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}$$

$$B = \begin{bmatrix} -1 & 0 \\ 1 & 3 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 2 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & 3 \end{bmatrix}$$

$$D = \begin{bmatrix} 2 & 2 \\ 1 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix} - \begin{bmatrix} -1 & 0 \\ 1 & 3 \end{bmatrix}$$

23.4.2. Multiplication and power of matrices

To simplify the entries, I've used a few functions. You can find their definitions in the sources section of this documentation. n integer > or < 0 or 'T'

```
\directlua{
  init_elements()
  M.A = matrix({ { 1, 2 }, { 2 ,-1 } })
  M.B = matrix({ { -1, 0 }, { 1, 3 } })
  M.P = M.A * M.B
  M.I = M.A ^ -1
  M.C = M.A ^ 3
  M.K = 2 * M.A}
```

$$P = \begin{bmatrix} 1 & 6 \\ -3 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix} * \begin{bmatrix} -1 & 0 \\ 1 & 3 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$K = \begin{bmatrix} 2 & 4 \\ 4 & -2 \end{bmatrix}$$

23.4.3. Metamethod `eq`

Tests whether two matrices are equal. The result is **true** or **false**.

23.5. Methods of the class matrix

Table 33: Matrix functions and methods.

Functions	Reference
new(...)	See [23.5.2; 23.1]
matrix.square()	[23.5.6]
matrix.vector()	[23.5.3]
matrix.row_vector()	[23.5.4]
matrix.create()	
matrix.identity()	[23.5.7]
matrix.htm()	[23.5.18]
Methods	**Reference**
print(s,n)	
htm_apply(...)	[23.5.20]
get_htm_point()	[23.5.19]
get()	[23.5.11]
inverse()	[23.5.12]
adjugate()	[23.5.15]
transpose()	[23.5.14]
is_diagonal()	[23.5.9]
is_orthogonal()	[23.5.8]
homogenization()	[23.5.17]
gauss_jordan()	[23.5.21]
rank()	[23.5.22]
augment_right(B)	[23.5.23]
submatrix(r1,r2,c1,c2)	[23.5.23]

23.5.1. Method print

With the amsmath package loaded, this method can be used. By default, the **bmatrix** environment is selected, although you can choose from **matrix**, **pmatrix**, **Bmatrix**, "vmatrix", "Vmatrix". Another option lets you set the number of digits after the decimal point. The "tkz_dc" global variable is used to set the number of decimal places. Here's an example:

```
\directlua{
  init_elements()
  M.n = matrix({ { math.sqrt(2), math.sqrt(3) }, { math.sqrt(4), math.sqrt(5) } })
  M.n:print('pmatrix')}
```

$$\begin{pmatrix} 1.41 & 1.73 \\ 2 & 2.24 \end{pmatrix}$$

You can also display the matrix as a simple array using the `print_array (M)` function. see the next example. In the case of a square matrix, it is possible to transmit a list of values whose first element is the order of the matrix.

```
\directlua{
init_elements()
 M.s = matrix.square(2, 1, 0, 0, 2)
 M.s:print()}
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

23.5.2. Function new

This is the main method for creating a matrix. Here's an example of a 2x3 matrix with complex coefficients:

```
\directlua{
 init_elements()
 a = point(1, 0)
 b = point(1, 1)
 c = point(-1, 1)
 d = point(0, 1)
 e = point(1, -1)
 f = point(0, -1)
 M.n = matrix({ { a, b, c }, { d, e, f } })
 M.n:print()}
```

$$\begin{bmatrix} 1 & 1+i & -1+i \\ i & 1-i & -i \end{bmatrix}$$

23.5.3. Function matrix.vector

The special case of a column matrix, frequently used to represent a vector, can be treated as follows:

```
\directlua{
 init_elements()
 M.v = matrix.vector(1, 2, 3)
 M.v:print()}
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

23.5.4. Function matrix.row_vector

```
M.rv = matrix.row_vector (1, 2, 3)
```

$$\mathrm{m.rv} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

23.5.5. Function matrix.create(n,m)

```
M.c = matrix.create (2, 3)
```

$$\mathrm{M.c} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

23.5.6. Function matrix.square(liste)

We have already seen this method in the presentation of matrices. We first need to give the order of the matrix, then the coefficients, row by row.

```
\directlua{
 init_elements()
 M.s = matrix.square(2, 2, 3, -5, 4)
 M.s:print()}
```

$$\begin{bmatrix} 2 & 3 \\ -5 & 4 \end{bmatrix}$$

23.5.7. Function matrix.identity

Creating the identity matrix order 3

```
\directlua{
  init_elements()
  M.Id_3 = matrix.identity(3)
  M.Id_3:print()}
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

23.5.8. Method is_orthogonal

The method returns **true** if the matrix is orthogonal and **false** otherwise.

```
\directlua{
 init_elements()
 local cos = math.cos
 local sin = math.sin
 local pi = math.pi
 M.A = matrix({ { cos(pi / 6), -sin(pi / 6) }, { sin(pi / 6), cos(pi / 6) } })
```

```
M.A:print()
bool = M.A:is_orthogonal()
tex.print("\\\\")
if bool then
 tex.print("The matrix is orthogonal")
else
 tex.print("The matrix is not orthogonal")
end
tex.print("\\\\")
tex.print("Test: $M.A^T = M.A^{-1} ?$")
print_matrix(transposeMatrix(M.A))
tex.print("=")
inv_matrix(M.A):print()}
```

$$\begin{bmatrix} 0.87 & -0.50 \\ 0.50 & 0.87 \end{bmatrix}$$
The matrix is not orthogonal
Test: $M.A^T = M.A^{-1}?$ $\begin{bmatrix} 0.87 & 0.50 \\ -0.50 & 0.87 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

23.5.9. Method is_diagonal

The method returns **true** if the matrix is diagonal and **false** otherwise.

23.5.10. Function print_array

We'll need to display results, so let's look at the different ways of displaying them, and distinguish the differences between arrays and matrices.
Below, A is an array. It can be displayed as a simple array or as a matrix, but we can't use the attributes and `A:print()` is not possible because A is not an object of the class **matrix**. If you want to display an array like a matrix you can use the function `print_matrix` (see the next example).

```
\directlua{
 init_elements()
 A = { { 1, 2 }, { 1, -1 } }
 tex.print("A = ")
 print_array(A)
 tex.print(" or ")
 print_matrix(A)
 M.A = matrix({ { 1, 1 }, { 0, 2 } })
 tex.print("\\\\")
 tex.print("M = ")
 M.A:print()}
```

A = {{ 1 , 2 },{ 1 , -1 }}or $\begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}$

M = $\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$

23.5.11. Method get

Get an element of a matrix.

```
\directlua{
  init_elements()
  M.n = matrix{ { 1, 2 }, { 2, -1 } }
  S = M.n:get(1, 1) + M.n:get(2, 2)
  tex.print(S)}
```

0

23.5.12. Method inverse

```
\directlua{
 init_elements()
 M.A = matrix({ { 1, 2 }, { 2, -1 } })
 tex.print("Inverse of $A = $")
 M.B = M.A:inverse()
 M.B:print()}
```

Inverse of $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

23.5.13. Inverse matrix with power syntax

```
\directlua{
  init_elements()
  M.n = matrix({ { 1, 0, 1 }, { 1, 2, 1 }, { 0, -
1, 2 } })
  tex.print("$M = $")  print_matrix (M.n)
  tex.print('\\\\')
  tex.print("Inverse of $M = M^{-1}$")
  tex.print('\\\\','=') print_matrix(M.n ^ -1)}
```

$$M = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 0 & -1 & 2 \end{bmatrix}$$

$$M = M^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

23.5.14. Method transpose

A transposed matrix can be accessed with `A: transpose ()` or with `A^{'T'}`.

```
\directlua{
 init_elements()
 M.A = matrix({ { 1, 2 }, { 2, -1 } })
 M.AT = M.A:transpose()
 tex.print("$A^{'T'} = $")
 M.AT:print()}
```

$$A^{'T'} = \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}$$

Remark: `(A ^'T')^'T' = A`

23.5.15. Method adjugate

```
\directlua{
  init_elements()
  M.N =  matrix({ {1, 0, 3}, {2, 1, 0},
                  {-1, 2, 0} })
  tex.print('N = ') print_matrix(M.N)
  tex.print('\\\\')
  M.N.a = M.N:adjugate()
  M.N.i = M.N * M.N.a
  tex.print('adj(M) = ') M.N.a:print()
  tex.print('\\\\')
  tex.print('N $\\times$ adj(N) = ')
  print_matrix(M.N.i)
  tex.print('\\\\')
  tex.print('det(N) = ')
  tex.print(M.N.det)}
```

$$\mathrm{N} = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 1 & 0 \\ -1 & 2 & 0 \end{bmatrix}$$

$$\mathrm{adj(M)} = \begin{bmatrix} 0 & 6 & -3 \\ 0 & 3 & 6 \\ 5 & -2 & 1 \end{bmatrix}$$

$$\mathrm{N} \times \mathrm{adj(N)} = \begin{bmatrix} 15 & 0 & 0 \\ 0 & 15 & 0 \\ 0 & 0 & 15 \end{bmatrix}$$

$$\det(\mathrm{N}) = 15.0$$

23.5.16. Method diagonalize

For the moment, this method only concerns matrices of order 2.

```
\directlua{
 init_elements()
 M.A = matrix({ { 5, -3 }, { 6, -4 } })
 tex.print("A = ")
 M.A:print()
 M.D, M.P = M.A:diagonalize()
 tex.print("D = ")
 M.D:print()
 tex.print("P = ")
 M.P:print()
 M.R = M.P ^ -1 * M.A * M.P
 tex.print("\\\\")
 tex.print("Test: $D = P^{-1}AP = $ ")
 M.R:print()
 tex.print("\\\\")
 tex.print("Verification: $P^{-1}P = $ ")
 M.T = M.P ^ -1 * M.P
 M.T:print()}
```

$$A = \begin{bmatrix} 5 & -3 \\ 6 & -4 \end{bmatrix} D = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix} P = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

Test: $D = P^{-1}AP = \begin{bmatrix} 2 & -1 \\ 2 & -2 \end{bmatrix}$

Verification: $P^{-1}P = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$

23.5.17. Method homogenization

The goal of **homogenization** is to to be able to use a homogeneous transformation matrix
Let's take a point A such that `z.A = point(2,-1)`. In order to apply a **htm** matrix, we need to perform a few operations on this point. The first is to determine the vector (matrix) associated with the point. This is straightforward, since there's a point attribute called **mtx** which gives this vector:

```
z.A = point(2,0)
M.V = z.A.mtx:homogenization()
```

which gives:

```
\directlua{
 init_elements()
 pi = math.pi
 M.h = matrix.htm(pi / 4, 3, 1)
 z.A = point(2, 0)
 M.V = z.A.mtx:homogenization()
 z.A.mtx:print()
 tex.print("then after homogenization: ")
 M.V:print()}
```

$$\begin{bmatrix} 2 \\ 0 \end{bmatrix} \text{then after homogenization: } \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

23.5.18. Function matrix.htm

Homogeneous transformation matrix.
There are several ways of using this transformation. First, we need to create a matrix that can associate a rotation with a translation.
The main method is to create the matrix:

```
pi = math.pi
M.h = matrix.htm(pi / 4, 3, 1)
```

A 3x3 matrix is created which combines a $\pi/4$ rotation and a $\vec{t} = (3,1)$ translation.

$$\begin{bmatrix} 0.71 & -0.71 & 3 \\ 0.71 & 0.71 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can apply the matrix M. Let A be the point defined here: 23.5.17. By homogenization, we obtain the column matrix V.

```
M.W = M.A * M.V
```

$$\begin{bmatrix} 0.71 & -0.71 & 3 \\ 0.71 & 0.71 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.41 \\ 2.41 \\ 1 \end{bmatrix}$$

All that remains is to extract the coordinates of the new point.

23.5.19. Method get_htm_point

In the previous section, we obtained the W matrix. Now we need to obtain the point it defines.
The method **get_htm_point** extracts a point from a vector obtained after applying a **htm** matrix.

```
\directlua{
  init_elements()
  pi  = math.pi
  M.h   = matrix.htm(pi / 4 , 3 , 1)
  z.A = point(2,0)
  M.V = z.A.mtx:homogenization()
  M.W = M.h * M.V
  M.W:print()
  z.P = get_htm_point(M.W)
  tex.print("The affix of $P$ is: ")
  tex.print(tkz.display(z.P))}
```

$$\begin{bmatrix} 4.41 \\ 2.41 \\ 1 \end{bmatrix}$$ The affix of P is: 4.41+2.41i

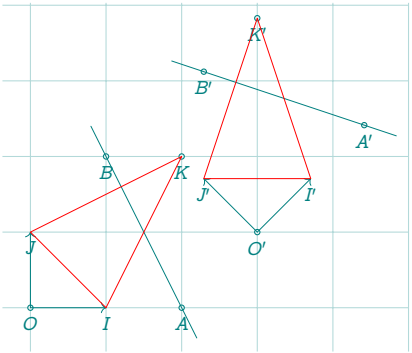
23.5.20. Method htm_apply

The above operations can be simplified by using the **htm_apply** method directly at point A.

```
z.Ap = M: htm_apply (z.A)
```

Then the method **htm_apply** transforms a point, a list of points or an object.

```
\directlua{
 init_elements()
 pi = math.pi
 M.h = matrix.htm(pi / 4, 3, 1)
 z.O = point(0, 0)
 z.I = point(1, 0)
 z.J = point(0, 1)
 z.A = point(2, 0)
 z.B = point(1, 2)
 L.AB = line(z.A, z.B)
 z.Op, z.Ip, z.Jp = M.h:htm_apply(z.O, z.I, z.J)
 L.ApBp = M.h:htm_apply(L.AB)
 z.Ap = L.ApBp.pa
 z.Bp = L.ApBp.pb
 z.K = point(2, 2)
 T.IJK = triangle(z.I, z.J, z.K)
 Tp = M.h:htm_apply(T.IJK)
 z.Kp = Tp.pc}
```



New cartesian coordinates system:

```
\directlua{
 init_elements()
 pi = math.pi
 tp = tex.print
 nl = "\\\\"
 a = point(1, 0)
 b = point(0, 1)
 M.R = matrix.htm(pi / 5, 2, 1)
 M.R:print()
 tp(nl)
 M.v = matrix.vector(1, 2)
 M.v:print()
 M.v.h = M.v:homogenization()
 M.v.h:print()
 tp(nl)
 M.V = M.R * M.v.h
 M.V:print()
 z.N = get_htm_point(M.V)
tex.print(tkz.display(z.N))}
```

$$\begin{bmatrix} 0.81 & -0.59 & 2 \\ 0.59 & 0.81 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1.63 \\ 3.21 \\ 1 \end{bmatrix} 1.63+3.21i$$

23.5.21. Method gauss_jordan()

This method applies the Gauss–Jordan elimination algorithm to the matrix and returns its reduced row echelon form (RREF).

The Gauss–Jordan algorithm transforms a matrix into an equivalent matrix by a sequence of elementary row operations:

- swapping two rows;
- multiplying a row by a non-zero scalar;
- adding a multiple of one row to another.

The resulting matrix satisfies the following properties:

- each leading entry of a non-zero row is equal to 1;
- each leading 1 is the only non-zero entry in its column;
- all zero rows, if any, are at the bottom of the matrix.

`Return value:`
The method returns a new matrix object corresponding to the reduced row echelon form of the original matrix. The original matrix is not modified.

`Numerical considerations:`
Since computations are performed using floating-point arithmetic, a numerical tolerance is used internally to detect zero pivots. Very small values (typically below a fixed threshold) are treated as zero.

`Typical uses:`

- solving linear systems;
- computing the rank of a matrix;
- testing linear independence of vectors;
- computing the inverse of a square matrix (when it exists).

Example with matrix 2x2:

$$\begin{bmatrix} 3 & -1 \\ -5 & 2 \end{bmatrix}$$

```
\directlua{
  M.A    = matrix({{2,1},{5,3}})
  M.Ainv = M.A:augment_right(matrix.identity(2))
              :gauss_jordan()
              :submatrix(1, 2, 3, 4)

  M.Ainv:print()
}
```

Example with matrix 3x3:

$$\begin{bmatrix} 0 & 0 & 1 \\ -2 & 1 & 3 \\ 3 & -1 & -5 \end{bmatrix}$$

```
\directlua{
  % Matrix 3x3
  M.A = matrix({
    {2, 1, 1},
    {1, 3, 2},
    {1, 0, 0}
  })

  % Inverse via augmentation + Gauss-Jordan
  M.Ainv = M.A
            :augment_right(matrix.identity(3))
            :gauss_jordan()
            :submatrix(1, 3, 4, 6)

  % Print the inverse
  M.Ainv:print()
}
```

23.5.22. Method rank()

This method returns the rank of the matrix, that is, the dimension of the vector space spanned by its rows (or equivalently, by its columns).

The rank is computed using the Gauss–Jordan elimination process: it is equal to the number of non-zero rows in the reduced row echelon form of the matrix.

Return value: The method returns a non-negative integer equal to the rank of the matrix.

Remarks:

- The rank is always less than or equal to the minimum of the number of rows and columns.
- A matrix has full rank if its rank is equal to this minimum.
- A square matrix is invertible if and only if its rank is maximal.

Numerical stability: As for **gauss_jordan()**, a numerical tolerance is used to decide whether a row should be considered zero.

Note. The method **rank()** internally relies on the Gauss–Jordan reduction, but provides a direct and convenient access to the rank without exposing the intermediate reduced matrix.
Example 1:

rank(A) =3

```
\directlua{
  M.A = matrix({
    {2, 1, 1},
    {1, 3, 2},
    {1, 0, 0}
  })

  tex.print("rank(A) = ", M.A:rank())
}
```

Example 2:

rank(B) =2

```
\directlua{
  M.B = matrix({
    {1, 2, 3},
    {2, 4, 6},
    {1, 1, 1}
  })

  tex.print("rank(B) = ", M.B:rank())
}
```

Example 3:

rank(C) =2

```
\directlua{
  M.C = matrix({
    {1, 2, 3},
    {2, 4, 6},
    {1, 1, 1}
  })

  M.R = M.C:gauss_jordan()

  tex.print("rank(C) = ", M.C:rank())
}
```

23.5.23. Augmented matrices and submatrices

The inversion of a square matrix and the resolution of linear systems are both based on the same fundamental idea: the manipulation of an *augmented matrix* followed by a block extraction after a Gauss–Jordan reduction.

Augmented matrix. Let A be an $m \times n$ matrix and B an $m \times p$ matrix. The augmented matrix $[A \mid B]$ is obtained by appending the columns of B to the right of A:

$$[A \mid B] = \begin{pmatrix} A_{11} & \cdots & A_{1n} & B_{11} & \cdots & B_{1p} \\ \vdots & & \vdots & \vdots & & \vdots \\ A_{m1} & \cdots & A_{mn} & B_{m1} & \cdots & B_{mp} \end{pmatrix}.$$

In the `matrix` class, this operation is performed by the method `augment_right`.

Example (matrix inversion). Let

$$A = \begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix}.$$

The augmented matrix $[A \mid I]$ is

$$\begin{pmatrix} 2 & 1 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{pmatrix}.$$

Applying the Gauss–Jordan algorithm yields

$$\begin{pmatrix} 1 & 0 & 3 & -1 \\ 0 & 1 & -5 & 2 \end{pmatrix} = [I \mid A^{-1}],$$

from which the inverse matrix can be read directly.

Submatrix extraction. The method `submatrix(r1, r2, c1, c2)` extracts a rectangular block from a matrix, defined by:

- rows from `r1` to `r2`,
- columns from `c1` to `c2`.

All indices start at 1.

Meaning of submatrix(1,2,3,4). In the previous example, the reduced augmented matrix has:

- 2 rows,
- 4 columns.

The inverse matrix occupies:

- rows 1 to 2,
- columns 3 to 4.

Thus the extraction

`submatrix(1,2,3,4)`

returns exactly the block

$$A^{-1} = \begin{pmatrix} 3 & -1 \\ -5 & 2 \end{pmatrix}.$$

General rule. For a square matrix of order n, the inverse matrix is obtained by

$$[A \mid I_n] \xrightarrow{\text{Gauss–Jordan}} [I_n \mid A^{-1}],$$

and extracted using

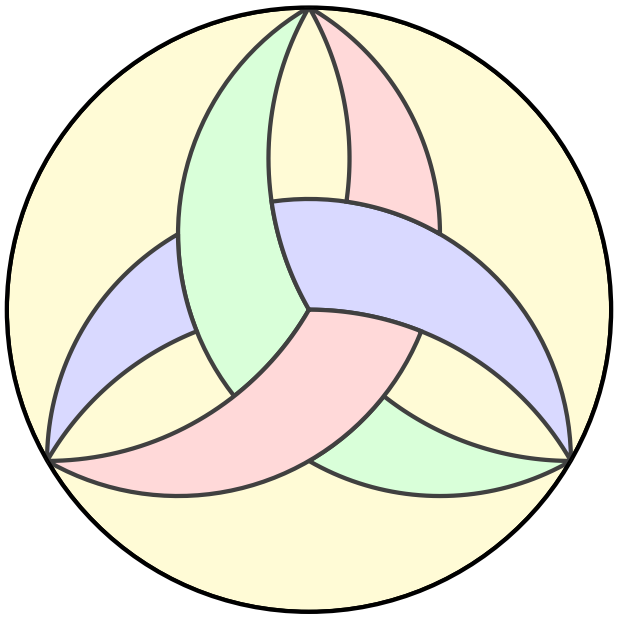
`submatrix(1, n, n+1, 2n)`.

Summary.

- `augment_right` constructs an augmented matrix,
- `gauss_jordan` performs the row reduction,
- `submatrix` extracts the desired result block.

24. Class path

This class was developed in response to a question posted on **tex.stackexchange.com** [fill-a-space-made-by-three-arcs].
The concept was missing from the package, although in some cases it was possible to work around this using Ti*k*Z paths. When creating the **conic** class, I frequently had to manipulate tables of coordinates, so it became natural to formalize these as **paths**.



```
\directlua{
 z.O = point(0, 0)
 z.A = point(8, 0)
 z.E = z.O:rotation(2*math.pi / 3, z.A)
 z.C = z.O:rotation(-2*math.pi / 3, z.A)
 z.F, z.D, z.B = z.O:rotation(math.pi / 3, z.A, z.E, z.C)
 L.AC = line(z.A, z.C)
 L.OB = line(z.O, z.B)
 L.OD = line(z.O, z.D)
 L.CE = line(z.C, z.E)
 L.AE = line(z.A, z.E)
 L.OF = line(z.O, z.F)
 z.G = intersection(L.AC, L.OB)
 z.H = intersection(L.CE, L.OD)
 z.I = intersection(L.AE, L.OF)
 C.GA = circle(z.G, z.A)
 C.FE = circle(z.F, z.E)
 C.BA = circle(z.B, z.A)
 C.IE = circle(z.I, z.E)
 C.DE = circle(z.D, z.E)
 C.FA = circle(z.F, z.A)
 C.HE = circle(z.H, z.E)
 C.GC = circle(z.G, z.C)
 _,z.J = intersection(C.GA, C.FE)
 _,z.K = intersection(C.BA, C.IE)
 _,z.L = intersection(C.DE, C.IE)
 _,z.M = intersection(C.FA, C.HE)
 z.N = intersection(C.BA, C.HE)
 z.P = intersection(C.DE, C.GC)
 % the paths
```

```
 local nb = 40
 PA.th1 = C.IE:path(z.E, z.L, nb) + C.DE:path(z.L, z.O, nb) - C.FA:path(z.E, z.O, nb)
 PA.th2 = C.IE:path(z.G, z.A, nb) - C.FA:path(z.M, z.A, nb) - C.HE:path(z.G, z.M, nb)
 PA.th3 = C.IE:path(z.H, z.K, nb) + C.BA:path(z.K, z.C, nb) - C.GC:path(z.H, z.C, nb)
 PA.th4 = C.GA:path(z.A, z.J, nb) + C.FA:path(z.J, z.O, nb) - C.BA:path(z.A, z.O, nb)
 PA.th5 = C.HE:path(z.C, z.N, nb) + C.BA:path(z.N, z.O, nb) - C.DE:path(z.C, z.O, nb)
 PA.th6 = C.HE:path(z.I, z.E, nb) - C.DE:path(z.P, z.E, nb) - C.GA:path(z.I, z.P, nb)}
\begin{center}
  \begin{tikzpicture}[scale = .5, rotate = -30]
    \tkzGetNodes
    \tkzFillCircle[fill=yellow!20](O,A)
    \tkzDrawCircle[ultra thick,black](O,A)
    \tkzClipCircle(O,A)
    \tkzDrawCoordinates[draw,ultra thick,fill=green!15](PA.th1)
    \tkzDrawCoordinates[draw,ultra thick,fill=green!15](PA.th2)
    \tkzDrawCoordinates[draw,ultra thick,fill=blue!15](PA.th3)
    \tkzDrawCoordinates[draw,ultra thick,fill=blue!15](PA.th4)
    \tkzDrawCoordinates[draw,ultra thick,fill=red!15](PA.th5)
    \tkzDrawCoordinates[draw,ultra thick,fill=red!15](PA.th6)
    \tkzDrawCircle[ultra thick,black](O,A)
    \end{tikzpicture}
\end{center}
```

In addition to manually constructing paths from a list of points, you can also automatically generate paths using geometric classes. This is particularly useful when drawing arcs, segments, or interpolated curves. Currently, the following classes provide built-in methods to generate path objects:

- **line** — segment interpolation between two points See [12.13.6]
- **triangle** — triangle outline as a path. See [14.11.2]
- **circle** — circular arcs between two points on the circumference See [13.10.3]
- **conic** — arcs or full curves parameterized. See [16.5.10]

These methods typically return a **path** object with a default or user-defined number of points. For instance:

```
PA.arc = C.OA:path(z.A, z.B, 40) -- circular arc from A to B on circle OA
PA.seg = L.AB:path(10) -- line segment from A to B with 10 points
PA.tri = T.ABC:path() -- triangle outline path
PA.co = CO.EL:points(0, 1, 50) -- conic arc from t=0 to t=1
```

These generated paths can be combined, reversed, filled, or used for Ti*k*Z decorations.

24.1. Overview

The path object represents an ordered sequence of 2D complex points, stored as TikZ-compatible strings in the form "(x,y)". It supports geometric operations such as translation, homothety, and rotation, and can be combined with other paths using Lua operator overloading.
The variable **PA** holds a table used to store paths. It is optional, and you are free to choose the variable name. However, using **PA** is a recommended convention for clarity and consistency. If you use a custom variable (e.g., Paths), you must initialize it manually. The **init_elements()** function reinitializes the **PA** table if used.
When working with path objects—especially when several intermediate curves are involved—it is strongly recommended to use indexed names such as **PA.p1**, **PA.p2**, etc., rather than a generic name.

> **Rule — Prefer explicit, indexed names:** Using **PA.p1**, **PA.p2**, **PA.result**, etc., improves clarity, avoids accidental overwriting, and ensures that all geometric data can be reset with a single call to **.**

This approach aligns with the overall philosophy of tkz-elements, where named tables like **PA**, **T**, **C**, and **L** are used to store structured geometric information. Indexing also helps make diagrams more readable, reproducible, and easier to debug.
It supports arithmetic operations for combining and manipulating paths, as well as geometric transformations such as translation, rotation, homothety, and reversal.

24.2. Notes

- Points are internally stored as strings like "(x,y)", and parsing is done via a utility function parse_point.

