Towards a Ruby-driven Mathematical Diagram Tool

Table of Contents

[Introduction 2](#_Toc244338688)

[What I want from Ruby 3](#_Toc244338689)

[Source code 4](#_Toc244338690)

[METAPOST 4](#_Toc244338691)

[Eukleides 5](#_Toc244338692)

[Ruby 6](#_Toc244338693)

[Summary 7](#_Toc244338694)

[Further reading 7](#_Toc244338695)

[Sample Code: Eukleides Examples 8](#_Toc244338696)

[Problem of Thales 8](#_Toc244338697)

[A property of parallelograms 9](#_Toc244338698)

[Isosceles triangle 10](#_Toc244338699)

[Collinear points 10](#_Toc244338700)

[Incircle and bisectors 11](#_Toc244338701)

[Fueurbach circle 12](#_Toc244338702)

[Tangents to a circle 13](#_Toc244338703)

[Addition of vectors 13](#_Toc244338704)

[Spiral 14](#_Toc244338705)

[Sample Code: HSC Questions 15](#_Toc244338706)

[2008 Extension 1 Q5(c) 15](#_Toc244338707)

[2008 Mathematics Q8(b) 16](#_Toc244338708)

[2008 Extension 2 Q4(a) 17](#_Toc244338709)

[2008 Extension 2 Q4(a)(ii) 18](#_Toc244338710)

[2003 General Q9 19](#_Toc244338711)

[Sample code: PGF/TikZ examples 21](#_Toc244338712)

[Proposition II 21](#_Toc244338713)

[Annulus Sector 22](#_Toc244338714)

[Asymptote 23](#_Toc244338715)

[Further images and Eukleides code to turn into examples one day 28](#_Toc244338716)

# Introduction

Mathematical diagrams are cumbersome to create. All sorts of tools have all sorts of pros and cons. In the GUI camp: FXDraw has many good features, but has a long way to go; Geometers Sketchpad is wonderful and scriptable, but is not designed for diagrams *per se*, and therefore can’t easily show segments as being equal, for instance. Nor can it emit a scalable graphic. Among the text-based tools, METAPOST seems powerful but very low-level, and Eulkeides is admirably intelligent and geometrical, but not sufficiently high-level (you have to give every point a name; you can’t create reusable components).

GUI or text? Both have their advantages, and which tool is best depends on the particular job (and the skill of the user). For the kinds of documents I create, Microsoft Word is best. Were I to write a serious article or book, I’d certainly head for LaTeX, though. With mathematical diagrams, simple things can easily be created in FXDraw, and the fact of having the *object* embedded in the Word document is very handy: one double-click and I’m editing the diagram. It quickly runs into limitations, though: it’s cumbersome to label points; complex diagrams (e.g. solids, needing equal and parallel segments to represent depth) are tricky to achieve. It’s clever in some ways, but often not in the way I want.

Text-driven diagrams, though, have a high barrier to entry, an innate complexity. Only a complex diagram would really warrant their use, unless you were already an expert. Not only is the act of creating the diagram more complicated, so is the toolchain: type in text, compile picture, include in document. The two options mentioned so far – METAPOST and Eukleides – are tightly coupled to TeX: they can’t even generate general-purpose images. Finally, imagine creating a complex diagram, then wanting to edit it months later. Without a high-level input language, this is a daunting proposition.

The key sentence in the above paragraph is: *only a complex diagram would warrant their use*. It was the act of reading about one such complex diagram in a METAPOST article that got me thinking about the desirability of generating such a beast.

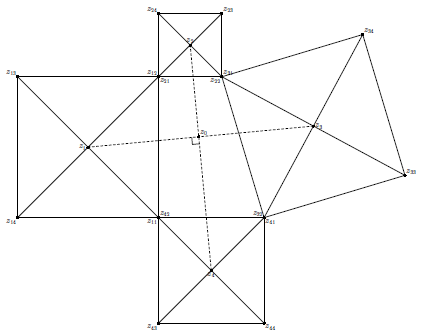
The thought of creating this in FXDraw makes me shudder. As it turns out, it’s easily done in Sketchpad (but not as flexibly; in METAPOST you can easily generate variations of the image to show that the centre angle is always 90°). So I thought about how it might be achieved using Ruby. A friend referred me to Eukleides, and after a little reading I managed it.

Figure 1 – Quadrilateral diagram in METAPOST

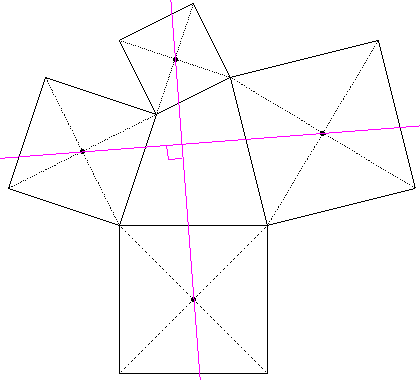
As we will see later, the Eukleides code is much nicer than the METAPOST code. But it lacks high-level definition. Each square must be defined from scratch. There *should* be a way to say: I want to create a square, draw its diagonals in dotted lines, and mark the centre, but there doesn’t seem to be. Instead, it’s four repetitions of:

Figure 2 – Quadrilateral diagram in METAPOST

B A s11 s12 square

draw(B, A, s11, s12)

draw(B, s11, dotted)

draw(A, s12, dotted)

cs1 = barycenter(A, B, s11, s12)

draw(cs1)

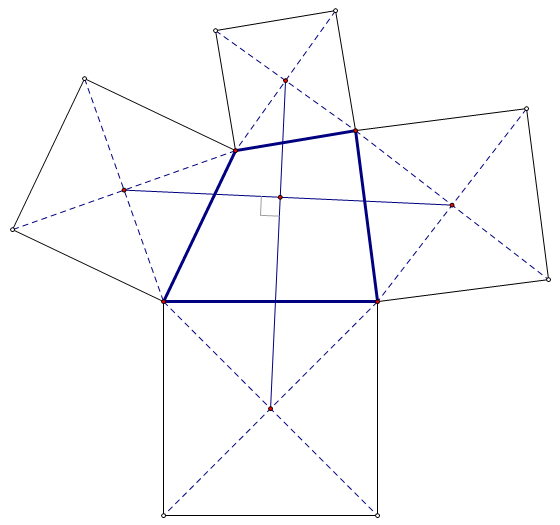
The Geometers’ Sketchpad version of this diagram was easy to create using its “Tools” feature. I created one square with the diagonals and centre dot rendered as required, turned that into a “tool” and stamped out all four in no time. The custom tool for marking a right angle made the finishing touch easy, too. GSP is extendable in a way that Eukleides is not.

Figure 3 – Quadrilateral diagram in The Geometers Sketchpad

However, if I wanted to mark sides as being equal, for instance, I’m out of luck.

## What I want from Ruby

I want to be able to define points, segments, lines, polygons, circles, etc., just like in Eukleides, especially being able to use existing points to constrain new shapes intelligently.

