Asymptote: The Vector Graphics Language

Andy Hammerlindl and John Bowman

University of Toronto and University of Alberta

August 16, 2007

`http://asymptote.sf.net`

History

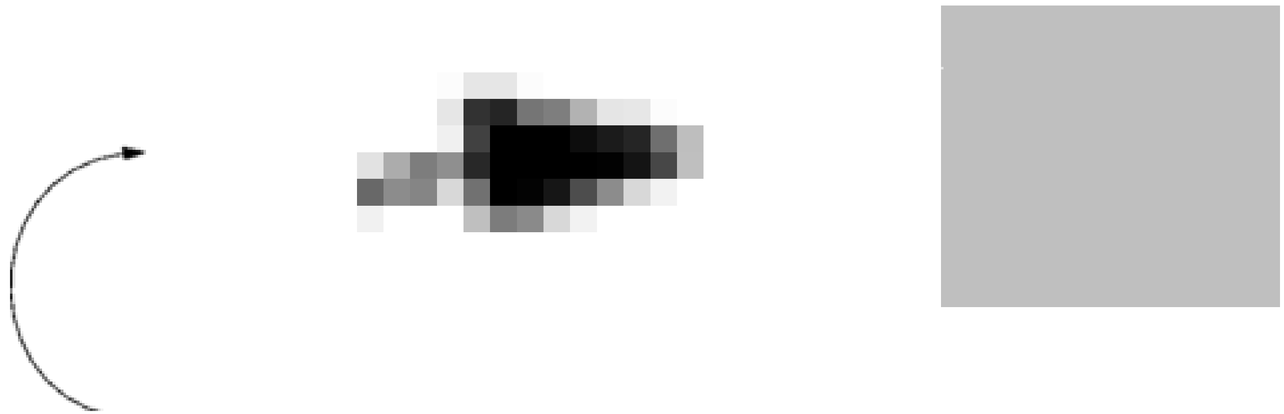
- $\TeX$ and METAFONT (Knuth, 1979)
- MetaPost (Hobby, 1989): 2D Bezier Control Point Selection
- Asymptote (Hammerlindl, Bowman, Prince, 2004): 2D & 3D Graphics

Statistics (as of April, 2007)

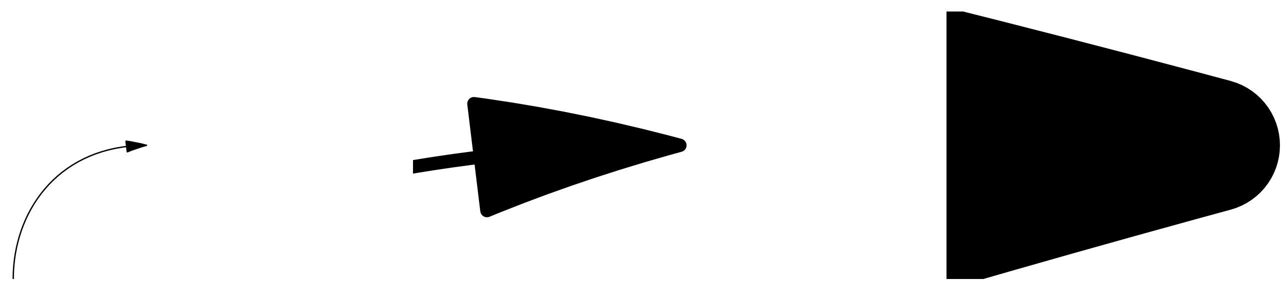
- Runs on Windows, Mac OS X, Linux, etc.
- 1800 downloads a month from `asymptote.sourceforge.net`.
- 33 000 lines of C++ code.
- 18 000 lines of Asymptote code.

Vector Graphics

- Raster graphics assign colors to a grid of pixels.

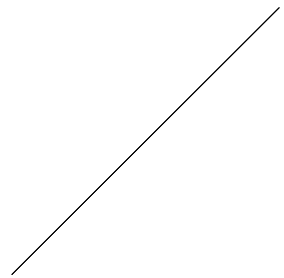


- Vector graphics are graphics which still maintain their look when inspected at arbitrarily small scales.



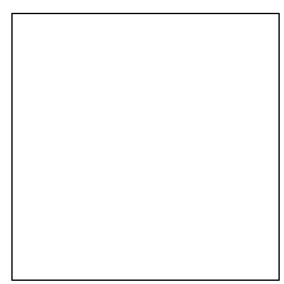
Cartesian Coordinates

```
draw((0,0)--(100,100));
```



- units are `PostScript` *big points* (1 `bp` = 1/72 `inch`)
- `--` means join the points with a linear segment to create a *path*
- cyclic path:

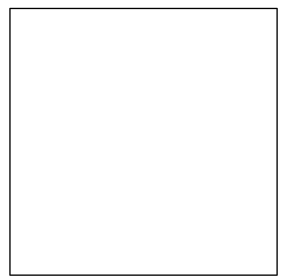
```
draw((0,0)--(100,0)--(100,100)--(0,100)--cycle);
```



Scaling to a Given Size

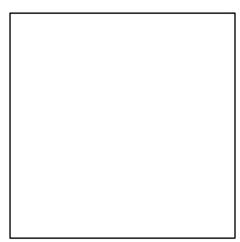
- `PostScript` units are often inconvenient.
- Instead, scale user coordinates to a specified final size:

```
size(101,101);
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle);
```



- One can also specify the size in `cm`:

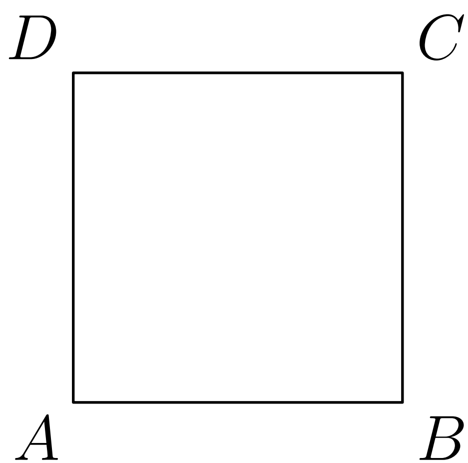
```
size(0,3cm);
draw(unitsquare);
```



Labels

- Adding and aligning LaTeX labels is easy:

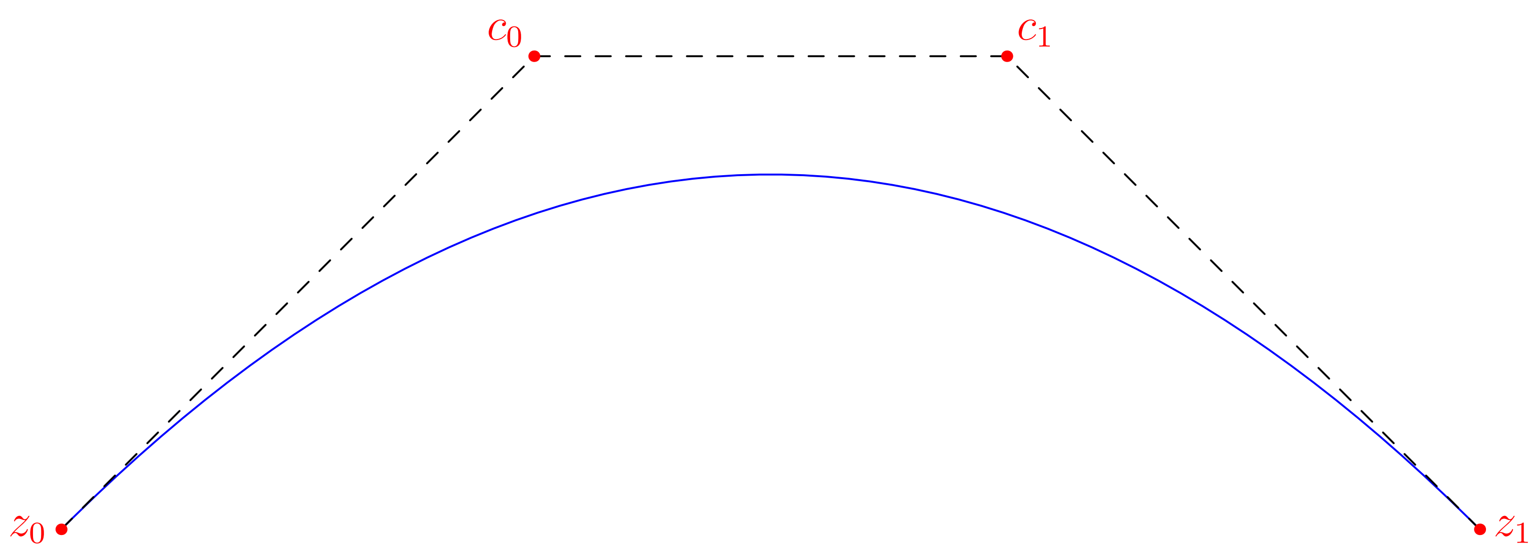
```
size(0,3cm);
draw(unitsquare);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
```



2D Bezier Splines

- Using `..` instead of `--` specifies a *Bezier cubic spline*:

```
draw(z0 .. controls c0 and c1 .. z1,blue);
```

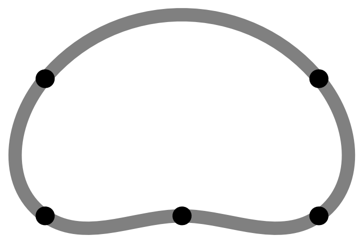


$$(1-t)^3 z_0 + 3t(1-t)^2 c_0 + 3t^2(1-t)c_1 + t^3 z_1, \qquad t \in [0,1].$$

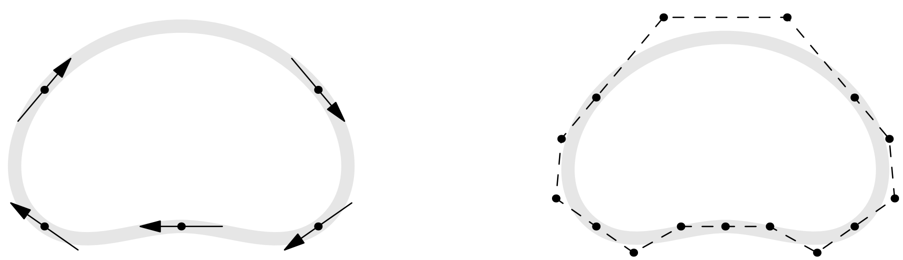
Smooth Paths

- Asymptote can choose control points for you, using the algorithms of Hobby and Knuth [**?**, **?**]:

```
pair[] z={(0,0), (0,1), (2,1), (2,0), (1,0)};

draw(z[0]..z[1]..z[2]..z[3]..z[4]..cycle,
     grey+linewidth(5));
dot(z,linewidth(7));
```