```
local x, y = utils.parse_point("(3.5, -2)")
-- x = 3.5, y = -2.0
```

- Number formatting (e.g., for Ti*k*Z compatibility) should be handled by checknumber, assumed to be defined globally or in a utility module.
- The class is designed to be lightweight and compatible with TikZ/LuaLaTeX workflows.

24.3. Constructor

```
PA.name = path(table_of_points) -- Creates a new path object.
```

If data is provided, the input should be a table of points written as strings, e.g., `{ "(0,0)", "(1,0)","(1,1)" }` otherwise creates an empty path.
Here is a **path** representing a simple triangle:

```
PA.triangle = path({ "(0, 0)", "(1, 0)","(1, 1)" })
```

24.4. Operator Overloading; metamethods

24.4.1. Table of metamethods

Table 34: Methods of the class vector.

Metamethods	Reference
add(path1,path2)	[24.4.2]
sub(path1,path2)	[24.4.3]
unm(path1)	[24.4.4]
tostring(path1)	[24.4.5]

24.4.2. Metamethod add

If p1 and p2 are two paths, we can obtain a third path with `p3 = p1:add(p2)`
More easily with an operator
p3 = p1 + p2 — Concatenation
Returns a new path by appending **p2** after **p1**.

24.4.3. Metamethod unm

`p2 = p1:unm()`
More easily with an operator
-p — Reversal
Returns a copy of the path in reverse order.

24.4.4. Metamethod sub

If p1 and p2 are two paths, we can obtain a third path with `p3 = p1:sub(p2)`
More easily with an operator
p1 - p2 — Subtraction
Equivalent to **p1 + (-p2)** (concatenates **p1** with the reversed **p2**).

24.4.5. Metamethod tostring

String Representation
The **tostring** metamethod provides a readable form, mainly for debugging:

```
tostring(p) --> path: { (x1,y1) , (x2,y2) , ... }
```

24.5. Methods

Table 35: Methods of the path class.

Method	Reference
add_point(z)	[24.5.1]
get(i)	[24.5.2]
copy()	[24.5.3]
count()	[24.5.4]
translate(dx, dy)	[24.5.5]
homothety(pt, k)	[24.5.6]
rotate(pt, an)	[24.5.7]
close()	[24.5.8]
sub(i1, i2)	[24.5.9]
show()	[24.5.10]
add_pair_to_path(z1, z2, n)	or add_pair [24.5.11]
concat(sep)	[24.5.12]

24.5.1. Method add_point(pt,<n>)

This method appends a point to the path.

Arguments:

- **pt**: a point object.
- **n** (optional): number of decimal places used when formatting the point coordinates for TikZ output.

If **n** is provided, the coordinates are rounded to **n** decimal places before being stored in the path. If omitted, a default formatting precision is used.

Note: The parameter **n** affects only the numerical representation in the generated TikZ code. It does not modify the internal geometric computation. Adds a point (given as a complex table) to the path. Let's take one of the first examples:

path: (0,0) , (1,0) , (1,1)

```
\directlua{
  PA.triangle = path()
  z.A = point(0, 0)
  z.B = point(1, 0)
  z.C = point(1, 1)
  PA.triangle:add_point(z.A, 0)
  PA.triangle:add_point(z.B, 0)
  PA.triangle:add_point(z.C, 0)
  tex.print(tostring(PA.triangle))}
```

24.5.2. Method get(i)

The method **get(i)** returns the i-th point stored in the path PA.p.
Paths in tkz-elements store their points as formatted coordinate strings (e.g. "(1.000,2.000)") for compatibility with TikZ and for efficient transfer from Lua to TEX. When **get(i)** is called, these coordinates are automatically converted back into a proper Lua point object.

Return value:
A point object corresponding to the i-th element of the path.

Errors:
If the index is outside the valid range, an error is raised:

24.5.3. Method copy()

Returns a deep copy of the path

```
\directlua{
  PA.p1 = path({ "(0, 0)", "(1, 0)","(1, 1)" })
  PA.p2 = PA.p1:copy()
  PA.p2:add_point(point(0, 0),0)}
\begin{tikzpicture}
  \tkzDrawCoordinates(PA.p2)
\end{tikzpicture}
```

24.5.4. Method count()

Returns a deep copy of the path

4

```
\directlua{
  PA.p1 = path({ "(0, 0)", "(1, 0)","(1, 1)" })
  PA.p2 = PA.p1:copy()
  PA.p2:add_point(point(0, 0),0)
  tex.print(PA.p2:count())
  }
```

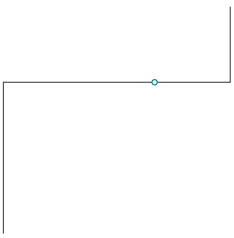
24.5.5. Method translate(dx,dy)

Translates the path by a given vector (dx, dy)

```
\directlua{
  PA.p1 = path({ "(0,0)", "(1,0)", "(1,1)" })
  PA.p2 = PA.p1:translate(2,1)}
\begin{tikzpicture}
\tkzDrawCoordinates(PA.p2)
\end{tikzpicture}
```

24.5.6. Method homothety(pt,r)

Applies a homothety centered at pt with ratio k



```
\directlua{
   z.O = point(0, 0)
   PA.base = path({ "(0,0)", "(1,0)", "(1,1)" })
   PA.scaled = PA.base:homothety(z.O, -2)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates(PA.base)
  \tkzDrawCoordinates(PA.scaled)
  \tkzDrawPoint(O)
\end{tikzpicture}
```

24.5.7. Method rotate(pt, an)

Rotates the path around *pt* by angle θ (radians)

```
\directlua{
   z.O = point(1, 1)
   PA.base = path({ "(1,0)", "(2,0)", "(2,1)" })
   PA.rotated = PA.base:rotate(z.O, math.pi / 2)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawCoordinates(PA.base)
  \tkzDrawCoordinates(PA.rotated)
  \tkzDrawPoint(O)
\end{tikzpicture}
```

24.5.8. Method close()

Closes the path by repeating the first point at the end

```
\directlua{
  PA.p = path({ "(0,0)", "(1,0)", "(1,1)" })
  PA.closed = PA.p:close()}
\begin{tikzpicture}
  \tkzDrawCoordinates[fill=green!20](PA.closed)
\end{tikzpicture}
```

24.5.9. Method sub(i1, i2)

Returns a subpath between two indices

```
\directlua{
  PA.full = path({ "(0,0)", "(1,0)", "(1,1)", "(0,1)", "(0,0)" })
  PA.part = PA.full:sub(2, 4)}
\begin{tikzpicture}
  \tkzDrawCoordinates[fill=green!20](PA.part)
\end{tikzpicture}
```

24.5.10. Method show()

Prints the path in the terminal (for debugging)

(0,0)(1,0)(1,1)

```
\directlua{
  PA.p = path({ "(0,0)", "(1,0)", "(1,1)" })
  PA.p:show()}
```

24.5.11. Method add_pair_to_path(p, p, n) or add_pair(p, p, n)

```
path:add_pair(z1, z2, decimals)
```

or equivalently:

```
path:add_pair_to_path(z1, z2, decimals)
```

Purpose:
The coordinates of **z1** and **z2** are extracted, formatted with a fixed number of decimals n, and combined into a string **"x1/y1/x2/y2"** appended to the path. This string is then inserted into the current path table.This method appends to a **path** object. a formatted point pair. It is mainly used to store segments, by keeping together the coordinates of two points.

Arguments:

- **z1**, **z2** – points given as tables **{re, im}**.
- **decimals** (optional) – number of decimals (default: **5**).

Returns:
Nothing (modifies the path in place).

Example usage:

```
local PA.p = path() -- empty path
local z.A = point(1, 2)
local z.B = point(3, 4)
PA.p:add_pair(z.A, z.B, 2)
-- PA.p = {"1.00/2.00/3.00/4.00"}
```

24.5.12. Method concat(sep)

This method returns a string obtained by concatenating all the coordinates stored in a **path** object. The coordinates are joined in order using a separator.

Internally, a **path** contains an ordered Lua table of points written in TikZ syntax, such as `(x,y)`. The **concat** method simply joins these entries into a single textual path.

Syntax:

```
S = PA.myPath:concat(sep)
```

Arguments:

- **sep** — optional string used to separate the coordinates (default: a single space `" "`).

Returns: A Lua string containing all points of the path, separated by **sep**. The resulting string is suitable for direct insertion into a TikZ path.
Example:

```
\directlua{
init_elements()
local A = point(0, 0)
local B = point(2, 1)
local C = point(3, 0)
local P = path:new()
P:add_point(A)
P:add_point(B)
P:add_point(C)
local S = P:concat(" -- ")
}
```

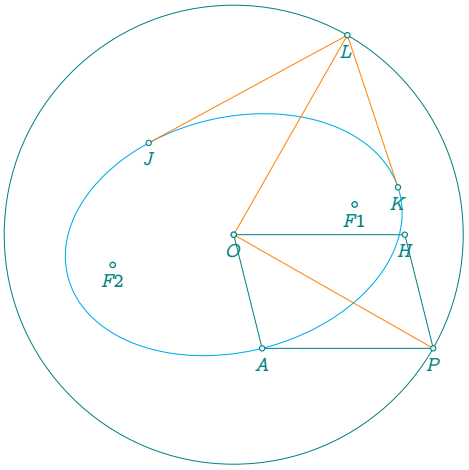
Usefulness:
This method is useful when:

- building TikZ paths dynamically from Lua;
- exporting or combining geometric paths;
- generating the argument of macros such as `\tkzDrawPath`.

24.5.13. Example : Director circle

Conics are drawn using the **path** tool.

```
   \directlua{
    init_elements()
    z.O = point(0, 0)
    z.F1 = point(4, 1)
    z.F2 = point(-4, -1)
    z.H = point(4 * math.sqrt(2), 0)
    local a = tkz.length(z.O, z.H)
    CO.EL = conic(EL_bifocal(z.F2, z.F1, a))
    PA.curve = CO.EL:points(0, 1, 50)
    z.A = CO.EL.covertex
    T.HOA = triangle(z.H, z.O, z.A)
    z.P = T.HOA:parallelogram()
    C.OP = circle(z.O, z.P)
    z.L = C.OP:point(0.25)
    T.LJ, T.LK = CO.EL:tangent_from(z.L)
    z.J = T.LJ.pb
    z.K = T.LK.pb}
```

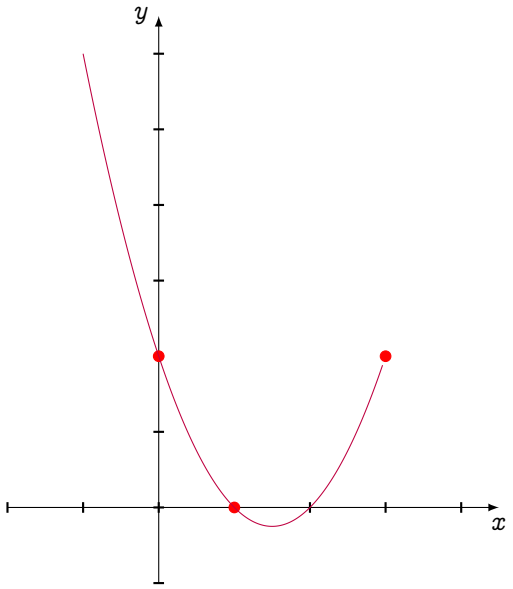



```
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPoints(F1,F2,O)
 \tkzDrawCircles[teal](O,P)
 \tkzDrawPolygon(H,O,A,P)
 \tkzDrawCoordinates[smooth,cyan](PA.curve)
 \tkzDrawSegments[orange](O,P O,L L,J L,K)
 \tkzDrawPoints(F1,F2,O,H,A,P,L,J,K)
 \tkzLabelPoints(F1,F2,O,H,A,P,L,J,K)
\end{tikzpicture}
```

24.5.14. Classic parabola

This example moves away from the concept of Euclidean geometry, but the method used can be reused. Here we want to draw a parabola whose axis of symmetry is parallel to the y-axis. Knowing three of its points, we can determine the equation. The set of points for tracing the parabola is obtained using the tools provided by the **path** class.

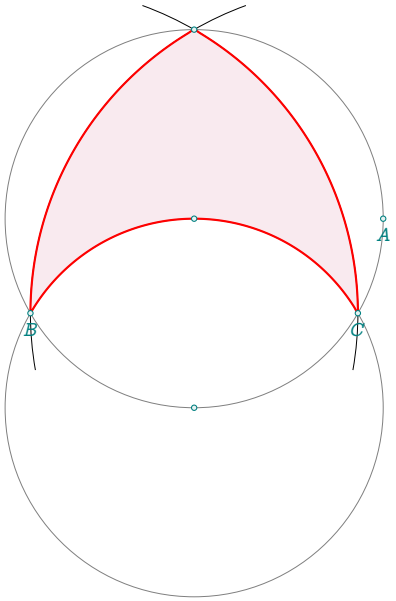
```
\directlua{
 init_elements()
 z.a = point(1, 0)
 z.b = point(3, 2)
 z.c = point(0, 2)
 local A, B, C = parabola(z.a, z.b, z.c)

 function f(t0, t1, n)
  local PA.tbl = path()
  for t = t0, t1, (t1 - t0) / n do
   local y = A * t ^ 2 + B * t + C
    local pt = point(t, y)
    PA.tbl:add_point(pt)
  end
  return PA.tbl
 end}
```




```
\begin{tikzpicture}
 \tkzGetNodes
 \tkzInit[xmin = -2,xmax=4,ymin =-1,ymax=6]
 \tkzDrawX\tkzDrawY
```

```
 \tkzDrawPoints[red,size=4pt](a,b,c)
 \draw[smooth] plot coordinates {%
 \directlua{tex.print(f(-1,3,100))}};
\end{tikzpicture}
```

24.6. Example with several paths




```
\directlua{
  z.O = point(0, 0)
  z.A = point(5, 0)
  C.OA = circle(z.O, z.A)
  z.S = C.OA.south
  C.SO = circle(z.S, z.O)
  z.B, z.C = intersection(C.OA, C.SO)
  C.BC = circle(z.B, z.C)
  z.D = intersection(C.OA, C.BC)
  C.CD = circle(z.C, z.D)

  local p1 = C.SO:path(z.C, z.B, 50)
  local p2 = C.BC:path(z.C, z.D, 50)
  local p3 = C.CD:path(z.D, z.B, 50)
  thepath = (-p1) + p2 + p3 }
\begin{tikzpicture}[scale=.5]
  \tkzGetNodes
  \tkzDrawCircles(O,A S,O)
  \tkzDrawArc(B,C)(D)
  \tkzDrawArc(C,D)(B)
  \tkzDrawCoordinates[fill = purple!20,
                      opacity=.4](thepath)
  \tkzDrawCoordinates[smooth,red,thick](thepath)
  \tkzDrawPoints(A,O,B,C,S,D)
  \tkzLabelPoints(A,B,C)
\end{tikzpicture}
```

25. Class list_point

The variable LP holds a table used to store **list_point** objects. Its use is optional, and users are free to choose any variable name.

However, for the sake of consistency and readability throughout the documentation, it is recommended to use the predefined variable LP. The function **init_elements()** automatically reinitializes this table.

The **list_point** class represents an ordered list of points. It is primarily intended as a lightweight container used to store and manipulate multiple points produced by geometric constructions, intersections, or iterative algorithms.
Unlike geometric objects such as lines or circles, a **list_point** object does not define intrinsic geometry. Its purpose is to facilitate grouping, iteration, and data transfer between **Lua** and **TikZ**.

A **list_point** object behaves like a Lua table indexed by integers, while providing a set of convenience methods adapted to geometric workflows.

25.1. Attributes

Table 36: list_point attributes.

Attribute	Meaning	Reference
n	[25.1.1]	
items	[25.1.2]	

25.1.1. Attribute n

The attribute **n** denotes the number of points stored in the list. In practice, it corresponds to the Lua length operator #P (array part).

25.1.2. Attribute items

The attribute **items** refers to the ordered sequence of points stored in the list. Individual points are accessed using standard Lua indexing, e.g. P[i].

25.2. Creating an object

25.2.1. Method new(...)

Creates a new **list_point** object.
In user code, the recommended constructor is the callable form **list_point(...)**. Internally, this calls the method **new(...)**.

A **list_point** object can be created either as a local Lua variable or as an entry of the table **LP**, which is reserved for storing lists of points.

```
local P = list_point()
LP.Q = list_point(z.A, z.B, z.C)
```

Both forms are valid and produce independent **list_point** objects.

Using a local variable is convenient for temporary computations or intermediate results. However, for objects intended to be reused, transferred to **TikZ**, or accessed outside a local scope, it is recommended to store them in the LP table.

The function **init_elements()** automatically reinitializes the LP table.

25.3. Methods or Basic accessors

Table 37: list_point methods.

Method	Reference
Constructor	
new(...)	[25.2.1]
Methods Returning a Integer Number	
len()	[25.3.1]
Methods Returning a Point	
get(i)	[25.3.2]
barycenter()	[25.6.1]
Methods Returning a List_Point	
add(p)	[25.4.1]
extend(pl)	[25.4.2]
map(f)	[25.5.2]
unpack()	[25.3.3]
clear()	[25.4.3]
foreach(f)	[25.5.1]
bbox()	[25.6.2]
to_path()	[25.7.1]

25.3.1. Method len()

Returns the number of points stored in the list.

25.3.2. Method get(i)

Returns the point at index **i**.

25.3.3. Method unpack()

Returns all points as separate values using **table.unpack**. This method is useful when interfacing with functions expecting individual point arguments.

```
\directlua{
init_elements()
LP = list_point()
LP:add(point(0,0))
LP:add(point(1,0))
LP:add(point(1,1))
z.A, z.B, z.C = LP:unpack()
}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C)
\end{tikzpicture}
\end{center}
```

25.4. Modification methods

25.4.1. Method add(p)

Appends the point **p** to the list and returns the object itself.

25.4.2. Method extend(pl)

Appends all points from another **list_point** object **pl**.

25.4.3. Method clear()

Removes all points from the list.
These methods modify the object in place and allow method chaining.

25.5. Iteration helpers

25.5.1. Method foreach(f)

Applies the function **f(p, i)** to each point **p** with its index **i**.

25.5.2. Method map(f)

Applies the function **f(p, i)** to each point and returns a new **list_point** object containing the results.
These methods support a functional programming style for geometric transformations.

25.6. Geometric utilities

25.6.1. Method barycenter()

Returns the barycenter (centroid) of all points in the list. If the list is empty, the method returns **nil**.

25.6.2. Method bbox()

Returns the bounding box of the point set as four numbers:

$$(x_{\min}, y_{\min}, x_{\max}, y_{\max})$$

25.7. TikZ output

25.7.1. Method to_path()

Converts the list of points into a TikZ path string of the form

$$(x_1, y_1) - - (x_2, y_2) - - ...$$

suitable for drawing with TikZ commands.

This class provides a minimal yet effective bridge between **Lua** computations and **TikZ** drawing instructions.

25.8. Examples

25.8.1. Temporary list of points

This example illustrates the use of a local `list_point` object to store temporary results inside a computation.

```
local P = list_point()
for i = 1, 10 do
  P:add(z.A + i * (z.B - z.A) / 10)
end
local G = P:barycenter()
```

The list **P** is used only locally to compute the barycenter of a set of points.

25.8.2. Storing construction results in LP

This example shows how to store a list of intersection points for later use.

```
LP.I = list_point()
local X, Y = intersection(L.AB, C.circ)
LP.I:add(X)
LP.I:add(Y)
```

Storing the points in LP makes them accessible for drawing or further constructions.

25.8.3. Iterative construction

A typical use case is the progressive construction of a family of points.

```
LP.P = list_point()
for i = 0, 24 do
  local t = i / 24
  LP.P:add(L.AB:point(t))
end
```

The list **LP.P** contains a discrete sampling of a segment.

25.8.4. Functional transformation

This example uses the `map` method to transform a set of points.

```
local P = list_point(z.A, z.B, z.C)
LP.Q = P:map(function(p)
  return rotation_(p, tkz.tau / 3, z.O)
end)
```

The original list **P** is preserved, while **LP.Q** stores the transformed points.

25.8.5. Conversion to a TikZ path

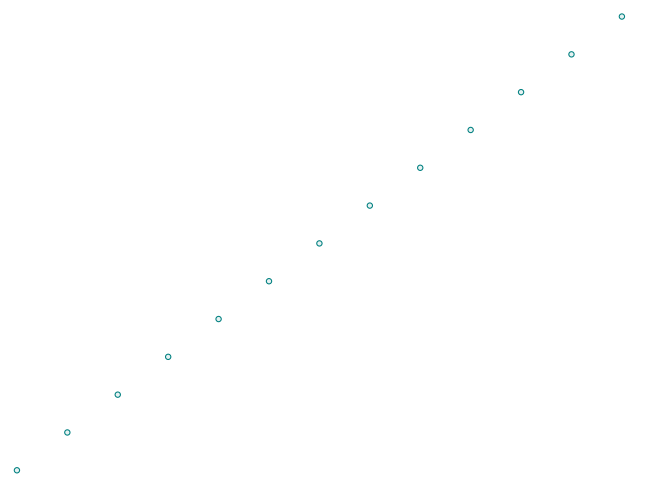
This example illustrates the interaction between **list_point** and TikZ.

```
LP.P = list_point(z.A, z.B, z.C)
local path = LP.P:to_path()
```

The string returned by **to_path** can be passed directly to TikZ drawing commands.

25.8.6. From list_point to path and drawing with tkz-euclide

This example shows a typical workflow: compute a list of points in **Lua**, convert it into a **path**, and then draw the resulting points with **tkzDrawPointsFromPath**.



