

## Documentation for Enkidu v3

### Contents

1	Introduction	1
2	Typical structure of an Enkidu program	1
3	Running an Enkidu program	3
4	Examples	4
4.1	Vector arithmetic	4
4.2	More vector arithmetic: orthogonal projection	5
4.3	Intersections: inversion as a composition of projections	6
4.4	More intersections: line meets parabola	7

### Example

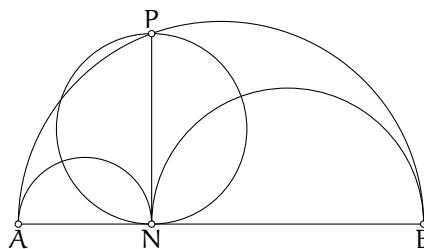
```
from __future__ import division
from enkidu import *

circ = circle(vec.zero, 1)
a = circ.pt(180)
b = circ.pt(0)
p = circ.pt(110)
n = foot(p, join(a, b))

lft, bot, rt, top = bbox(a, b, circ.pt(90))
margin = 3
width = 158 - 2*margin
height = (top - bot)*width/(rt - lft)
f = figure(width, height, margin, lft, bot, rt, top)

for pt in [a,b,n,p]:
    f.clipdot(pt)
f.polyline(a, b)
f.polyline(p, n)
f.circle(circ, 0, 180)
f.circle(circle.ondiam(a, n), 0, 180)
f.circle(circle.ondiam(n, b), 0, 180)
f.circle(circle.ondiam(p, n), 0, 360)
f.label('$A$', 't', a)
f.label('$B$', 't', b)
f.label('$N$', 't', n)
f.label('$P$', 'b', p)

if __name__ == '__main__':
    main(f)
```



## 1 Introduction

Enkidu is software for generating geometric diagrams in  $\text{\LaTeX}$  documents. To make a diagram using Enkidu, you write a Python program describing the diagram; running this program creates PostScript and  $\text{\LaTeX}$  files which together produce the diagram in your document.

The program on the title page, for example, produces the diagram shown there, which is the figure for Proposition 4 of Archimedes' *Book of Lemmas*, on the area of an arbelos.<sup>1</sup>

To use Enkidu at all, you need basic knowledge of Python. To label objects in your figure with mathematical notations, you need to know some  $\text{\LaTeX}$ . To produce effects that I haven't already arranged for, you will probably need to know some PostScript.

## 2 Typical structure of an Enkidu program

The example on the title page illustrates the typical structure of an Enkidu program, which consists of three main parts: geometric computations, creating a figure object, and drawing the figure.

The first section computes the geometric objects needed to draw the figure.

```
circ = circle(vec.zero, 1)
a = circ.pt(180)
b = circ.pt(0)
p = circ.pt(110)
n = foot(p, join(a, b))
```

The circle `circ` is centred at the origin and has radius 1. The points `a`, `b`, and `p` lie on this circle at angles  $180^\circ$ ,  $0^\circ$ , and  $110^\circ$  respectively. (Angles in Enkidu are always denoted in degrees;  $0^\circ$  is east and  $90^\circ$  is north.) The point `n` is computed geometrically: the expression `join(a, b)` computes the line through the points `a` and `b`, and `n` is computed as the foot of the perpendicular from the point `p` to that line.

Having computed all points of interest, the program creates the figure object `f`.

```
lft, bot, rt, top = bbox(a, b, circ.pt(90))
margin = 3
width = 158 - 2*margin
height = (top - bot)*width/(rt - lft)
f = figure(width, height, margin, lft, bot, rt, top)
```

There are two coordinate systems at work in an Enkidu diagram: the “drawing” coordinate system and the “figure” coordinate system. In the drawing coordinate system, the bottom left corner of the figure is the origin, and  $x$ - and  $y$ -coordinates are measured in PostScript points (which  $\text{\TeX}$  calls “big points”). The figure coordinate system may have its origin anywhere, and  $x$ - and  $y$ -units

---

<sup>1</sup>*The Works of Archimedes*, ed. T. L. Heath (Cambridge UP, 1897; Dover, 2002), 305.

of any size; this coordinate system is used for points in the figure, and so can be chosen for convenience in that purpose, without regard to the actual size of the figure on the page.