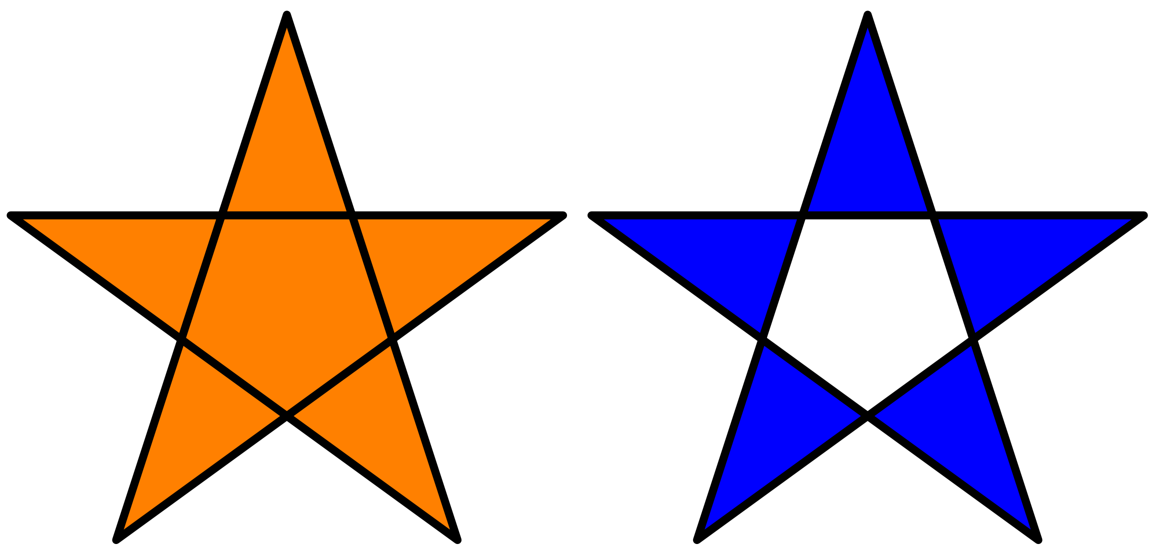
- First, linear equations involving the curvature are solved to find the direction through each knot. Then, control points along those directions are chosen:



Filling

- Use `fill` to fill the inside of a path:

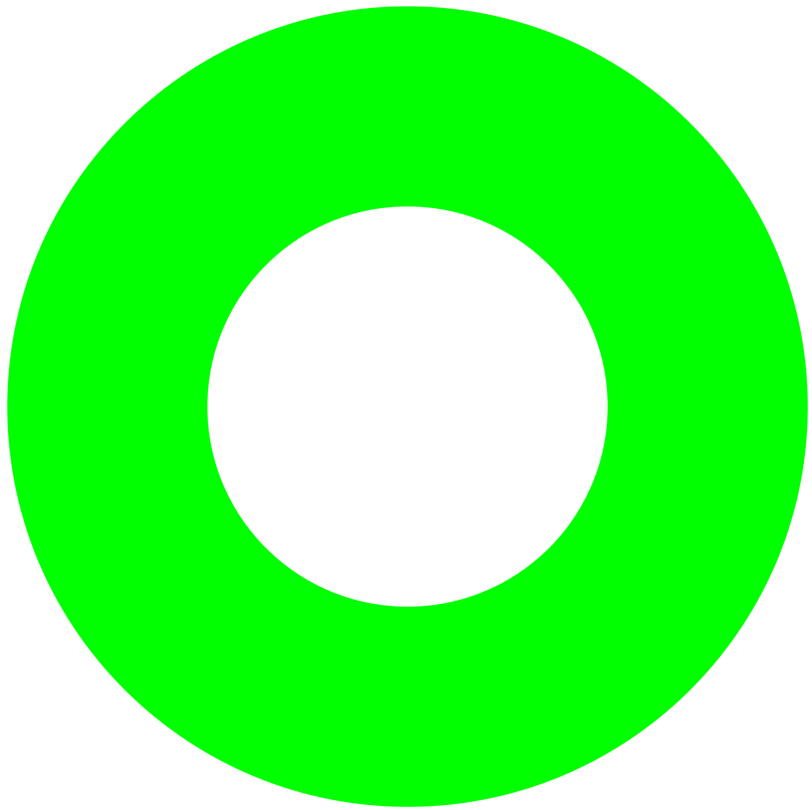
```
path star;
for (int i=0; i<5; ++i)
  star=star--dir(90+144i);
star=star--cycle;
fill(shift(-1,0)*star,orange+zerowinding);
draw(shift(-1,0)*star,linewidth(3));
fill(shift(1,0)*star,blue+evenodd);
draw(shift(1,0)*star,linewidth(3));
```



Filling

- Use a list of paths to fill a region with holes:

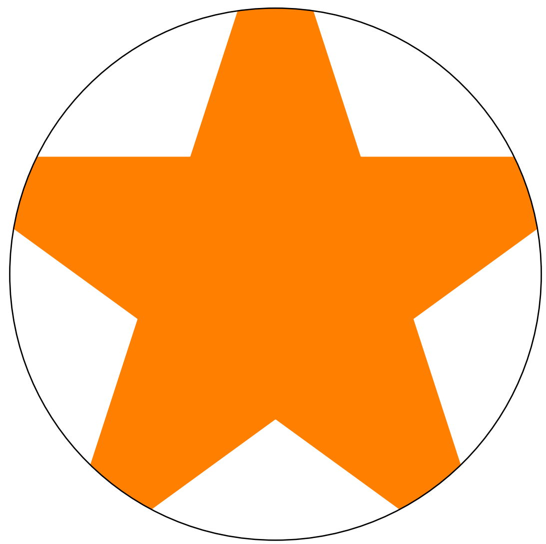
```
path[] p={scale(2)*unitcircle, reverse(unitcircle)};
fill(p,green+zerowinding);
```



Clipping

- Pictures can be clipped to lie inside a path:

```
fill(star,orange+zerowinding);
clip(scale(0.7)*unitcircle);
draw(scale(0.7)*unitcircle);
```



- All of Asymptote's graphical capabilities are based on four primitive commands: `draw`, `fill`, `clip`, and `label`.

Affine Transforms

- Affine transformations: shifts, rotations, reflections, and scalings.

```
transform t=rotate(90);
write(t*(1,0));  // Writes (0,1).
```

- Pairs, paths, pens, strings, and whole pictures can be transformed.

```
fill(P,blue);
fill(shift(2,0)*reflect((0,0),(0,1))*P, red);
fill(shift(4,0)*rotate(30)*P, yellow);
fill(shift(6,0)*yscale(0.7)*xscale(2)*P, green);
```

C++/Java-like Programming Syntax

```
// Declaration: Declare x to be real:
real x;

// Assignment: Assign x the value 1.
x=1.0;

// Conditional: Test if x equals 1 or not.
if(x == 1.0) {
  write("x equals 1.0");
} else {
  write("x is not equal to 1.0");
}

// Loop: iterate 10 times
for(int i=0; i < 10; ++i) {
  write(i);
}
```

Helpful Math Notation

- Integer division returns a **`real`**. Use **`quotient`** for an integer result:

```
3/4==0.75          quotient(3,4)==0
```

- Caret for real and integer exponentiation:

```
2^3    2.7^3    2.7^3.2
```

- Many expressions can be implicitly scaled by a numeric constant:

```
2pi    10cm    2x^2    3sin(x)    2(a+b)
```

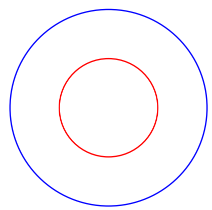
- Pairs are complex numbers:

```
(0,1)*(0,1)==(-1,0)
```

Function Calls

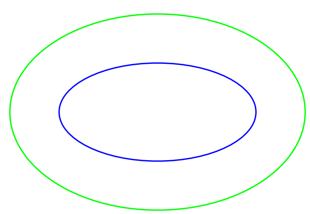
- Functions can take default arguments in any position. Arguments are matched to the first possible location:

```
void drawEllipse(real xsize=1, real ysize=xsize, pen p=blue) {
  draw(xscale(xsize)*yscale(ysize)*unitcircle, p);
}

drawEllipse(2);
drawEllipse(red);
```



- Arguments can be given by name:

```
drawEllipse(xsize=2, ysize=1);
drawEllipse(ysize=2, xsize=3, green);
```



Rest Arguments

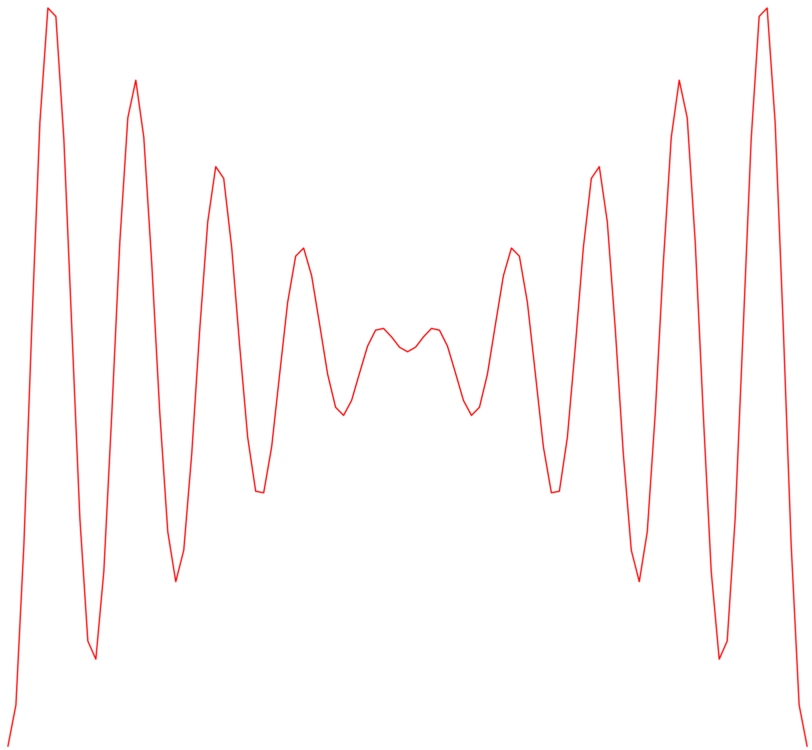
- Rest arguments allow one to write a function that takes an arbitrary number of arguments:

```
int sum(... int[] nums) {
  int total=0;
  for (int i=0; i < nums.length; ++i)
    total += nums[i];
  return total;
}

sum(1,2,3,4);                         // returns 10
sum();                                // returns 0
sum(1,2,3 ... new int[] {4,5,6});     // returns 21

int subtract(int start ... int[] subs) {
  return start - sum(... subs);
}
```

Higher-Order Functions

- Functions are first-class values. They can be passed to other functions:

```
real f(real x) {
    return x*sin(10x);
}
draw(graph(f,-3,3),red);
```