I want each shape to be an object, so that shapes know how to draw themselves, label their vertices, draw their diagonals; segments can mark themselves with single or double ticks.

I want elements of a DSL,where things can be defined in a way that seems natural, e.g.

triangle(:ABC, :base\_angles => [20.deg, 45.deg], :upside\_down)

## Source code

### METAPOST

boolean show\_labels; show\_labels := false;

def draw\_problem (expr p, q, r, s ) =

save x, y, a, b, c, d, e, f, g, h;

z11 = p ; z12 = q ; z22 = r ; z32 = s ;

z31 = z22 ; z21 = z12 ; z41 = z32 ; z42 = z11 ;

a = x12 - x11 ; b = y12 - y11 ;

z11 = (x11, y11) ; z12 = (x12, y12) ;

z13 = (x12-b, y12+a) ; z14 = (x11-b, y11+a) ;

c = x22 - x21 ; d = y22 - y21 ;

z21 = (x21, y21) ; z22 = (x22, y22) ;

z23 = (x22-d, y22+c) ; z24 = (x21-d, y21+c) ;

e = x32 - x31 ; f = y32 - y31 ;

z31 = (x31, y31) ; z32 = (x32, y32) ;

z33 = (x32-f, y32+e) ; z34 = (x31-f, y31+e) ;

g = x42 - x41 ; h = y42 - y41 ;

z41 = (x41, y41) ; z42 = (x42, y42) ;

z43 = (x42-h, y42+g) ; z44 = (x41-h, y41+g) ;

pickup pencircle scaled .5pt;

draw z11--z12--z13--z14--z11; draw z11--z13; draw z12--z14;

draw z21--z22--z23--z24--z21; draw z21--z23; draw z22--z24;

draw z31--z32--z33--z34--z31; draw z31--z33; draw z32--z34;

draw z41--z42--z43--z44--z41; draw z41--z43; draw z42--z44;

z1 = 0.5[z11,z13] ; z2 = 0.5[z21,z23] ;

z3 = 0.5[z31,z33] ; z4 = 0.5[z41,z43] ;

draw z1--z3 dashed evenly;

draw z2--z4 dashed evenly;

z0 = whatever[z1,z3] = whatever[z2,z4];

mark\_rt\_angle (z1, z0, z2) ;

if show\_labels:

pickup pencircle scaled 5pt;

verbatimtex \def\Z #1{$z\_{\scriptscriptstyle #1}$} etex;

verbatimtex \def\ZZ#1#2{$z\_{\scriptscriptstyle #1#2}$} etex;

dotlabel.llft(btex \ZZ 11 etex, z11); dotlabel.ulft(btex \ZZ 12 etex, z12);

dotlabel.ulft(btex \ZZ 13 etex, z13); dotlabel.llft(btex \ZZ 14 etex, z14);

dotlabel.lrt (btex \ZZ 21 etex, z21); dotlabel.llft(btex \ZZ 22 etex, z22);

dotlabel.urt (btex \ZZ 23 etex, z23); dotlabel.ulft(btex \ZZ 24 etex, z24);

dotlabel.urt (btex \ZZ 31 etex, z31); dotlabel.ulft(btex \ZZ 32 etex, z32);

dotlabel.urt (btex \ZZ 33 etex, z33); dotlabel.urt (btex \ZZ 34 etex, z34);

dotlabel.lrt (btex \ZZ 41 etex, z41); dotlabel.urt (btex \ZZ 42 etex, z42);

dotlabel.llft(btex \ZZ 43 etex, z43); dotlabel.lrt (btex \ZZ 44 etex, z44);

dotlabel.urt (btex \Z 0 etex, z0);

dotlabel.lft (btex \Z 1 etex, z1); dotlabel.top (btex \Z 2 etex, z2);

dotlabel.rt (btex \Z 3 etex, z3); dotlabel.bot (btex \Z 4 etex, z4);

fi;

enddef;

angle\_radius = 10pt;

def mark\_rt\_angle (expr a, b, c) =

draw ((1,0)--(1,1)--(0,1)) zscaled (angle\_radius\*unitvector(a-b)) shifted b

enddef;

def do\_draw\_problem ( expr n, i, j ) =

beginfig ( n );

draw\_problem

( (400,400), (300,600),

i[(300,600),(550,800)], j[(400,400),(550,500)] );

endfig;

enddef;

do\_draw\_problem ( 40 , 1.0 , 1.0 );

### Eukleides

Presented in two columns, as it is sufficiently narrow.

frame(-3, -5, 8, 8)

**% ABCD is our central quadrilateral.**

A = point(0,0)

B = point(4,0)

C = point(3,4)

D = point(1,3)

draw(A, B, C, D)

**% Now we want a square on each side.**

**% Each square has dotted diagonals.**

B A s11 s12 square

draw(B, A, s11, s12)

draw(B, s11, dotted)

draw(A, s12, dotted)

C B s21 s22 square

draw(C, B, s21, s22)

draw(C, s21, dotted)

draw(B, s22, dotted)

D C s31 s32 square

draw(D, C, s31, s32)

draw(D, s31, dotted)

draw(C, s32, dotted)

A D s41 s42 square

draw(A, D, s41, s42)

draw(A, s41, dotted)

draw(D, s42, dotted)

**% Mark the centre of each square, and**

**% join up the centres of opposite**

**% squares. These two segments ought**

**% to be perpendicular.**

cs1 = barycenter(A, B, s11, s12)

draw(cs1)

cs2 = barycenter(B, C, s21, s22)

draw(cs2)

cs3 = barycenter(C, D, s31, s32)

draw(cs3)

cs4 = barycenter(D, A, s41, s42)

draw(cs4)

color(magenta)

line1 = line(cs1, cs3)

line2 = line(cs2, cs4)

X = intersection(line1, line2)

draw(line1)

draw(line2)

mark(cs4, X, cs1, right)

%%% Done!

The things that grate about this, from a general-purpose programmer’s point of view (Ruby in particular):

* Repeated code: each of the four squares must be spelled out in full.
* Needless point names: things like s21; yes they’re needed to make segments, but I don’t care about a *name* for them. Anyway, I should be able to tell the software to draw the squares’ diagonals, rather than have to specify segments.
* No structure: the color command simply flicks a switch for everything that follows; I’d like to see it (optionally) have a scope.