When a figure object is constructed, the user specifies seven values. The first three — `width`, `height`, and `margin` — specify the size of the figure on the page: `width + 2*margin` points wide and `height + 2*margin` points tall. The next four values specify the coordinates, in the figure coordinate system, of the edges of the drawing region, that is, the rectangle which is `width` points wide and `height` points tall. (Note that this region excludes the margin.)

The example program computes these seven values by a method appropriate when the figure's desired total width (in the example, 158 points) is known, and all other values should be accommodated to that value. The range of coordinates needed to represent the points of interest is computed with the `bbox` (bounding box) function; `width` is computed from the desired total width and the margin size; and `height` is computed in proportion to `width`.

The third and last section of the program draws the figure:

```
for pt in [a,b,n,p]:
    f.clipdot(pt)
f.polyline(a, b)
f.polyline(p, n)
f.circle(circ, 0, 180)
f.circle(circle.ondiam(a, n), 0, 180)
f.circle(circle.ondiam(n, b), 0, 180)
f.circle(circle.ondiam(p, n), 0, 360)
f.label('$A$', 't', a)
f.label('$B$', 't', b)
f.label('$N$', 't', n)
f.label('$P$', 'b', p)
```

This section is largely self-explanatory. Note that geometric computations may well take place here; for example, the semicircle on AN has no relevance to any prior computations, so it is created (by the expression `circle.ondiam(a, n)`) only when drawn.

### 3 Running an Enkidu program

An Enkidu program `foo.py` should end with these lines:

```
if __name__ == '__main__':  
    main(f)
```

Then a command such as

```
python foo.py foo-fig
```

will produce files `foo-fig.eps` and `foo-fig.tex`.

The `.tex` file includes a `\LaTeX` `picture` environment which superimposes any labels produced by `f.label()` onto the figure; the line drawing itself is in the `.eps` file, and is included by the `\includegraphics*` command (from the `graphics` package, which the including document must load).

The `.tex` file can be `\input` directly into your document.

PDF $\LaTeX$  wants graphics files in PDF format instead of EPS. Standard tools will convert `foo-fig.eps` to `foo-fig.pdf`; but the `\includegraphics*` command in `foo-fig.tex` will then refer to the wrong file. In this situation, invoke your Enkidu program with

```
python foo.py foo-fig pdf
```

The last argument specifies a “fake suffix”: with this command, `foo-fig.eps` is created as usual, but the `\includegraphics*` command in `foo-fig.tex` includes the file `foo-fig.pdf`. (It’s up to you to convert the EPS file into a PDF before running PDF $\LaTeX$  on your document.)

The Enkidu source distribution includes a complete example of how to use Enkidu, to wit, the source for this document and all its figures, as well as a `Makefile` for generating this PDF file from that source.

## 4 Examples

The following examples demonstrate the basic facilities of Enkidu. The detailed documentation for all Enkidu classes and functions is in the source code; read it there or by way of the Python online help system.

The examples omit boilerplate and the second section (the creation of the figure object) in each program; consult the source distribution for the complete code.

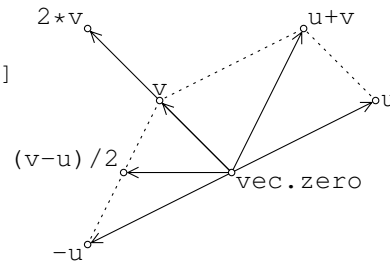
The text accompanying the examples doesn't discuss everything of interest in every program; read the code.

### 4.1 Vector arithmetic

Vectors and points (Enkidu doesn't distinguish) are represented by instances of the `vec` class, which support the expected arithmetic operators.

```
u = vec(2, 1)
v = vec(-1, 1)
vecs = [u, v, u+v, -u, (v-u)/2, 2*v]

for pt in vecs + [vec.zero]:
    f.clipdot(pt)
for pt in vecs:
    f.arrowtodot(vec.zero, pt)
f.dashed()
f.polyline(-u, v, u+v, u)
f.undashed()
f.label('\texttt{vec.zero}', 'tl', vec.zero)
f.label('\texttt{u}', 'l', u)
f.label('\texttt{v}', 'b', v)
f.label('\texttt{2*v}', 'br', 2*v)
f.label('\texttt{u+v}', 'bl', u+v)
f.label('\texttt{-u}', 'tr', -u)
f.label('\texttt{(v-u)/2}', 'br', (v-u)/2)
```