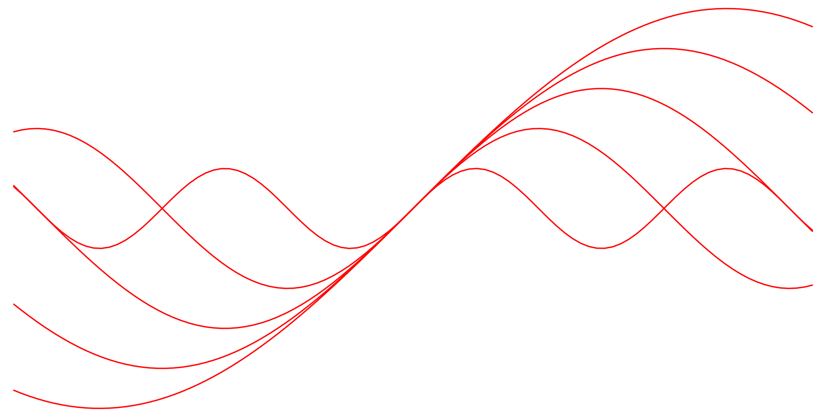


Higher-Order Functions

- Functions can return functions:

$$f_n(x) = n \sin\left(\frac{x}{n}\right).$$

```
typedef real func(real);
func f(int n) {
  real fn(real x) {
    return n*sin(x/n);
  }
  return fn;
}

func f1=f(1);
real y=f1(pi);

for (int i=1; i<=5; ++i)
  draw(graph(f(i),-10,10),red);
```



Anonymous Functions

- Create new functions with **new**:

```
path p=graph(new real (real x) { return x*sin(10x); },-3,3,red);

func f(int n) {
  return new real (real x) { return n*sin(x/n); };
}
```

- Function definitions are just syntactic sugar for assigning function objects to variables.

```
real square(real x) {
  return x^2;
}
```

is equivalent to

```
real square(real x);
square=new real (real x) {
  return x^2;
};
```

Structures

- As in other languages, structures group together data.

```
struct Person {
  string firstname, lastname;
  int age;
}
Person bob=new Person;
bob.firstname="Bob";
bob.lastname="Chesterton";
bob.age=24;
```

- Any code in the structure body will be executed every time a new structure is allocated...

```
struct Person {
  write("Making a person.");
  string firstname, lastname;
  int age=18;
}
Person eve=new Person;   // Writes "Making a person."
write(eve.age);          // Writes 18.
```

Object-Oriented Programming

- Functions are defined for each instance of a structure.

```
struct Quadratic {
  real a,b,c;
  real discriminant() {
    return b^2-4*a*c;
  }
  real eval(real x) {
    return a*x^2 + b*x + c;
  }
}
```

- This allows us to construct "methods" which are just normal functions declared in the environment of a particular object:

```
Quadratic poly=new Quadratic;
poly.a=-1; poly.b=1; poly.c=2;

real f(real x)=poly.eval;
real y=f(2);
draw(graph(poly.eval, -5, 5));
```

Specialization

- Can create specialized objects just by redefining methods:

```
struct Shape {
    void draw();
    real area();
}

Shape rectangle(real w, real h) {
  Shape s=new Shape;
  s.draw = new void () {
                   fill((0,0)--(w,0)--(w,h)--(0,h)--cycle); };
  s.area = new real () { return w*h; };
  return s;
}

Shape circle(real radius) {
  Shape s=new Shape;
  s.draw = new void () { fill(scale(radius)*unitcircle); };
  s.area = new real () { return pi*radius^2; }
  return s;
}
```

Overloading

- Consider the code:

```
int x1=2;
int x2() {
  return 7;
}
int x3(int y) {
  return 2y;
}

write(x1+x2());  // Writes 9.
write(x3(x1)+x2());  // Writes 11.
```

Overloading

- `x1`, `x2`, and `x3` are never used in the same context, so they can all be renamed `x` without ambiguity:

```
int x=2;
int x() {
  return 7;
}
int x(int y) {
  return 2y;
}

write(x+x());  // Writes 9.
write(x(x)+x());  // Writes 11.
```

- Function definitions are just variable definitions, but variables are distinguished by their signatures to allow overloading.

Operators

- Operators are just syntactic sugar for functions, and can be addressed or defined as functions with the `operator` keyword.

```
int add(int x, int y)=operator +;
write(add(2,3));  // Writes 5.

// Don't try this at home.
int operator +(int x, int y) {
  return add(2x,y);
}
write(2+3);  // Writes 7.
```

- This allows operators to be defined for new types.

Operators

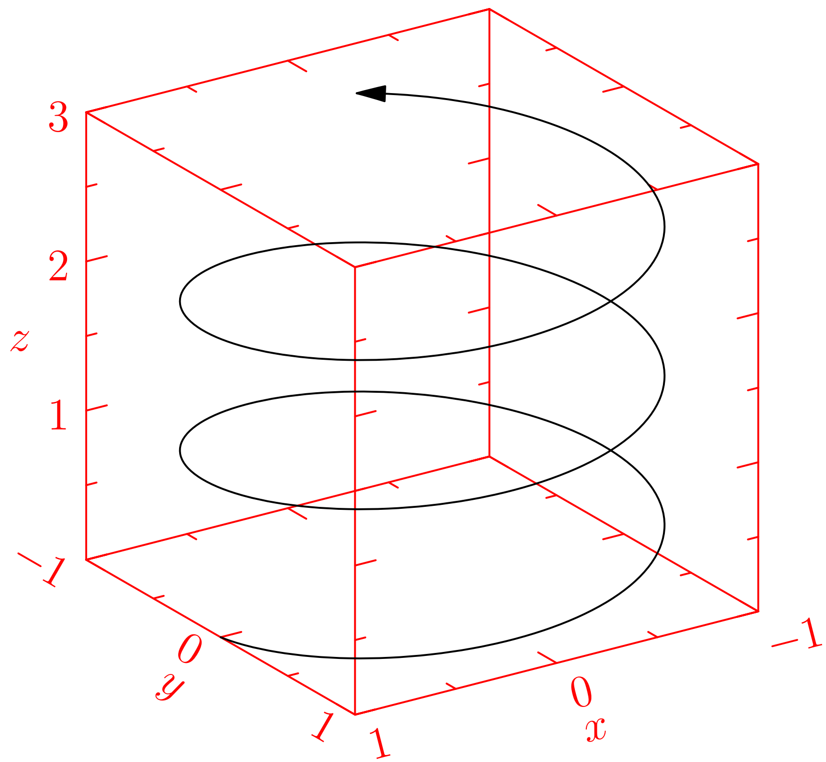
- Operators for constructing paths are also functions:

```
a.. controls b and c .. d--e
```

  is equivalent to

```
operator --(operator ..(a, operator controls(b,c), d), e)
```

- This allowed us to redefine all of the path operators for 3D paths.



Packages

- Function and structure definitions can be grouped into packages:

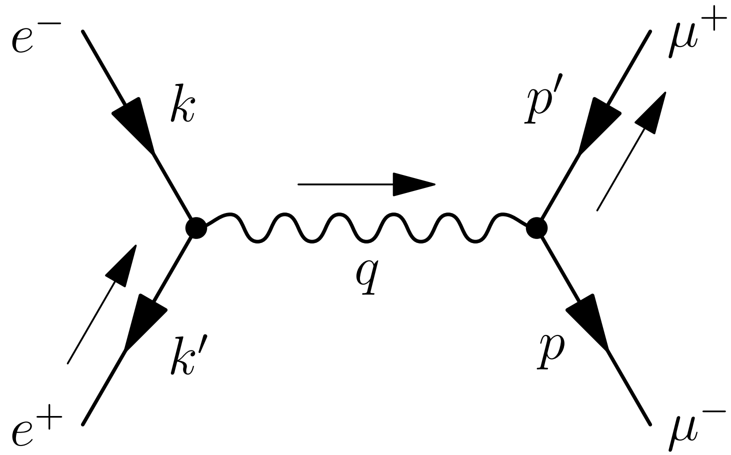
```
// powers.asy
real square(real x) { return x^2; }
real cube(real x) { return x^3; }
```

and imported:

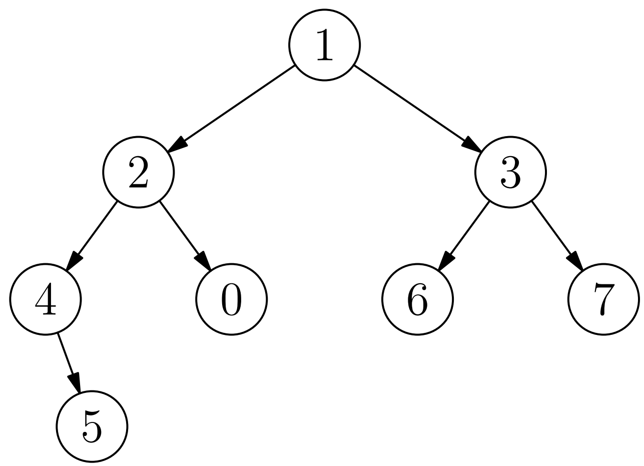
```
import powers;
real eight=cube(2.0);
draw(graph(powers.square, -1, 1));
```

Packages

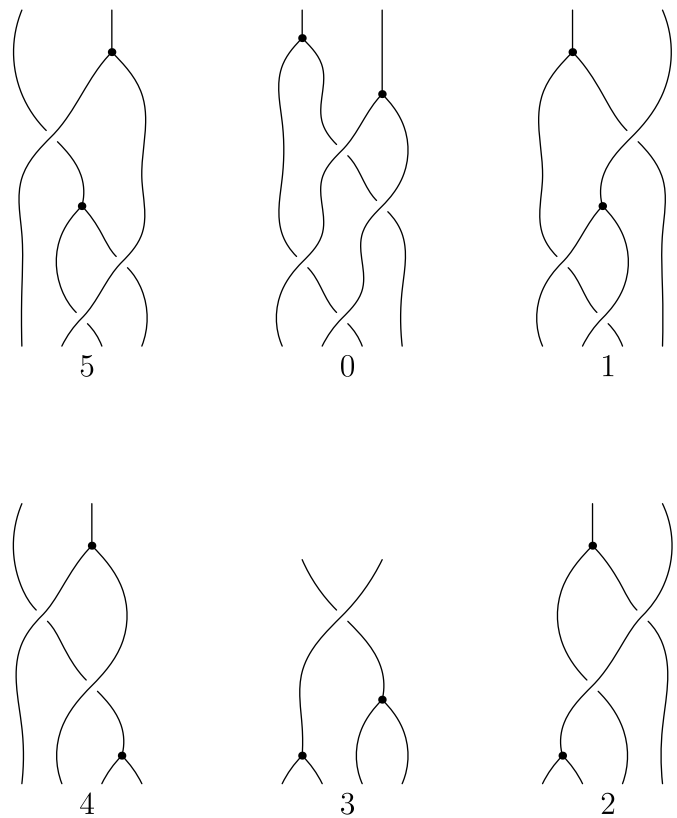
- There are packages for Feynman diagrams,