### Ruby

This is the *proposed* source code.

points :A => p(0,0),

:B => p(4,0),

:C => p(3,4),

:D => p(1,3)

q = quadrilateral(:ABCD)

squares = [:AB, :BC, :CD, :DA].map { |seg|

square(:base => seg, :away\_from => q.centre) do |s|

s.diagonals(:dotted)

s.centre(:dot)

}

scope do

style(:grey) do

points :WXYZ => squares.map { |s| s.centre }

segments(:WY, :XZ)

point :P => intersection(:WY, :XZ)

angle(:ZPY).mark(:right)

end

end

Some points:

* I envisage doing away with the cartesian definition of points *for this example* (not in general) in favour of defining a quadrilateral with side lengths and/or angles, or just accepting a default (scalene) shape. I defined the points simply because that’s what I did in the Eukleides example.
* The four squares are defined all in one go, using the appealingly simple :AB, :BC etc. to specify the base segments. The :away\_from option ensures the squares are oriented the desired way.
* A square object knows about its diagonals and its centre, making it trivial to render these things. If this software were packaged and distributed, a user could add extra capabilities to a square because Ruby allows additions to existing classes.
* scope introduces a new scope for point definitions. The points W, X, Y, Z and P are needed only temporarily; there is no point littering the global scope with them. I could have accessed the same points without naming them, like

c = squares.map { |s| s.centre }

segments([c[0],c[2]], [c[1],c[3]])

however that is far less readable. With scopes, we can name points to our heart’s content, then forget about those names and reuse them later if needed.

* style takes a block, so that the colour, thickness, etc. it specifies only applies to a limited area.
* Angles are cheerfully specified with a symbol as well – :ZPY

It may not be possible to implement a library that allows the exact source code above, and in any case I expect design improvements will be made. But it is certainly possible to implement something in that spirit. It is no more complicated than the Eukleides code; it merely represents the same information in a more compact way, using Ruby’s control structures and some easily implemented methods (diagonals, etc.).

### Summary

By studying one motivating problem, I have demonstrated a nice potential Ruby interface for drawing geometric diagrams. The problem was particularly apposite because of its medium complexity and its repeated structures.

The next section explores several further examples – some Eukleides examples (no need for further source code though), some diagrams from textbooks and exam papers, and a variety of very simple diagrams to nut out basic construction commands.

All of it is psuedocode at this stage: the idea is to round out the design of the library by considering lots of cases.

At this point, it’s worth mentioning that many ideas presented here come from Eukleides. I would be scarcely able to conceive of an effective Ruby-driven geometry library without having studied its approach.

## Further reading

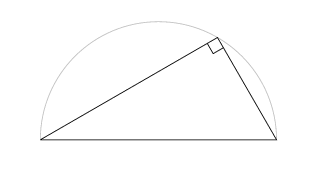
The PGF/TikZ library for TeX looks extremely accomplished. It has a wonderful manual with an extended section on Geometry (specifically Euclid’s *Elements*). Furthermore, an examples webpage shows off its caliber very well. I can’t believe I didn’t discover this sooner.

* Examples: <http://www.texample.net/tikz/examples/>
* Manual: search for “PGF manual”

# Sample Code: Eukleides Examples

All of the examples in this section come from the Eukleides website (samples section), where the images and Eukleides code can be found.

## Problem of Thales



triangle(:ABC, :base => 4, :angles => [30.d, 60.d])

angle(:ACB).mark(:right)

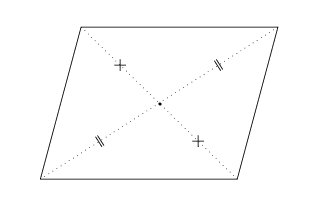
circle(:diameter => :AB, :arc => :tophalf, :style => :light)

triangle(:ABC).label\_vertices # Not shown above

Notes:

* Line 1 shows how a triangle can be constucted with a base length and the base angles.
* Line 4 shows how we can access an existing triangle without having maintained a reference to it. The first reference will create the triangle object and draw it (unless we specify :nodraw or something). Subsequent references merely recall the object so we can do extra things to it.
* The code calls Triangle#label\_vertices, but the diagram (copied from the ’net) doesn’t show it. It’s showoff code. All shapes know how to label their own vertices in the most appealing way: away from their centre. It’s not merely labelling a point; it’s making an intelligent decision where to put that label. Of course, that may clash with some other part of the diagram, so the labels can be adjusted if necessary.

## A property of parallelograms



parallelogram(:ABCD, :base => 6, :angle => 80.deg) do |p|

point :E => p.centre

style(:dotted) do

segments(:AE, :EC).equal(:double)

segments(:BE, :ED).equal(:cross)

end

end

# To mark parallel lines (not shown in image)...

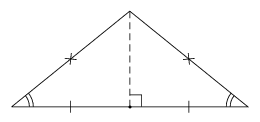
segments(:AB, :CD).parallel(:double)

segments(:AD, :BC).parallel(:single)

Notes:

* methods equal and parallel to mark two segments as equal or parallel

## Isosceles triangle



triangle(:ABC, :iscosceles, :base => 8) do |t|

t.altitude(:B, :dashed)

end

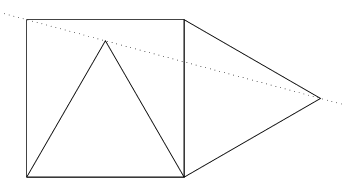
point :M => midpoint(:AB).draw(:dot)

segments(:AM, :MB).equal(:single)

segments(:AC, :BC).equal(:cross)

angles(:CAM, :CBM).equal(:double)

## Collinear points



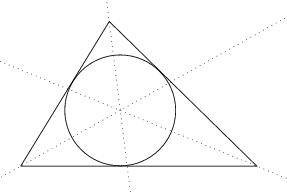
square(:ABCD)

triangle(:ABE, :equilateral)

triangle(:BCF, :equilateral, :away\_from => :E)

line(:EF, :dotted)

## Incircle and bisectors



triangle(:ABC).do |t|

t.incircle

t.each\_angle do |a|

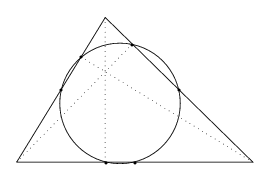
bisector(a, :dotted)

end

end

## Fueurbach circle

Demonstrates that the circle through the feet of the three altitudes of a triangle also passes through the three medians.



This diagram is fairly complex, using altitudes and medians of a triangle, and a circle through three points. A key aim of this tool is to avoid naming points explicitly where possible, and provide a rich API. Of course a triangle can determine and draw its own medians and altitudes, but how do you then access the points? What data would the method return? If you want to label the medians M1, M2 and M3, for instance, how do you do that nicely? Is a median a point or a line? (A line.) What if you just want a segment? Altitude segment or altitude line?