```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(8, 6)
  L.AB = line(z.A, z.B)
  % list of sampled points
  LP.P = list_point()
  for i = 0, 12 do
    LP.P:add(L.AB:point(i/12))
  end
  % convert list_point -> path
  PA.sample = path()
  for i = 1, LP.P:len() do
    PA.sample:add_point(LP.P:get(i))
  end
}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
% draw the points stored in the path
\tkzDrawPointsFromPath(PA.sample)
\end{tikzpicture}
\end{center}
```

The points are stored in **LP.P** for later reuse, while **PA.sample** contains the corresponding TikZ path representation used for drawing.

26. Class angle

The **angle** class is an experimental helper object used to represent an angle defined by three points. It is currently self-contained and does not interact with other classes. Its main purpose is to provide a simple and direct interface for obtaining:

- the oriented angle in radians,
- the normalised oriented angle in $[0, 2\pi]$
- the interior (non-oriented) angle,
- the measure of the interior angle in degrees.

An **angle** object is *static*: all values are computed at creation time and never updated.

26.1. Creating an object

```
local alpha  = angle(z.A, z.B, z.C)   -- equivalent to angle:new(A,B,C)
local beta   = angle(z.B, z.C, z.A)
local gamma  = angle(z.C, z.A, z.B)
```

The three arguments are:

- : the vertex of the angle,
- : the first point defining the first ray,
- : the second point defining the second ray.

26.2. Attributes

The following attributes are stored inside every **angle** object:

Table 38: Angle attributes.

Attribute	Meaning	Reference
ps	Vertex of the angle	[26.2]
pa	First defining point (ray [*pspa*])	[26.2]
pb	Second defining point (ray [*pspb*])	[26.2]
raw	Oriented angle (radians), may be negative	[26.2]
value	Non-oriented angle in the range $[0, \pi]$	[26.2]
deg	Interior angle in degrees	[26.2]

ps The vertex (summit) of the angle.

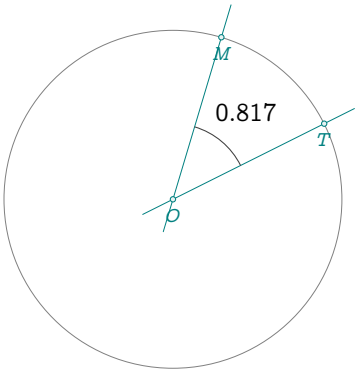
pa A point defining the first ray [*pspa*].

pb A point defining the second ray [*pspb*].

raw The oriented angle in radians as returned by `get_angle_(ps,pa,pb)`; it may be negative.

norm The oriented angle normalised to the interval $[0, 2\pi)$.

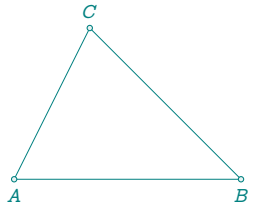
value Returns the interior (non-oriented) angle in the range $[0, \pi]$:
Example:




```
\directlua{%
  init_elements()
  z.O = point(0, 1)
  z.T = point(2, 2)
  C.OT = circle(z.O, z.T)
  z.M = C.OT:point(.13)
  A.OTM = angle(z.O, z.T, z.M)
  tkzA = A.OTM.value}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(O,T O,M)
  \tkzDrawCircle(O,T)
  \tkzDrawPoints(O,T,M)
  \tkzLabelPoints(O,T,M)
  \tkzMarkAngle(T,O,M)
  \tkzLabelAngle[pos=1.5](T,O,M){%
  \tkzPN[3]{\tkzUseLua{tkzA}}}
\end{tikzpicture}
\end{center}
```

deg Returns an interior angle in degrees.
Example:

Angle at A = 45.0 degrees



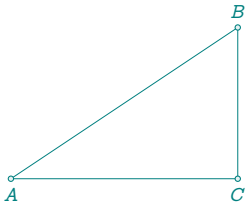

```
\directlua{
  init_elements()
  z.A = point(0,0)
  z.B = point(3,0)
  z.C = point(1,2)
  A.alpha = angle(z.B, z.A, z.C)
  tex.print("Angle at A = "..A.alpha.deg.." degrees")
}
\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C)
\end{tikzpicture}
\end{center}
```

Also possible:

```
T.ABC = triangle(z.A, z.B, z.C)
local val = T.ABC.alpha_.deg
```

All values are numerical scalars and remain fixed once the object is created.
Example:

A(value) = 0.58800260354757
A(raw) = -0.58800260354757
A(deg) = 33.69006752598




```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.C = point(3, 0)
  z.B = point(3, 2)
  T.ABC = triangle(z.A, z.B, z.C)
  A.A = T.ABC.alpha_
  A.B = T.ABC.beta_
  T.C = T.ABC.gamma_
  tex.print("A(value) = \\", A.A.value)
  tex.print('\\\\')
  tex.print("A(raw) = \\", A.A.raw)
  tex.print('\\\\')
  tex.print("A(deg) = \\", A.A.deg)}

\begin{center}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,C)
  \tkzLabelPoints[above](B)
\end{tikzpicture}
\end{center}
```

26.3. Methods

Table 39: angle methods.

Methods	Reference
Creation	
angle(ps, pa, pb)	[26.1]
Accessors	
get()	[26.3.1]
Tests	
is_direct()	[26.3.2]

26.3.1. get()

Returns the three defining points:

```
local ps, pa, pb = alpha:get()
```

26.3.2. is_direct()

Returns true when the angle is positive (counterclockwise orientation).

This class is *experimental* and may evolve in future versions.

27. Intersections

The **intersection** function is an essential geometric tool. It computes the intersection of two geometric objects, which may belong to the following classes:

- **line** with **line**,
- **line** with **circle**,
- **circle** with **circle**,
- **line** with **conic**.

Note that **circle** is a distinct class from **conic**. The **conic** class includes parabolas, hyperbolas, and ellipses.

The function takes a pair of objects as arguments, regardless of order. The result typically consists of one or two points, depending on the geometric configuration. If there is no intersection, the function may return **false** or **_**, depending on the context.

When two intersection points exist and you either:

- already know one of them, or
- want to select the one closest to a reference point,

you may use optional keys in the **opts** table.
In addition, numerical robustness can be controlled through an optional tolerance parameter **eps**, also stored in **opts**.

27.1. Optional arguments: known, near, and EPS

The function **intersection(X, Y, opts)** computes the intersection between two geometric objects **X** and **Y**. The optional table **opts** may contain the following keys:

- known (*point*)
  This option is used when one of the two intersection points is already known. The function will return the *other* solution as the first value. This is particularly useful in constructions where one intersection is already defined:

```
z.X, z.Y = intersection(C1, C2, {known = z.Y})
```

  If **z.Y** is one of the intersection points, the function ensures that **z.X** receives the other.

- near (*point*)
  With this option, the function returns first the solution closest to the given reference point:

```
z.A, z.B = intersection(C1, L, {near = z.O})
```

  Here, **z.A** will be the intersection point closest to **z.O**.

- EPS (*number*)
  This optional key sets the numerical tolerance used internally when determining whether objects intersect, touch, or are tangential. If omitted, the global value **tkz.epsilon** is used:

```
z.A, z.B = intersection(C1, C2, {eps = 1e-6})
```

  This option improves robustness in configurations involving near-tangency or small numerical uncertainties.

If none of these options is supplied, the intersection points are returned in the default order determined by the geometric computation, and the global tolerance **tkz.epsilon** is used.

Note: These options only affect cases where the intersection returns multiple points. For single-point intersections (e.g., tangency), the result is unaffected. The **eps** option however always controls the numerical tolerance used internally.

Compatibility note: For convenience and backward compatibility, the third argument of **intersection** may also be a numerical tolerance:

```
z.A, z.B = intersection(C1, C2, 1e-5)
```

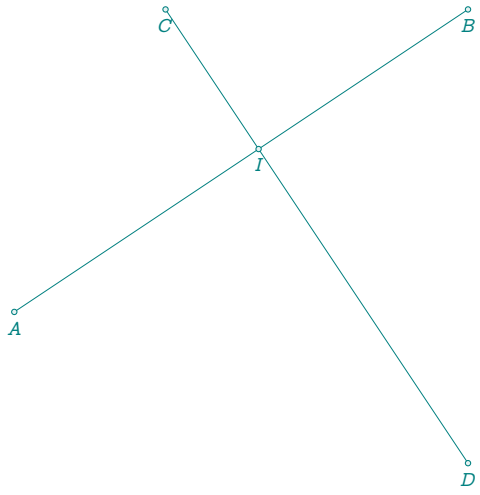
In this case, the number is interpreted as the value of **eps**, and no other option is provided. The recommended modern form is:

```
z.A, z.B = intersection(C1, C2, {eps = 1e-5})
```

27.2. Line-line

The result is of the form: point or false.

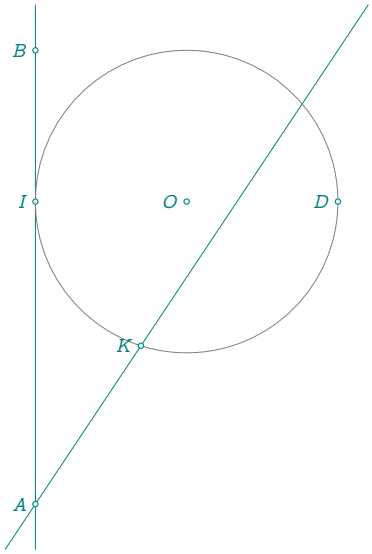
```
\directlua{
 init_elements()
 z.A = point(1, -1)
 z.B = point(4, 1)
 z.C = point(2, 1)
 z.D = point(4, -2)
 z.I = point(0, 0)
 L.AB = line(z.A, z.B)
 L.CD = line(z.C, z.D)
 x = intersection(L.AB, L.CD)
 if x == false then
  tex.print("error")
 else
  z.I = x
end}

\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawSegments(A,B C,D)
   \tkzDrawPoints(A,B,C,D,I)
   \tkzLabelPoints(A,B,C,D,I)
\end{tikzpicture}
```



27.3. Line-circle

The result is of the form: `point,point` or `false,false`. If the line is tangent to the circle, then the two points are identical. You can ignore one of the points by using the underscore: `_, point` or `point, _`. When the intersection yields two solutions, the order of the points is determined by the argument of `(z.p - z.c)` with c center of the circle and p point of intersection. The first solution corresponds to the smallest argument (arguments are between 0 and 2π).

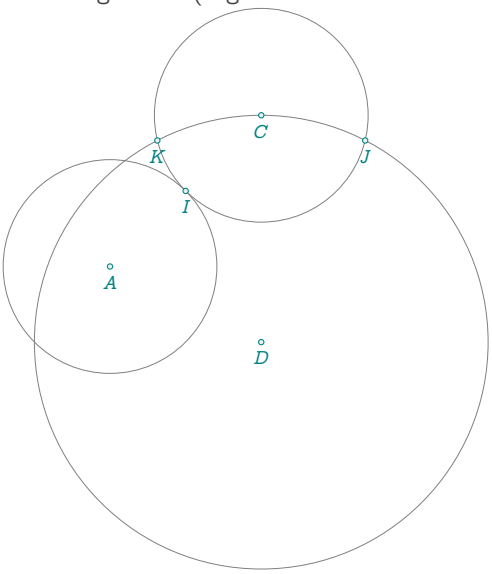
```
\directlua{
 init_elements()
 z.A = point(1, -1)
 z.B = point(1, 2)
 L.AB = line(z.A, z.B)
 z.O = point(2, 1)
 z.D = point(3, 1)
 z.E = point(3, 2)
 L.AE = line(z.A, z.E)
 C.OD = circle(z.O, z.D)
 z.I, _ = intersection(L.AB, C.OD)
 _, z.K = intersection(C.OD, L.AE)}

\begin{tikzpicture}
\tkzGetNodes
   \tkzDrawLines(A,B A,E)
   \tkzDrawCircle(O,D)
   \tkzDrawPoints(A,B,O,D,I,K)
   \tkzLabelPoints[left](A,B,O,D,I,K)
\end{tikzpicture}
```



27.4. Circle-circle

The result is of the form: `point,point` or `false,false`. If the circles are tangent, then the two points are identical. You can ignore one of the points by using the underscore: `_ , point` or `point , _`. As for the intersection of a line and a circle, consider the argument of `z.p-z.c` with `c` center of the first circle and `p` point of intersection. The first solution corresponds to the smallest argument (arguments are between 0 and 2π).

```
\directlua{
 init_elements()
 z.A = point(1, 1)
 z.B = point(2, 2)
 z.C = point(3, 3)
 z.D = point(3, 0)
 C.AB = circle(z.A, z.B)
 C.CB = circle(z.C, z.B)
 z.I, _ = intersection(C.AB, C.CB)
 C.DC = circle(z.D, z.C)
 z.J, z.K = intersection(C.DC, C.CB)}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawCircles(A,B C,B D,C)
   \tkzDrawPoints(A,I,C,D,J,K)
   \tkzLabelPoints(A,I,C,D,J,K)
\end{tikzpicture}
```

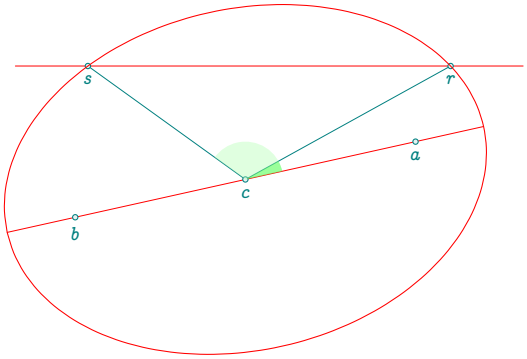


Other example: 4.4

27.5. Line-conic

The following example is complex, but it shows the possibilities of Lua. The designation of intersection points is a little more complicated than the previous one, as the argument characterizing the major axis must be taken into account. The principle is the same, but this argument must be subtracted. In concrete terms, you need to consider the slopes of the lines formed by the center of the ellipse and the points of intersection, and the slope of the major axis.

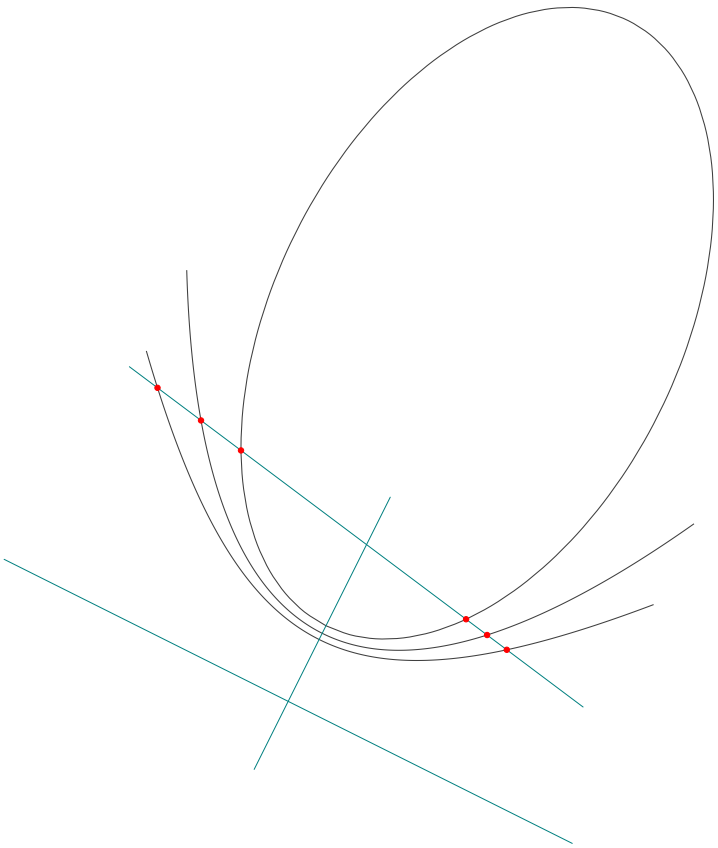
```
\directlua{
 init_elements()
 z.a = point(5, 2)
 z.b = point(-4, 0)
 L.ab = line(z.a, z.b)
 z.c = L.ab.mid
 z.v = L.ab:point(-0.2)
 local a = tkz.length(z.c, z.v)
 local c = 0.5 * tkz.length(z.a, z.b)
 local e = c / a
 z.K = L.ab:report(a ^ 2 / c, z.c)
 z.Kp = (z.K - z.a):orthogonal(2):at(z.K)
 L.dir = line(z.K, z.Kp)
 CO.EL = conic(z.b, L.dir, e)
 PA.EL = CO.EL:points(0, 1, 50)
 z.m = point(2, 4)
 z.n = point(4, 4)
 L.mn = line(z.m, z.n)
 z.r, z.s = intersection(CO.EL, L.mn)}


  \begin{tikzpicture}[scale =.5]
     \tkzGetNodes
     \tkzDrawLines[red](a,b r,s)
     \tkzDrawSegments(c,r c,s)
     \tkzDrawPoints(a,b,c,r,s)
     \tkzLabelPoints(a,b,c,r,s)
     \tkzDrawCoordinates[smooth,red](PA.EL)
     \tkzFillAngles[green!30,opacity=.4](v,c,s)
     \tkzFillAngles[green!80,opacity=.4](v,c,r)
  \end{tikzpicture}
```



27.5.1. Intersection all subtypes of conics

```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(4, -2)
 L.dir = line(z.A, z.B)
 z.F = point(2, 2)
 CO.EL = conic(z.F, L.dir, 0.8)
 CO.PA = conic(z.F, L.dir, 1)
 CO.HY = conic(z.F, L.dir, 1.2)
 PA.EL = CO.EL:points(0, 1, 50)
 PA.PA = CO.PA:points(-5, 5, 50)
 PA.HY = CO.HY:points(-5, 5, 50)
 z.K = CO.EL.K
 z.u, z.v = CO.EL.major_axis:get()
 z.x = L.dir:report(-4, z.K)
 z.y = L.dir:report(4, z.K)
 z.r = point(0, 4)
```

```
 z.s = point(4, 1)
 L.rs = line(z.r, z.s)
 z.u_1, z.u_2 = intersection(L.rs, CO.EL)
 z.v_1, z.v_2 = intersection(L.rs, CO.PA)
 z.w_1, z.w_2 = intersection(L.rs, CO.HY)}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawCoordinates[smooth](PA.EL)
 \tkzDrawCoordinates[smooth](PA.PA)
 \tkzDrawCoordinates[smooth](PA.HY)
 \tkzDrawLines[add =.5 and .5](r,s u,v)
 \tkzDrawLines(x,y)
 \tkzDrawPoints[red](u_1,u_2,v_2,v_1,w_1,w_2)
\end{tikzpicture}
```



27.5.2. Intersection line-parabola, explained

In this example, we're looking for a parabola inscribed in a triangle, i.e. tangent to the triangle's three sides. I won't go into detail about the first part to obtain the parabola. You'll notice this line

```
L.euler = T:euler_line():swap_line()
```

it swaps the ends of the Euler line, as we'll see later.
To construct the points of contact, it is necessary to find the intersections of the parabola with the sides:

```
z.ta = intersection(PA, T.bc)
z.tb = intersection(PA, T.ca)
z.tc = intersection(PA, T.ab)
```

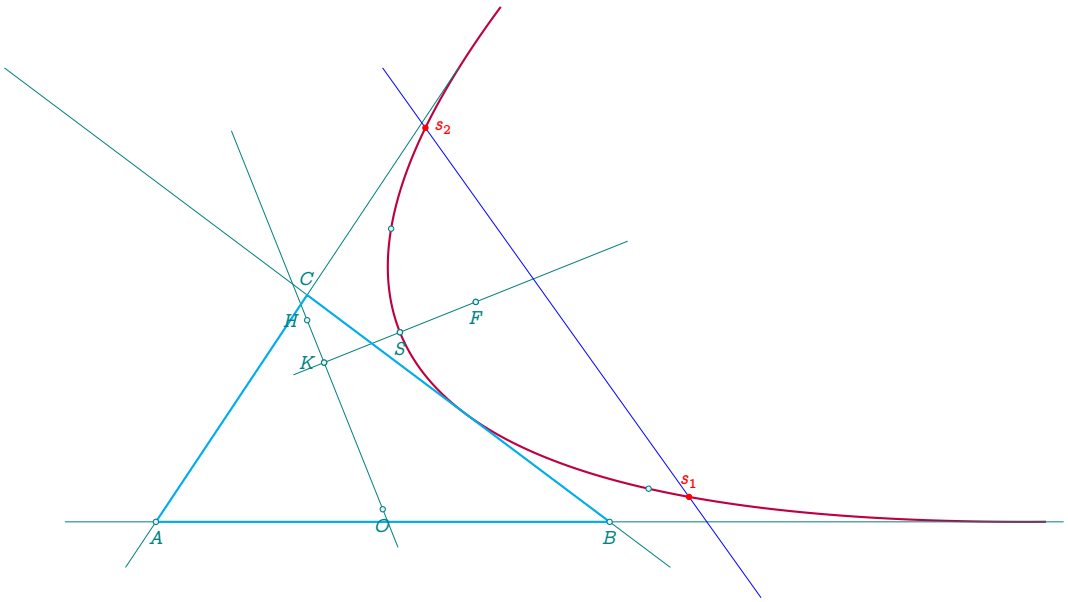
We will now detail how to determine the intersection of a line (ab) with the parabola. In this case, Euler's line serves as the directrix of the parabola. Its points have been swapped to maintain the correct order of abscissas—that is, negative values on the left and positive values on the right.
To simplify calculations, it is useful to change the coordinate system by setting the vertex of the parabola as the origin. The focal axis (**major_axis**), oriented from K to F, becomes the ordinate axis, while the abscissa axis is chosen so that the new system is **direct**.
I have kept `z.U = OCCS.x` and `z.V = OCCS.y` in the code to visualize the new coordinate system, for example, using `\tkzDrawSegments[red,->](S,U S,V)`. This new system is created with:

```
O.SKF  = occs(L.KF,z.S)
```

The line (KF), the axis of symmetry of the parabola, becomes the ordinate axis. In this new coordinate system, the equation of the parabola is $y = \dfrac{x^2}{2p}$, where p is the distance KF, also known as the **latus rectum**.
The **coordinates** method of the **occs** class allows you to obtain the new coordinates of each point. The **param_line** function calculates the coefficients of the line's equation (this function is currently internal and its name may change). Then, **solve_para_line** is used to find the common points between the line and the parabola (again, this function is internal and subject to modification).
The result is two abscissas that must be placed on the axis passing through S and orthogonal to the focal axis. This is why it was important to position the curve correctly. If you remove `swap_line` for Euler's line, you will see that the curve becomes the reflection of the previous one. While the parabola remains unchanged overall, the intersection points will not.
Finally, the abscissas of the intersection points must be placed, and then the intersections of the lines orthogonal to Euler's line passing through these abscissas with the line (ab) must be determined.
Note: This geometric method is more appropriate than determining the intersection points' coordinates using formulas. Indeed, those coordinates would be expressed in the new coordinate system, requiring an additional transformation to return to the original system.