data structures,



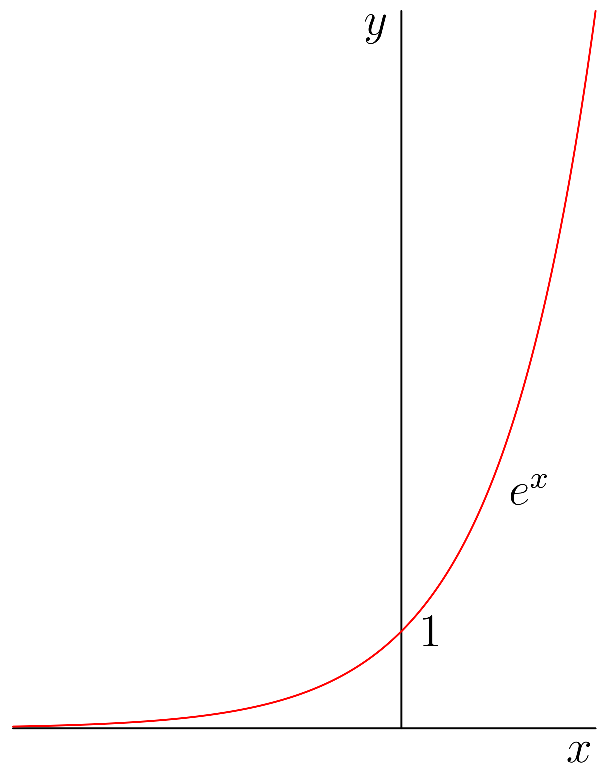
algebraic knot theory:



$$\begin{aligned}\Phi\Phi(x_1,x_2,x_3,x_4,x_5) = \quad & \rho_{4b}(x_1+x_4,x_2,x_3,x_5)+\rho_{4b}(x_1,x_2,x_3,x_4)\\ + \; & \rho_{4a}(x_1,x_2+x_3,x_4,x_5)-\rho_{4b}(x_1,x_2,x_3,x_4+x_5)\\ - \quad & \rho_{4a}(x_1+x_2,x_3,x_4,x_5)-\rho_{4a}(x_1,x_2,x_4,x_5).\end{aligned}$$

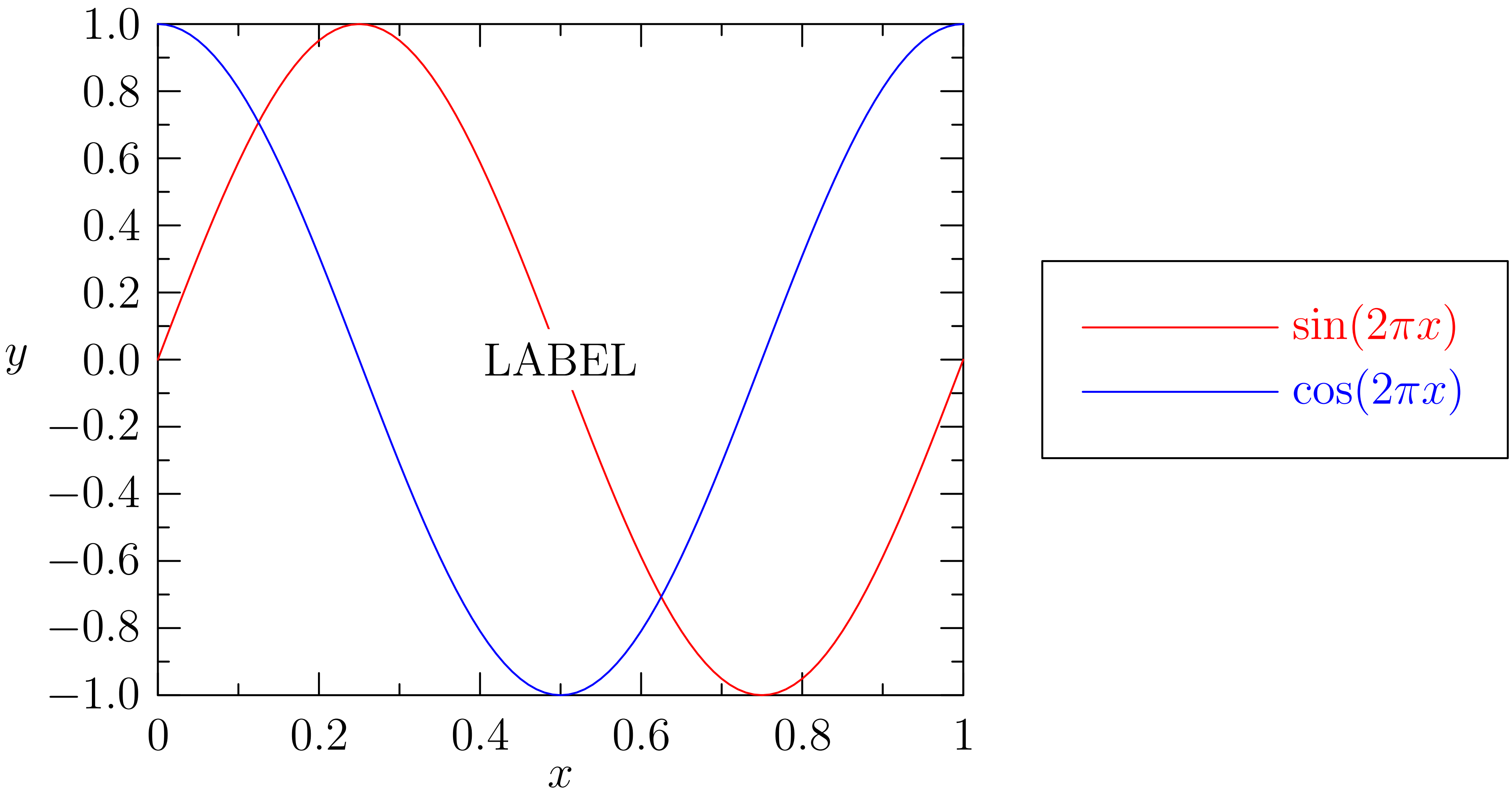
Textbook Graph

```
import graph;
size(150,0);

real f(real x) {return exp(x);}
pair F(real x) {return (x,f(x));}

xaxis("$x$");
yaxis("$y$",0);

draw(graph(f,-4,2,operator ..),red);

labely(1,E);
label("$e^x$",F(1),SE);
```



Scientific Graph

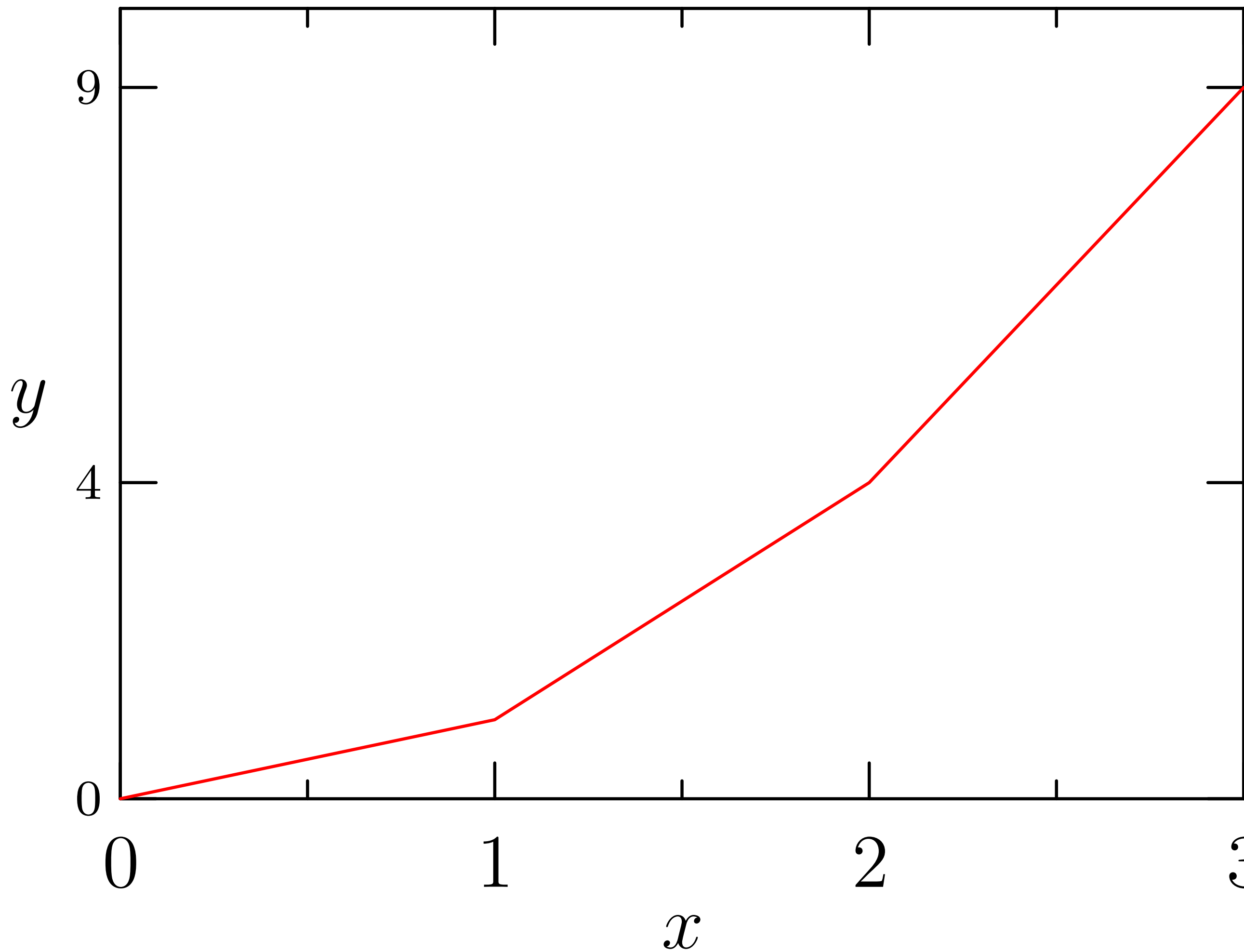
```
import graph;

size(250,200,IgnoreAspect);

real Sin(real t) {return sin(2pi*t);}
real Cos(real t) {return cos(2pi*t);}

draw(graph(Sin,0,1),red,"$\sin(2\pi x)$");
draw(graph(Cos,0,1),blue,"$\cos(2\pi x)$");

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks(trailingzero));

label("LABEL",point(0),UnFill(1mm));

attach(legend(),point(E),20E,UnFill);
```



1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
y
0
0.2
0.4
0.6
0.8
1
x
LABEL
sin(2πx)
cos(2πx)