Therefore I present two approaches to this, starting with low-level (where references to points are kept, etc.) and proceeding to high-level (very object-oriented).

triangle(:ABC) do |t|

midpoints = t.sides.map { |side| side.midpoint }

midpoints.draw(:dot).label(:Mn, :away\_from => t.centre)

t.each\_side do |side|

v = t.opposite\_vertex(side)

p = side.projection(v) # a point on the segment

segment(p, v, :dotted).label(:Fn) # auto-numbering

end

circle(points(:Bn)) # points(:Bn) -> [:B1, :B2, :B3]

end

triangle(:ABC) do |t|

t.midpoints(:dot).name(:Mn)

t.altitudes.each do |a|

a.foot.draw(:dot).name(:Fn)

a.segment.draw(:dotted)

a.angle.mark(:right)

end

end

circle(points(:Fn))

# Labelling – automatically away from the centre of the triangle

triangle(:ABC).label(:vertices, :Mn, :Fn)

## Tangents to a circle

## 

points :O => p(2,2).cross,

:A => (6.5,2).dot

c1 = circle(:O, 2)

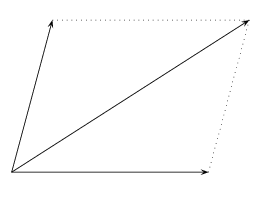
c2 = circle(:diameter => :OA, :dotted)

intersection(c1, c2).each do |pt|

line(:A, pt, :grey)

end

## Addition of vectors



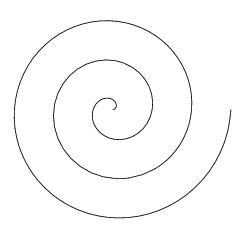
# Copying Eukleides example approach

parallelogram(:ABCD, :nodraw) # first time we’ve used :nodraw

vector(:AB, :AC, :AD) # vectors are drawn with arrows

segments(:BC, :DC, :dotted)

## Spiral



This is not so much a geometrical drawing, more a locus (the Euleides command is trace). I should be able to include it.

Eukleides code:

frame(-5, -4, 5, 4)

trace(t, 0, 3\*360)

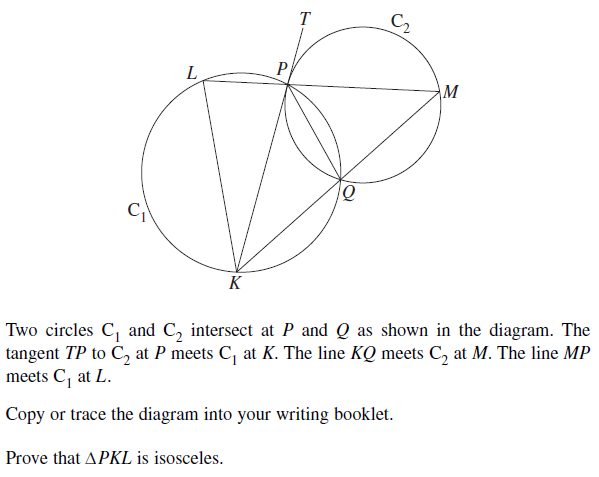
{ point(t/360, t:) }

Ruby code:

trace(:t => [0,360]) { |t| polar(t/360.0, t.deg) }

# Sample Code: HSC Questions

## 2008 Extension 1 Q5(c)



point :O1 => :origin, :O2 => p(:O1) + [6.5, 20.deg]

c1 = circle(:C1, :centre => :O1, :radius => 5)

c2 = circle(:C2, :centre => :O2, :radius => 3)

style(:nodraw)

points :PQ => intersection(:C1, :C2) # Which one’s which?

point :K => intersection(:C1, c2.tangent(:P))

segment(:KP).extend(:T, 1.5)

point :M => intersection(:C2, line(:KQ))

point :L => intersection(:C1, line(:MP))

end

# All points defined; now to draw and label everything.

c1.label(190.d)

c2.label(70.d)

segment(:PQ).label

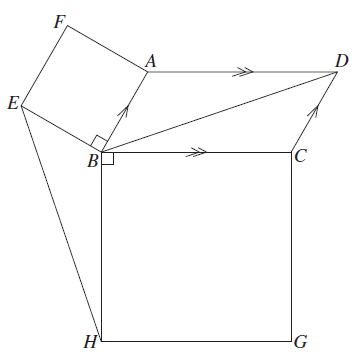
segment(:TK).label

segment(:KM).label # doesn’t relabel K

segment(:ML).label # doesn’t relabel M

segment(:LK)

## 2008 Mathematics Q8(b)



parallelogram(:BCDA).mark\_parallel\_lines.label

segment(:BD)

square(:BAFE).label

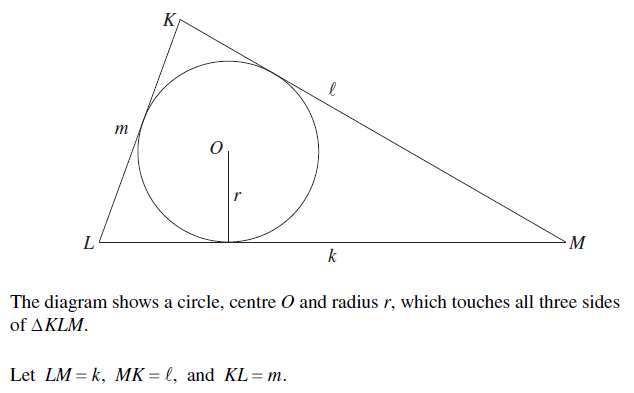
square(:CBHG).label

segment(:EH)

angle(:CBH).mark(:right)

angle(:ABE).mark(:right)

## 2008 Extension 2 Q4(a)



triangle(:LMK, :base => 8, :angles => [75.d, 22.d]) do |t|

t.label\_vertices

t.label\_sides

t.incircle do |c|

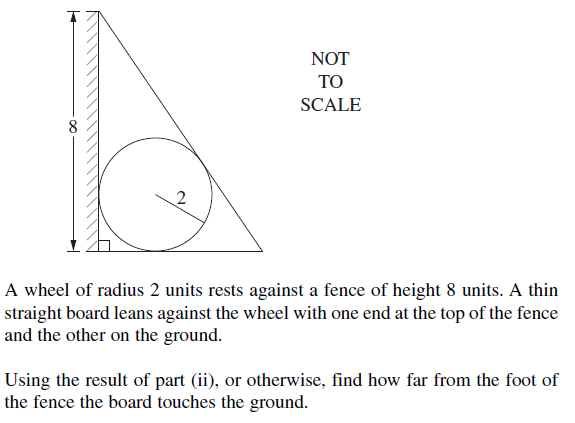
c.label\_centre(:O)

c.radius(270.d).label(:r)

end

end

## 2008 Extension 2 Q4(a)(ii)



triangle(:ABC, :right => :A, :base => 3, :height => 5) do |t|

t.angle(:A).mark(:right)

t.incircle do |c|

c.radius(-20.d).mark(2)

end

end

rectangle(:base => :AC, :height => 0.2, :nodraw) do |r|

r.top.indicate\_length(8, :left)

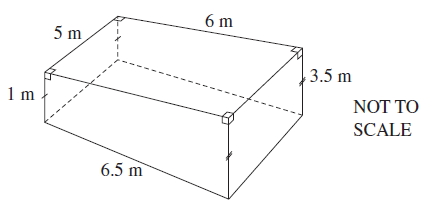
r.fill(:downleft)

r.sides.draw

end

## 2003 General Q9

This is a very nice diagram that I imagine would be a headache to create in any software. Asymptote *may* make it easy by specifying a 3D solid and rendering a 2D view of it, but I doubt it. I’ve never really had 3D in mind when thinking about RGeom, but it would be nice to be able to handle this. One feature this really motivates is generating a shape that you can then translate etc (e.g. to draw the front, then form the back), and labelling the points within that namespace.