```
\directlua{
   init_elements()
   z.A = point(0, 0)
   z.B = point(6, 0)
   z.C = point(2, 3)
   T.ABC  = triangle(z.A, z.B, z.C)
   L.euler = T.ABC:euler_line():swap_line()
   z.F = T.ABC:kimberling(110)
   z.H = T.ABC.orthocenter
   z.O = T.ABC.circumcenter
   z.Ω = point(0, 0)
   z.i = point(1, 0)
   z.j = point(0, 1)
   CO.PA  = conic(z.F, L.euler, 1)
   PA.curve = CO.PA:points(-3.5, 5.5, 50)
   local p = CO.PA.p
   z.K  = CO.PA.K
   z.S  = tkz.midpoint(z.F,z.K)
   L.KF = CO.PA.major_axis
   z.ta = intersection(CO.PA,T.ABC.bc)
   z.tb = intersection(CO.PA,T.ABC.ca)
   z.tc = intersection(CO.PA,T.ABC.ab)
% new occs
  O.SKF = occs(L.KF, z.S)
  z.U = O.SKF.x
  z.V = O.SKF.y
% line a,b
  z.a = point(3, 6)
  z.b = point(8, -1)
  L.ab = line(z.a, z.b)
```

```
% coordinates in the new occs
  Xa,Ya = O.SKF:coordinates(z.a)
  Xb,Yb = O.SKF:coordinates(z.b)
% solve in the new occs
  local r,s = tkz.line_coefficients(Xa, Ya ,Xb, Yb)
  r1,r2 = tkz.solve_quadratic_(1, -2 * p * r, -2 * p * s)
  z.x = O.SKF.abscissa:report(r1, z.K)
  z.y = O.SKF.abscissa:report(r2, z.K)
  L1 = L.euler:ortho_from(z.x)
  L2 = L.euler:ortho_from(z.y)
  z.s_1 = intersection(L.ab, L1)
  z.s_2 = intersection(L.ab, L2)}
  \begin{tikzpicture}
  \tkzGetNodes
   \tkzDrawCoordinates[smooth,purple,thick](PA.curve)
   \tkzDrawLines[add = .2 and 1](A,B A,C B,C K,F O,H)
   \tkzDrawPolygon[thick,cyan](A,B,C)
   \tkzDrawSegment[blue](a,b)
   \tkzDrawPoints(F,K,H,S,O)
   \tkzDrawPoints(A,B,F,K,S,ta,tb,tc)
   \tkzDrawPoints[red,size=2](s_1,s_2)
   \tkzLabelPoints[red,above](s_1)
   \tkzLabelPoints[red,right](s_2)
   \tkzLabelPoints(F,S,O,A,B)
   \tkzLabelPoints[above](C)
   \tkzLabelPoints[left](H,K)
  \end{tikzpicture}
```



28. Global Variables and constants

This module defines default values and mathematical constants used for precision control and symbolic computations in tkz-elements.

28.1. Global Variables

Table 40: Default settings and constants.

Variable	Description
tkz.nb_dec	Number of decimals used for formatting (default: 10)
tkz.epsilon	Tolerance used for floating-point comparisons ($10^{-\text{tkz.nb\_dec}}$)
tkz.dc	Number of decimals shown in output (default: 2)
Constant	**Description**
tkz.phi	Golden ratio $\varphi = \dfrac{1+\sqrt{5}}{2}$
tkz.invphi	Inverse golden ratio $1/\varphi = \dfrac{\sqrt{5}-1}{2}$
tkz.sqrtphi	Square root of the golden ratio $\sqrt{\varphi}$
tkz.pt	254 / 7227
tkz.deg	math.pi /1 80
tkz.rad	180 / math.pi

28.2. Functions

Table 41: Functions related to settings.

Function	Reference
tkz.reset_defaults()	Section 28.2.1
tkz.set_nb_dec(n)	Section 28.2.2

28.2.1. Function reset_defaults()

Restores default values for numerical precision and formatting:

- **tkz.nb_dec = 10**
- **tkz.epsilon = 1e-10**
- **tkz.dc = 2**

28.2.2. Function tkz.set_nb_dec(n)

Sets the number of decimals used in floating-point comparisons and updates the tolerance accordingly:

- **tkz.nb_dec =** n
- **tkz.epsilon =** 10^{-n}

29. Various functions

In addition to object-oriented constructions, the package provides a collection of auxiliary functions. These functions are not strictly required but serve to simplify many geometric operations and shorten code. Their use depends on the context, but in the spirit of the package, it's preferable to use objects.

They allow:

- direct geometric computation without having to define intermediate objects (e.g., lines or triangles),
- quick tests (alignment, orthogonality),
- or access to commonly used constructs (midpoints, barycenters, bisectors).

These tools follow the same convention as the rest of the package and accept either point objects or complex numbers (when appropriate). They are typically used in Lua code blocks when you want to keep your scripts minimal and avoid explicit variable declarations.

Example:
To compute the midpoint of a segment without defining the line:

```
z.M = tkz.midpoint(z.A, z.B)
```

This avoids having to define **L.AB = line(z.A, z.B)** and then access **L.AB.mid**.
Overview: The table below summarizes the available functions:

Table 42: Functions.

Functions	Reference
tkz.length(z1, z2)	[29.1]
tkz.midpoint(z1, z2)	[29.2]
tkz.midpoints(z1, z2, ..., zn)	[29.2]
tkz.is_linear(z1, z2, z3)	[29.13]
tkz.is_ortho(z1, z2, z3)	[29.13]
tkz.bisector(z1, z2, z3)	[29.3; 29.14; 29.6]
tkz.bisector_ext(z1, z2, z3)	[29.14]
tkz.altitude(z1, z2, z3)	[29.14]
tkz.get_angle(z1, z2, z3)	[29.6]
tkz.inner_angle(z1, z2, z3)	[29.7]
tkz.angle_normalize(an)	[29.8]
tkz.get_angle_normalize	[29.10]
tkz.angle_between_vectors	[29.17]
tkz.barycenter ({z1,n1},{z2,n2}, ...)	[29.4]
tkz.dot_product(z1, z2, z3)	[29.12]
tkz.parabola(pta, ptb, ptc)	[29.18]
tkz.nodes_from_paths	29.19
tkz.fsolve(f, a, b, n [, opts])	29.20
tkz.derivative(f, x0 [, accuracy])	29.21

29.1. Length of a segment

The function **tkz.length(z1, z2)** returns the Euclidean distance between two points. It is equivalent to:

```
point.abs(z1 - z2)
```

This shortcut allows you to compute the length of a segment without manipulating complex numbers explicitly.

Alternative:
If you need to define the segment as a **line** object for later use (e.g., to draw it or to access other attributes such as the midpoint or direction), you can use:

```
L.AB = line(z.A, z.B)
l = L.AB.length
```

Recommendation:
Use **tkz.length(z1, z2)** for simple distance computations, especially within calculations or conditions. If you're building a full construction and need more geometric attributes, prefer defining the segment explicitly as a line object.

29.2. Midpoint and midpoints

As with **length**, the function **tkz.midpoint(z1, z2)** provides a shortcut for computing the midpoint of a segment:

```
z.M = tkz.midpoint(z.A, z.B)
```

This is equivalent to creating a **line** object and accessing its midpoint:

```
L.AB = line(z.A, z.B)
z.M = L.AB.mid
```

Polygonal midpoints: To compute the midpoints of a polygonal chain, use the function tkz.midpoints(...). This function returns the midpoints of each successive segment: z_1z_2, z_2z_3, ..., $z_{n-1}z_n$.

```
z.MA, z.MB, z.MC = tkz.midpoints(z.A, z.B, z.C, z.D)
```

Medial triangle: For triangles, it is often useful to compute the medial triangle — the triangle formed by the midpoints of the sides. After defining a triangle object:

```
T.abc = triangle(z.a, z.b, z.c)
z.ma, z.mb, z.mc = T.abc:medial()
```

This is equivalent to calling **midpoints** directly:

```
z.mc, z.ma, z.mb = tkz.midpoints(z.a, z.b, z.c)
```

If you already have a triangle object, you may also write:

```
z.mc, z.ma, z.mb = tkz.midpoints(T.ABC:get())
```

This avoids the need to extract the triangle's vertices manually, but T.abc:medial() is preferable.

Recommendation:
Use **midpoint** or **midpoints** for quick calculations when no object is needed. Prefer object methods like **medial()** when working within a structured construction.

29.3. Bisectors

The functions **tkz.bisector(z1, z2, z3)** and **tkz.bisector_ext(z1, z2, z3)** define internal and external angle bisectors, respectively, for a given vertex.
These functions return a **line** object that represents the bisector of angle $\widehat{z_2z_1z_3}$, originating from point **z1**.

Internal bisector: This defines the internal bisector of angle $\widehat{BAC}$, starting at **z.A**. The resulting line is stored as **L.Aa**.

```
  L.Aa = tkz.bisector(z.A, z.B, z.C)
```

External bisector:
This defines the external bisector at vertex **z.A**. It is orthogonal to the internal one and also originates at **z.A**.

```
L.Aa = tkz.bisector_ext(z.A, z.B, z.C)
```

Recommendation: Use these functions when you need the bisector explicitly as a **line** object (e.g., for intersection or drawing). If you are working within a triangle object, some methods may also provide access to bisectors directly.

29.4. Barycenter

The function **tkz.barycenter** computes the barycenter (or center of mass) of any number of weighted points. It is a general-purpose function that works independently of geometric objects.

Syntax:
The function takes a variable number of arguments. Each argument must be a table of the form:

{point, weight}

Example with three equally weighted points:

```
z.G = tkz.barycenter({z.A, 1}, {z.B, 1}, {z.C, 1})
```

This computes the centroid of triangle ABC. The same result can be obtained using the object attribute:

```
T.ABC = triangle(z.A, z.B, z.C)
z.G = T.ABC.centroid
```

General case:
The function can be applied to any number of points with arbitrary weights:

```
z.G = tkz.barycenter({z.A, 2}, {z.B, 3}, {z.C, 1}, {z.D, 4})
```

This computes:

$$G = \frac{2A + 3B + C + 4D}{10}$$

Use cases:

- Use **tkz.barycenter** when dealing with arbitrary sets of weighted points.
- In triangle geometry, prefer **T.ABC.centroid**, which is automatically computed and stored.

Note:
The result is returned as a **point** object and can be used in any subsequent construction.

29.5. Angles and the constant tkz.tau

Most trigonometric and geometric computations in **tkz-elements** are expressed in radians. A full turn is therefore represented by the constant

$$\tau = 2\pi.$$

For convenience, the value of τ is available to the user as
`tkz.tau = 2 * math.pi`
This constant is particularly useful when dealing with normalized or periodic angles. For instance:

- **tkz.tau / 4** corresponds to a right angle (90°),
- **tkz.tau / 2** corresponds to 180°,

- **3 * tkz.tau / 2** corresponds to 270°,
- **a % tkz.tau** normalizes any angle **a** to the interval $[0, \tau)$.

Using τ avoids the explicit appearance of 2π in numerical code and improves readability in geometric computations.

29.6. Function tkz.get_angle(pa, pb, pc)

The function **tkz.get_angle** returns the oriented angle at a point, given three points **A**, **B**, **C**.
It computes the oriented angle $\widehat{BAC}$ at vertex A, corresponding to the rotation from vector $\overrightarrow{AB}$ to vector $\overrightarrow{AC}$, in radians.

Syntax:

```
an = tkz.get_angle(pa, pb, pc)
```

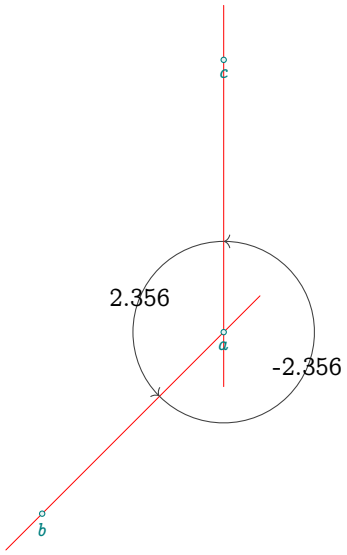
Arguments:

- **pa**, **pb**, **pc** — three points (Lua complex numbers).

Return value:

- an oriented angle (Lua number) in radians, in the interval $(-\pi, \pi]$.

Example:
The following code computes both angles $\widehat{CAB}$ and $\widehat{CAB}$ using point **a** as the vertex, and formats them for display using **tkzround**:




```
\directlua{
  init_elements()
  z.a = point(0, 0)
  z.b = point(-2, -2)
  z.c = point(0, 3)
  ang_cb = tkz.round(
    tkz.get_angle(z.a, z.c, z.b), 3)
  ang_bc = tkz.round(
    tkz.get_angle(z.a, z.b, z.c), 3)
}
\begin{center}
\begin{tikzpicture}[scale = 1.2]
  \tkzGetNodes
  \tkzDrawLines[red](a,b a,c)
  \tkzDrawPoints(a,b,c)
  \tkzLabelPoints(a,b,c)
  \tkzMarkAngle[->](c,a,b)
  \tkzLabelAngle(c,a,b){%
      \tkzUseLua{ang_cb}}
  \tkzMarkAngle[->](b,a,c)
  \tkzLabelAngle(b,a,c){%
     \tkzUseLua{ang_bc}}
\end{tikzpicture}
\end{center}
```

29.7. Function tkz.inner_angle(pa, pb, pc)

The function \tkzFct{tkz}{tkz.inner_angle} returns the interior (non-oriented) angle at a point, given three points **pa**, **pb**, **pc**.
It corresponds to the absolute value of the oriented angle $\widehat{BAC}$ in radians.

Syntax:

```
an = tkz.inner_angle(pa, pb, pc)
```

Arguments:

- **pa**, **pb**, **pc** — three points.

Return value:

- a non-oriented angle in radians, in the interval $[0,\pi]$.

This function is especially useful for working with triangle angles:

$$\alpha = \texttt{tkz.inner\_angle}(A,B,C),\quad \beta = \texttt{tkz.inner\_angle}(B,C,A),\quad \gamma = \texttt{tkz.inner\_angle}(C,A,B).$$

29.8. Function tkz.angle_normalize

The function \tkzFct{tkz}{tkz.angle_normalize} normalizes a real angle to the interval $[0,2\pi)$.

Syntax:
an2 = tkz.angle_normalize(an1)
Arguments:

- **an1** — a real number (angle in radians).

Return value:

- **an2** — the same angle, reduced modulo 2π into $[0,2\pi)$.

Application:
To compute the orientation of a line segment in the plane, we use the function **arg(z)** which returns the angle (in radians) between the positive horizontal axis and the vector represented by the complex number **z**. This value may be negative or greater than 2π depending on the direction of the vector.

To ensure consistency, especially when comparing angles or sorting them, the function **tkz.angle_normalize(an)** maps any angle to the interval $[0,2\pi]$.

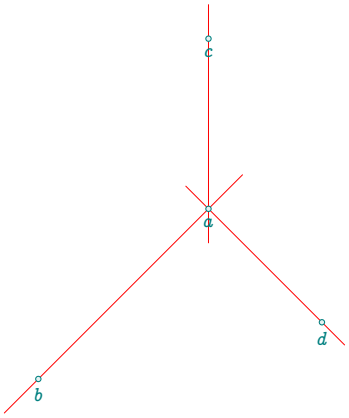
Example:
The following example computes and displays the slope (angle) of three vectors **(ab)**, **(ac)**, and **(ad)**. It shows both the raw angle and the normalized version.

```
\directlua{%
init_elements()
   z.a = point(0, 0)
   z.b = point(-3, -3)
   z.c = point(0, 3)
   z.d = point(2, -2)
   local angle = point.arg(z.b - z.a)
   tex.print('slope of (ab): '..tostring(angle)..'\\\\')
   tex.print('slope normalized of (ab): '..tostring(tkz.angle_normalize(angle))..'\\\\')
   local angle    = point.arg (z.c-z.a)
   tex.print('slope of (ac): '..tostring(angle)..'\\\\')
   tex.print('slope normalized of (ac): '..tostring(tkz.angle_normalize(angle))..'\\\\')
   local angle    = point.arg (z.d-z.a)
   tex.print('slope of (ad): '..tostring(angle)..'\\\\')
   tex.print('slope normalized of (ad): '..tostring(tkz.angle_normalize(angle))..'\\\\')
}
```

slope of (ab): -2.3561944901923
slope normalized of (ab): 3.9269908169872
slope of (ac): 1.5707963267949
slope normalized of (ac): 1.5707963267949
slope of (ad): -0.78539816339745

slope normalized of (ad): 5.4977871437821

```
\begin{tikzpicture}[scale = .75]
   \tkzGetNodes
   \tkzDrawLines[red](a,b a,c a,d)
   \tkzDrawPoints(a,b,c,d)
   \tkzLabelPoints(a,b,c,d)
\end{tikzpicture}
```



Note:
This technique is essential when working with angular comparisons, sorting directions (e.g., in convex hull algorithms), or standardizing outputs for further geometric processing.

29.9. Function tkz.is_direct

The function **tkz.is_direct** tests whether the oriented angle at a point is direct (counter-clockwise).

Syntax:

```
ok = tkz.is_direct(pa, pb, pc)
```

Arguments:

- **pa**, **pb**, **pc** — three points.

Return value:

- **true** if the oriented angle $\widehat{BAC}$ (from $\overrightarrow{AB}$ to $\overrightarrow{AC}$) is > 0 (direct, counter-clockwise),
- **false** otherwise.

29.10. Function tkz.get_angle_normalize

The function **tkz.get_angle_normalize** combines **tkz.get_angle** and **tkz.angle_normalize**.
It returns the oriented angle at a point, normalized to $[0, 2\pi)$.

Syntax:

```
an = tkz.get_angle_normalize(pa, pb, pc)
```

Arguments:

- **pa**, **pb**, **pc** — three points.

Return value:

- a real number in $[0, 2\pi)$: the normalized oriented angle at **pa**.

29.11. Function tkz.angle_between_vectors

The function **tkz.angle_between_vectors** returns the oriented angle between two vectors $\overrightarrow{AB}$ and $\overrightarrow{CD}$.

Syntax:

```
an = tkz.angle_between_vectors(a, b, c, d)
```

Arguments:

- **a**, **b** — endpoints of the first vector (from **a** to **b**),
- **c**, **d** — endpoints of the second vector (from **c** to **d**).

Return value:

- an oriented angle in radians, in the interval $(-\pi,\pi]$, corresponding to the rotation from $\overrightarrow{AB}$ to $\overrightarrow{CD}$.

Internally, the angle is computed using the standard atan2 formula based on the dot product and the determinant of the two vectors.

29.12. Function tkz.dot_product(z1, z2, z3)

The function computes the dot (scalar) product of the vectors $\overrightarrow{z_1z_3}$ and $\overrightarrow{z_1z_2}$:

$$(z_3-z_1)\cdot(z_2-z_1)$$

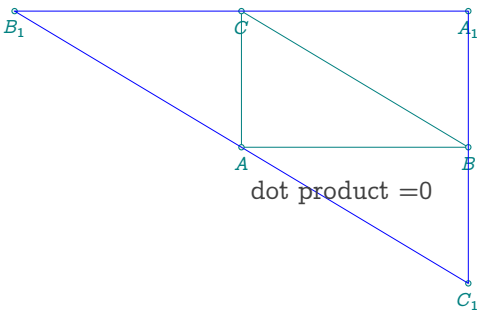
Note:
This is equivalent to **(z2 - z1)..(z3 - z1)** See also [22.2.7]
It is used to test orthogonality or to measure projection components. A result of zero indicates that the vectors are perpendicular.

Syntax:

tkz.dot_product(origin, point1, point2)

This returns the scalar product $\overrightarrow{z_1z_3}\cdot\overrightarrow{z_1z_2}$.
Example.
In the example below, the triangle ABC is constructed along with its triangle of antiparallels. The dot product of $\overrightarrow{AC}$ and $\overrightarrow{AB}$ is computed:

```
\directlua{%
 init_elements()
 z.A = point(0, 0)
 z.B = point(5, 0)
 z.C = point(0, 3)
 T.ABC = triangle(z.A, z.B, z.C)
 z.A_1,
 z.B_1,
 z.C_1  = T.ABC:anti():get()
 x  = tkz.dot_product(z.A, z.B, z.C)}
\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawPolygon(A,B,C)
   \tkzDrawPoints(A,B,C,A_1,B_1,C_1)
   \tkzLabelPoints(A,B,C,A_1,B_1,C_1)
   \tkzDrawPolygon[blue](A_1,B_1,C_1)
   \tkzText[right](0,-1){dot product =
                         \tkzUseLua{x}}
\end{tikzpicture}
```



Interpretation:
In this example, the dot product of vectors $\overrightarrow{AC}$ and $\overrightarrow{AB}$ is 0. Since the result is zero, the vectors are orthogonal.

29.13. Alignment and orthogonality tests

The functions **tkz.is_linear** and **tkz.is_ortho** are used to test the geometric relationships between three points, namely alignment and orthogonality.

tkz.is_linear(z1, z2, z3) This function returns **true** if the three points are aligned — that is, if the vectors

$overrightarrowz_1z_2$ and
$overrightarrowz_1z_3$ are collinear. Geometrically, this means the three points lie on the same straight line:

$$(z_2 - z_1) \parallel (z_3 - z_1)$$

```
if tkz.is_linear(z.A, z.B, z.C) then
  -- the points are collinear
end
```

This code replaces:

```
  L.AB = line(z.A, z.B)
  L.BC = line(z.B, z.C)
  if L.AB:is_parallel(L.BC) then
  --  the points are collinear
end
```

tkz.is_ortho(z1, z2, z3) This function returns **true** if the vectors $\overrightarrow{z_1z_2}$ and $\overrightarrow{z_1z_3}$ are orthogonal — that is, the scalar product between them is zero:

$$(z_2 - z_1) \cdot (z_3 - z_1) = 0$$

```
if tkz.is_ortho(z.A, z.B, z.C) then
  -- the angle between the segments is 90°
end
```

This code replaces:

```
  L.AB = line(z.A, z.B)
  L.BC = line(z.B, z.C)
  if L.AB:is_orthogonal(L.BC) then
  -- the angle between the segments is 90°
end
```

Use case:
These functions are particularly useful in decision structures or automated constructions (e.g., checking if a triangle is right-angled or degenerate). They return boolean values and do not create any geometric object.

29.14. Bisector and altitude

The functions **bisector**, **bisector_ext**, and **altitude** return **line** objects corresponding to classical triangle constructions. They are useful when you don't need to create a full **triangle** object and simply want the desired line or foot point directly.

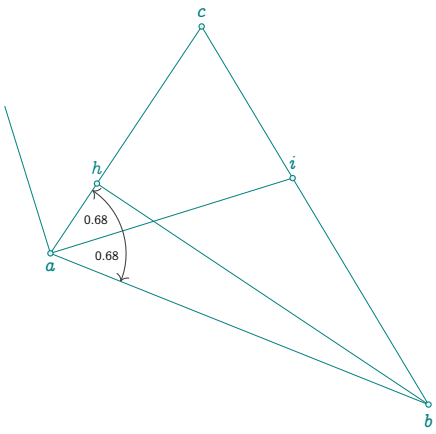
Internal bisector:
tkz.bisector(z1, z2, z3) returns the internal angle bisector at point **z1**, computed between vectors $\overrightarrow{z_1z_2}$ and $\overrightarrow{z_1z_3}$. The result is a **line** object; its foot (point **pb**) lies on the opposite side.

External bisector:
tkz.bisector_ext(z1, z2, z3) returns the external angle bisector at **z1**. It is orthogonal to the internal bisector.

Altitude:

tkz.altitude(vertex, foot1, foot2) returns the perpendicular from **vertex** to the line passing through **foot1** and **foot2**. The result is also a **line** object; the foot is stored in its **pb** attribute.

```
\directlua{
 z.a = point(0, 0)
 z.b = point(5, -2)
 z.c = point(2, 3)
 z.i = tkz.bisector(z.a, z.c, z.b).pb
 z.h = tkz.altitude(z.b, z.a, z.c).pb
 angic = tkz.round(tkz.get_angle(z.a, z.i, z.c), 2)
 angci = tkz.round(tkz.get_angle(z.a, z.b, z.i), 2)
 z.e = tkz.bisector_ext(z.a, z.b, z.c).pb

\begin{tikzpicture}
   \tkzGetNodes
   \tkzDrawPolygon(a,b,c)
   \tkzDrawSegments(a,i b,h a,e)
   \tkzDrawPoints(a,b,c,i,h)
   \tkzLabelPoints(a,b)
   \tkzLabelPoints[above](c,i,h)
   \tkzMarkAngle[->](i,a,c)
   \tkzLabelAngle[font=\tiny,pos=.75](i,a,c){\tkzUseLua{angci}}
   \tkzMarkAngle[<-](b,a,i)
   \tkzLabelAngle[font=\tiny,pos=.75](b,a,i){\tkzUseLua{angic}}
\end{tikzpicture}
```



Note:
These functions return full **line** objects, allowing access to their points and directions. This makes them ideal for flexible constructions where you do not need a structured **triangle** object.

29.15. Function tkz.is_linear

The function **tkz.is_linear(z1, z2, z3)** checks whether three points are collinear. It returns **true** if the vectors $\overrightarrow{z_1z_2}$ and $\overrightarrow{z_1z_3}$ are parallel — i.e., if the points lie on the same line:

$$(z_2 - z_1) \parallel (z_3 - z_1)$$

This is equivalent to testing whether the oriented area of triangle (z_1, z_2, z_3) is zero.

Syntax:

tkz.is_linear(z1, z2, z3) → **boolean**

Example:
The following code tests whether the points A, B and C are aligned. If so, the new point D is set to $(0,0)$; otherwise it is set to $(-1,-1)$.

Note:
The function does not create any geometric object. It is intended for use in conditionals or assertions to verify configurations before constructing further elements.

29.16. Function tkz.round(num, idp)

This function performs rounding of a real number to a specified number of decimal places and returns the result as a numerical value.

Syntax:

```
local r = tkz.round(3.14159, 2)          → 3.14
```

Arguments:

- **num** – A real number to round.
- **idp** – Optional number of decimal digits (default: 0).

Returns:
A rounded number, of type **number**.

Example.

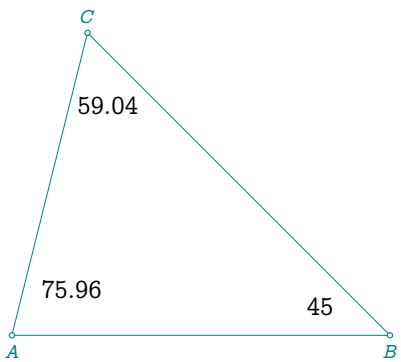
```
tkz.round(3.14159, 0) --> 3
tkz.round(3.14159, 2) --> 3.14
tkz.round(-2.71828, 3) --> -2.718
```

Related functions:
format_number(x, decimals) – Converts rounded number to string

29.17. Function tkz.angle_between_vectors(a, b, c, d)

Purpose:
The function computes the angle between the oriented vector $\vec{AB}$ (from **a** to **b**) and the oriented vector $\vec{CD}$ (from **c** to **d**).
This function is especially useful for computing turning angles or verifying geometric configurations such as orthogonality (angle $= \pm\pi/2$), alignment (angle $= 0$ or π), and orientation.

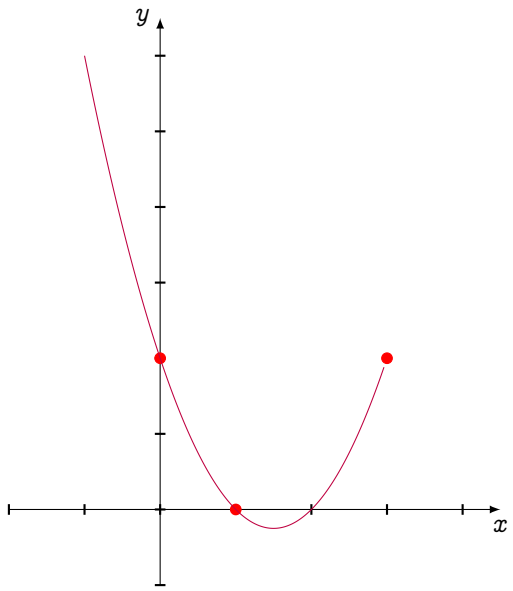
Arguments: **a**, **b**, **c**, **d** (points as complex numbers)
Returns: Angle in radians (real number)
Example:
In this example, several methods are used to determine the angles for each vertex.