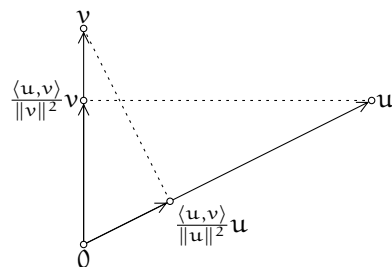
This example uses a single forward slash to divide a `vec` by an `int` (in the expression `(v-u)/2`). This practice requires that `__future__.division` be in effect; otherwise `/2` is interpreted as integer division, which `vecs` do not support.

## 4.2 More vector arithmetic: orthogonal projection

`vecs` support many other operations, including, as shown below, dot product and norm; see the online documentation for a complete list.

```
u = vec(2, 1)
v = vec(0, 1.5)
projuv = v*dot(u, v)/v.normsq
projvu = u*dot(u, v)/u.normsq
vecs = [u, v, projuv, projvu]

for pt in vecs + [vec.zero]:
    f.clipdot(pt)
for pt in vecs:
    f.arrowtodot(vec.zero, pt)
f.dashed()
f.polyline(u, projuv)
f.polyline(v, projvu)
f.undashed()
f.label('$0$', 't', vec.zero)
f.label('$u$', 'l', u)
f.label('$v$', 'b', v)
f.label(r'$\frac{\langle u,v \rangle}{\|v\|^2}v$', 'r', projuv)
f.label(r'$\frac{\langle u,v \rangle}{\|u\|^2}u$', 'tl', projvu)
```



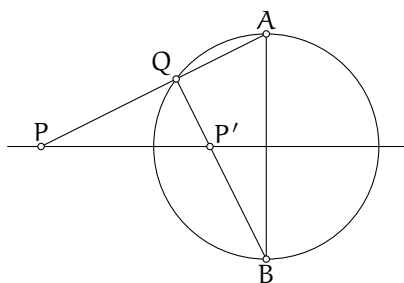
As the last two lines illustrate, when many  $\text{\LaTeX}$  macros are needed in a label, it is convenient to use the Python “raw string” syntax `r' . . . '` to suppress the usual interpretation of backslashes.

### 4.3 Intersections: inversion as a composition of projections

This figure shows an interpretation of inversion as a composition of two stereographic projections. The horizontal line is a plane, viewed edge-on, bisecting a sphere at its equator; points A and B are the north and south poles of the sphere. Let P be a point on the plane. Project P onto the sphere through the north pole; that is, let Q be the (other) intersection of AP with the sphere. Then project Q back onto the plane through the south pole; that is, let P' be the intersection of BQ with the plane. Then, it turns out, P' is the inverse of P with respect to the equator of the sphere.

```
circ = circle(vec.zero, 1)
a = circ.pt(90)
b = circ.pt(-90)
p = circ.o - 2*vec(circ.r, 0)
q = withmax(distfrom(a), intersect(circ, join(p, a)))
p2 = meet(join(p, circ.o), join(q, b))

for pt in [a, b, p, q, p2]:
    f.clipdot(pt)
f.circle(circ)
f.polyline(p, a, b, q)
f.line(join(p, circ.o))
f.label('$A$', 'b', a)
f.label('$B$', 't', b)
f.label('$P$', 'b', p)
f.label('$Q$', 'br', q)
f.label("$P'$", 'bl', p2)
```



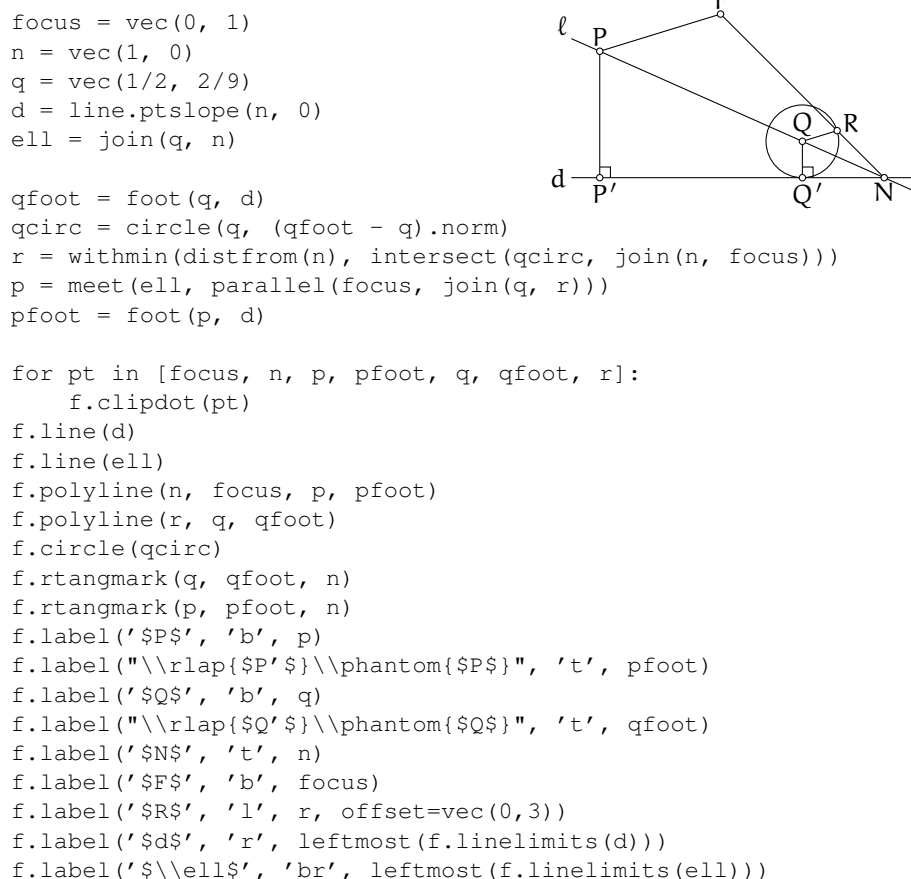
The point p2 (P' in the figure) is the intersection of BQ with the horizontal line; it is obtained with the `meet` function, which operates on lines and always returns a point (unless the lines are parallel, in which case it raises an exception).

The point q is the intersection of AP with the circle; it is obtained with the `intersect` function, which operates on lines and/or circles, and returns a list of points of intersection. The line AP intersects the circle twice, at A and at Q, so `intersect` returns a list containing those two points; the code must determine which one is Q. The obvious way is to scan the list for an element pt such that `pt != a`; but since floating point arithmetic is inexact, it is not certain that the point in the list representing the intersection at A will actually be equal to a in the sense of `==`. The example takes a more reliable course: q is chosen to be the point of intersection which is furthest from a.

#### 4.4 More intersections: line meets parabola

This figure shows a construction for finding an intersection  $P$  of the line  $\ell$  with the parabola with focus  $F$  and directrix  $d$ , or rather, a construction for the case where  $\ell$  and  $d$  meet.

Let  $\ell$  and  $d$  meet at  $N$ . Pick any point  $Q$  on  $\ell$  other than  $N$ . Draw a circle centred at  $Q$  and tangent to  $d$  at  $Q'$ . Join  $NF$ , meeting the circle at  $R$ . Draw  $PF$  parallel to  $QR$ , meeting  $\ell$  at  $P$ . Let  $P'$  be the foot of the perpendicular from  $P$  to  $d$ . By parallels,  $PF : PP' = QR : QQ' = 1$ , and so  $P$  lies on the parabola as well as on  $\ell$ .



The line  $d$  is created by point and slope; the line  $\ell$  is created by two points; in the computation of  $p$ , a line is created by a point and a parallel line. Other options exist; see the online documentation.

Note that, in the program, the lines  $\ell$  and  $d$  are constructed from  $n$  and  $q$ , reversing the procedure in the construction.

The `\rlap\phantom` dance centres the labels  $P'$  and  $Q'$  nicely on the letter; otherwise the `'` exerts an undue influence on the positioning.