**### Create front and back shapes ###**

shape :front do

points :A => p(0,0), :B => p(0,3), :C => (-8,5), :D => p(-8,2.5)

end

vector = v(3,3)

shape :back => translate(:front, vector)

f, b = Shape[:front, :back]

shape :front { segments :ABCDA }

shape :back { segments :ABC; segments :CDA, :dashed }

**### Segments joining front and back ###**

segment f[:A], b[:A]

segment f[:B], b[:B]

segment f[:C], b[:C]

segment f[:D], b[:D], :dashed

**### Right angles ###**

shape :front { mark\_right\_angles :ABC, :BCD }

shape :top do

points :W => f[:B], :X => f[:B], :Y => b[:B], :Z => b[:C]

mark\_right\_angles :WXY, :XYZ, :YZW, :ZWY

end

mark\_right\_angle [f[:A], f[:B], b[:B]]

mark\_right\_angle [f[:B], b[:B], b[:A]]

**### Segment ticks ###**

mark\_segments(f[:AB], b[:AB], :double, :dir => vector)

mark\_segments(f[:CD], b[:CD], :single, :dir => vector)

**### Segment lengths ###**

label\_segment(f[:CD], “1 m”, :left)

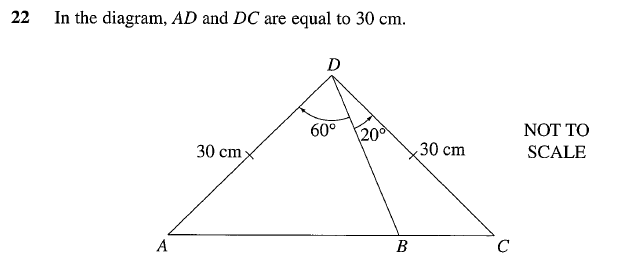
label\_segment(f[:AD], “6.5 m”, :down)

label\_segment(b[:AB], “3.5 m”, :right)

label\_segment(f[:BC], “6 m”, :up)

label\_segment( [f[:C], b[:C]], “5 m”, :NW )

## 2010 General Q22



# Sample code: PGF/TikZ examples

PGF is a low-level inline graphics package for TeX. TikZ is a higher-level drawing language on top of PGF. It’s a compelling combination. The manual is a very well-written document with some good examples. The example below is taken from Euclid’s *Elements* (Proposition II).

## Proposition II

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | Given: | Interval BC and point A. | | Aim: | Create interval AL equal to BC. Direction is unimportant. | | Method: | Construct equilateral triangle ABD.  Construct circle BC (called H).  Extend DA to E and DB to F.  Construct point G where DF intersects circle H.  Construct circle DG (called K).  Construct point L where DE meets circle K.  Now AL = BC, as required. | | Reason: | DG = DL. But DG = DB + BG and DL = DA + AL. DA = DB, which means AL = BG. But BG = BC, so AL = BC. | |

# Givens

points :B => p(0,0), :C => p(.3,2), :A => p(-1,-.4)

segment(:BC, :blue)

# Constructions

triangle(:ABD, :equilateral)

circle(:H, :radius => :BC)

segment(:DA).extend(4, :E)

segment(:DB).extend(4, :F)

point :G => intersection(:H, :DF)

circle(:K, :radius => :DG)

point :L => intersection(:K, :DE)

# Labels etc.

dot :C, :B, :A, :D, :G, :L

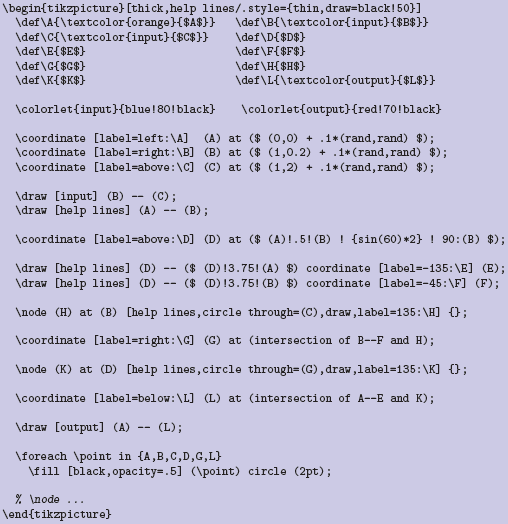
label :BC, :blue

label :D => :N, :A => [:W, :orange], :G => :W,

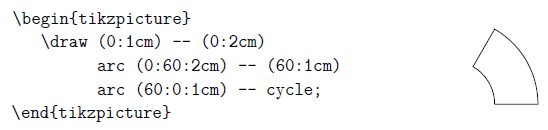
:E => :SW, :F => :SE, :H => :NW, :K => :NW

segment(:AL, :red)

This Ruby code is pretty lovely, whereas the TikZ code (below) is pretty gross.



## Annulus Sector



This example shows off TikZ very well. The ease with which it can combine intervals and arcs within a single *draw* command is impressive. Explanation: (60:1cm) is a polar point definition.

In Ruby, this is not so easy. At the moment, I haven’t thought of a way to do it with four anonymous points, as done above.

points :A => p(1,0), :B => p(2,0)

a1 = circle(:radius => :OA, :arc => [0,60])

a2 = circle(:radius => :OB, :arc => [0,60])

segment(a1.start, a2.start)

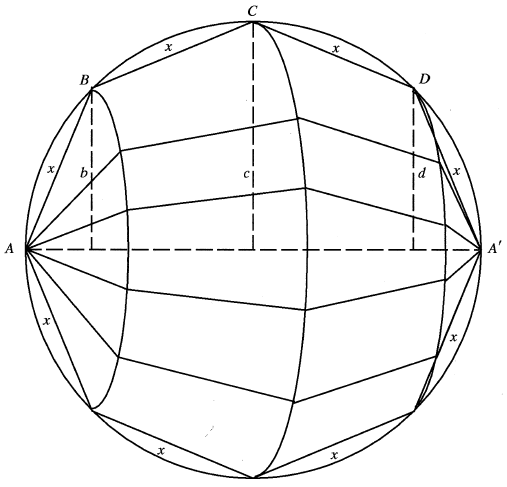
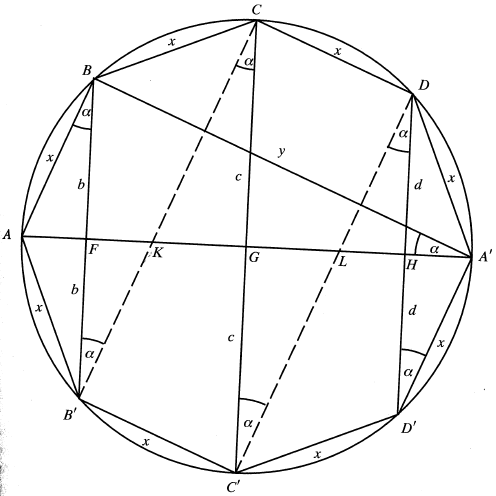
segment(a1.end, a2.end)