```
\directlua{
z.A = point(0, 0)
z.B = point(5, 0)
z.C = point(1, 4)
T.ABC = triangle(z.A, z.B, z.C)
}
\begin{center}
  \begin{tikzpicture}
  \tkzGetNodes
  %\tkzDrawCircles()
  \tkzDrawPolygon(A,B,C)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B)
  \tkzLabelPoints[above](C)
  \tkzLabelAngle(B,A,C){$\tkzPrintNumber[2]{%
  \tkzUseLua{math.deg(T.ABC.alpha)}}$}
  \tkzLabelAngle(C,B,A){$\tkzPrintNumber[2]{%
  \tkzUseLua{math.deg(tkz.get_angle(z.B,z.C,z.A))}}$}
  \tkzLabelAngle(A,C,B){$\tkzPrintNumber[2]{%
\tkzUseLua{math.deg(tkz.angle_between_vectors(z.C,z.A,z.C,z.B))}}$}
  \end{tikzpicture}
\end{center}
```

29.18. Function tkz.parabola(pta, ptb, ptc)

Given three non-collinear points, there exists a unique parabola passing through them. The function **parabola** computes the coefficients A,B,C of the quadratic function $y = Ax^2 + Bx + C$ that interpolates these points.




```
\directlua{
 init_elements()
 z.a = point(1, 0)
 z.b = point(3, 2)
 z.c = point(0, 2)
 local A,
 B,
 C = tkz.parabola(z.a, z.b, z.c)

 function def_curve(t0, t1, n)
 local p = path()
 local dt = (t1-t0)/n
 for t = t0, t1, dt  do
     local y = A * t^2 + B * t + C
     local pt = point(t, y)
     p:add_point(pt)
  end
  return p
 end

 PA.curve = def_curve(-1,3,100)
}

\begin{tikzpicture}
   \tkzGetNodes
   \tkzInit[xmin = -2,xmax=4,ymin =-1,ymax=6]
   \tkzDrawX\tkzDrawY
   \tkzDrawPoints[red,size=4pt](a,b,c)
  \tkzDrawCoordinates[smooth,purple](PA.curve)
\end{tikzpicture}
```

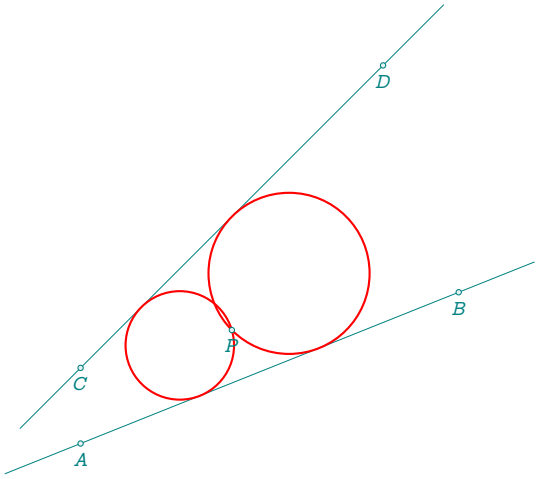
29.19. Function tkz.nodes_from_paths

Syntax: tkz.nodes_from_paths(PAcenters, PAthrough [, wbase, tbase, indice])

Purpose: This function transfers the points contained in two Lua paths — one representing the circle centers and the other the corresponding through points — into the global Lua table . Each pair of points is assigned a name based on two prefixes (for instance “w” and “t”) followed by an index number.
Arguments:

- PAcenters – a path containing the centers of the circles.
- PAthrough – a path containing the corresponding through points.
- wbase (optional) – base name for the centers (default: "w").
- tbase (optional) – base name for the through points (default: "t").
- indice (optional) – starting index (default: 1).

Returned values: None. The function creates Lua entries in the global table such as:
z["w1"]=z.w1, z["t1"]=z.t1, z["w2"], z["t2"], ...
Example: The task here is to determine the two circles passing through point P and tangent to the two lines (AB) and (CD).




```
\directlua{
  z.A = point(0, 0)
  z.B = point(5, 2)
  z.C = point(0, 1)
  z.D = point(4, 5)
  z.P = point(2, 1.5)
  L.AB = line(z.A, z.B)
  L.CD = line(z.C, z.D)
  pc, pt = L.AB:LLP(L.CD, z.P)
  tkz.nodes_from_paths(pc, pt)}
\begin{tikzpicture}
  \tkzGetNodes
  \tkzDrawLines(A,B C,D)
  \tkzDrawCircles[thick,red](w1,t1 w2,t2)
  \tkzDrawPoints(A,B,C,D,P)
  \tkzLabelPoints(A,B,C,D,P)
\end{tikzpicture}
```

Remarks:

- Both paths must have the same number of points; otherwise, the error originates from the previous computation.
- The optional indice argument is useful when several sets of contact circles are created successively, avoiding overwriting previously defined nodes.
- This function does not perform any drawing. It only defines the Lua variables for later use in TikZ constructions.

29.20. Function tkz.fsolve(f, a, b, n [, opts])

Purpose: Searches numerically for real roots of a function $f(x)$ over the interval $[a,b]$.
The interval is divided into n subintervals, and on each one a few Newton-type iterations are performed (with numerical derivative).

Syntax: roots = tkz.fsolve(f, a, b [, n, opts])

Arguments:

- **f** — function of one variable (function(x) → number);
- **a**, **b** — interval bounds (number);
- **n** — number of subintervals (default: 25);
- **opts** — optional table of parameters:

- **tol** — tolerance on f(r) (default: 1e-8);
- **step_tol** — tolerance on the Newton step (default: 1e-10);
- **h** — step size for the numerical derivative (default: 1e-6);
- **max_iter** — maximum iterations per seed (default: 8);
- **merge_eps** — distance threshold to merge nearby roots (default: 1e-5).

Returns: A Lua table (array) containing all distinct roots found in $[a,b]$, or nil if no root exists.

Notes:

- Each subinterval $[x_i, x_{i+1}]$ is explored independently.
- The derivative is approximated by the finite difference $(f(x+h)-f(x))/h$.
- Nearby roots (closer than **merge_eps**) are merged automatically.
- This method is simple and robust for basic searches but not intended as a high-precision or multi-dimensional solver.

29.21. tkz.derivative(f, x0 [, accuracy])

Purpose: Computes the numerical derivative $f'(x_0)$ using a symmetric finite difference.

Syntax: df = tkz.derivative(f, x0 [, accuracy])

Arguments:

- **f** — function of one variable (function(x) → number);
- **x0** — point where the derivative is evaluated;
- **accuracy** — optional increment controlling the precision of the finite difference (default: 1e-6).

Returns: The approximate derivative value $f'(x_0)$, or nil if inputs are invalid.

Notes:

- The derivative is approximated by the central formula:

$$f'(x_0) \approx \frac{f(x_0+accuracy)-f(x_0-accuracy)}{2accuracy}.$$

- The parameter **accuracy** controls the trade-off between precision and numerical stability.
- Too small a value of **accuracy** may amplify floating-point rounding errors.

29.22. Function tkz.range

The **tkz.range** function creates a Lua table containing a sequence of numbers between two bounds, using a specified step (similar to the Python range function).
Syntax:

```
tkz.range(a, b [, step])
```

- **a**, **b** — the lower and upper bounds.
- **step** — optional increment (default: 1). Can be negative for decreasing sequences.

Returns: a Lua table **tbl** containing the values from **a** to **b** (inclusive).

Example:

```
\directlua{
local T = tkz.range(5, 1, -1)
for i, v in ipairs(T) do
  tex.print(string.format("Value i: v))
end
}
```

Output:

```
Value 1: 5
Value 2: 4
Value 3: 3
Value 4: 2
Value 5: 1
```

30. Module utils

The **utils** module provides a collection of general-purpose utility functions used throughout the tkz-elements library. These functions are designed to support common tasks such as numerical rounding, type checking, floating-point comparisons, and table operations.

Although these functions are not directly related to geometric constructions, they play a vital role in ensuring the consistency, robustness, and readability of the core algorithms. Most of them are small, efficient, and reusable in other contexts.

This module is loaded automatically by tkz-elements, but its functions can also be used independently if needed.

30.1. Table of module functions utils

Table 43: Functions of the module **utils**.

Function	Reference
utils.parse_point(str)	[30.2]
utils.format_number(r, n)	[30.3]
utils.format_coord(x, decimals)	[30.4]
utils.format_point(z, decimals)	[30.6]
utils.checknumber(x, decimals)	[30.5]
utils.almost_equal(a, b, eps)	[30.7]
utils.wlog(...)	[30.8]

30.2. Function parse_point(str)

Parses a string of the form **"(x,y)"** and returns the corresponding numeric coordinates. This function supports optional spaces and scientific notation.
Syntax:

```
local x, y = utils.parse_point("(1.5, -2.3)")
```

Purpose:
The function takes a string argument and parses it to extract the **x** and **y** components as numbers. The input string must follow the format **"(x, y)"** where **x** and **y** can be floating-point values written in decimal or scientific notation.
Arguments: **str** – A string representing a point, e.g., **"(3.5, -2.0)"**.
Returns:
x, **y** – numeric coordinates as Lua numbers. Two numerical values: the real and imaginary parts of the point.
Features:

- Accepts optional spaces around numbers and commas.
- Accepts scientific notation (**1e-2**, **3.4E+1**).
- Raises an error for invalid formats.

Example usage:

```
local x, y = utils.parse_point("(3.5, -2)")
-- x = 3.5, y = -2.0
```

Related functions:

- **format_point(z, decimals)**
- **format_number(x, decimals)**

30.3. Function format_number(x, decimals)

This function formats a numeric value (or a numeric string) into a string representation with a fixed number of decimal places.
Syntax:

```
local str = utils.format_number(math.pi, 3)
```

Purpose:
The function converts a number (or a string that can be converted to a number) into a string with the specified number of decimal digits. It is especially useful when generating clean numerical output for display or export to Ti*k*Z coordinates.
Arguments:

- **x** – A number or a string convertible to a number.
- **decimals** – Optional. The number of decimal places (default is 5).

Returns: A string representing the value of **x** with the specified number of decimals.
Features:

- Automatically converts strings to numbers if possible.
- Ensures consistent formatting for Ti*k*Z coordinates or LaTeX output.
- Raises an error if the input is not valid.

Example usage:

```
local a = utils.format_number(math.pi, 3)
% a = "3.142"

local b = utils.format_number("2.718281828", 2)
% b = "2.72"
```

Error handling.
An error is raised if **x** is not a valid number or numeric string.
Related functions:

- **to_decimal_string(x, decimals)**
- **format_point(z, decimals)**

30.4. Function format_coord(x, decimals)

This function formats a numerical value into a string with a fixed number of decimal places. It is a lighter version of **format_number**, intended for internal use when inputs are guaranteed to be numeric.
Syntax:

```
local s = utils.format_coord(3.14159, 2)                    → "3.14"
```

Arguments:

- **x** – A number (not validated).
- **decimals** – Optional number of decimal places (default: 5).

Returns:
A string with fixed decimal formatting.

Notes:
This function is used internally by **add_pair_to_path** and other path-building methods.
Unlike **format_number**, it does not perform input validation and should only be used with known numeric inputs.

Related functions:

- **format_number(x, decimals)** – safer alternative with validation

30.5. Function checknumber(x, decimals)

Validates and converts a number or numeric string into a fixed-format decimal string.

Syntax:

```
local s = utils.checknumber("2.71828", 4)                    → "2.7183"
```

Arguments:

- **x** – A number or numeric string.
- **decimals** – Optional number of decimal digits (default: 5).

Returns:
A formatted string representing the value rounded to the specified number of decimal places.

Remarks:
Used internally to validate input before formatting. Returns an error if the input is not convertible.

Related functions:

- format_number

30.6. Function format_point(z, decimals)

Converts a complex point into a string representation suitable for coordinate output.
Syntax:

```
local s = utils.format_point(z, 4)                    → "(1.0000,2.0000)"
```

Arguments:

- **z** – A table with fields **re** and **im**.
- **decimals** – Optional precision (default: 5).

Returns:
A string representing the point as **"(x,y)"**.

Error handling.
Raises an error if **z** does not have numeric **re** and **im** components.

Related functions:

- format_coord

30.7. Function almost_equal(a, b, epsilon)

Returns **true** if two numbers are approximately equal within a given tolerance.

Syntax:

```
if utils.almost_equal(x, y) then ... end
```

Arguments:

- **a**, **b** – Two numbers to compare.
- **epsilon** – Optional tolerance (default: **tkz_epsilon**).

Returns:
A boolean: **true** if the values differ by less than the tolerance.

30.8. Function wlog(...)

Logs a formatted message to the .log file only, with a **[tkz-elements]** prefix.

Syntax:

```
utils.wlog("Internal value: %s", tostring(value))
```

Returns:
No return value. Logging only.

31. Maths tools

The **maths tools** module provides general-purpose mathematical functions for solving algebraic equations and linear systems. These tools serve as computational backbones for various geometric or algebraic operations in the tkz-elements library.
The key features include:

- Solving polynomial equations of degree 1, 2, or 3.
- Solving linear systems via augmented matrix transformation.

These functions handle both symbolic and numerical workflows and are suitable for educational, demonstrative, or computational geometry contexts.

Functions overview

Table 44: Math functions

Name	Reference
tkz.solve(...)	
tkz.solve_linear_system	

31.1. solve(...)

This general-purpose function solves polynomial equations of degree 1, 2, or 3 with real or complex coefficients. It delegates to specific solvers depending on the number of parameters.

Syntax:

```
x = solve(a, b) -- solves ax + b = 0
x1, x2 = solve(a, b, c) -- solves ax^2 + bx + c = 0
x1, x2, x3 = solve(a, b, c, d) -- solves ax^3 + bx^2 + cx + d = 0
```

Arguments:

- 2 parameters: a, b — coefficients of a linear equation $ax+b=0$.
- 3 parameters: a, b, c — coefficients of a quadratic equation.
- 4 parameters: a, b, c, d — coefficients of a cubic equation.

Return:
Depending on the degree of the equation:

- **solve(a,b)** returns one value (or an error if $a=0$).
- **solve(a,b,c)** returns two roots (real or **false** if complex and unsupported).
- **solve(a,b,c,d)** returns up to three real roots (complex roots not currently supported).

Examples:

```
x = solve(2, -4) -- x = 2
x1, x2 = solve(1, -3, 2) -- x1 = 2, x2 = 1
x1, x2, x3 = solve(1, -6, 11, -6) -- x1 = 1, x2 = 2, x3 = 3
```

Notes:

- For quadratics with complex solutions, the function currently returns **false, false**.
- Cubic solving is limited to real roots using Cardano's method and assumes $a \neq 0$.
- Internally uses the functions **tkz.solve_quadratic** and **tkz.solve_cubic**.

31.2. Function solve_linear_system(M, N)

Solves the linear system $MX = N$ using Gauss–Jordan elimination on the augmented matrix $[M \| N]$.

- `M` is the coefficient matrix $(m \times n)$.
- `N` is the right-hand side column vector $(m \times 1)$.
- Returns the unique solution vector X $(n \times 1)$, if it exists.

`Return value:`

- Returns a new **`matrix`** representing the solution.
- Returns `nil` and an error message if:
  - the system is inconsistent (no solution),
  - the system is underdetermined (infinite solutions),
  - dimensions are incompatible.

`Example usage:`

```
M = matrix({{2,1}, {4,-6}})
N = matrix({{5}, {-2}})
X = solve_linear_system(M, N)
```

Part III.

Lua and Integration

32. LuaLaTeX for Beginners: An Introduction to Lua Scripting

32.1. Introduction to Lua with LaTeX

LuaLaTeX is a variant of LaTeX that integrates Lua as a scripting language. This enables:

- Perform advanced calculations.
- Manipulate text and data.
- Generate dynamic content in a document.

32.2. Using Lua in a LaTeX document

In tkz-elements, I only use two ways of integrating code into **Lua**, which are as follows:

- Lua code block with **\directlua**. Note[14]

  Lua code can be executed directly in a document using **\directlua**:

  `\directlua{tex.print(math.exp(1))}` –> 2.718281828459

- Loading an external Lua file:

  You can write a separate Lua script and load it with: **dofile**, **loadfile** or **require**. Each of these functions has its own advantages, depending on the circumstances; initially, we'll be focusing on the first two.

  dofile: Immediately executes a Lua file. Equivalent to loadfile(filename)()

  loadfile: Loads a Lua file into memory (without executing it immediately); returns a function to execute.

  require: Loads a Lua module once (via package.loaded) and executes it; uses package.path to search for the file.

  Examples are provided in this documentation, which you can see:

  [6] `\directlua{dofile("lua/sangaku.lua")}`

☞How to comment in a file under LuaLaTeX. There are several cases to consider: in the LaTeX part outside a `directlua` directive, the % symbol is always used. Within a `\directlua` directive, % is still used. In an external file `fic.lua`, the symbol for a single-line comment in Lua is `--` (two hyphens).
With **luacode** environment, you need to use two hyphens.
The next subsection describes some of the differences between TeX and LuaTeX for special characters and catcodes.

32.3. Special Characters and Catcodes in LuaTeX

☞ Warning: Working with LuaTeX exposes deep differences between how Lua and TeX handle characters—especially special characters—and how category codes (catcodes) are assigned and interpreted.

32.3.1. What Are Catcodes in LaTeX?

In LaTeX, every character is associated with a *category code*, or *catcode*, which determines how TeX interprets it during input processing.

Character	Catcode	Meaning
\	0	Escape character (starts a control sequence)
{	1	Begin group
}	2	End group
#	6	Macro parameter
^	7	Superscript (in math mode)
%	14	Comment character

These catcodes can change dynamically within a macro or document. This flexibility is powerful, but also risky when interfacing with Lua.

14 You can use the **luacode** environment, after loading the package of the same name. The **tkzelements** environment can also be used.

32.3.2. Lua Does Not Understand Catcodes

Lua interprets strings as sequences of bytes or UTF-8 code points. It has *no concept of catcodes*. When Lua sends content to TEX using **\directlua**, the interpretation is performed by TEX according to the current catcode settings.
This discrepancy leads to common issues:

- Special characters such as #, %, or ^ are not escaped, which can break the LATEX parser.
- Control sequences may be misinterpreted if \ is improperly used or redefined.
- Group delimiters ({ and }) inserted from Lua may be incorrectly parsed, especially when injected into macros using **tex.sprint**.

32.3.3. Typical Problem Examples

- **Hash (#)**: Printing `\mycmd{#1}` from Lua may raise an error if # does not have catcode 6 or is not escaped correctly.
- **Newlines (\n)**: Lua uses \n for line breaks, which are not valid in TeX input unless passed through **\scantokens**.
- **Percent (%)**: The % (percent) symbol is special to both (La)TeX and Lua, but in very different ways. In (La)TeX, it's the default comment character. In Lua, it's one of the "magic" characters in pattern matching operations. (Lua's other "magic" characters are `^$()[]*+-?`.)

32.4. Interaction between Lua and LaTeX

LuaLaTeX allows you to interact with LaTeX, in particular via:

- `tex.print()`: displays text in LaTeX.
- `tex.sprint()`: inserts content without extra space.

```
|\directlua{tex.sprint("\\textbf{Bold text generated by Lua}")}|
```

Bold text generated by Lua

32.5. The variables

> In Lua, a variable is a named label that stores a value. Its type is determined dynamically by the value assigned to it, and its scope (global or local) influences where it can be used in your program.

Let's analyze what we've just written. Four points are important: "named label", "type of value", "dynamically" and "scope".

32.5.1. Named label

The name you assign to a variable must follow certain rules:

- They can contain letters (a-z, A-Z), digit (0-9) and the underscore symbol (_).
- They cannot start with a digit.
- They are case-sensitive.
- Some Lua keywords (such as if, then, function, local, etc.) are reserved and cannot be used as variable names.

32.5.2. Data types

What types of values can be assigned to a variable? Here's a list of the simplest:

- **nil**: undefined value.
- **boolean**: `true` or `false`.
- **number**: double-precision floating-point numbers (reals).
- **string**: strings.
- **table**: associative data structures(key-value) and arrays.
- **function**: anonymous or named functions.

32.5.3. Dynamically

The type is defined by the assigned value. So when `x = 5` then the type of **x** is **number**, but if `x = true` then its type changes to **boolean**.
☞This warning was issued earlier concerning classes and variables reserved by the tkz-elements. For example, you cannot use: **circle** with $circle = \ldots$ which would result in losing the use of the class **circle**. Similarly, $C = 1$ would result in the loss of the contents of the table **C**.

32.5.4. Scope and block

1. **Block**
   In Lua, a block is a sequence of statements that are treated as a single unit. Blocks in Lua are delimited by specific keywords. They are crucial for structuring code, defining the scope of local variables, and controlling the flow of execution.

   Here are the primary ways blocks are defined in Lua:

   - `do ... end`. This explicitly creates a block. Any local variables declared within a do...end block are only accessible within that block.
   - `if ... then ... else ... end` or `if...then...elseif...then...end`. The statements between then and end (or else and end, or the then following elseif) form a block that is executed conditionally.
   - `for...do...end` or `while...do...end` The body of the loop, between do and end, is a block that is executed repeatedly.
   - `function...end`. The body of a function, between function and end, is a block that contains the function's code.

2. **Scope**
   Lua has two main types of scope: global or local. The scope of a variable determines where in your code that variable is accessible. By default, a variable that is assigned a value without being explicitly declared (with local) is global. It is recommended and preferable to declare only local variables in a function or block. This is done by adding **local** in front of the variable name.

   A variable declared as **local** in a block is not accessible outside it.

   Example:

```
\directlua{
 do
  local x = 10
  tex.print(x) % Accessible here
end
 tex.print(x) % Error! x is out of scope here}
```

32.6. The values

Let's see what you need to know about the main values.

32.6.1. nil

- **nil** is the most basic and its own data type. **nil** is used to indicate that a variable has not yet been assigned a value, or that a value has been explicitly suppressed.

```
    \directlua{
    tex.print(tostring(no_assigned))}
    and
    tex.print(type(no_assigned)) gives


nil nil
```

- Table and **nil**.

  Assigning **nil** to a key removes that entry from the table. It's possible to delete a table using **nil**. This can free up memory if there are no other references to this table.

32.6.2. Booleans

- In Lua, there are only two boolean values: **true** and **false**.
- Booleans belong to the boolean data type.
- They are manipulated using logical operators (**and**, **or**, **not**) to form more complex boolean expressions.
- Unlike many other languages, in Lua, only the values false and nil are considered false in a boolean context (e.g., in an if condition). If you want to distinguish between nil and false you need to be more explicit

```
if v == false then
 ...
elseif v == nil then
 ...
else
 ...
end
```

32.6.3. Number

- Single Numeric Type. Lua has only one numeric type for representing real (double-precision floating-point) numbers. This means that all numbers, whether they look like integers or have decimal points, are treated the same.
- Lua uses a standard 64-bit floating-point representation.
- Even though there's only one numeric type, Lua handles integer values efficiently.
- Arithmetic Operators.Lua supports the standard arithmetic operators. It is worth noting% (modulo) The remainder of the division and // (floor division) - Divides and rounds the result down to the nearest integer. You can use functions defined in the math library.
- A number can be transformed into a string with the **tostring** function. Example: a = 10 b = "1" c = tostring(a) .. b. Finaly, `tex.print(c)` gives 101.
  Remark: in some cases, operations like **addition** or **concatenation** transform the type of one of the operands to obtain a result, but there are many special cases.

32.6.4. Strings

(a) Construction

You can define string literals in Lua using single quotes ('...'), double quotes ("..."), or double square brackets ([[...]]), but there are differences between these methods. There are no major problems if the strings don't contain any special characters.

A first small problem arises if we want to include a single quote or a double quote into the string. The simplest method is to define the string with the double square brackets. You can include a single quote inside a string defined with double quote or vice versa.

a string with "

```
\directlua{tex.print('a string with " ')}
```

And if you like a bit of a challenge, it's still possible to include a double quote inside a string defined with double quotes. This time, to use a common method with TEX, we try to write ¨, but this leads to an error. Simply inactivate ẅith \" to obtain the correct result.

a string with "

```
\directlua{tex.print("a string with \string\" ")}
```

(b) Concatenation: `...` An interesting example is the ability to create multiple variables:

```
local t= {}
local i = 5
t["x_" .. tostring(i)] = 11
tex.print(t.x_5)
```

(c) Lua provides a powerful string library (string) with many useful functions for manipulating strings. Here are some common ones:

- **`string.char(n)`**,
- **`string.len(s)`**,
- **`string.sub(s, i [, j])`**,
- **`string.find(s, pattern [, init [, plain]])`**,
- **`string.gsub(s, pattern, repl [, n])`**,
- **`string.format(formatstring, ...)`**, etc.

(d) Special or magic characters in the library **String**

Character	Meaning
.	Matches any character
%	Escape character (for magic characters)
+	One or more repetitions
-	Zero or more repetitions (minimal)
*	Zero or more repetitions (greedy)
?	Zero or one occurrence
[ ]	Character class
^	Beginning of string
$	End of string

(e) Character Classes (after %)

Code	Meaning
a	Letters (A-Z, a-z)
c	Control characters
d	Digits (0-9)
g	Printable characters (except space)
l	Lowercase letters
p	Punctuation characters
s	Space characters
u	Uppercase letters
w	Alphanumeric characters
x	Hexadecimal digits (0-9, A-F, a-f)
z	Null character (ASCII 0)

(f) Notes

- To match a literal magic character (like . or *), prefix it with %.
- Uppercase codes (%A, %D, etc.) match anything *except* the corresponding class.

(g) String.format in LuaLaTeX - Memo Sheet I've chosen to place the code **Lua** as an argument to the `directlua` directive, which means you'll have to keep an eye out for any special symbols (#, %, _ etc.) you may be tempted to use.

Lua format	Description
%d	Decimal integer (7 or 7.0)
%i	Placeholder for an integer or hexadecimal)
%f	Floating-point number (fixed precision)
%g	Floating-point number (automatic precision)
%s	Placeholder for a string
%e	Scientific notation (exponent)
%%	Percent sign (escaping)
%-5d	Left-align the integer with a field width of 5
%10.2f	Right-align a floating-point number with width 10 and 2 decimal places
%.2f	Floating-point number with 2 decimal places
%g	Floating-point number with general format

* Basic Formatting
  - `string.format(%d, 5)` outputs: `5`
  - `string.format(%f, 2.5)` outputs: `2.500000`
  - `string.format(%g, 1234567.89)` outputs: `1.23457e+06`
  - `string.format(%s, "z.A")` outputs: `z.A`

* Mathematical Expressions in LaTeX You can easily format mathematical expressions like powers and equations:
  - `string.format($ %d ^3 = %d $, x, x^3)` will output something like `$8^3 = 512$`

* Special Characters
  - `string.format(%%)` outputs: %

(h) If s = "example" then `tex.print(s:len())` gives 7.

32.6.5. The Tables

- Definition
  In Lua, the table type implements associative arrays and are the only data structure available. This means that an association is created between two entities. The first is named key and the second value. A key is associated with a unique value. A key can be an integer, a real (rare but possible), a string, a boolean, a table, a function, but not nil.

  A very frequent special case concerns tables whose keys are consecutive integers from 1. These are simply referred to as arrays, in which case the keys are indices.

– Table creation.
The simplest way to create a table is with the constructor expression {}, but the following example uses sugar syntax

```
t = {one = 1, two = 4, three = 9, four = 16, five = 25}
```
Pairs (key,value) are separated by commas or semicolons. The key is to the left of the = sign and the value to the right. `t.one = 1; t.two = 4; etc.`

In fact, this corresponds to

```
t = {["one"] = 1, ["two"] = 4, ["three"] = 9, ["four"] = 16, ["five"] = 25}
```

which is the most general definition of a table.

`t.["one"] = 1; t.["two"] = 4; etc.`

The use of sugar syntax is pleasant for the user, but it hides important points. First, the **dot notation** for string keys is valid only if the string keys follow the rules for Lua identifiers (alphanumeric and underscore, not starting with a digit).

– Table Manipulation.
There are in built functions for table manipulation and they are listed in the following table.

Table 45: Built-in Table Manipulation Functions in Lua

Function	Description
`table.insert(t, [pos,] value)`	Inserts a value into table `t` at position `pos` (default: end).
`table.remove(t, [pos])`	Removes the element at position `pos` from table `t`.
`table.sort(t [, comp])`	Sorts the elements of table `t` in-place. Optional comparator.
`table.concat(t [, sep [, i [, j]]])`	Concatenates elements of `t` from `i` to `j` using separator `sep`.
`table.unpack(t [, i [, j]])`	Returns the elements of `t` from index `i` to `j`.
`table.pack(...)`	Returns a new table with all arguments stored as elements.

– Accessing elements: You can access table elements using square brackets [] with the corresponding key. If the key is a valid string identifier, you can also use dot notation (.).

– Deleting elements: To remove an element from a table, simply assign the value nil to its key.

32.6.6. Functions

In Lua, functions possess first-class status, which means they can be assigned to variables, provided as arguments to other functions, and used as return values from functions.

(a) Basic Syntax

125.0

```
\directlua{
  function cubic(x)
     return x^3
  end
  tex.print(cubic(5))}
```

(b) Parameters and Return Values

Functions can take any number of parameters and return multiple values. Let's look at an example that illustrates this case and shows some of the difficulties associated with Lua's special characters.

This involves dividing with whole numbers. The simplest solution is to create this function in an external file: "lua/divide.lua".

```
function divide_and_remainder(dividend, divisor)
  if divisor == 0 then
    tex.error("Error: Cannot divide by zero")
    return nil
  else
    local quotient = math.floor(dividend / divisor)
    local remainder = dividend % divisor
    return quotient, remainder
  end
end
```

```
\directlua{
 dofile("lua/divide.lua")
local pc = string.char(37)
local num1 = 17
local num2 = 5
local q, r = divide_and_remainder(num1, num2)
tex.sprint(string.format("The quotient of ".. pc.. "d divided by"
.. pc.." d is:".. pc.." d\\par", num1, num2, q))
tex.sprint(string.format("The remainder is: "..pc.."d\\par", r))
}
```

The quotient of 17 divided by 5 is: 3

The remainder is: 2

Another solution is to use a local variable, for example:

```
local format = string.char(37) .. ".2f"
```

and then `tex.print(string.format(format , ...))`

Let's examine the version of the function if it is defined in the macro **\directlua**.

The problem is the use of l'operator % (Modulus Operator and remainder of after an integer division). The same problem occurs when using the string.format function and the %d or %i: placeholder for an integer.

One solution for these two cases is to define a macro **pc** which replaces the %.

```
\makeatletter
\let\pc\@percentchar
\makeatother
```

The code becomes:

```
\directlua{
 function divide_and_remainder(dividend, divisor)
   if divisor == 0 then
     tex.error("Error: Cannot divide by zero")
     return nil
   else
     local quotient = math.floor(dividend / divisor)
     local remainder = dividend \pc divisor
     return quotient, remainder
   end
 end

local num1 = 17
local num2 = 5
local q, r = divide_and_remainder(num1, num2)

tex.sprint(string.format([[The quotient of \pc i divided by
 \pc i is: \pc i\\]], num1, num2, q))
tex.sprint(string.format([[The remainder is: \pc i]], r))}
```

(c) Variadic Functions

Use ... to handle a variable number of arguments:

First example:

It's the easiest way to avoid problems linked to the #.

10

```
\directlua{
local function sum_all(...)
  local total = 0
  for _, value in ipairs{...} do
    total = total + value
  end
  return total
end

tex.print(sum_all(1, 2, 3, 4))
}
```

– Second example:

Here, I'm using a new function from the **table_getn** package, which replaces an old Lua function **table.getn**, which was used to obtain the size of the table.

100

```
\directlua{
local function sum_all(...)
  local arg = {...}
  local total = 0
  for i = 1, utils.table_getn(arg) do
    total = total + arg[i]
  end
  return total
end
local result = sum_all(10, 20, 30, 40)
tex.print(result)}
```

– Third example:

It's always possible to use #, with a little effort (texhnic).

100

```
\bgroup
\catcode`\#=12
\directlua{
local function sum_all(...)
  local arg = {...}
  local total = 0
  for i = 1, #arg do
    total = total + arg[i]
  end
  return total
end
local result = sum_all(10, 20, 30, 40)
tex.print(result)}
\egroup
```

(d) Recursion

Lua supports recursive functions:

$8! = 40320$

```
\directlua{dofile("lua/fact.lua")}
\newcommand*{\luafact}[1]{\directlua{
  tex.write(fact(\the\numexpr(#1)\relax))}%
}
$8! = \luafact{8}$
```

(e) Methods (Object-like Syntax)

When defining functions for tables (objects), you can use the colon syntax (:) to automatically pass the object itself as the first argument, usually called self.

Here's an example:

5+2.50i

```
\directlua{
 z.A = point(0, 0)
 z.B = point(5, 0)
 L.AB = line(z.A, z.B)
 T.ABC = L.AB:half()
 _,_,z.C = T.ABC:get()
 tex.print(tostring(z.C))}
```

Without the colon syntax, you would need to pass the object explicitly as the first argument:

```
T.ABC = L.AB.half(L.AB)
_,_,z.C = T.ABC.get(T.ABC)
```

half and **get** are methods defined for object tables. When called with the colon syntax, they automatically receive the instance (self) they operate on.

32.7. Control structures

Control structures let your program make decisions and repeat actions. Lua has a small set of easy-to-use control structures to help you write flexible and powerful code.

1. Conditional statements

Use if, elseif, and else when you want your code to do different things depending on conditions. This is how your program can "choose" between different options.

2. Loops

Loops are used to repeat actions:
while repeats as long as a condition is true. repeat ... until repeats until a condition becomes true (it always runs at least once). for is used to count or go through elements in a table.

3. Local variables and blocks

When you create a variable using local, it only exists inside the block where you wrote it. A block can be a function, a loop, or an if statement.

4. Breaking and returning

break lets you stop a loop early. return sends a value back from a function or stops it completely. Lua keeps things simple, so once you learn these basic structures, you can write all kinds of logic in your programs!

1. **if then else**

x =1.2246467991474e-16
$x = 0$

```
\directlua{
x = math.sin(math.pi)
 tex.print("x = ",x)
   tex.print([[\\]])
if math.abs(x) < 1e-12 then
tex.print("$x = 0$")
elseif x >0 then
 tex.print("$x > 0$")
 else
  tex.print("$x < 0$")
end
}
```

2. **while**

12345

```
\directlua{
 i = 1
 while i <= 5 do
   tex.print(i)
   i = i + 1
 end}
```

3. **repeat**

246

```
\directlua{
 x = 0
 repeat
   x = x + 2
   tex.print(x)
 until x == 6}
```

4. Numeric **for**:
   `for init,max/min value, increment do statement(s)  end`

10, 20, 30, 40, 50.

```
\directlua{
  numbers = {10, 20, 30, 40, 50}
  local nb = utils.table_getn(numbers)
  for i = 1, nb do
    local sep = (i < nb) and ", " or "."
    tex.sprint(numbers[i] .. sep)
  end}
```

5. Generic **for**:
   `for i,v in ipairs(t) do tex.print(v) end`

a1b2

```
\directlua{
   t = {a = 1, b = 2}
   for k, v in pairs(t) do
     tex.print(k, v)
   end}
```

an example for the package:

C: (1+i)
A: (i)
B: (2-i)

```
\directlua{
init_elements()
z.A = point(0, 1)
z.B = point(2, -1)
z.C = point(1, 1)
for k, v in pairs(z) do
  tex.sprint(k,": ","(".. tostring(v)..")")
  tex.print('\\\\')
end}
```

6. Summary about **for**

Table 46: Comparison of `ipairs` and `pairs` in Lua

Feature	`ipairs(t)`	`pairs(t)`
Iterates over	Numeric keys `1..n`	All keys (strings, numbers, etc.)
Order	Guaranteed (1, 2, 3, ...)	Not guaranteed
Stops at	First `nil` in sequence	After all keys
Use case	Arrays / Lists	Tables / Dictionaries
Skips non-numeric	✓	✗
Includes holes	✗	✓

7. **break** and **return**
   - break: exits a loop.

1234

```
\directlua{  for i = 1, 10 do
    if i == 5 then break end
    tex.print(i)
  end}
```

   - return: exits a function (or a chunk) with or without values.

     Here's an example with the following constraints:

(i) The function loops over a list of numbers.

(ii) If it finds an even number (value % 2 == 0), it returns it immediately.

(iii) If it sees −1, it uses break to stop the loop without returning a value right away.

(iv) If no even number is found before a −1, the function returns nil at the end.

nil

```
\makeatletter
\let\pc\@percentchar
\makeatother

\directlua{%
  function find_first_even(t)
    for i = 1, utils.table_getn(t) do
      local value = t[i]
      if value \pc 2 == 0 then
        return value
      elseif value == -1 then
        break
      end
    end
    return nil
  end

  local numbers = {1, 3, 5, -1, 4, 6}
  tex.print(tostring(find_first_even(numbers))) }
```

32.8. Lua Sugar Syntax

The "sugar syntax" in Lua refers to syntactic elements that make the language more concise and readable changing its fundamental meaning. Here are the main forms of sugar syntax in Lua:

1. Method calls with ":"
   obj:method(arg) Is sugar for: **obj.method(obj, arg)**

   The ":" implicitly passes obj as the first argument (self), which is very helpful in object-oriented programming.

2. Table field access with "."
   t.key Is sugar for: **t["key"]**

   This is more readable when the key is a valid identifier (no spaces, punctuation, etc.).

3. Shorthand table constructors
   local t = {1, 2, 3} Is equivalent to: **local t = {[1]=1, [2]=2, [3]=3}**

   And:

   local t = {a = 1, b = 2} Is sugar for: **local t = {["a"] = 1, ["b"] = 2}**

4. Loop sugar (e.g., for k, v in pairs(t) do)
   Even though pairs() is just a function, the way Lua loops over tables using for is very clean and readable. k for keys , v for value.

   ```
   for k, v in pairs(t) do
      ...
   end
   ```

5. Logical expressions as default value tricks
   x = a or b Is often used to mean: "if a is truthy, use it, otherwise use b". Great for setting default values.

6. do ... end blocks
   This is a way to create a local scope block — useful for limiting variable visibility.

32.9. Example: Calculating the sum of 1 to 10

Lua is useful for advanced calculations that LaTeX alone cannot handle efficiently.

Sum of 1 to 10: 55

```
\directlua{
  local somme = 0
  for i = 1, 10 do
    somme = somme + i
  end
  tex.print("Sum of 1 to 10: " .. somme)}
```

32.9.1. Example: Fibonacci

Calculation of the rank term 10

The term of rank 10 in the Fibonacci sequence is $u_{10} = 55$

```
\directlua{
 function fibonacci(n)
   if n == 0 then
     return 0
   elseif n == 1 then
     return 1
   else
     return fibonacci(n - 1) + fibonacci(n - 2)
   end
 end}
    The term of rank $10$ in the Fibonacci sequence
    is
    \(u_{10}=\directlua{tex.print(
    fibonacci(10))}\)
```

33. Transfers

The tkz-elements package strictly separates *mathematical computation* from *graphical rendering*. All geometric objects are created and manipulated internally in Lua, while drawing is performed by TikZ or tkz-euclide.
A *transfer* is the explicit operation that communicates data between the Lua computational layer and the TEX rendering layer.

Conceptually, the architecture may be summarized as:

$$\text{Computation (Lua)} \longrightarrow \text{Transfer} \longrightarrow \text{Rendering (TikZ/TEX)}$$

Objects remain invisible to TEX until they are explicitly transferred. This design guarantees numerical robustness and a clear separation between computation and presentation.

33.1. Conceptual Overview

Three types of transfer mechanisms are available in tkz-elements:

1. Immediate transfer via the TEX input stream;
2. Object-based transfer through dedicated macros;
3. File-based transfer via external data files.

Each mechanism serves a different purpose depending on the size and nature of the data being communicated.

33.2. Object Serialization via tostring

The macro **\tkzUseLua** is the basic bridge from Lua to TEX. It evaluates a Lua expression and prints its result into the TEX input stream. Internally, it uses Lua's `tostring()` mechanism:

```
\def\tkzUseLua#1{\directlua{tex.sprint(#1)}}
```

As a consequence, any tkz-elements object may define a Lua `__tostring` metamethod to control its textual TEX representation. This is used in particular for complex numbers (**point**).

Example with a point:

1-2.00i

```
\directlua{
  init_elements()
  z.A = point(1, -2)
}
\tkzUseLua{tostring(z.A)}
```

Thus, the *complex affix* associated with point *A* is printed using Lua's `__tostring` metamethod (via `tostring`).

Example with a matrix:
The macro **\tkzUseLua** prints the *value* of a Lua expression as text (internally via `tostring(<expr>)`).[15]

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
\directlua{
  init_elements()
  local a, b, c, d = 1, 2, 3, 4
  M.new = matrix({ { a, b }, { c, d } })}
  \tkzUseLua{M.new:print()}
```

Other cases To transfer **nil**, **true**, and **false**, you must use **tostring**, However, you can directly transfer a number or a string of characters.

```
\tkzUseLua{tostring(nil)}
\tkzUseLua{tostring(true)}
\tkzUseLua{"TANGENT"}
\tkzUseLua{math.pi}
```

15 Hence objects may customize their inline representation by defining a Lua `__tostring` metamethod.

33.3. Immediate Transfer

Immediate transfer occurs when Lua prints data directly into the TEX input stream using `tex.print`.

Numerical example

5.0

```
\directlua{
  init_elements()
  z.A = point(3,4)
  tex.print(point.abs(z.A))
}
```

The computed value is inserted directly into the document.

Boolean example

false

```
\directlua{
  init_elements()
  z.A = point(0, 0)
  z.B = point(3, 0)
  z.C = point(0, 2)
  T.ABC = triangle(z.A, z.B, z.C)
  tex.print(tostring(T.ABC:check_equilateral()))
}
```

This mechanism allows Lua computations to influence TEX logic through conditional statements.

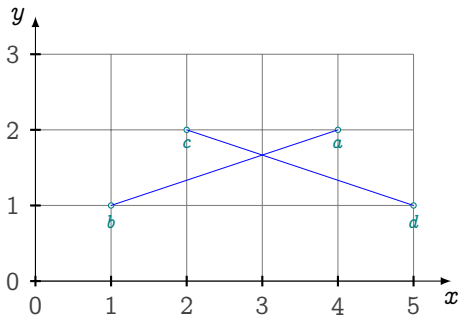
Note: It is possible to choose when to perform the transfer. A macro has been created for this purpose, but you can also create your own: it is called **\tkzUseLua**. It is defined as follows:

```
\def\tkzUseLua#1{\directlua{tex.print(#1)}}
```

This macro prints the value of a Lua variable or expression directly into the TeX stream.

Example.
The following Lua code computes whether two lines intersect:
The intersection of the two lines lies at a point whose affix is:

```
\directlua{
  z.a = point(4, 2)
  z.b = point(1, 1)
  z.c = point(2, 2)
  z.d = point(5, 1)
  L.ab = line(z.a, z.b)
  L.cd = line(z.c, z.d)
  det = (z.b - z.a) ^ (z.d - z.c)
  if det == 0 then bool = true
    else bool = false
  end
  x = intersection (L.ab, L.cd)}

The intersection of the two lines lies at
    a point whose affix is:\tkzUseLua{tostring(x)}
```



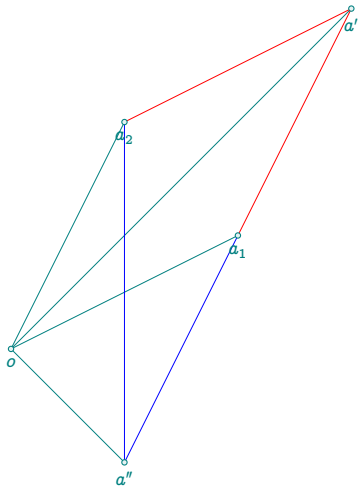

```
\begin{tikzpicture}
  \tkzGetNodes
  \tkzInit[xmin =0,ymin=0,xmax=5,ymax=3]
  \tkzGrid\tkzAxeX\tkzAxeY
  \tkzDrawPoints(a,...,d)
  \ifthenelse{\equal{\tkzUseLua{bool}}{%
  true}}{\tkzDrawSegments[red](a,b c,d)}{%
  \tkzDrawSegments[blue](a,b c,d)}
  \tkzLabelPoints(a,...,d)
\end{tikzpicture}
```

33.4. Object-Based Transfer

Geometric objects stored in Lua tables must be converted into TikZ-compatible structures before drawing.

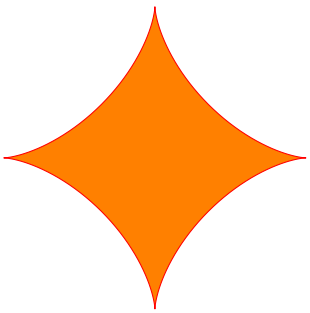
33.4.1. Transferring points

Points are internally stored as complex affixes. The macro **\tkzGetNodes** converts them into TikZ coordinates.




```
\directlua{
   init_elements()
   z.o   = point(0, 0)
   z.a_1 = point(2, 1)
   z.a_2 = point(1, 2)
   z.ap  = z.a_1 + z.a_2
   z.app = z.a_1 - z.a_2
}

\begin{center}
  \begin{tikzpicture}[ scale = 1.5]
     \tkzGetNodes
     \tkzDrawSegments(o,a_1 o,a_2 o,a' o,a'')
     \tkzDrawSegments[red](a_1,a' a_2,a')
     \tkzDrawSegments[blue](a_1,a'' a_2,a'')
     \tkzDrawPoints(a_1,a_2,a',o,a'')
     \tkzLabelPoints(o,a_1,a_2,a',a'')
  \end{tikzpicture}
\end{center}
```

33.4.2. Transferring paths and curves

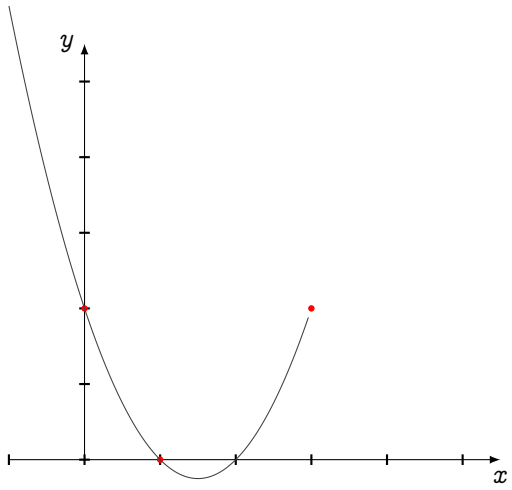


```
\directlua{
 init_elements()
 PF.ex = pfct("(cos(t))^3", "(sin(t))^3")
 PA.curve = PF.ex:path(0, 2*math.pi, 100)
 }

\begin{center}
 \begin{tikzpicture}[scale=2]
  \tkzDrawCoordinates[smooth,red,fill=orange](PA.curve)
\end{tikzpicture}
 \end{center}
```

The Lua object is serialized into a coordinate list interpreted by TikZ.

33.4.3. File-Based Transfer

For large datasets or high-resolution curves, it may be preferable to generate an external file.
This time, the transfer will be carried out using an external file. The following example is based on this one, but using a table.
`Generating a data file:` The file `tmp.table` is created using the f function.
The file `tmp.table` now contains numerical data. `Using the generated file:` Here, the curve is plotted using TikZ.




```
\directlua{
 init_elements()
  z.a = point(1, 0)
  z.b = point(3, 2)
  z.c = point(0, 2)
  A,B,C = tkz.parabola (z.a, z.b, z.c)

function f(t0, t1, n)
 local out=assert(io.open("tmp.table","w"))
 local y
 for t = t0,t1,(t1-t0)/n  do
  y = A*t^2+B*t +C
  out:write(utils.checknumber(t), " ",
    utils.checknumber(y), " i\string\n")
 end
 out:close()
end}

\begin{tikzpicture}
   \tkzGetNodes
   \tkzInit[xmin=-1,xmax=5,ymin=0,ymax=5]
   \tkzDrawX\tkzDrawY
   \tkzDrawPoints[red,size=2](a,b,c)
   \directlua{f(-1,3,100)}%
   \draw[domain=-1:3] plot[smooth]
        file {tmp.table};
\end{tikzpicture}
```

File-based transfer provides scalability and allows interoperability with external plotting tools such as `pgfplots`.

33.5. Dynamic TEX--Lua Interaction

Transfers are bidirectional. TEX may pass parameters to Lua for computation.

```
\newcommand{\ComputeSquare}[1]{%
  \directlua{tex.print(#1 * #1)}%
}
```

Usage:

```
The square of 5 is \ComputeSquare{5}.
```

The square of 5 is 25.

This interaction enables dynamic parameterized geometry, where document input directly controls Lua computations.

Transfers in tkz-elements may therefore be immediate, object-based, or file-based, depending on the size and structure of the data. This layered architecture ensures a clean separation between computation and graphical representation.

34. TeX Interface Macros (tkz-elements.sty)

34.1. Purpose

The file tkz-elements.sty provides a small set of macros that act as a bridge between TEX and the Lua engine used by tkz-elements. Their main goals are:

- printing Lua values safely in TEX;
- transferring geometric objects (points, paths, circles) from Lua to TikZ;
- simplifying the drawing stage when computations are done in Lua.