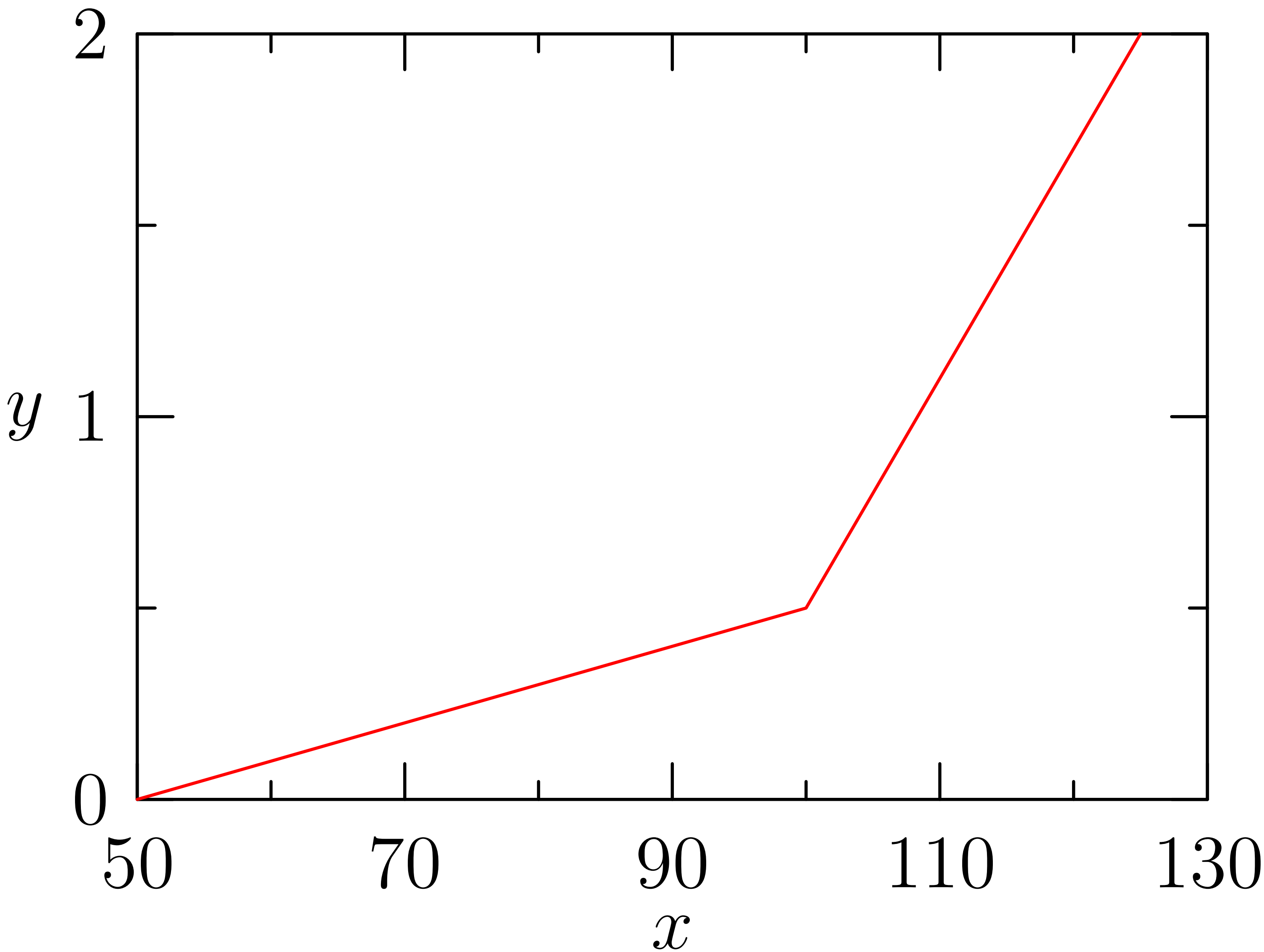
Data Graph

```
import graph;

size(200,150,IgnoreAspect);

real[] x={0,1,2,3};
real[] y=x^2;

draw(graph(x,y),red);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,
      RightTicks(Label(fontsize(8)),new real[]{0,4,9}));
```


9
4
0
y
0
1
2
3
x

Imported Data Graph

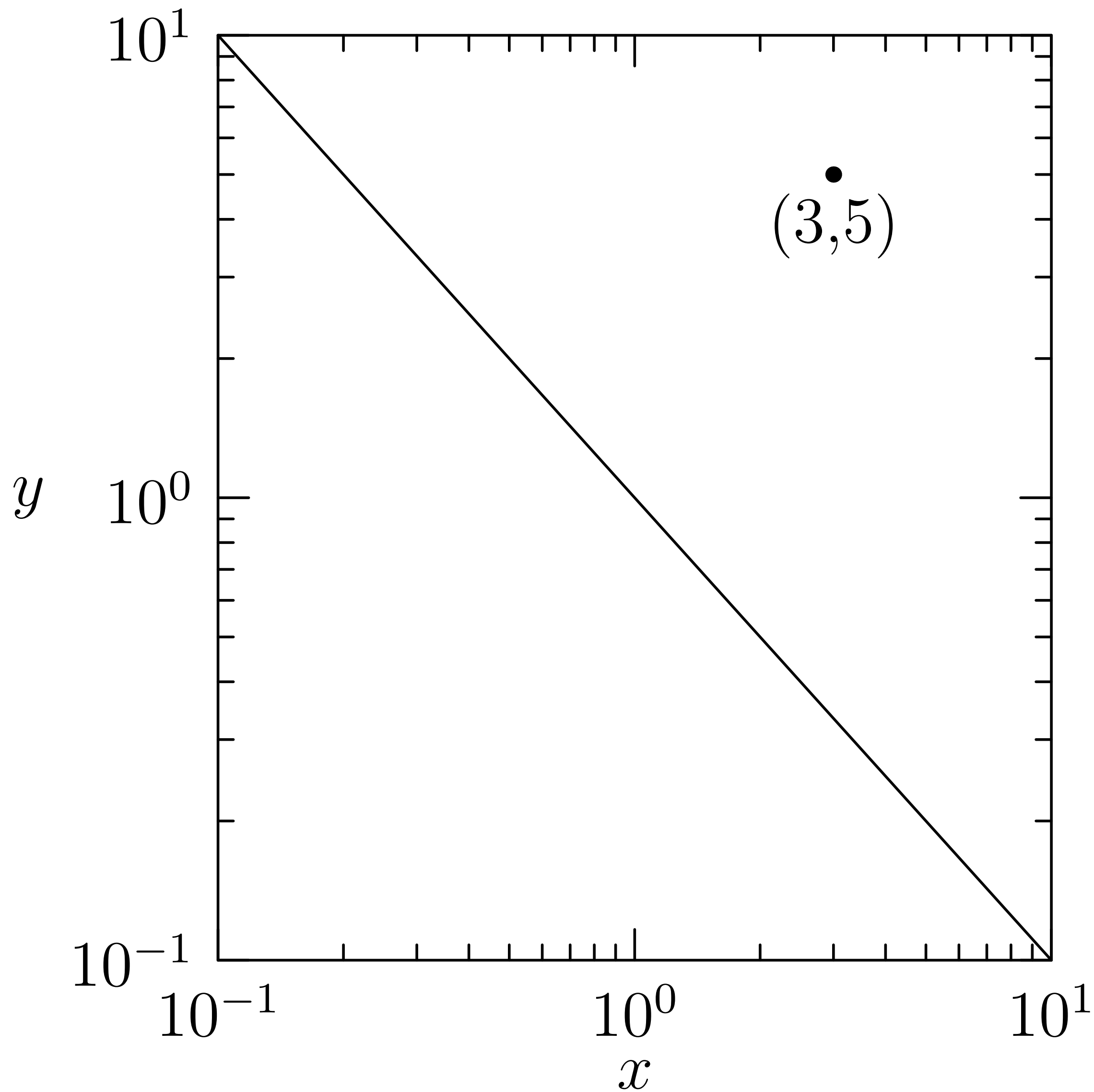
```
import graph;

size(200,150,IgnoreAspect);

file in=line(input("filegraph.dat"));
real[][] a=dimension(in,0,0);
a=transpose(a);

real[] x=a[0];
real[] y=a[1];

draw(graph(x,y),red);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);
```



2
y
1
0
50
70
90
110
130
x

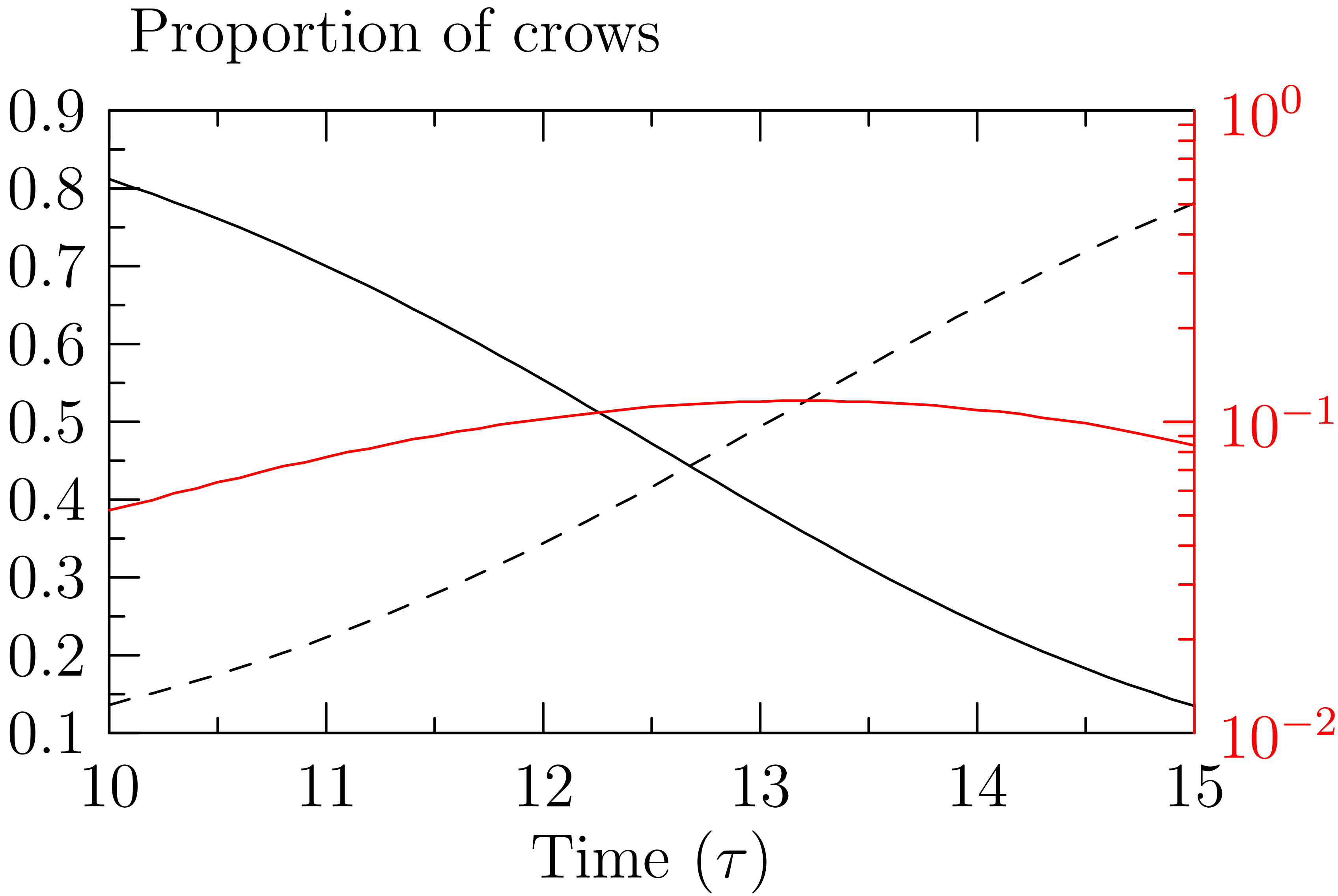
Logarithmic Graph

```
import graph;

size(200,200,IgnoreAspect);

real f(real t) {return 1/t;}

scale(Log,Log);

draw(graph(f,0.1,10));

//xlimits(1,10);
//ylimits(0.1,1);

dot(Label("(3,5)",align=S),Scale((3,5)));

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,RightTicks);
```

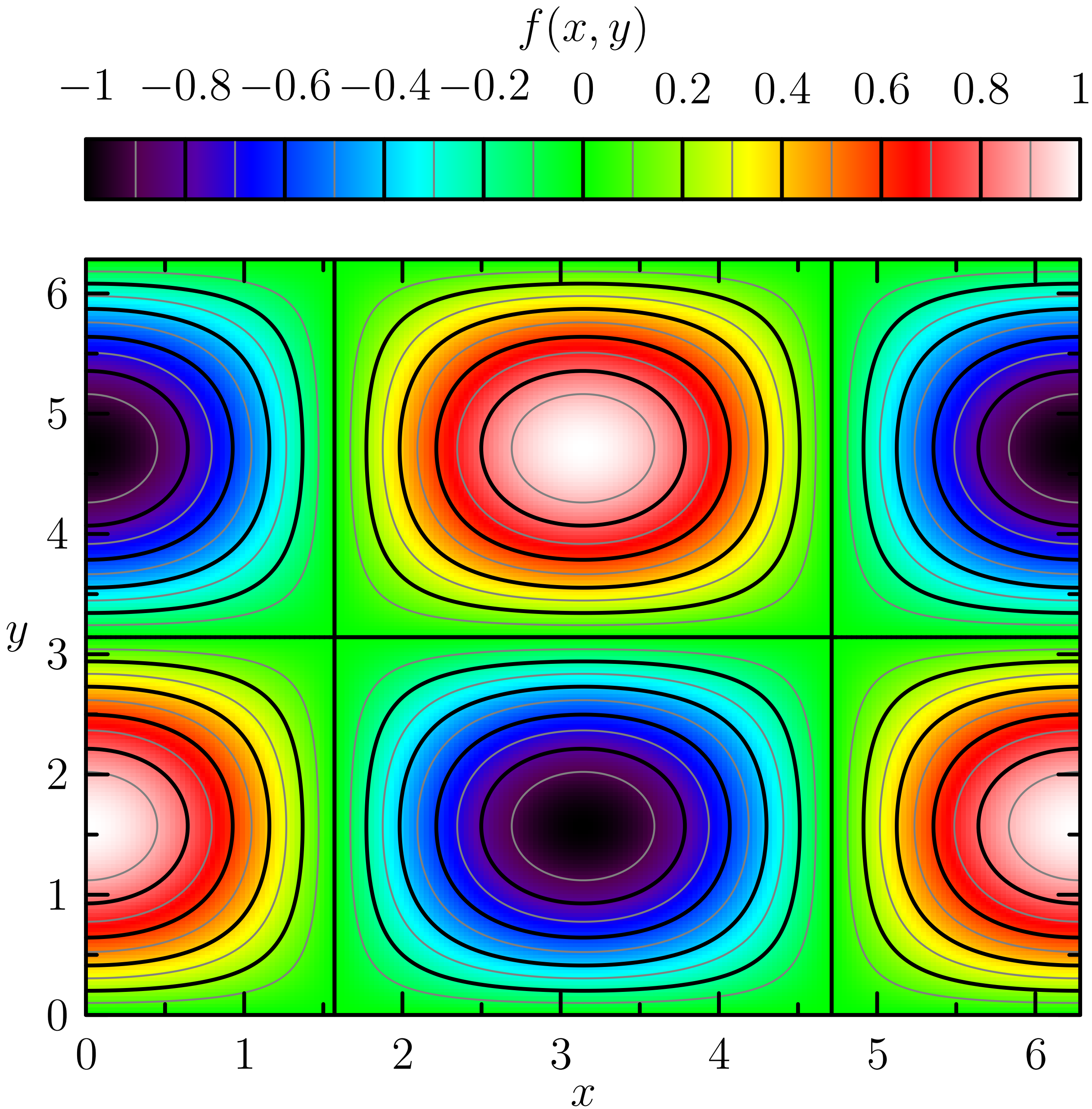


10^1
10^0
10^{-1}
y
(3,5)
10^{-1}
10^0
10^1
x

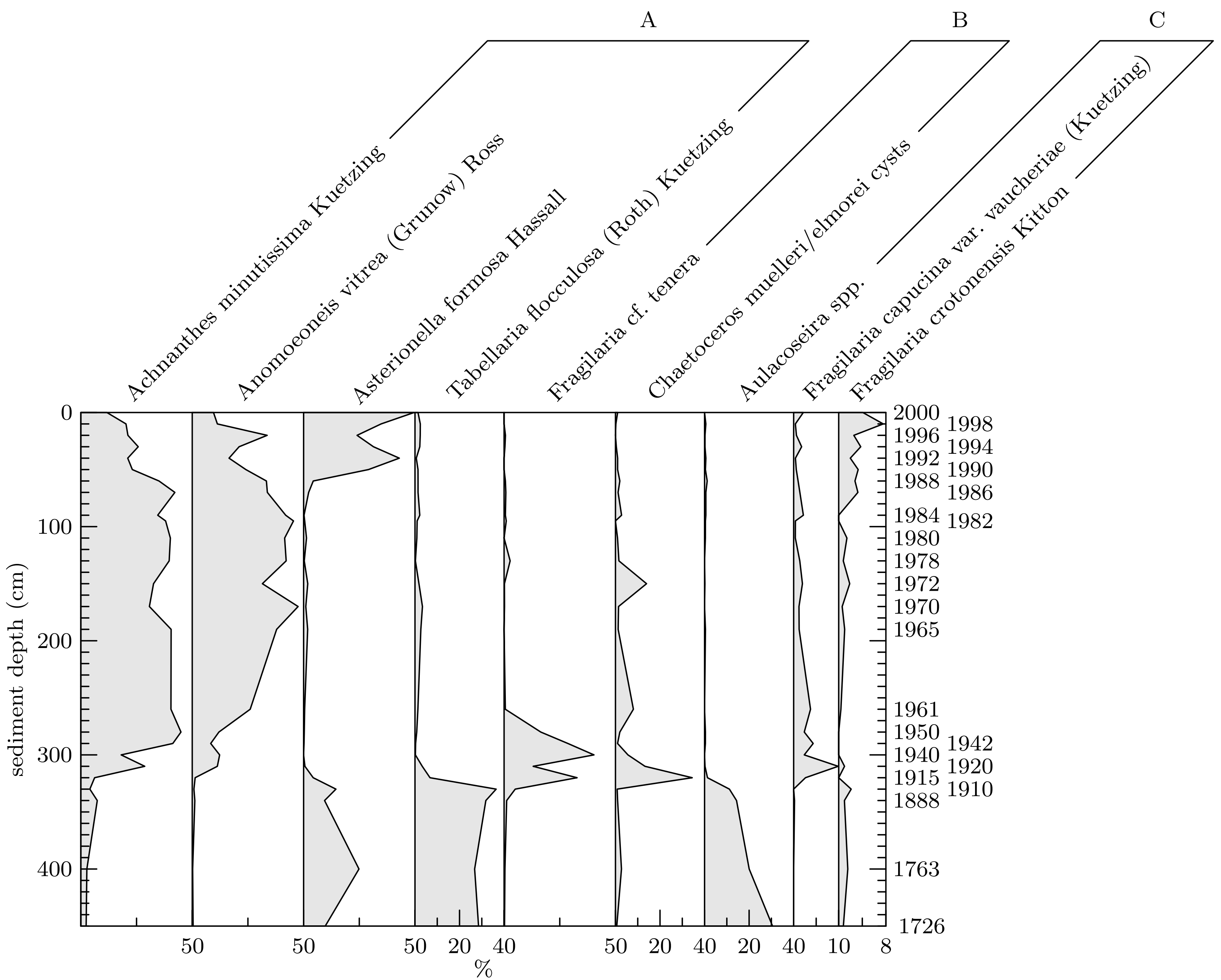
Secondary Axis



Images



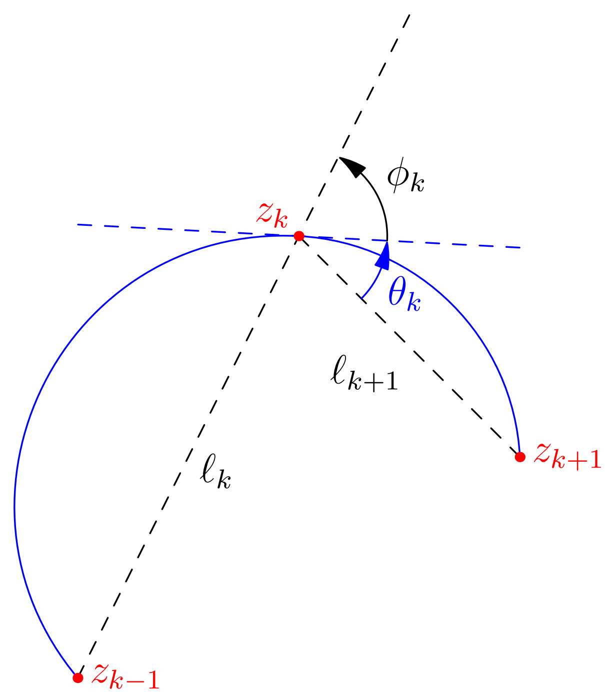
Multiple Graphs



Hobby's 2D Direction Algorithm

- A tridiagonal system of linear equations is solved to determine any unspecified directions θ_k and ϕ_k through each knot z_k:

$$\frac{\theta_{k-1} - 2\phi_k}{\ell_k} = \frac{\phi_{k+1} - 2\theta_k}{\ell_{k+1}}.$$



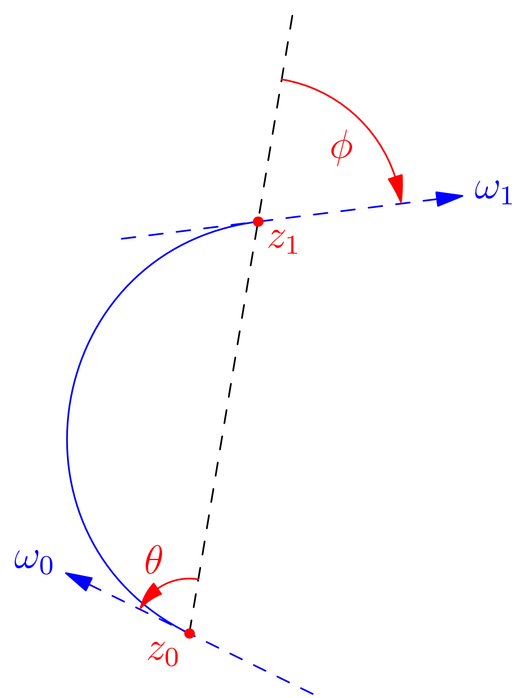
- The resulting shape may be adjusted by modifying optional *tension* parameters and *curl* boundary conditions.

Hobby's 2D Control Point Algorithm

- Having prescribed outgoing and incoming path directions $e^{i\theta}$ at node z_0 and $e^{i\phi}$ at node z_1 relative to the vector $z_1 - z_0$, the control points are determined as:

$$\begin{aligned} u &= z_0 + e^{i\theta}(z_1 - z_0) f(\theta, -\phi), \\ v &= z_1 - e^{i\phi}(z_1 - z_0) f(-\phi, \theta), \end{aligned}$$

where the relative distance function $f(\theta, \phi)$ is given by Hobby [1986].



Bezier Curves in 3D

- Apply an affine transformation

$$x'_i = A_{ij} x_j + C_i$$

to a Bezier curve:

$$x(t) = \sum_{k=0}^{3} B_k(t) P_k, \qquad t \in [0,1].$$

- The resulting curve is also a Bezier curve:

$$\begin{aligned} x'_i(t) &= \sum_{k=0}^{3} B_k(t) A_{ij} (P_k)_j + C_i \\ &= \sum_{k=0}^{3} B_k(t) P'_k, \end{aligned}$$

where P'_k is the transformed k^{th} control point, noting $\Sigma_{k=0}^{3} B_k(t) = 1$.

3D Generalization of Hobby's algorithm

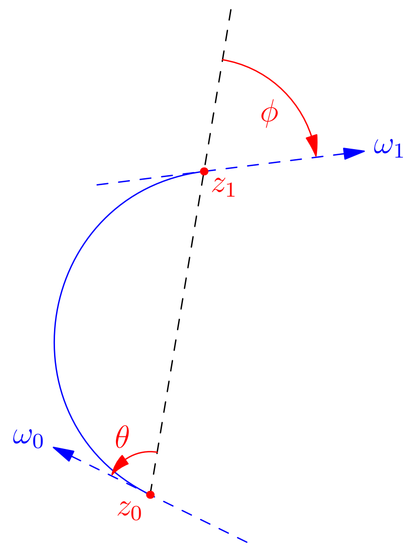
- Must reduce to 2D algorithm in planar case.
- Determine directions by applying Hobby's algorithm in the plane containing z_{k-1}, z_k, z_{k+1}.
- The only ambiguity that can arise is the overall sign of the angles, which relates to viewing each 2D plane from opposing normal directions.
- A reference vector based on the mean unit normal of successive segments can be used to resolve such ambiguities.

3D Control Point Algorithm

- Hobby's control point algorithm can be generalized to 3D by expressing it in terms of the absolute directions ω_0 and ω_1:

$$u = z_0 + \omega_0 \left| z_1 - z_0 \right| f(\theta, -\phi),$$

$$v = z_1 - \omega_1 \left| z_1 - z_0 \right| f(-\phi, \theta),$$

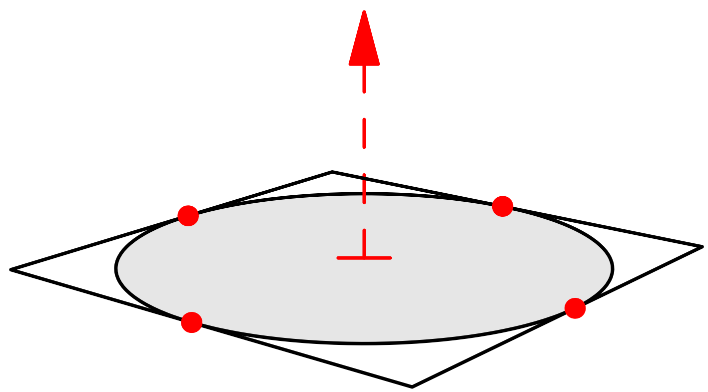


interpreting θ and ϕ as the angle between the corresponding path direction vector and $z_1 - z_0$.

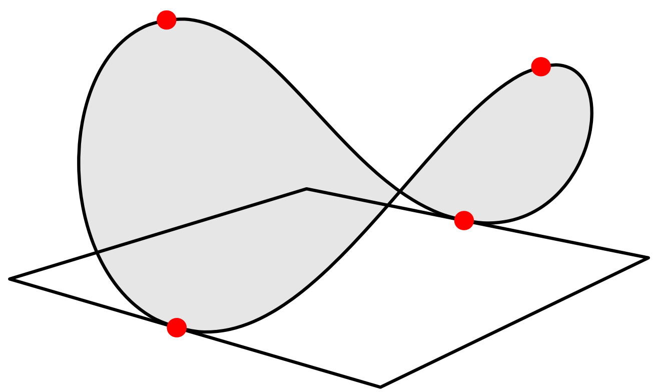
- In this case there is an unambiguous reference vector for determining the relative sign of the angles ϕ and θ.

3D saddle example

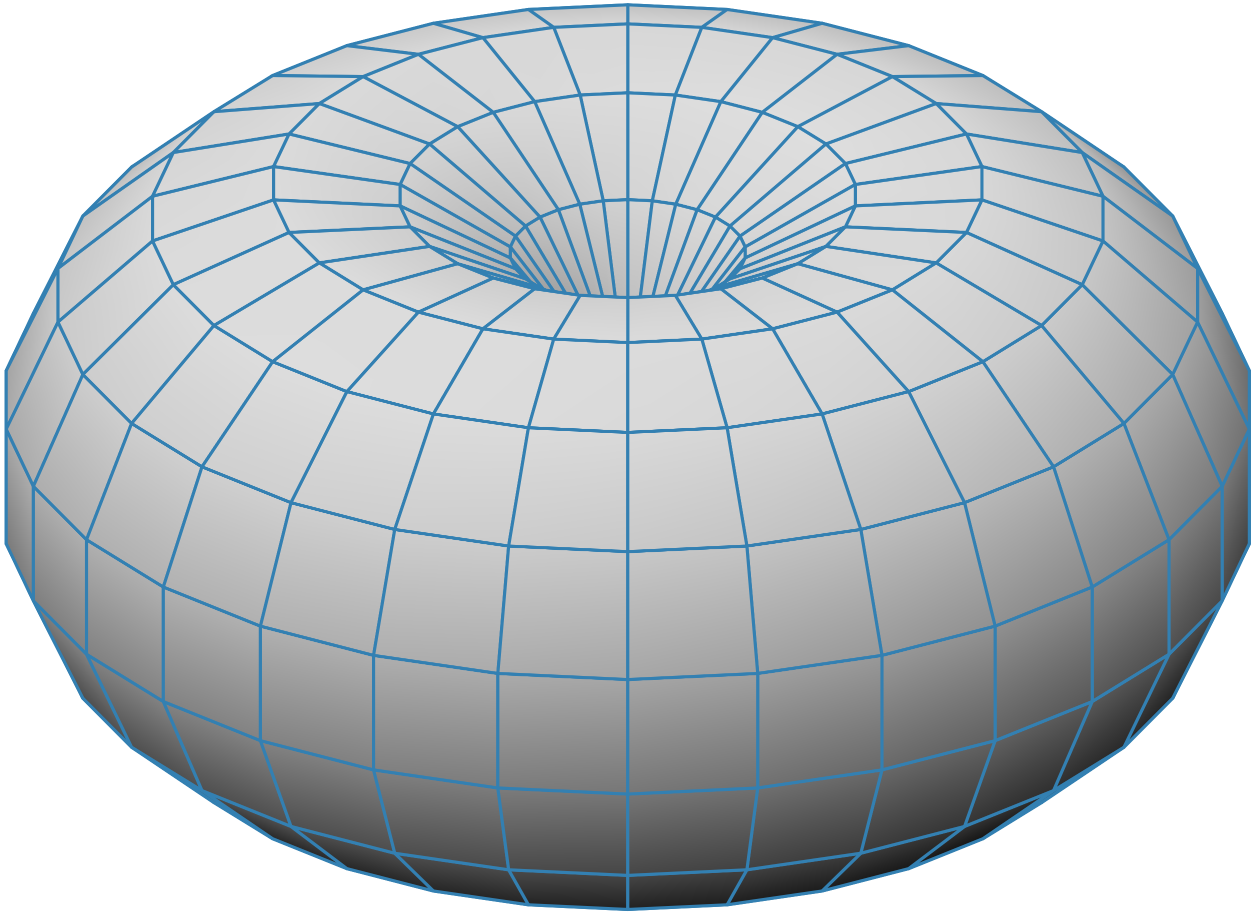
- A unit circle in the X–Y plane may be filled and drawn with:
(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle



and then distorted into a saddle:
(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle



3D surfaces



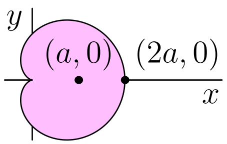
Slide Presentations

- Asymptote has a package for preparing slides.
- It even supports embedded hi-resolution PDF movies.

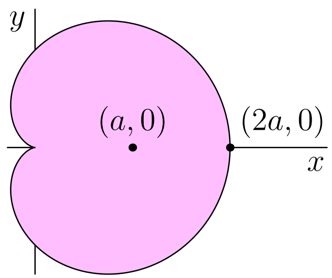
```
title("Slide Presentations");
item("Asymptote has a package for preparing slides.");
item("It even supports embedded hi-resolution PDF movies.");
  ...
```

Automatic Sizing

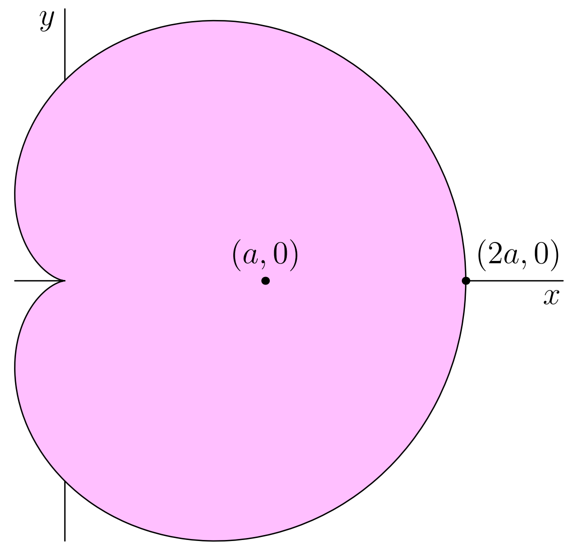
- Figures can be specified in user coordinates, then automatically scaled to the desired final size.



`size(0,50);`



`size(0,100);`



`size(0,200);`

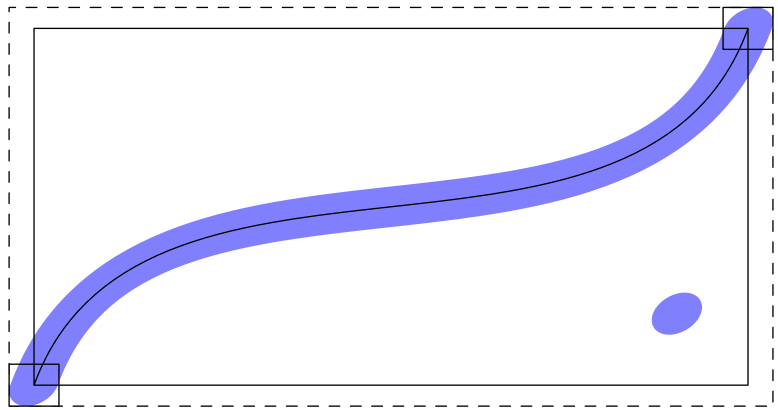
Deferred Drawing

- We can't draw a graphical object until we know the scaling factors for the user coordinates.
- Instead, store a function that when given the scaling information, draws the scaled object.

```
void draw(picture pic=currentpicture, path g, pen p=currentpen) {
  pic.add(new void(frame f, transform t) {
      draw(f,t*g,p);
    });
  pic.addPoint(min(g),min(p));
  pic.addPoint(max(g),max(p));
}
```

Coordinates

- Store bounding box information as the sum of user and true-size coordinates:



```
pic.addPoint(min(g),min(p));
pic.addPoint(max(g),max(p));
```

- Filling ignores the pen width:

```
pic.addPoint(min(g),(0,0));
pic.addPoint(max(g),(0,0));
```

- Communicate with LaTeX to determine label sizes:

$$\boxed{E = mc^2}$$

Sizing

- When scaling the final figure to a given size S, we first need to determine a scaling factor $a > 0$ and a shift b so that all of the coordinates when transformed will lie in the interval $[0, S]$. That is, if u and t are the user and truesize components:

$$0 \leq au + t + b \leq S.$$

- We are maximizing the variable a subject to a number of inequalities. This is a linear programming problem that can be solved by the simplex method.

Sizing

- Every addition of a coordinate (t, u) adds two restrictions

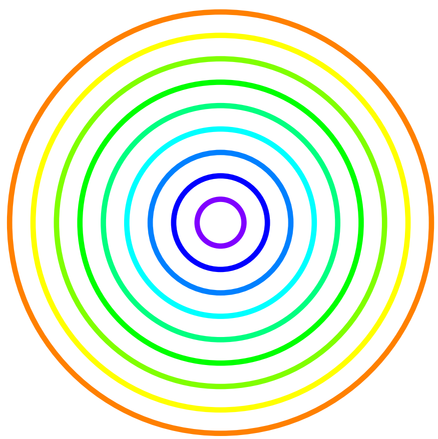
$$au + t + b \geq 0,$$

$$au + t + b \leq S,$$

and each drawing component adds two coordinates.

- A figure could easily produce thousands of restrictions, making the simplex method impractical.

- Most of these restrictions are redundant, however. For instance, with concentric circles, only the largest circle needs to be accounted for.



Redundant Restrictions

- In general, if $u \le u'$ and $t \le t'$ then

$$au + t + b \le au' + t' + b$$

for all choices of $a > 0$ and b, so

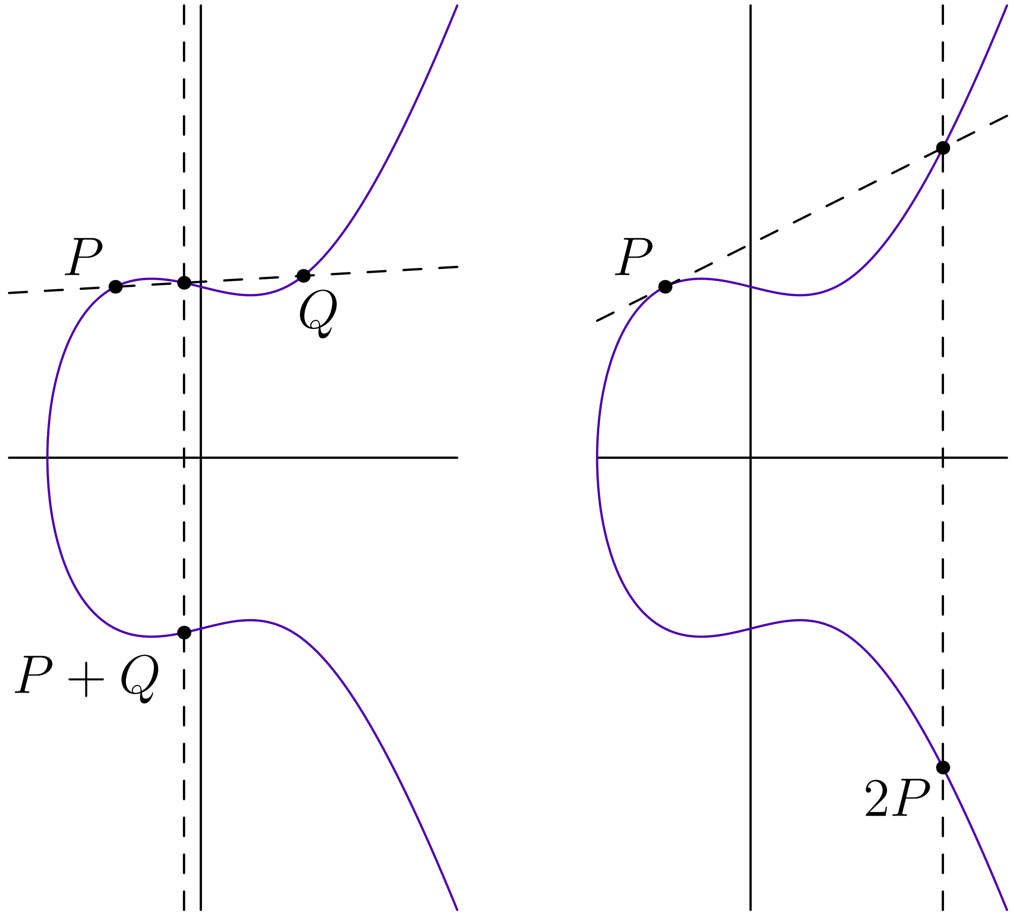
$$0 \le au + t + b \le au' + t' + b \le S.$$

- This defines a partial ordering on coordinates. When sizing a picture, the program first computes which coordinates are maximal (or minimal) and only sends effective constraints to the simplex algorithm.
- In practice, the linear programming problem will have less than a dozen restraints.
- All picture sizing is implemented in Asymptote code.

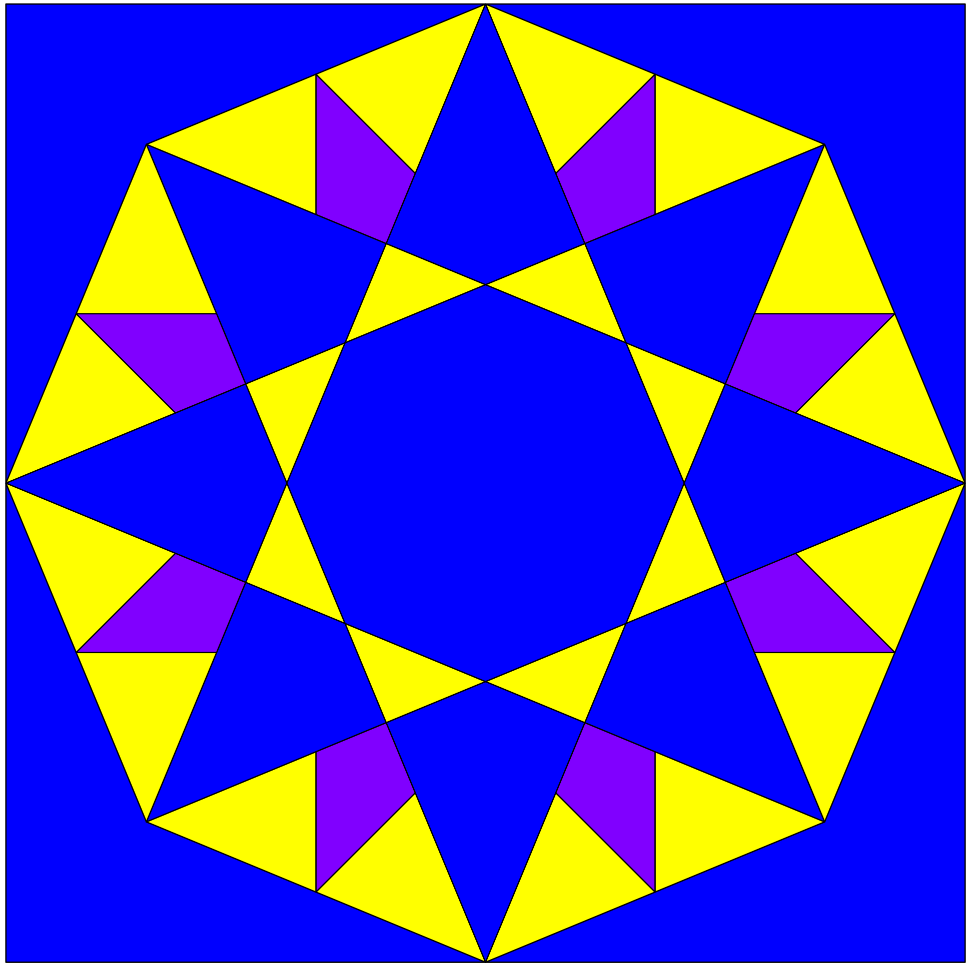
Infinite Lines

- Deferred drawing allows us to draw infinite lines.

```
drawline(P, Q);
```



A Final Example: Quilting



Asymptote: The Vector Graphics Language

`http://asymptote.sf.net`

(freely available under the GNU public license)