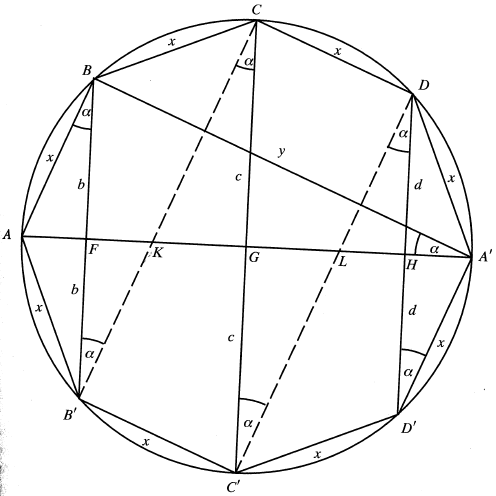
# Sample Code: Archimedes diagrams for spherical surface area

These images are scanned from William Dunham’s excellent book *The Mathematical Universe*, chapter S (for “Spherical Surface”).

|  |  |
| --- | --- |
|  | triangle(:EFX, :right => E, :base => 1,  :height => 4)  \_segment(:XE).extend(:D, 1.3).mark(:D)  \_segment(:XF).extend(:C, 1.3).mark(:C)  \_segment(:EF).mark(:E, :F)  triangle(:DCX)  \_segment(:EF).label(:r, :above)  \_segment(:DC).label(:R, :above)  \_segment(:XF).label(:t, :right)  \_segment(:FC).label(:s, :right)  mark\_right\_angle(:XDC, :XEF, 0.15)  polygon(:DCFE).shade(:lightgrey) |

The following images appear over the next two pages.





point :G => p(0,0), :A => p(-1,0), “A’” => p(1,0)

circle(:diameter => “AA’”)

polygon(“A’DCBAB’C’D’”, :radius => “GA’”) do |poly|

poly.mark\_each\_side(:x)

poly.mark\_vertices

end

segment(“BB’CC’DD’”)

segments(“B’C”, “C’D”).dashed

point :F => intersection(“AA’”, “BB’”), :G => intersection(“AA’”, “CC’”),

:H => intersection(“AA’”, “DD’”), :K => intersection(“AA’”,“B’C”),

:L => intersection(“AA’”, “C’D”)

points(:F, :G, :H, :K, :L).mark(:SW)

point :K => intersection( segment

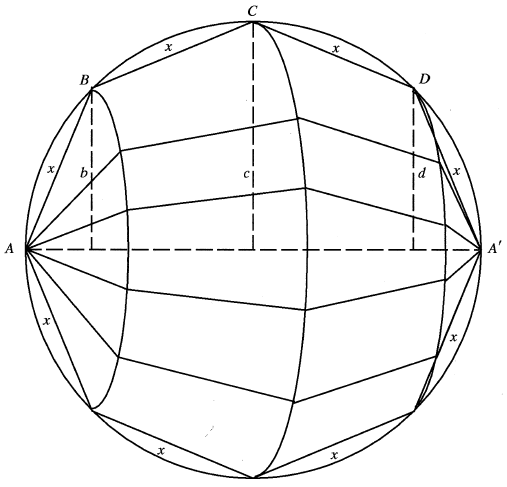
segment(“A’B”).mark(:y, :above)

\_segments(“FB”, “FB’”).mark(:b, :left, 0.3)

\_segments(“FC”, “FC’”).mark(:c, :left, [0.3, 0.45])

\_segments(“HD”, “HD’”).mark(:d, :right, [0.5, 0.4])

angles(%w[ABB’ BB’C B’CD CC’D C’DD’ DD’A’ AA’B]).mark(:alpha)



# Asymptote

Asymptote is a highly accomplished graphics/diagram tool for LaTeX. It doesn’t have as much high-level geometrical awareness as Eukleides (well, not that I’ve seen yet), but it has some excellent lower-level primitives, like transforms, vectors, etc. The following examples capture some of the interesting ideas.

|  |  |  |
| --- | --- | --- |
|  |  | * The bisection being the average of the direction of the two lines, and being able to act as a vector (drawing the dotted line) and a point (drawing the dot). * Placing the label A on the path of the dotted line. |

|  |  |
| --- | --- |
|  | These arcs are drawn separately, and filled as they are drawn. Colour is determined via the counter. Some of the code is chopped off; the colour is:  i/n\*blue + (1-i/n)\*green + gray |
|  | |

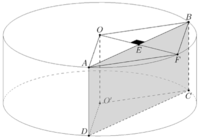
|  |  |
| --- | --- |
|  |  |
| Nice demonstration of shifting and scaling a unit circle to get what you want, instead of creating custom circles. | |

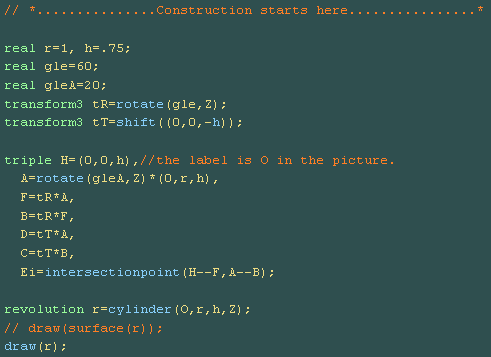
|  |  |
| --- | --- |
|  | Really nice way to specify the length of the segment relative to *A* and *B*. |
|  |

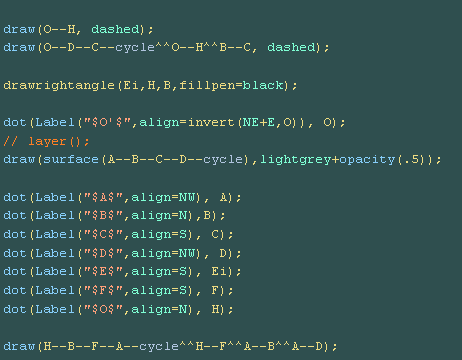
|  |  |
| --- | --- |
|  |  |
| Points along a path. Very easy definition of the path, as well.  fig1370 is similar, but uses the *direction* of the path to draw a bunch of arrows as well. Very neat. | |

|  |  |
| --- | --- |
|  | Drawing the dotted horizontal and vertical lines from *M* to the axes.  Another nice touch is the way of ensuring the label *M* is “away from” the *O*. |
|  | |