Unless stated otherwise, these macros assume that tkz-elements has been initialized in Lua (typically via \directlua{init_elements()}.

34.2. Summary of provided macros

Macro	Reference
\tkzUseLua{...}	
\tkzPrintNumber{...}	
\tkzEraseLuaObj{name}	
\tkzDrawCoordinates(…)	
\tkzGetPointsFromPath{P}{A}	
\tkzDrawPointsFromPath(…)	
\tkzDrawSegmentsFromPaths(…)	
\tkzDrawCirclesFromPaths(…)	

Note: The exact internal Lua object layout may evolve; these macros are designed to remain stable at the user level.

34.3. Macro \tkzUseLua

Syntax:
tkzUseLua{<lua-expression>\}

Description: Evaluates the Lua expression and prints its result in the TEX stream.

Argument:

- **<lua-expression>**: a Lua expression returning a value.

Notes:

- This macro is meant for *values* (numbers, booleans, strings).
- For Lua objects (point, line, circle, path, ...), prefer dedicated transfer/draw macros or call an explicit Lua method that returns a value.

```
Example:
          \directlua{init_elements()
            z.A = point(0,0)
            z.B = point(4,2)
          }
          The slope is \tkzUseLua{(z.B.im-z.A.im)/(z.B.re-
          z.A.re)}.
```

This macro evaluates a Lua expression and inserts its result in the document.
Syntax: \tkzUseLua{<lua-code>}
Example usage:

```
\tkzUseLua{checknumber(math.pi)}
```

34.4. Macro \tkzPrintNumber

This macro formats and prints a number using PGF's fixed-point output, with a default precision of 2 decimal places.

Syntax:

```
\tkzPrintNumber{<lua-expression>}     \tkzPN{<lua-expression>}
```

Example:

```
\tkzPrintNumber{pi} % outputs 3.14
\tkzPrintNumber[4]{sqrt(2)} % outputs 1.4142
```

Alias: The macro **\tkzPN** is a shorthand for **\tkzPrintNumber**.

```
Example:
          \directlua{init_elements()
            x = math.pi/7
          }
          $\alpha = \tkzPN{x}$.
```

34.5. Macro \tkzEraseLuaObj

Syntax:

```
\tkzEraseLuaObj{<name>}
```

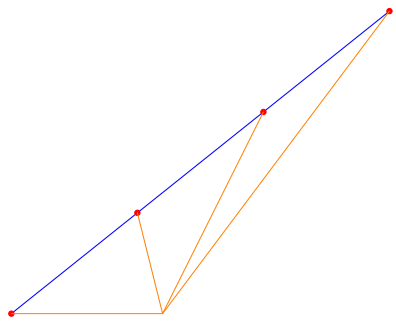
Description: Removes a stored Lua object from the global storage table (typically **z**, **L**, **C**, **CO**, etc., depending on your conventions).
Syntax: \tkzEraseLuaObj{<object>}
Example usage:

```
\tkzEraseLuaObj{z.A}
\tkzEraseLuaObj{T.ABC}
```

Notes: Use this macro to avoid name collisions when compiling large documents or when reusing the same object names across figures.

34.6. Macro \tkzPathCount

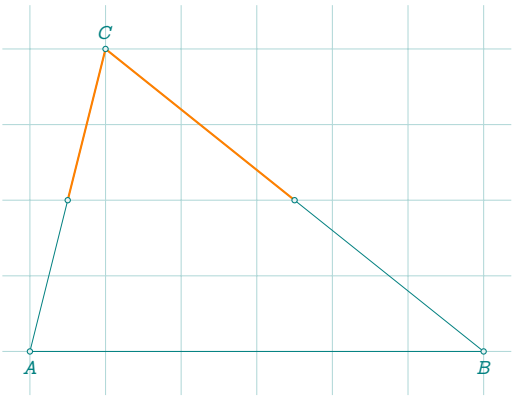
This macro retrieves the number of vertices in a Lua path and stores the result in a TeX macro.
Syntax: \tkzPathCount(<lua-path>){<MacroName>}
Arguments:
<lua-path>: identifier of a Lua path object. <MacroName>: name of a TeX macro (without the backslash) that will store the count.
Example usage:



```
\directlua{
 z.A = point(0, 0)
 z.B = point(5, 4)
 L.AB = line(z.A, z.B)
 PA.AB = L.AB:path(3)
 z.O = point(2, 0)
 }
\tkzPathCount(PA.AB){N}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawCoordinates[blue](PA.AB)
\tkzDrawPointsFromPath[red,size=2](PA.AB)
  \foreach \i in {1,...,\N}{
 \tkzGetPointFromPath(PA.AB,\i){P\i}
 \tkzDrawSegment[orange](O,P\i)
  }
\end{tikzpicture}
\end{center}
```

34.7. Macro \tkzDrawCoordinates

This macro draws a curve using the TikZ **plot coordinates** syntax, with coordinates generated by Lua.
Syntax: \tkzDrawCoordinates[<options>](lua-path)
Example usage:




```
\directlua{
  z.A = point(0, 0)
  z.B = point(6, 0)
  z.C = point(1, 4)
  T.ABC = triangle(z.A, z.B, z.C)
  z.M = T.ABC.bc.mid
  z.N = T.ABC.ca.mid
  PA.path = T.ABC:path(z.M,z.N,4)
}
\begin{tikzpicture}[gridded]
\tkzGetNodes
\tkzDrawPolygon(A,B,C)
\tkzDrawCoordinates[orange,thick](PA.path)
\tkzDrawPoints(A,B,C,M,N)
\tkzLabelPoints(A,B)
\tkzLabelPoints[above](C)
\end{tikzpicture}
```

Description: Draws a Lua **path** object as a TikZ path. This macro is typically used after the computational phase in Lua. Draws the polyline defined by the sequence of points in a Lua path, using explicit **(x,y)** coordinates.

Alias:

\tkzDrawPath[<tikz-options>](lua-path)

34.8. Macro \tkzDrawPointsFromPath

This macro draws all points of a path using **tkzDrawPoint**, without exporting point names.

Syntax: \tkzDrawPointsFromPath[<options>](lua-path)

Example usage:

```
\directlua{
init_elements()
LP.test = list_point()
LP.test:add(point(0, 0))
LP.test:add(point(1, 0))
LP.test:add(point(1, 1))
PA.curve = LP.test:as_path()
}
\begin{tikzpicture}
\tkzDrawCoordinates[blue](PA.curve)
\tkzDrawPointsFromPath[red,size=2](PA.curve)
\end{tikzpicture}
```

34.9. Macro \tkzGetPointsFromPath

This macro defines TikZ points from a Lua path. Each point is named with a base name followed by an index.
Syntax: \tkzGetPointsFromPath[<options>](<path>,<basename>)

Description: Exports the points of a Lua path as named TEX/TikZ points. The naming scheme is **<prefix>1**, **<prefix>2**, ...

Example:

```
 \directlua{
  init_elements()
  PA.g = path({ "(0, 0)", "(1, 0)","(1, 1)" })
}
\begin{tikzpicture}
  \tkzGetPointsFromPath(PA.g,A)
  \tkzDrawPolygon(A_1,A_2,A_3)
  \tkzDrawPoints(A_1,A_2,A_3)
\end{tikzpicture}
```

34.10. Macro \tkzGetPointFromPath

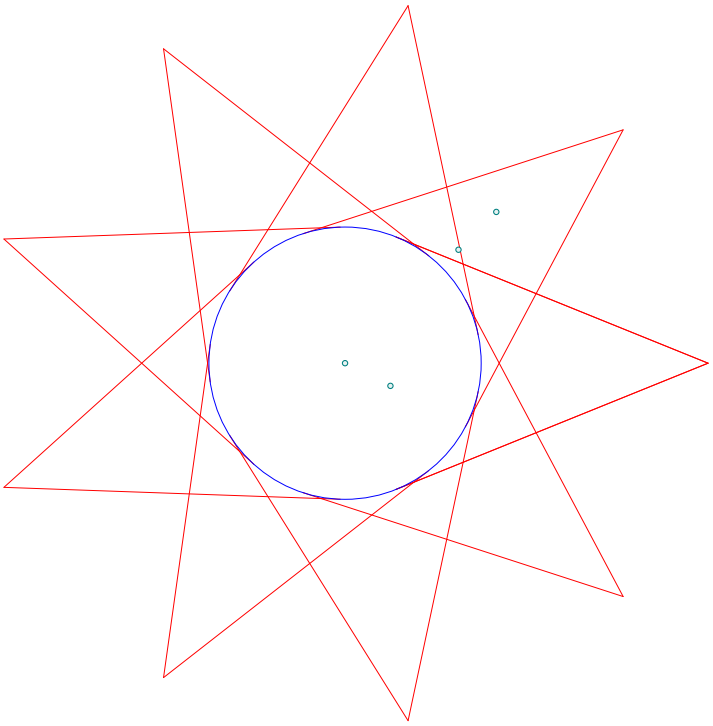
This macro extracts the i-th vertex of a Lua path and defines it as a TikZ point with a given name.
Syntax: \tkzGetPointFromPath(<PathLua>,<index>){<PointName>}
Arguments:
<PathLua>: identifier of a Lua path object. <index>: index of the vertex to retrieve (TeX number or numeric macro). <PointName>: name of the TikZ point to create.

Example usage:

```
% Get the 3rd vertex of PA.A and name it P3
\tkzGetPointFromPath(PA.A,3){P3}
% With counting and a loop
\tkzPathCount(PA.A){N}
\foreach \i in {1,...,\N}{
\expandafter\tkzGetPointFromPath\expandafter(PA.A,\i){P\i}
}
```

34.11. Macro \tkzDrawSegmentsFromPaths

Description: Draws segments encoded in one or more Lua paths (pairs of points or consecutive points, depending on the chosen convention).
Syntax: \tkzDrawSegmentsFromPaths[<options>](<lua-path-start>,<lua-path-end>)
Example:



```
\directlua{
 init_elements()
 z.A = point(0, 0)
 z.B = point(3, 0)
 C.AB = circle(z.A, z.B)
 PA.c = path()
 PA.u = path()
 PA.v = path()
 for i = 0, 18, 2 do
   local angle = i * 2 * math.pi / 18
   local p = point(polar(8, angle))
   local Lu, Lv = C.AB:tangent_from(p)
   local u , v  = Lu.pb, Lv.pb
   PA.c:add_point(p)
   PA.u:add_point(u)
   PA.v:add_point(v)
 end}

\begin{center}
 \begin{tikzpicture}[scale =.6]
  \tkzGetNodes
  \tkzDrawSegmentsFromPaths[draw,red](PA.c,PA.u)
  \tkzDrawSegmentsFromPaths[draw,red](PA.c,PA.v)
  \tkzDrawCircle[blue](A,B)
  \tkzDrawPoints(A,I,u,v)
\end{tikzpicture}
 \end{center}
```

34.12. Macro \tkzDrawCirclesFromPaths

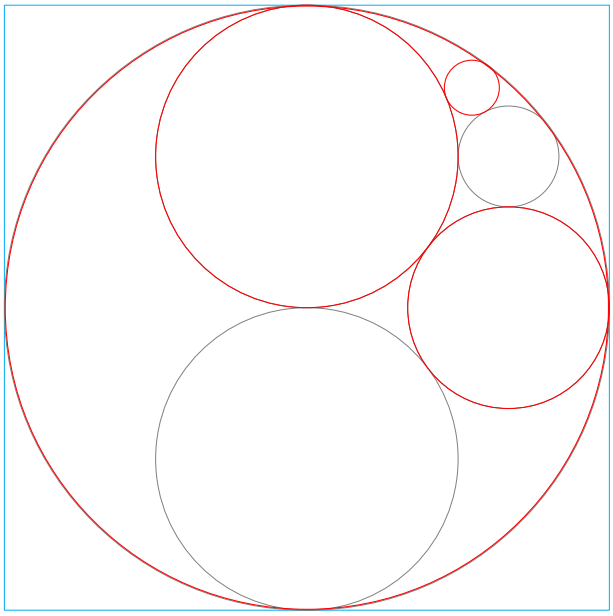
This macro draws circles defined by two paths: one for the centers and one for the points through which each circle passes.

Syntax: \tkzDrawCirclesFromPaths[<options>](<PA.center>,<PA.through>)

Example usage:

```
\tkzDrawCirclesFromPaths[blue](PA.O, PA.A)
```

See the document **Euclidean Geometry** presented in altermundus.frfor other examples.



```
\directlua{
 init_elements()
 z.A = point(-4, -4)
 z.B = point(4, -4)
 L.AB = line(z.A, z.B)
 C.AB = circle(z.A, z.B)
 S.AB = L.AB:square()
 _, _, z.C, z.D = S.AB:get()
 z.O = S.AB.ac.mid
 z.a = S.AB.ab.mid
 z.c = S.AB.cd.mid
 z.o1 = tkz.midpoint(z.a, z.O)
 z.o = tkz.midpoint(z.c, z.O)
 z.b = S.AB.bc.mid
 z.g = (z.o1 - z.O) + (z.b - z.O)
 z.o2 = intersection(line(z.c, z.g), line(z.O,z.b))
 z.o3 = (z.o - z.O) + (z.o2 - z.O)
 z.j = intersection(line(z.c, z.b), line(z.o2,z.o3))
 local C2 = circle(z.O, z.a)
 local C1 = circle(z.o, z.c)
 local C3 = circle(z.o3, z.j)
 PA.pc, PA.pt, n = C1:CCC(C2, C3)
}

\begin{center}
\begin{tikzpicture}[scale=1]
\tkzGetNodes
\tkzDrawPolygon[cyan](A,B,C,D)
\tkzDrawCircles(O,a o,c o1,a o2,b o3,j)
\tkzDrawCirclesFromPaths[draw,red](PA.pc,PA.pt)
\end{tikzpicture}
\end{center}
```

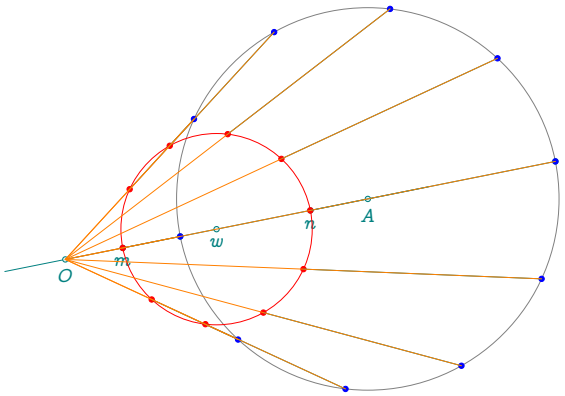
34.13. Macro \tkzDrawFromPointToPath

This macro draws a set of line segments from a given TikZ point to every point of a Lua *path* object.

Syntax: \tkzDrawFromPointToPath[<options>](<point-tikz>,<lua-path>)

Arguments:
<options> (optional): pgf/tikz drawing options (color, thickness, etc.).
<point-tikz>: the name of a TikZ point (already defined).
<lua-path>: identifier of a Lua path object.

Example usage:




```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = point(5, 1)
 z.Ta = point(8, 2)
 C.A = circle(z.A, z.Ta)
 L.OA = line(z.O,z.A)
 z.x, z.y = intersection(C.A, L.OA,{near=z.O})
 C.A.through = z.y
 z.m = tkz.midpoint(z.O, z.x)
 z.n = tkz.midpoint(z.O, z.y)
 z.w = tkz.midpoint(z.m, z.n)
 PA.A = path()
 PA.c = path()
 for t = 0, 1 - 1e-12, 0.1 do
   PA.A:add_point(C.A:point(t))
   local m = tkz.midpoint(z.O, C.A:point(t))
   PA.c:add_point(m)
 end
}
 \begin{center}
  \begin{tikzpicture}[scale=.8]
   \tkzGetNodes
   \tkzDrawCircle(A,Ta)
   \tkzDrawLine(O,A)
   \tkzDrawPoints(O,A,m,n,w,x,y)
   \tkzLabelPoints(O,A,m,n,w)
   \tkzDrawCircle[red](w,n)
   \tkzDrawPointsFromPath[red](PA.c)
   \tkzDrawPointsFromPath[blue](PA.A)
   \tkzDrawSegmentsFromPaths[draw,teal](PA.c,PA.A)
   \tkzDrawFromPointToPath[orange](O,PA.A)
 \end{tikzpicture}
  \end{center}
```

34.14. Macros \tkzDrawPointOnGraph and \tkzDrawPointsOnGraph

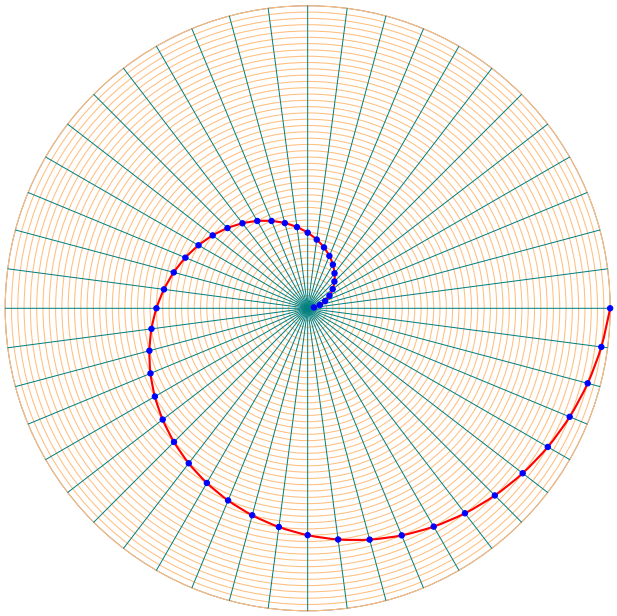
These macros are described in the section [35]

34.15. Macros \tkzDrawPointOnParamGraph and \tkzDrawPointsOnParamGraph

These macros are described in the section [36]

34.16. Archimedes spiral, a complete example

This example is built with the help of some of the macros presented in this section. It is inspired by an example created with tkz-euclide by Jean-Marc Desbonnez.



```
\directlua{
  z.O = point(0, 0)
  z.A = point(5, 0)
  L.OA = line(z.O, z.A)
  PA.spiral = path()
  PA.center = path()
  PA.through = path()
  PA.ray = path()
  for i = 1, 48 do
      local k = i / 48
      z["P"..i] = L.OA:point(k)
      local angle = i * tkz.tau / 48
      z["r"..i] = point(polar(5, angle))
      ray = line(z.O, z["r"..i])
      circ = circle(z.O, z["P"..i])
      z["X"..i], _ = intersection(ray, circ, { near = z["r"..i] })
      PA.spiral:add_point(z["X"..i])
      PA.center:add_point(z.O)
      PA.through:add_point(z["P"..i])
      PA.ray:add_point(z["r"..i])
  end}
\begin{center}
  \begin{tikzpicture}[scale = .8]
  \tkzGetNodes
  \tkzDrawCircle(O,A)
  \tkzDrawCirclesFromPaths[draw,
      orange!50](PA.center,PA.through)
  \tkzDrawSegmentsFromPaths[draw,teal](PA.center,PA.ray)
  \tkzDrawPath[red,thick,smooth](PA.spiral)
  \tkzDrawPointsFromPath[blue, size=2](PA.spiral)
  \end{tikzpicture}
\end{center}
```

35. Class fct

The **fct** class represents a real-valued function

$$x \longmapsto f(x),$$

defined and evaluated in Lua. A **fct** object encapsulates either (i) a numerical expression (given as a string in the variable x), or (ii) a callable Lua function, together with methods for evaluation, sampling, and geometric construction.
By convention, function objects are stored in a Lua table named **F**. Although this name is not mandatory, using F is strongly recommended for consistency across examples and documentation. Each entry of F associates a symbolic name (such as fa) with an object of class **fct**. If a custom table name is used, it must be initialized manually. The **init_elements()** function will reset the F table if it has already been defined.

Mathematical expressions. All expressions defining a **fct** object are evaluated using standard Lua rules. A **fct** object can be evaluated at a real number, sampled on an interval, converted into a **path** (for drawing with TikZ), or exported to a data file.

Important. In function and parametric function definitions (**fct** and **pfct**), expressions must be written using standard mathematical notation. The Lua prefix math. must *not* be used.
In contrast, in all other methods and Lua code fragments of tkz-elements, standard Lua syntax applies, and the use of the prefix math. is required whenever a Lua mathematical function is called. For convenience, users may define local aliases in their Lua code, such as **local sin, cos, pi = math.sin, math.cos, math.pi**.

Correct:

```
sin(x), exp(-x^2)
```

Incorrect (in **fct** and **pfct**):

```
math.sin(x), math.exp(-x^2)
```

35.1. Methods of the class fct

The **fct** class offers a compact set of methods, centered around: *(i)* creating a function object, *(ii)* evaluating it, and *(iii)* producing a **path** or a data file.

Table 47: fct methods.

Methods	Reference
Creation	
new(expr_or_fn)	[35.1.1]
compile(expr)	[35.1.2]
Reals / evaluation	
eval(x)	[35.1.3]
Points / Paths	
point(x)	[35.1.4]
path(xmin,xmax,n)	[35.1.5]

35.1.1. Constructor new(expr_or_fn)

Description: Creates a function object from either a Lua expression (string) or a callable Lua function. When a string is provided, it represents an expression in the variable x, evaluated using standard Lua rules (therefore requiring math. prefixes).
Arguments:

- **expr_or_fn**: either
  - a string representing an expression in x, or

– a callable Lua function f(x).

Returns: a **fct** object.
Example:

```
\directlua{
  init_elements()
  F.f = fct:new("x*math.exp(-x^2)+1")
  tex.print(F.f:value(0))
}
```

35.1.2. Function compile(expr)

Description: Compiles an expression and returns a callable Lua function (or wraps it into a **fct**, depending on the implementation). This is a low-level helper mainly intended for internal use.
Argument: **expr** (string, expression in variable x). Returns: a callable function or a **fct** object (depending on implementation).

35.1.3. Method eval(x)

Description: Evaluates the function at the real number **x**.
Returns: a number (which may be nan or infinite if the expression is not defined).

```
  init_elements()
  F.fa = fct("sin(x) + x")
  PA.curve = F.fa:path(-2, 5, 200)
  z.A = F.fa:point(1.3)
  tex.print(F.fa:eval(math.pi/2))
```

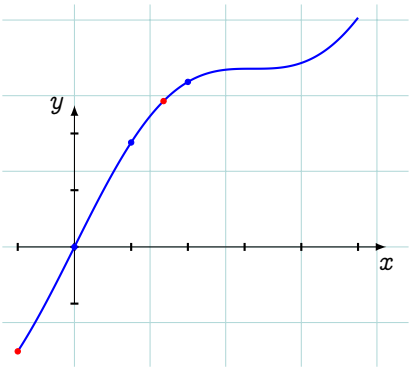
35.1.4. Method point(x)

Description: Constructs the point $(x, f(x))$ associated with the function. This method provides a direct bridge between numerical evaluation and geometric construction.
Returns: a **point**.
Example:

```
  init_elements()
  F.fa = fct("sin(x)+x")
  PA.curve = F.fa:path(-2, 5, 200)
  z.A = F.fa:point(1.5)
  % or z.A = point(1.3, F.fa:eval(1.3) )
```

35.1.5. Method path(xmin,xmax,n)

Description: Samples the function on $[xmin, xmax]$ using **n** subdivisions and returns a **path**. Invalid values are skipped (depending on the internal policy).
Returns: a **path**.
Example:




```
\directlua{
  init_elements()
  F.fa = fct("sin(x) + x")
  PA.curve = F.fa:path(-1, 5, 200)
  z.A = F.fa:point(math.pi/2)
}
\begin{tikzpicture}[scale=.75,gridded]
  \tkzInit[xmin=-1,xmax=5,ymin=-1,ymax=2]
  \tkzDrawX\tkzDrawY
  \tkzGetNodes
  \tkzDrawCoordinates[smooth,blue,thick](PA.curve)
  \tkzDrawPoint[red](A)
  \tkzDrawPointsOnGraph[blue]{0,1,2}{fa}
  \tkzDrawPointOnGraph[red]{-1}{fa}
\end{tikzpicture}
```

35.2. Macros \tkzDrawPointOnGraph and \tkzDrawPointsOnGraph

These macros allow drawing points on the graph of a function defined in the module system.
The function must already exist in the module table F.

35.2.1. Macro \tkzDrawPointOnGraph

Syntax

```
\tkzDrawPointOnGraph[<TikZ options>]{<x>}{<name>}
```

Description This macro draws a point belonging to the graph of a real function stored in the module F.
The drawn point has coordinates

$$(x, f(x)),$$

where f denotes the function object F.<name>.