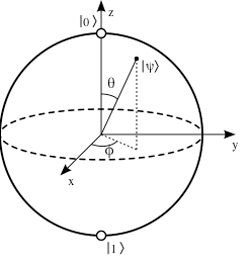
|  |  |
| --- | --- |
| Extension of one segment onto another.  Nice way of defining the points as well. |  |

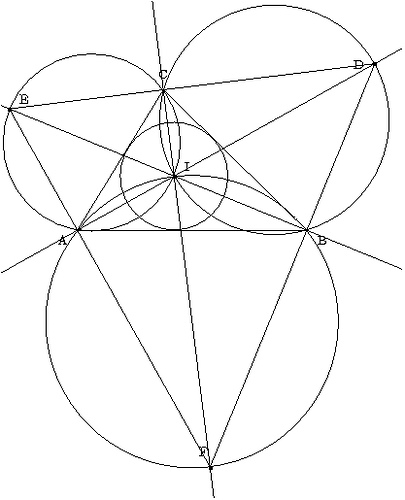
  



# Further images and Eukleides code to turn into examples one day





frame(-2,-7,8.5,6)

A B C triangle

draw(A,B,C)

draw("A",A, 213:)

draw("B", B, -33:)

draw("C", C, 90:)

inc = incircle(A,B,C)

draw(inc)

I = center(inc)

draw(I)

draw("I", I, 33:)

a\_i = line(A,I)

b\_i = line(B,I)

c\_i = line(C,I)

draw(a\_i)

draw(b\_i)

draw(c\_i)

o\_bic = circle(B,I,C)

o\_aic = circle(A,I,C)

o\_aib = circle(A,I,B)

draw(o\_bic)

draw(o\_aic)

draw(o\_aib)

NU D intersection(a\_i,o\_bic)

NU E intersection(b\_i,o\_aic)

NU F intersection(c\_i,o\_aib)

draw("D",D,183:)

draw(D)

draw("E",E,33:)

draw(E)

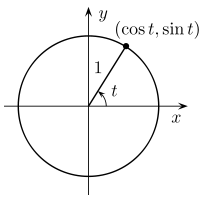
draw("F",F,113:)

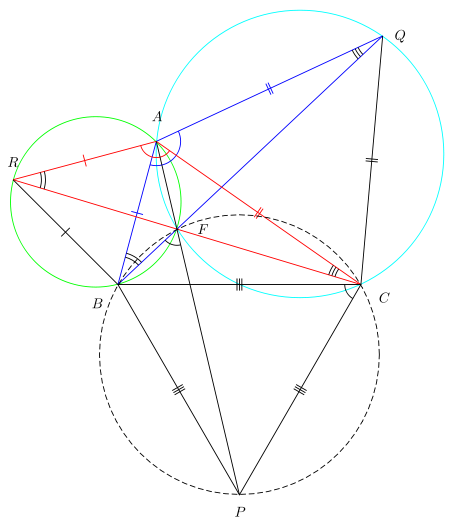
draw(F)

draw(segment(D,E))

draw(segment(E,F))

draw(segment(D,F))





% http://commons.wikimedia.org/wiki/File:Fermat\_Points\_Proof.png

frame(-2.8, -6, 8.2, 7.5)

B C A triangle(6, 75:, 35:)

C B P equilateral

A C Q equilateral

B A R equilateral

s = segment(A, P)

t = segment(B, Q)

u = segment(C, R)

c = circle(B, A, R)

d = circle(A, C, Q)

e = circle(C, B, P)

F = intersection(line(A, P), line(B, Q))

color(green)

draw(c)

color(cyan)

draw(d)

color(blue)

draw(t)

draw(segment(A, Q))

draw(segment(A, B))

mark(segment(A, B), simple)

mark(segment(A, Q), double)

mark(B, A, Q, simple, 1.2)

color(red)

draw(u)

draw(segment(A, R))

draw(segment(A, C))

mark(segment(A, R), simple)

mark(segment(A, C), double)

mark(R, A, C, simple, 0.8)

color(black)

draw(s)

draw(segment(B, R))

draw(segment(C, Q))

draw(C, B, P)

draw(e, dashed)

draw(F, dot, 1.3)

mark(segment(B, R), simple)

mark(segment(C, Q), double)

mark(segment(B, C), triple)

mark(segment(B, P), triple)

mark(segment(C, P), triple)

mark(C, R, A, double, 1.5)

mark(Q, B, A, double, 1.5)

mark(A, Q, B, triple, 1.5)

mark(A, C, R, triple, 1.5)

mark(B, F, P, dot, 0.8)

mark(B, C, P, dot, 0.8)

label(F, 0.5, 0:)

label(A, 0.5, 90:)

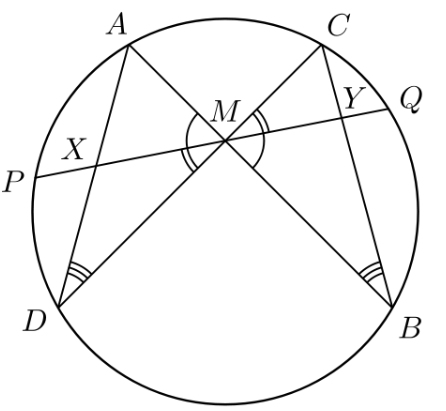
label(B, 0.5, 225:)

label(C, 0.5, 330:)

label(P, 0.3, 270:)

label(Q, 0.3, 0:)

label(R, 0.3, 90:)



% Butterfly proof

box(-1.2,-1.1,1.2,1.1,2.2)

thickness(1.2)

E = circle(point(0,0),1)

A = point(E,120:)

C = point(E,60:)

P = point(E,170:)

D = point(E,210:)

B = point(E,330:)

M = intersection(line(C,D),line(A,B))

G = line(M, P)

Q P intersection(G, E)

X = intersection(line(A,D),line(P,Q))

Y = intersection(line(C,B),line(P,Q))

draw(P,Q)

draw(E)

draw(A,D,C,B,A)

draw("$A$",A,.05,120:)

draw("$C$",C,.05,50:)

draw("$P$",P,.05,190:)

draw("$Q$",Q,.05,20:)

draw("$D$",D,.05,210:)

draw("$B$",B,.05,310:)

draw("$M$",M,.1,90:)

draw("$X$",X,.05,135:)

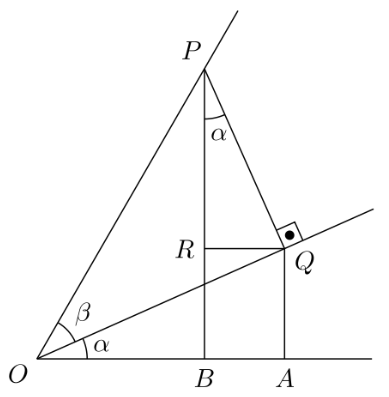
draw("$Y$",Y,.05,60:)

mark(B,M,C,simple,.4); mark(Q,M,C,simple,.46)

mark(A,M,D,simple,.4); mark(P,M,D,simple,.46)

mark(C,D,A,simple,.5); mark(C,D,A,simple,.44); mark(C,D,A,simple,.38)

mark(C,B,A,simple,.5); mark(C,B,A,simple,.44); mark(C,B,A,simple,.38)