Arguments

- <TikZ options> (optional): graphical options applied to the point
- <x>: abscissa of the point
- <name>: name of the function stored in F

Example

```
\tkzDrawPointOnGraph[blue]{4}{fa}
```

35.2.2. Macro \tkzDrawPointsOnGraph

Syntax

```
\tkzDrawPointsOnGraph[<TikZ options>]{<x1,x2,...,xn>}{<name>}
```

Description This macro draws several points belonging to the graph of a real function stored in the module fct.
For each value x_i in the list, a point of coordinates

$$(x_i, f(x_i))$$

is computed and drawn.

Arguments

- <TikZ options> (optional): graphical options applied to all points
- <x1,x2,...,xn>: comma-separated list of abscissas
- <name>: name of the function stored in F

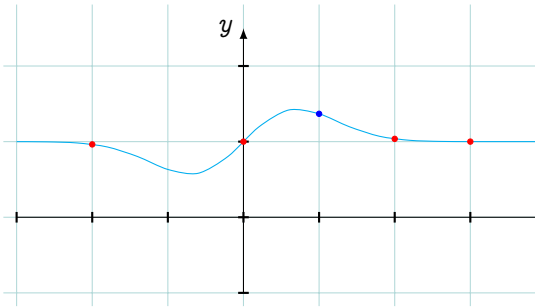
Example

```
\tkzDrawPointsOnGraph[red]{-2,0,2,3,5}{fa}
```

Remarks

- These macros assume that the function is already defined in the module F.
- All evaluations are performed in Lua.
- The macros are compatible with paths drawn using \tkzDrawCoordinates.

Example:




```
\directlua{
  init_elements()
  F.fa = fct("x*exp(-x^2)+1")
  PA.curve = F.fa:path(-3, 5, 20)
}
\begin{tikzpicture}[gridded]
  \tkzInit[xmin=-3,xmax=5,ymin=-1,ymax=2]
  \tkzDrawX\tkzDrawY
  \tkzDrawCoordinates[smooth,cyan](PA.curve)
  \tkzDrawPointsOnGraph[red]{-2,0,2,3,5}{fa}
  \tkzDrawPointOnGraph[blue]{1}{fa}
\end{tikzpicture}
```

36. Class pfct

The **pfct** class represents a parametric curve

$$t \longmapsto \big(x(t), y(t)\big),$$

defined and evaluated in Lua. A **pfct** object encapsulates two expressions (or callable Lua functions) describing the x- and y-components of the curve, together with methods for evaluation, sampling, and geometric construction.
By convention, parametric function objects are stored in a Lua table named **PF**. Although this name is not mandatory, using PF is strongly recommended for consistency across examples and documentation. Each entry of PF associates a symbolic name (such as `lis`) with an object of class **pfct**. If a custom table name is used, it must be initialized manually. The **init_elements()** function will reset the PF table if it has already been defined.

Mathematical expressions. All expressions defining a **pfct** object are evaluated using standard Lua rules. A **pfct** object can be evaluated at a parameter value, converted into points, sampled into a **path**, or exported to a data file suitable for TikZ `plot file` input.

Important. In function and parametric function definitions (**fct** and **pfct**), expressions must be written using standard mathematical notation. The Lua prefix `math.` must *not* be used.
In contrast, in all other methods and Lua code fragments of `tkz-elements`, standard Lua syntax applies, and the use of the prefix `math.` is required whenever a Lua mathematical function is called. For convenience, users may define local aliases in their Lua code, such as **local sin, cos, pi = math.sin, math.cos, math.pi**.

Correct:

```
sin(x), exp(-x^2)
```

Incorrect (in **fct** and **pfct**):

```
math.sin(x), math.exp(-x^2)
```

36.1. Methods of the class pfct

Table 48: pfct methods.

Methods	Reference
Constructor	
new(exprx,expry)	[36.1.1]
compile(exprx,expry)	[36.1.2]
Methods Returning a Real Number	
x(t)	[36.1.3]
y(t)	[36.1.4]
point(t)	[36.1.5]
Methods Returning a Path	
path(tmin,tmax,n)	[36.1.6]

36.1.1. Constructor new(exprx,expry)

Description: Creates a parametric function object from two expressions or callable Lua functions describing the components $x(t)$ and $y(t)$. When strings are provided, they represent Lua expressions in the variable `t`, evaluated using standard Lua rules (therefore requiring `math.` prefixes).
Arguments:

- **exprx**: a string expression in `t`, or a callable Lua function representing $x(t)$;
- **expry**: a string expression in `t`, or a callable Lua function representing $y(t)$.

Returns: a **pfct** object.

36.1.2. Function compile(exprx,expry)

Description: Compiles the two expressions defining $x(t)$ and $y(t)$ and returns the corresponding callable Lua functions (or wraps them into a **pfct** object, depending on the implementation). This is a low-level helper mainly intended for internal use.

36.1.3. Method x(t)

Description: Evaluates the x-component of the parametric curve at parameter **t**.
Returns: a number.

36.1.4. Method y(t)

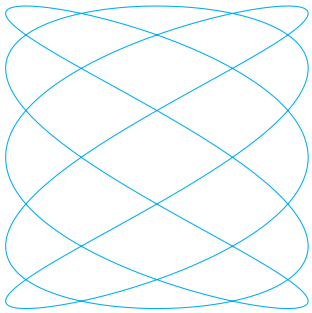
Description: Evaluates the y-component of the parametric curve at parameter **t**.
Returns: a number.

36.1.5. Method point(t)

Description: Constructs the point $(x(t),y(t))$ associated with the parametric curve. This method provides a direct bridge between numerical evaluation and geometric construction.
Returns: a **point**.

36.1.6. Method path(tmin,tmax,n)

Description: Samples the parameter t on the interval $[tmin,tmax]$ using **n** subdivisions and returns the corresponding **path**. Invalid values (NaN, infinities) may be skipped according to the internal policy.
Returns: a **path**.
Example:



```
\directlua{
 init_elements()
 PF.lis = pfct("sin(5*t)", "cos(3*t)")
 PA.curve = PF.lis:path(0, 2*math.pi, 400)
 }

\begin{center}
 \begin{tikzpicture}[scale=2]
  \tkzDrawCoordinates[smooth,cyan](PA.curve)
\end{tikzpicture}
 \end{center}
```

36.2. Macros \tkzDrawPointOnParamGraph and \tkzDrawPointsOnParamGraph

These macros allow drawing points on a *parametric curve* defined in the module system.
The parametric function must already exist in the module table PF.

36.2.1. Macro \tkzDrawPointOnParamGraph

Syntax

\tkzDrawPointOnParamGraph[<TikZ options>]{<t>}{<name>}

Description This macro draws a point belonging to a parametric curve stored in the module PF.
The drawn point has coordinates

$$\big(x(t), y(t)\big),$$

where the parametric curve is defined by the object PF.<name>.

Arguments

- <TikZ options> (optional): graphical options applied to the point
- <t>: value of the parameter
- <name>: name of the parametric function stored in PF

Example

```
\tkzDrawPointOnParamGraph[blue]{1.5}{lis}
```

36.2.2. Macro \tkzDrawPointsOnParamGraph

Syntax

```
\tkzDrawPointsOnParamGraph[<TikZ options>]{<t1,t2,...,tn>}{<name>}
```

Description This macro draws several points belonging to a parametric curve stored in the module PF. For each value t_i in the list, a point of coordinates

$$\big(x(t_i), y(t_i)\big)$$

is computed and drawn.

Arguments

- <TikZ options> (optional): graphical options applied to all points
- <t1,t2,...,tn>: comma-separated list of parameter values
- <name>: name of the parametric function stored in PF

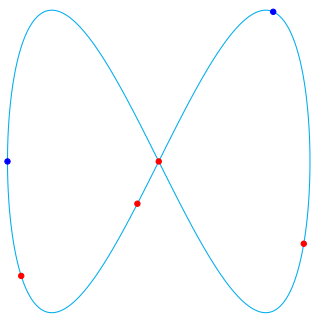
Example

```
\tkzDrawPointsOnParamGraph[red]{0,0.5,1,1.5,2}{lis}
```

Remarks

- These macros assume that the parametric function is already defined in the module PF.
- All evaluations are performed in Lua.
- The macros are compatible with parametric curves drawn using \tkzDrawCoordinates.

Example:



```
\directlua{
 init_elements()
 PF.lem = pfct("cos(t+pi/2)", "sin(2*t)")
 PA.courbe = PF.lem:path(0, 2*math.pi, 200)
 z.A = PF.lem:point(math.pi/2)
 }
\begin{center}
 \begin{tikzpicture}[scale=2]
\tkzGetNodes
  \tkzDrawCoordinates[smooth,cyan](PA.courbe)
  \tkzDrawPoint[blue](A)
  \tkzDrawPointsOnParamGraph[red]{0,2,3,5}{lem}
  \tkzDrawPointOnParamGraph[blue]{4}{lem}
\end{tikzpicture}
 \end{center}
```

37. Metapost

This is for guidance only. As I don't know metapost, I've simply developed the minimum to show how to proceed. The only way I've found to transfer points is via an external file. The macro **\tkzGetNodesMP** does the job. Here's its code.
Remark: This approach is minimal and experimental. The MetaPost integration shown here is primarily intended to demonstrate that point data from Lua can be reused in other graphical systems. While MetaPost is not officially supported by tkz-elements, this example illustrates how external file-based communication can serve as a general method for transferring geometric information.

```
\def\tkzGetNodesMP#1{\directlua{
  local out = assert(io.open("#1.mp", "w"))
  local names = {}
  for K, _ in pairs(z) do
    table.insert(names, tostring(K))
  end
  table.sort(names)
  out:write("pair ", table.concat(names, ", "), ";\string\n")
  for _, name in ipairs(names) do
    local V = z[name]
    if V then
      out:write(name, " := (", V.re, "cm,", V.im, "cm);\string\n")
    end
  end
  out:close()
}}
```

```
\directlua{
  z.A = point(0, 1)
  z.B = point(2, 0)
}
\tkzGetNodesMP{myfic}

\begin{mplibcode}
  input myfic.mp ;
   pickup pencircle scaled 1mm;
  draw A; draw B;
   pickup defaultpen;
  draw A--B;
\end{mplibcode}
```

Part IV.

Mathematical and Computational Foundations

38. Computational Model and Geometric Engine

38.1. Introduction

The package tkz-elements is not merely a collection of geometric construction commands. It implements a coherent computational geometry engine based on a unified mathematical model.
This chapter presents the mathematical and computational foundations underlying the package. While most users interact with high-level geometric objects such as **point**, **line**, **circle**, **triangle**, or **conic**, all constructions ultimately rely on a compact algebraic core and a carefully designed numerical strategy.
The purpose of this chapter is threefold:

- to describe the mathematical model used internally,
- to explain the numerical robustness strategy,
- to clarify the architectural principles governing the engine.

This chapter may be read independently as a technical overview of the geometric engine.

38.2. The Complex-Plane Model

All planar points are represented internally as complex numbers:

$$z = x + iy,$$

where x and y are real coordinates.
This choice provides a compact and algebraically coherent framework for geometric computations. Vector addition, subtraction, rotation, and homothety become natural algebraic operations.
The class **point** therefore has a dual interpretation: it behaves both as a geometric point in the Euclidean plane and as a complex number in the algebraic model. This design avoids maintaining separate vector and complex abstractions and provides a compact, coherent framework for geometric computations.
The complex-plane representation allows:

- direct implementation of vector arithmetic,
- concise expressions for rotations and symmetries,
- natural encoding of orientation via complex multiplication,
- simplified determinant and dot-product calculations.

The use of complex numbers ensures both elegance and computational efficiency.

38.2.1. The Point Class as a Complex Number

For example,

```
z.A = point(1, 2)
z.B = point(1, -1)
```

define two point objects whose associated affixes are

$$z_A = 1 + 2i, \qquad z_B = 1 - i.$$

The notation z.A refers to a Lua object stored in table z, whereas z_A denotes its associated complex number.

If one prefers to work with standalone variables (without using reserved tables), one may also write:

```
za = point(1, 2)
zb = point(1, -1)
```

The only difference is organizational: z.A is stored in the table z, while za is a regular Lua variable.

The algebraic structure is implemented through Lua metamethods and a small set of class methods, allowing standard operators to perform geometric operations in a natural way.

Table 49: Point (complex) metamethods.

Metamethod	Application	Result
add(z1,z2)	z.a + z.b	affix
sub(z1,z2)	z.a - z.b	affix
unm(z)	-z.a	affix
mul(z1,z2)	z.a * z.b	affix
concat(z1,z2)	z.a .. z.b	dot product (real number)[a]
pow(z1,z2)	z.a ^ z.b	determinant (real number)
div(z1,z2)	z.a / z.b	affix
tostring(z)	tostring(z.a)	TeX-friendly display
tonumber(z)	tonumber(z.a)	affix or nil
eq(z1,z2)	z.a == z.b	boolean

a If O is the origin of the complex plane, then $z\_1..z\_2$ corresponds to the dot product of the vectors $\overrightarrow{Oz\_1}$ and $\overrightarrow{Oz\_2}$.

Table 50: Point (complex) class methods.

Method	Application	Result
conj()	z.a:conj()	affix (conjugate)
mod()	z.a:mod()	real number (modulus)
abs()	z.a:abs()	real number (modulus)
norm()	z.a:norm()	real number (squared modulus)
arg()	z.a:arg()	real number (argument in radians)
get()	z.a:get()	re and im (two real numbers)
sqrt()	z.a:sqrt()	affix

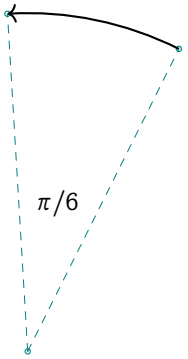
38.2.2. Example of complex use

Let za = math.cos(a) + i math.sin(a) . This is obtained from the library by writing

```
za = point(math.cos(a),math.sin(a)).
```

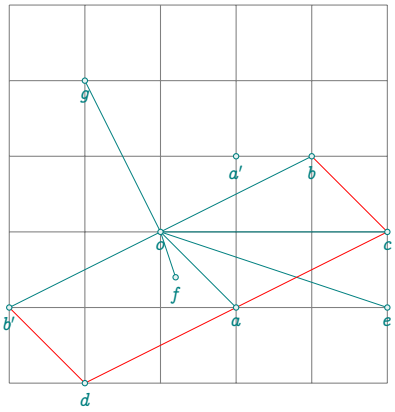
Then z.B = z.A * za describes a rotation of point A by an angle a.

```
\directlua{
 init_elements()
 z.O = point(0, 0)
 z.A = point(1, 2)
 a = math.pi / 6
 za = point(math.cos(a), math.sin(a))
 z.B = z.A * za}
\begin{tikzpicture}
 \tkzGetNodes
 \tkzDrawPoints(O,A,B)
 \tkzDrawArc[->,delta=0](O,A)(B)
 \tkzDrawSegments[dashed](O,A O,B)
 \tkzLabelAngle(A,O,B){$\pi/6$}
\end{tikzpicture}
```



38.2.3. Point operations (complex)

```
\directlua{
 init_elements()
 z.o = point(0, 0)
 z.a = point(1, -1)
 z.b = point(2, 1)
 z.bp = -z.b
 z.c = z.a + z.b
 z.d = z.a - z.b
 z.e = z.a * z.b
 z.f = z.a / z.b
 z.ap = point.conj(z.a)
 z.g = z.b * point(math.cos(math.pi / 2),
     math.sin(math.pi / 2))}

\begin{tikzpicture}
 \tkzGetNodes
 \tkzInit[xmin=-2,xmax=3,ymin=-2,ymax=3]
 \tkzGrid
 \tkzDrawSegments(o,a o,b o,c o,e o,b')
 \tkzDrawSegments(o,f o,g)
 \tkzDrawSegments[red](a,c b,c b',d a,d)
 \tkzDrawPoints(a,...,g,o,a',b')
 \tkzLabelPoints(o,a,b,c,d,e,f,g,a',b')
\end{tikzpicture}
```



38.2.4. Barycentric Combination

A fundamental operation in the computational engine is the barycentric combination of points. Given points $z_1, \ldots, z_n$ and associated real weights $w_1, \ldots, w_n$, the barycenter is defined as

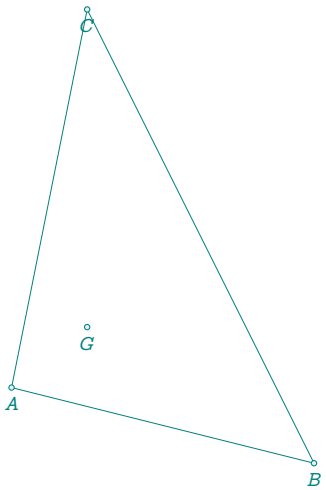
$$G = \frac{\sum_{i=1}^{n} w_i z_i}{\sum_{i=1}^{n} w_i}.$$

Because points are represented as complex numbers, this operation reduces to a weighted linear combination.

The internal function **barycenter_** implements this formula directly at the algebraic level. The public interface **barycenter** provides a user-friendly wrapper.

The barycentric construction is used extensively throughout the package (in triangle centers, homotheties, affine constructions, and other derived geometric algorithms).

Example.




```
\directlua{
 init_elements()
 z.A = point(1, 0)
 z.B = point(5, -1)
 z.C = point(2, 5)
 z.G = tkz.barycenter({z.A, 3}, {z.B, 1}, {z.C, 1})
}
\begin{center}
\begin{tikzpicture}
\tkzGetNodes
\tkzDrawPolygon(A,B,C)
\tkzDrawPoints(A,B,C,G)
\tkzLabelPoints(A,B,C,G)
\end{tikzpicture}
\end{center}
```

38.3. Algebraic Primitives and Geometric Operators

The computational engine relies on a minimal set of algebraic operations defined on complex numbers.
These primitives are sufficient to implement all higher-level geometric constructions such as projections, intersections, membership tests, orientation checks, and affine combinations.
Because points are represented as complex numbers, vector operations reduce to elementary algebra.

38.3.1. Vector Arithmetic

Given two points represented by complex numbers, vector subtraction is defined naturally:

$$\overrightarrow{AB} = z_B - z_A.$$

Addition and scalar multiplication follow directly from complex arithmetic.
These operations form the basis of translations, midpoints, affine combinations, and homotheties.

38.3.2. Dot Product

Given two vectors represented by complex numbers $z_1 = a + ib$ and $z_2 = c + id$, their scalar product is defined as

$$z_1 \,..\, z_2 = ac + bd.$$

In the implementation, this quantity is computed using the operator `..`:

$$z_1 \,..\, z_2 = ac + bd.$$

Equivalently, the dot product corresponds to the real part of the product of the first vector and the conjugate of the second:

$$z_1 \,..\, z_2 = \mathrm{Re}\left(z_1 \overline{z_2}\right).$$

The dot product plays a central role in planar geometry. It is used for:

- orthogonality tests,
- projections onto a line,
- distance computations,
- angle measurements.

Distance computation. The squared distance between two points A and B is

$$d(A,B)^2 = (z_B - z_A) \,..\, (z_B - z_A).$$

Angle measurement. For three points A, B, C, the angle $\widehat{ABC}$ satisfies

$$\cos(\theta) = \frac{(z_A - z_B) \,..\, (z_C - z_B)}{\|z_A - z_B\| \, \|z_C - z_B\|}.$$

38.3.3. Determinant and Oriented Area

Given two vectors represented by complex numbers $z_1 = a + ib$ and $z_2 = c + id$, their determinant is defined as

$$\det(z_1, z_2) = ad - bc.$$

This quantity represents the signed (or oriented) area of the parallelogram generated by the two vectors.
In the implementation, the determinant is computed using the operator `^`:

$$z_1 \hat{} z_2 = ad - bc.$$

Given three points A, B, C, the oriented area of triangle ABC is obtained from

$$\det(\overrightarrow{AB},\overrightarrow{AC}) = (z_B - z_A)\hat{}(z_C - z_A).$$

The sign of this quantity determines the orientation:

- positive: direct (counterclockwise),
- negative: indirect (clockwise),
- zero (up to tolerance): aligned points.

This primitive is fundamental for:

- triangle orientation,
- segment membership tests,
- line intersection,
- inside/outside classification.

38.4. Numerical Robustness and Tolerance Control

Geometric computations rely on floating-point arithmetic and are therefore subject to numerical approximation errors.
To ensure robustness, all membership and position tests use a configurable tolerance parameter:

`tkz.epsilon`

This tolerance is applied systematically in:

- alignment tests,
- intersection detection,
- tangency classification,
- equality comparisons,
- geometric membership tests.

Rather than relying on strict equality, the engine evaluates whether a quantity is sufficiently close to zero.
This strategy guarantees numerical stability across complex constructions.

38.5. Geometric Classification Model

Geometric objects provide a unified classification interface based on three possible states:

`"ON"  "IN"  "OUT"`

This tri-state model applies consistently across:

- **line**
- **circle**
- **triangle**
- **conic**

Membership and region tests are therefore harmonized throughout the system.
For backward compatibility, boolean interfaces are preserved where necessary, but the internal model is uniformly tri-state.

38.6. Degenerate Configurations

Special care is taken to manage degenerate geometric cases:

- coincident points,
- concentric circles,
- aligned triangle vertices,
- zero-radius constructions,
- tangency limit cases.

Rather than allowing numerical instability, the engine classifies these configurations explicitly.
This design choice prevents undefined behaviors and improves the reliability of higher-level constructions.

38.7. Architecture of the Computational Engine

The engine is structured in two layers:

Primary Computational Functions

Low-level functions perform algebraic computations. They are internal and are not part of the public API.

Object-Oriented Layer

Geometric classes provide user-facing methods. These methods call primary functions while ensuring:

- consistent tolerance handling,
- type safety,
- backward compatibility.

This separation maintains clarity between computation and abstraction.

38.7.1. Internal Data Tables

In Lua, the main data structure is the *table*. It functions both as an array and as a dictionary, allowing you to store sequences of values, associate keys with values, and even represent complex objects. Tables are the foundation for representing geometric structures such as points, lines, and triangles in tkz-elements.

38.7.2. General Use of Tables

Tables are the only data structure "container" integrated in Lua. They are associative arrays which associates a key (reference or index) with a value in the form of a field (set) of key/value pairs. Moreover, tables have no fixed size and can grow based on our need dynamically.
Tables are created using curly braces:

```
T = {} % T is an empty table.
```

In the next example, **coords** is a table containing four points written as coordinate strings. Lua indexes tables starting at 1.

1: (0,0)
2: (1,0)
3: (1,1)
4: (0,1)

```
\directlua{
 local coords = { "(0,0)", "(1,0)", "(1,1)", "(0,1)" }
 for i, pt in ipairs(coords) do
    tex.print(i .. ": " .. pt)
    tex.print([[\\]])
 end}
```

You can define a table with unordered indices as follows:

```
coords = {[1] = "(0,0)", [3] = "(1,1)", [2] = "(1,0)"}
```

Accessing an indexed element is straightforward:

```
tex.print(coords[3]) --> (1,1)
```

You can also append new elements:

```
coords[4] = "(0,1)"
```

To iterate over a table with numeric indices in order, use **ipairs**:

```
for i, v in ipairs(coords) do
print(i, v)
end
```

This will print:

```
1 (0,0)
2 (1,0)
3 (1,1)
4 (0,1)
```

Key–Value Tables.

Tables may also use string keys to associate names to values. This is especially useful in geometry for storing attributes like color or label:

```
properties = {
A = "blue",
B = "green",
C = "red"
}
properties.D = "black"
```

To iterate over all key–value pairs (order not guaranteed), use **pairs**:

```
for k, v in pairs(properties) do
print(k, v)
end
```

Deleting an entry.

To remove an entry, assign **nil** to the corresponding key:

```
properties.B = nil
```

38.7.3. Variable Number of Arguments

Tables can also be used to store a variable number of function arguments:

```
function ReturnTable(...)
return table.pack(...)
end
```

```
function ParamToTable(...)
local mytab = ReturnTable(...)
for i = 1, mytab.n do
print(mytab[i])
end
end

ParamToTable("A", "B", "C")
```

This technique is useful for collecting points, coordinates, or any variable-length data.

Accessing with Sugar Syntax.

In tables with string keys, there are two common syntaxes:

- **properties["A"]** — always valid;
- **properties.A** — shorter, but only works with string keys that are valid Lua identifiers (not numbers).

38.7.4. Table z

The most important table in tkz-elements is **z**, used to store geometric points. It is declared as:

```
z = {}
```

Each point is then stored with a named key:

```
z.M = point(3, 2)
```

The object **z.M** holds real and imaginary parts, accessible as:

```
z.M.re --> 3
z.M.im --> 2
```

If you print it:

```
tex.print(tostring(z.M))
```

You get the complex representation **3+2i**.

Moreover, points in the **z** table are not just data—they are objects with methods. You can perform geometric operations directly:

```
z.N = z.M:rotation(math.pi / 2)
```

This makes the **z** table central to object creation and manipulation in tkz-elements.

38.8. Design Principles and Trade-offs

The design of tkz-elements follows several guiding principles:

- Mathematical coherence,
- Minimal algebraic core,
- Numerical robustness,
- Backward compatibility,
- Performance efficiency.

The choice of complex representation, tri-state classification, and tolerance-based comparisons results from balancing mathematical rigor and practical usability.

Index