% Trigonometry sum formula

box(-0.15, -0.15, 1.1, 1.1, 4.5);

O = point(0, 0);

x\_axis = line(O, 0<);

l\_alpha = line(O, 24:);

l\_beta = line(O, 60:);

P = point(l\_beta, 1);

PQ = perpendicular(l\_alpha, P);

Q = intersection(l\_alpha, PQ);

QA = perpendicular(x\_axis, Q);

A = intersection(x\_axis, QA);

PB = perpendicular(x\_axis, P);

B = intersection(x\_axis, PB);

QR = parallel(x\_axis, Q);

R = intersection(PB, QR);

mark(R,P,Q, simple, 0.3); % alpha

mark(B,O,Q, simple, 0.3); % alpha

mark(Q,O,P, simple, 0.25); % beta

mark(point(l\_alpha, 2),Q,P, dotted, 0.2);

draw("$O$", O, .03, 220:);

draw("$P$", P, .03, 124:);

draw("$Q$", Q, .03, -34:);

draw("$R$", R, .03, 180:);

draw("$A$", A, .03, -90:);

draw("$B$", B, .03, -90:);

draw("$\alpha$", P, .18, -78:);

draw("$\alpha$", O, .17, 12:);

draw("$\beta$", O, .15, 42:);

draw(segment(P,Q));

draw(segment(Q,R));

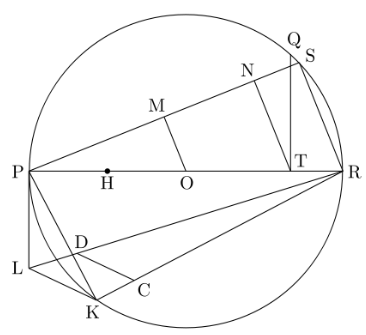
draw(segment(P,B));

draw(segment(Q,A));

draw(segment(O, point(x\_axis, 1.0)));

draw(segment(O, point(l\_alpha, 1.1)));

draw(segment(O, point(l\_beta, 1.2)));



% Approximately squaring the circle – Ramanujan’s attempt to

% construct sqrt(pi).

box(-1, 6, 7, -2)

P = point(0, 2)

R = point(6, 2)

PR = segment(P, R)

draw(PR)

c1 = circle(P, R)

O = center(c1)

draw(c1)

H = barycenter(P, O)

draw(H)

T = barycenter(O, 1, R, 2)

Q0 Q intersection(perpendicular(PR, T), c1)

TQ = segment(T, Q)

draw(TQ)

c2 = circle(R, length(TQ))

S S0 intersection(c1, c2)

RS = segment(R, S)

draw(RS)

PS = segment(P, S)

draw(PS)

N = intersection(parallel(RS, T), line(P, S))

TN = segment(T, N)

draw(TN)

M = intersection(parallel(RS, O), line(P, S))

OM = segment(O, M)

draw(OM)

PM = segment(P, M)

c3 = circle(P, length(PM))

K K0 intersection(c1, c3)

PK = segment(P, K)

draw(PK)

MN = segment(M, N)

c4 = circle(P, length(MN))

L L0 intersection(perpendicular(PR, P), c4)

PL = segment(P, L)

draw(PL)

RL = segment(R, L)

draw(RL)

RK = segment(R, K)

draw(RK)

KL = segment(K, L)

draw(KL)

RH = segment(R, H)

c5 = circle(R, length(RH))

C0 C intersection(line(R, K), c5)

D = intersection(parallel(KL, C), line(R, L))

CD = segment(C, D)

draw(CD)

draw("C", C, 0.1, 315:)

draw("D", D, 0.1, 70:)

draw("H", H, 0.1, 270:)

draw("K", K, 0.1, 250:)

draw("L", L, 0.1, 180:)

draw("M", M, 0.1, 120:)

draw("N", N, 0.1, 120:)

draw("O", O, 0.1, 270:)

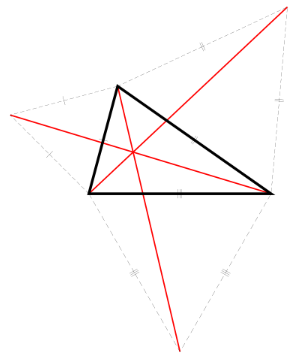
draw("P", P, 0.1, 180:)

draw("Q", Q, 0.1, 75:)

draw("R", R, 0.1, 0:)

draw("S", S, 0.1, 30:)

draw("T", T, 0.1, 45:)



frame(-3.5, -6, 7.5, 7)

B C A triangle(6, 75:, 35:)

C B P equilateral

A C Q equilateral

B A R equilateral

s = segment(A, P)

t = segment(B, Q)

u = segment(C, R)

F = intersection(line(A, P), line(B, Q))

color(red)

thickness(2)

draw(s)

draw(t)

draw(u)

draw(F)

thickness(0.5)

color(black)

thickness(4)

draw(B, C, A)

thickness(0.25)

draw(B, C, P, dashed)

draw(A, C, Q, dashed)

draw(B, A, R, dashed)

mark(segment(A, B), simple)

mark(segment(A, R), simple)

mark(segment(B, R), simple)

mark(segment(A, C), double)

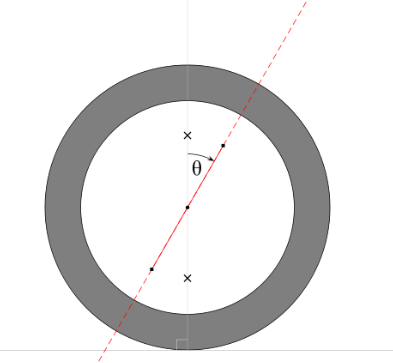
mark(segment(A, Q), double)

mark(segment(C, Q), double)

mark(segment(B, C), triple)

mark(segment(B, P), triple)

mark(segment(C, P), triple)



http://commons.wikimedia.org/wiki/File:Caster\_angle.svg

box(-2, -2, 10, 10)

cc = point(4,4)

Ci = circle(cc, 3)

Co = circle(cc, 4)

base = line(point(0,0), point(2,0))

vert = segment(point(4, 0), point(4,40))

ang = line(cc, 60:)

draw(Co)

draw(Ci)

mark(point(Ci, 60:), cc, point(4,6), backarrow, 3)

color(lightgray)

draw(base)

draw(vert)

mark(point(0,0), point(4,0), cc, right)

color(red)

draw(ang, dashed)

draw(segment(point(ang, -2), point(ang, 2)))

color(black)

draw(cc)

draw(point(ang, 2), box)

draw(point(ang, -2), box)

draw(point(vert, 2), cross)

draw(point(vert, 6), cross)

From NCM 10 5.2/5.3 Teachers Resouces:

