Welcome to the Asymptote Style Guide! This document outlines how to write Asymptote code for consistency and readability. (If you have done any programming, the same principles of good coding will generally apply.)

References

First, here are some references that may be useful.

- **The Official Asymptote Manual**
  https://asymptote.sourceforge.io/asymptote.pdf
  It's worth skimming this, just to get a sense of all the features that Asymptote provides.

- **Asymptote Wiki**
  https://artofproblemsolving.com/wiki/index.php/Asymptote_(Vector_Graphics_Language)

- **An Asymptote Tutorial**
  https://math.uchicago.edu/~cstaats/Charles_Staats_III/Notes_and_papers_files/asymptote_tutorial.pdf
  This document describes how to use Asymptote, in a narrative format.

- **Asymptote Galleries**
  https://asymptote.sourceforge.io/gallery/
  https://asy.marris.fr/asymptote/ (in French)
  Physics Asymptote Library (AoPS internal)

- Writing Style Guide

In the following examples, the Asymptote code can be copied into another editor.

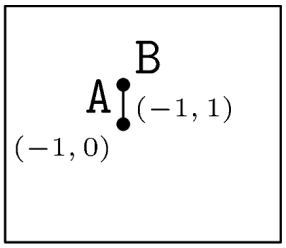
Common issues and weird quirks in Asymptote

Asymptote is a cool but cursed language. *Lasciate qualche speranza, voi ch'entrate*.

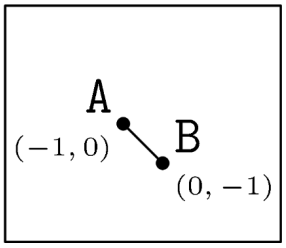
- TeXeR throws an error if the Asymptote code doesn't actually draw anything.
- Asymptote can be given units in "human" units like inches or centimeters, but internally it converts everything into PostScript units. Scroll down to "Converting Asymptote units to centimeters" for a helper function that converts Asymptote units into centimeters.
- The BBCode tags `[asy][/asy]` compile to `\begin{center}\begin{asy}\end{asy}\end{center}`.
    - This often creates unwanted vertical spacing, since LaTeX defines the center environment as `\def\center{\trivlist \centering\item\relax}`, which uses a list for some reason.
    - This spacing can be cancelled with `\vspace{-`*n*`\baselineskip}`, replacing *n* with how much spacing you want to cancel; 2–3 is usually a good amount.
- The variables N, E, S, W are defined as the pairs (0,1), (1,0), (0,-1), (-1,0), but can be overwritten (this often happens unintentionally when dealing with a sequence of, say, at least five points named A through E).
    - The in-between directions are similarly defined, NE as (√2,√2), NW as (-√2,√2), etc.
- `sequence(n)` returns `{0, …, n-1}`, but `sequence(0,n)` returns `{0, …, n}`.
- As the documentation very briefly mentions, the map function with signature `T2[] map(T2 f(T1), T1[] a)` requires a manual import with the line `from mapArray(Src=T1, Dst=T2) access map;`, replacing `T1` and `T2` with the desired types, before it can be used. Hence I (=KB) do not recommend ever using this lol, and instead just writing your own implementation of what you want.

- How `-`, `--`, and `---` interact:

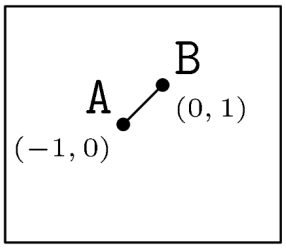
```
A-----B
=> A---(--B)
=> A---(--B.x, B.y)
```



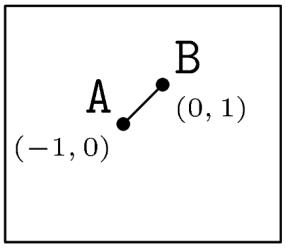

```
A----B
=> A---(-B)
=> A---(-B.x, -B.y)
```



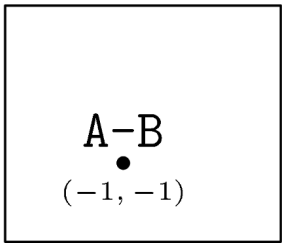

```
A---B
=> A..tension atleast infinity..B
```




```
A--B
```

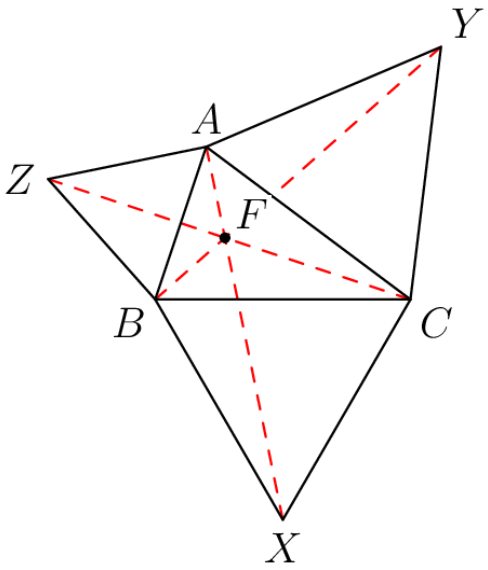



```
A-B
=> (A.x-B.x, A.y-B.y)
```



The Fermat Point of Triangle ABC

```
[asy]
unitsize(0.5 cm);

pair A, B, C, F, X, Y, Z;

A = (1,3);
B = (0,0);
C = (5,0);
X = rotate(60,C)*(B);
Y = rotate(60,A)*(C);
Z = rotate(60,B)*(A);
F = extension(A,X,B,Y);

draw(A--X, red + dashed);
draw(B--Y, red + dashed);
draw(C--Z, red + dashed);
draw(B--C--X--cycle);
draw(C--A--Y--cycle);
draw(A--B--Z--cycle);

label("$A$", A, N);
label("$B$", B, SW);
label("$C$", C, SE);
dot(F);
label("$F$", F, NE, UnFill);
label("$X$", X, S);
label("$Y$", Y, NE);
label("$Z$", Z, W);
[/asy]
```



This code can be broken down into the following parts:

- We start with the command

  ```
  unitsize(0.5 cm);
  ```

  which controls the size of the underlying unit length of the diagram. It's also possible to control the size of the diagram with the `size` command (which specifies the size of the entire diagram), but specifying the size of the unit ensures that any other diagrams based on the same code have the same scale.

- We then define the objects in the diagram. The line

  ```
  pair A, B, C, F, X, Y, Z;
  ```

  defines A, B, C, etc., as pairs. All objects in the code must be defined in terms of a data type, such as:

    - `int i, j;`
    - `path foo;`
    - `real x, y;`
    - `int[] bar;`

- The objects are then assigned specific values:

```
A = (1,3);
B = (0,0);
C = (5,0);
X = rotate(60,C)*(B);
Y = rotate(60,A)*(C);
Z = rotate(60,B)*(A);
F = extension(A,X,B,Y);
```

  One objective in your code should be to have as little “hard-coding” as possible. (In other words, use actual numbers/constants sparingly.) The inputs for this diagram are the coordinates of points A, B, and C. If these coordinates are changed, then all other points will change accordingly. The other points are defined in terms of A, B, and C using the natural geometric commands of Asymptote, which makes this possible.

- The next few lines draw the actual diagram:

```
draw(A--X, red + dashed);
draw(B--Y, red + dashed);
draw(C--Z, red + dashed);
draw(B--C--X--cycle);
draw(C--A--Y--cycle);
draw(A--B--Z--cycle);
```

- We end by labeling each point:

```
label("$A$", A, N);
label("$B$", B, SW);
label("$C$", C, SE);
label("$F$", F, NE, UnFill);
label("$X$", X, S);
label("$Y$", Y, NE);
label("$Z$", Z, W);
```

  Note the use of the `UnFill` command, which blocks out the red dashed line that the label of F overlaps with.

  Some of the code on the website shies away from using E as a point, because it messes up a line like this:

```
label("$A$", A, E);
```

It is possible (and preferable) to use E as a point, when appropriate. The solution is to change the line above to

```
label("$A$", A, dir(0));
```

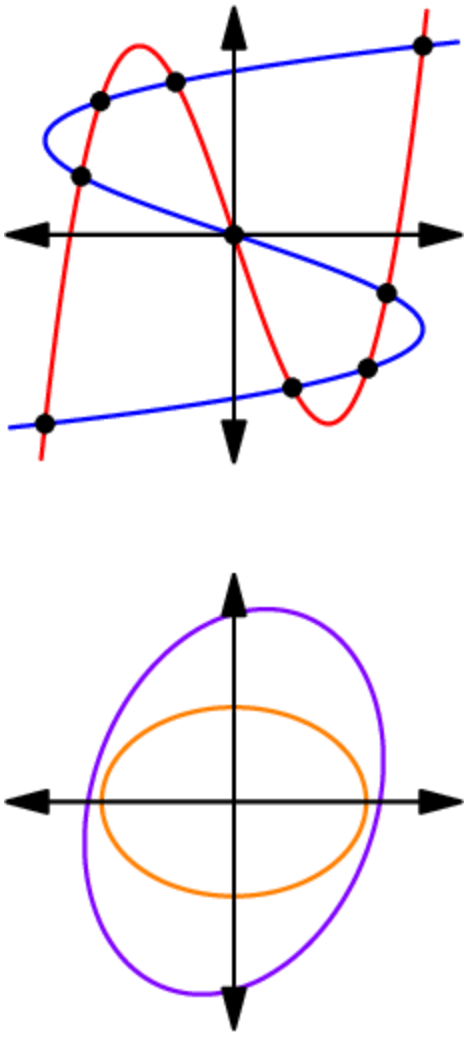
In short, we have divided the code into the following distinct sections:

- Using unitsize
- Defining the objects
- Assigning the objects
- Drawing the diagram
- Using labels

By dividing the code into these sections, we have made the code easy to parse.

Transformations of Graphs

```
[asy]
unitsize (1 cm);

real func (real x) {
  return(4*x^3 - 3*x);
}

pair trans = (3,0);
path ellipseone, ellipsetwo, graphone,
graphtwo;

graphone = graph(func,-1.02,1.02);
graphtwo =
reflect((0,0),(1,1))*(graphone);

draw(graphone, red);
draw(graphtwo, blue);
draw((-1.2,0)--(1.2,0), Arrows(6));
draw((0,-1.2)--(0,1.2), Arrows(6));
for (pair P :
intersectionpoints(graphone,graphtwo))
{
  dot(P);
}

ellipseone =
yscale(0.5)*xscale(0.7)*Circle((0,0),1
);
ellipsetwo =
scale(1.5)*rotate(70)*ellipseone;
draw(shift(trans)*(ellipseone),
orange);
draw(shift(trans)*(ellipsetwo),
purple);
draw(shift(trans)*((-1.2,0)--(1.2,0)),
Arrows(6));
draw(shift(trans)*((0,-1.2)--(0,1.2)),
Arrows(6));
[/asy]
```



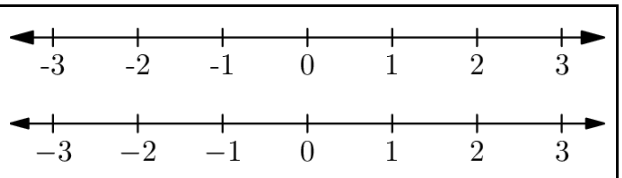
Asymptote has the following commands for transformations:

- `shift` (translation)
- `scale, xscale, yscale` (dilation)
- `rotate` (rotation)
- `reflect` (reflection)
- `slant` (shear transformation)

These commands can be quite useful, because they can be applied to paths as well as points. They can also be composed with each other.

Real Number Lines

```
[asy]
unitsize(1 cm);
defaultpen(fontsize(10));

int i;

draw((-3.5,0)--(3.5,0), Arrows);
draw((-3.5,-1)--(3.5,-1), Arrows(6));

for (i = -3; i <= 3; ++i) {
  draw((i,-0.1)--(i,0.1));
  draw((i,-1.1)--(i,-0.9));
  label(string(i), (i,-0.1), S);
  label("$" + string(i) + "$",
(i,-1.1), S);
}
[/asy]
```



This example illustrates the use of a for loop to label two real number lines.

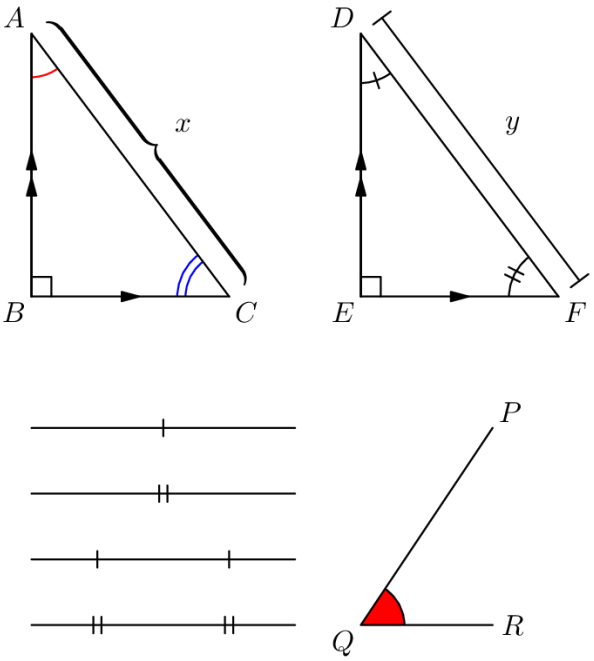
Note that Asymptote uses only prefix notation to increment variables, as in `++i`, not postfix notation, as in `i++`. If you really want to know why, you can find a short note here: https://asymptote.sourceforge.io/doc/Self-_0026-prefix-operators.html

One difference is that in the bottom real number line, the size of the arrows at the endpoints have been adjusted. This doesn't make much of a difference here, but in other diagrams, you will probably find it useful to be able to adjust the size of the arrows.

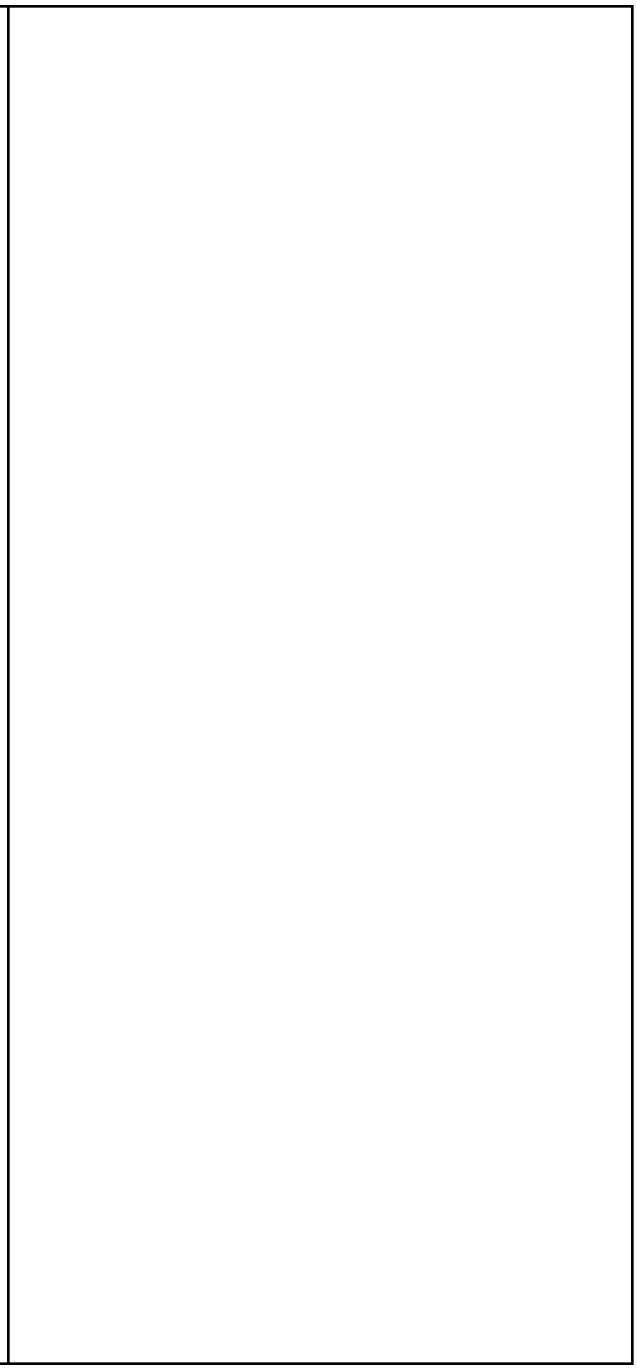
Another difference is that in the bottom real number line, the labels are typeset in math mode, which is preferred. (You can see this in the negative labels.)

Congruent Triangles

```
[asy]
import markers;

unitsize(0.8 cm);
defaultpen(fontsize(10));

pair A, B, C, D, E, F, P, Q, R;

A = (0,4);
B = (0,0);
C = (3,0);
D = A + (5,0);
E = B + (5,0);
F = C + (5,0);
P = (7,-2);
Q = (5,-5);
R = (7,-5);

// Mark the angles at vertices A, C,
D, and F.

markangle(B,A,C,radius=15,red);
markangle(2,A,C,B,radius=15,blue);
markangle(E, D, F, radius=6mm,
marker(markinterval(stickframe(n=1,2mm
),true)));
markangle(D, F, E, radius=6mm,
marker(markinterval(stickframe(n=2,2mm
),true)));

// Draw BC and EF with arrows in the
middle.

draw(B--C, MidArrow(6));
draw(E--F, MidArrow(6));

// Draw the rest of the sides.

draw(B--A--C);
draw(E--D--F);

// Add arrows to AB and DE.

add(arrow(B--A,6,Relative(0.45)));
add(arrow(B--A,6,Relative(0.55)));
add(arrow(E--D,6,Relative(0.45)));
add(arrow(E--D,6,Relative(0.55)));

// Draw right angles at vertices B and
E.

draw(rightanglemark(D,E,F,10));
draw(rightanglemark(A,B,C,10));

// Draw four lines with various tick
marks.
```



```
draw((0,-2)--(4,-2),
StickIntervalMarker(1,1,size=2mm));
draw((0,-3)--(4,-3),
StickIntervalMarker(1,2,size=2mm));
draw((0,-4)--(4,-4),
StickIntervalMarker(2,1,size=2mm));
draw((0,-5)--(4,-5),
StickIntervalMarker(2,2,size=2mm));

// Label vertices.

label("$A$", A, NW);
label("$B$", B, SW);
label("$C$", C, SE);
label("$D$", D, NW);
label("$E$", E, SW);
label("$F$", F, SE);

// Label side AC with brace.

label(rotate(180 -
aTan(4/3))*"$\underbrace{\hspace{4
cm}}$", (A + C)/2 +
0.4*dir(aTan(3/4)));
label("$x$", (A + C)/2 +
1.0*dir(aTan(3/4)));

// Label side DF with bar.

draw(shift(0.4*dir(aTan(3/4)))*(D--F),
bar = Bars);
label("$y$", (D + F)/2 +
1.0*dir(aTan(3/4)));

// Draw solid angle PQR

markangle(R,Q,P,radius=15,Fill(red));
draw(P--Q--R);

label("$P$", P, NE);
label("$Q$", Q, SW);
label("$R$", R, dir(0));
[/asy]
```

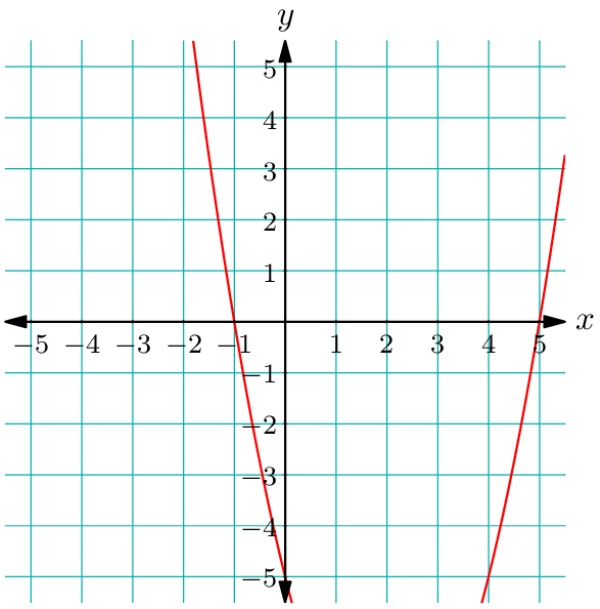


This example illustrates how to use geometric markings. Note that we must import the `markers` package for some of these markings.

Also, you can insert comments with two slashes.

Graph with Axes

```
[asy]
size(175);

// set bounds of the grid
real xmin = -5.5, xmax = 5.5, ymin =
-5.5, ymax = 5.5;

// draw your graph
real f (real x) {return (x^2 - 4x -
5);}
draw(graph(f,xmin,xmax),red);
limits((xmin,ymin),(xmax,ymax),Crop);

// set parameters of the grid lines
ticks tgrid = Ticks(Label("%"), Step =
1, extend = true, rgb(0,0.7,0.7) +
0.4*bp);
ticks taxes =
Ticks(Label(fontsize(8)),
NoZeroFormat, Step = 1, Size = 0.1pt);

// draw axes and grid lines
xaxis(xmin, xmax, taxes, Arrows(6),
above = true);
yaxis(ymin, ymax, taxes, Arrows(6),
above = true);
xaxis(BottomTop, xmin, xmax,
invisible, tgrid);
yaxis(LeftRight, ymin, ymax,
invisible, tgrid);

// add axis labels
label("$x$", (xmax,0), E,
fontsize(10));
label("$y$", (0,ymax), N,
fontsize(10));
[/asy]
```



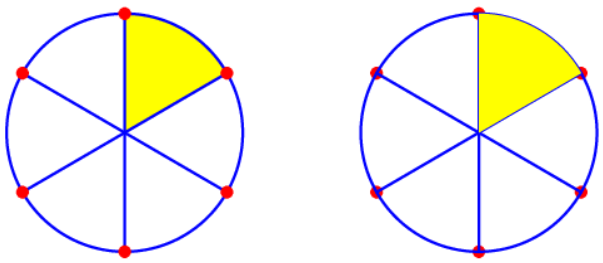
We have set up this template for drawing graphs. (The code may seem complicated, but drawing a reasonable grid in Asymptote using its natural commands is surprisingly difficult.)

Note that we are using the `size` command here. In this code, Asymptote is doing some automatic sizing, which `unitsize` does not work with.

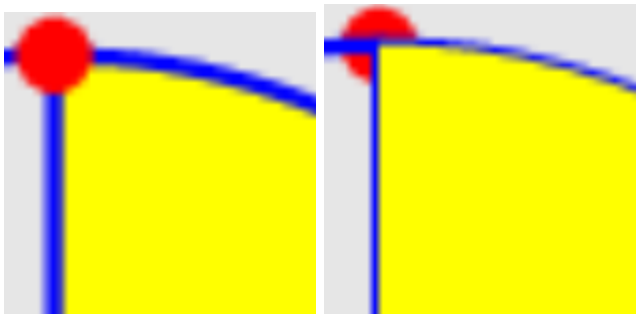
You can find other examples of graphs here:
https://artofproblemsolving.com/crypt/document/321/18748

Circular Sectors

```
[asy]
unitsize(1 cm);

int i;

// Left circle

fill((0,0)--arc((0,0),1,30,90)--cycle,
yellow);

draw(Circle((0,0),1), blue);

for (i = 0; i <= 2; ++i) {
  draw(dir(i*60 + 30)--dir(i*60 + 30 +
180), blue);
}

for (i = 0; i <= 5; ++i) {
  dot(dir(i*60 + 30), red);
}

// Right circle

for (i = 0; i <= 5; ++i) {
  dot(dir(i*60 + 30) + (3,0), red);
}

draw(Circle((3,0),1), blue);

for (i = 0; i <= 2; ++i) {
  draw(shift((3,0))*(dir(i*60 +
30)--dir(i*60 + 30 + 180)), blue);
}

fill((3,0)--arc((3,0),1,30,90)--cycle,
yellow);
[/asy]
```



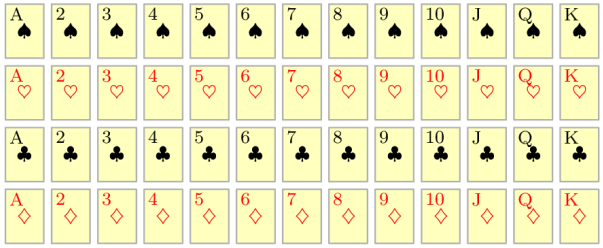
The main point of this example is that when filling in regions, we should fill in the region first, then draw its borders. (In other words, work from the inside-out.) The two close-ups below show the difference. So whenever you are using color, you want to make sure that you are drawing objects in an appropriate order. (Objects that should go “in front” should be drawn last.)



You can also fill in and draw a region at the same time using the `filldraw` command.

Deck of Cards

```
[asy]
unitsize(0.6 cm);

// the goal is to draw an entire deck
of cards

int i, j;
pen color; // we use this for the
color of the card later
path card =
(0,0)--(1,0)--(1,1.4)--(0,1.4)--cycle;
string[] ranks = {"A", "2", "3", "4",
"5", "6", "7", "8", "9", "10", "J",
"Q", "K"};
string[] suits = {"$\spadesuit$",
"$\heartsuit$", "$\clubsuit$",
"$\diamondsuit$"};

for (i = 0; i < ranks.length; ++i) {
  for (j = 0; j < suits.length; ++j) {
      // the suits alternate black,
red, black, red, so we choose the pen
below accordingly
      if (j % 2 == 0) {color = black;}
      if (j % 2 == 1) {color = red;}
      filldraw(shift((1.2*i,
-1.6*j))*(card), paleyellow,
gray(0.7)); // draw the card outline
      label(suits[j], (0.5 + 1.2*i,
0.7 - 1.6*j), color + fontsize(8)); //
label the suit
      label(ranks[i], (1.2*i, -1.6*j)
+ (0,1.4), SE, color + fontsize(8));
// label the rank
```

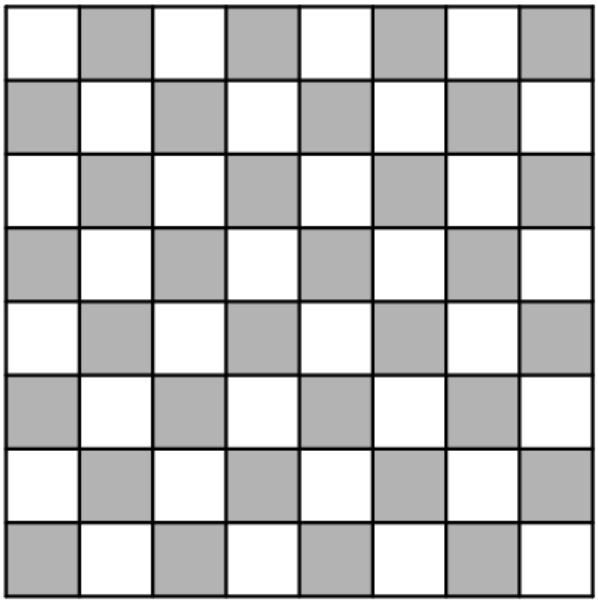


```
  }
}
[/asy]
```

For loops are great for drawing repeated objects.

Chessboard

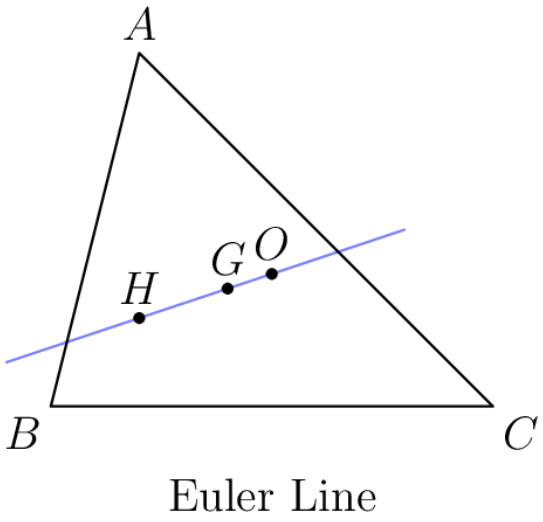
```
[asy]
unitsize(0.5 cm);

int i, j;

for (i = 0; i <= 7; ++i) {
for (j = 0; j <= 7; ++j) {
  if ((i + j) % 2 == 0) {

fill(shift((i,j))*((0,0)--(1,0)--(1,1)
--(0,1)--cycle), gray(0.7));
  }
}}

for (i = 0; i <= 8; ++i) {
  draw((0,i)--(8,i));
  draw((i,0)--(i,8));
}
[/asy]
```



Triangle Centers

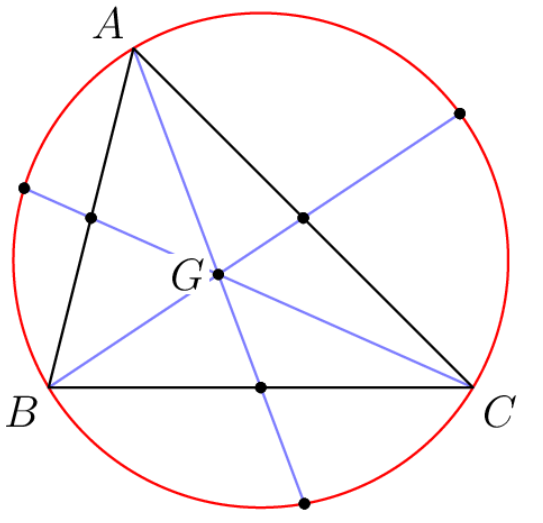
Euler Line

```
[asy]
unitsize(0.8 cm);

pair A, B, C, G, H, O;

A = (1,4);
B = (0,0);
C = (5,0);
G = (A + B + C)/3;
H = orthocenter(A,B,C);
O = circumcenter(A,B,C);

draw(interp(H,O,-1)--interp(H,O,2),
lightblue);
draw(A--B--C--cycle);

label("$A$", A, N);
label("$B$", B, SW);
label("$C$", C, SE);
dot("$G$", G, N);
dot("$H$", H, N);
dot("$O$", O, N);

label("Euler Line", (2.5,-1));
[/asy]
```



Centroid

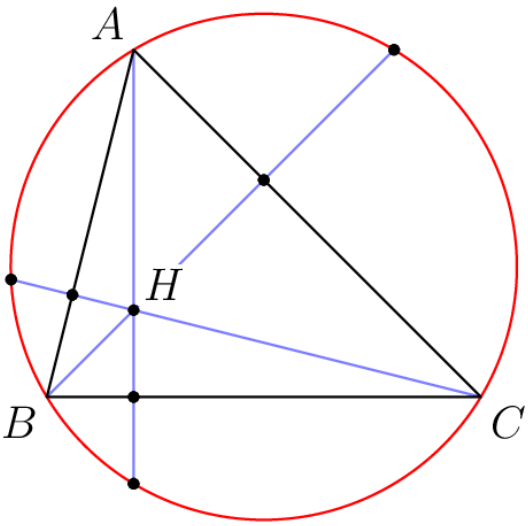
```
[asy]
unitsize(0.8 cm);

pair A, B, C;
pair[] G;

A = (1,4);
B = (0,0);
C = (5,0);
G[0] = (A + B + C)/3;
G[1] = (B + C)/2;
G[2] = (C + A)/2;
G[3] = (A + B)/2;
G[4] =
intersectionpoint(G[1]--interp(A,G[1],
5), circumcircle(A,B,C));
G[5] =
intersectionpoint(G[2]--interp(B,G[2],
5), circumcircle(A,B,C));
G[6] =
```



```
intersectionpoint(G[3]--interp(C,G[3],
5), circumcircle(A,B,C));

draw(circumcircle(A,B,C),red);
draw(A--G[4], lightblue);
draw(B--G[5], lightblue);
draw(C--G[6], lightblue);
draw(A--B--C--cycle);

label("$A$", A, NW);
label("$B$", B, SW);
label("$C$", C, SE);
dot(G);
label("$G$", G[0], W, UnFill);
dot(G[1]);
dot(G[2]);
dot(G[3]);
dot(G[4]);
dot(G[5]);
dot(G[6]);
[/asy]
```

Orthocenter

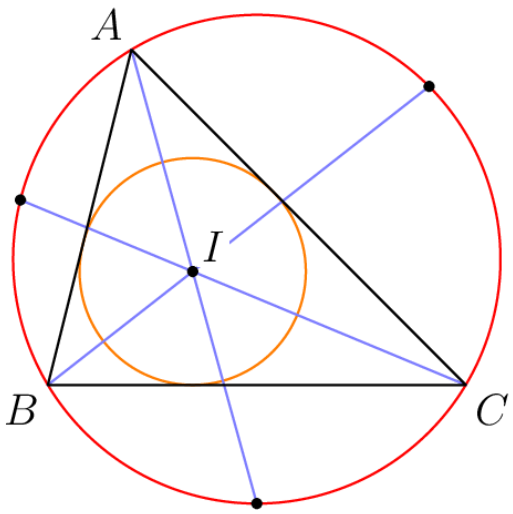
```
[asy]
unitsize(0.8 cm);

pair A, B, C;
pair[] H;

A = (1,4);
B = (0,0);
C = (5,0);
H[0] = orthocenter(A,B,C);
H[1] = (A + reflect(B,C)*(A))/2;
H[2] = (B + reflect(C,A)*(B))/2;
H[3] = (C + reflect(A,B)*(C))/2;
H[4] = reflect(B,C)*(H[0]);
H[5] = reflect(C,A)*(H[0]);
H[6] = reflect(A,B)*(H[0]);

draw(circumcircle(A,B,C),red);
draw(A--H[4], lightblue);
draw(B--H[5], lightblue);
draw(C--H[6], lightblue);
draw(A--B--C--cycle);

label("$A$", A, NW);
label("$B$", B, SW);
label("$C$", C, SE);
dot(H);
label("$H$", H[0], NE, UnFill);
dot(H[1]);
```



```
dot(H[2]);
dot(H[3]);
dot(H[4]);
dot(H[5]);
dot(H[6]);
[/asy]
```

Incenter

```
[asy]
unitsize(0.8 cm);

pair A, B, C;
pair[] I;

A = (1,4);
B = (0,0);
C = (5,0);
I[0] = incenter(A,B,C);
I[4] =
intersectionpoint(I[0]--interp(A,I[0],
5), circumcircle(A,B,C));
I[5] =
intersectionpoint(I[0]--interp(B,I[0],
5), circumcircle(A,B,C));
I[6] =
intersectionpoint(I[0]--interp(C,I[0],
5), circumcircle(A,B,C));

draw(circumcircle(A,B,C),red);
draw(incircle(A,B,C),orange);
draw(A--I[4], lightblue);
draw(B--I[5], lightblue);
draw(C--I[6], lightblue);
draw(A--B--C--cycle);

label("$A$", A, NW);
label("$B$", B, SW);
label("$C$", C, SE);
dot(I[0]);
label("$I$", I[0], NE, UnFill);
dot(I[4]);
dot(I[5]);
dot(I[6]);
[/asy]
```



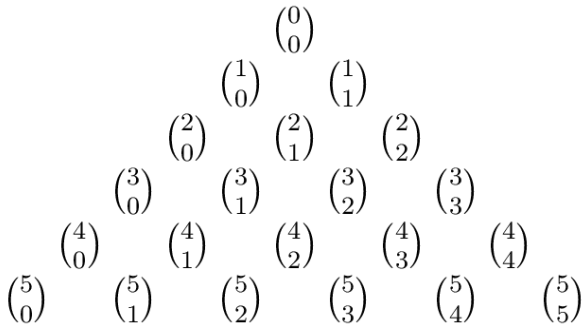
On the AoPS website, the olympiad package is imported automatically for any diagram. This package includes functions for triangle centers such as the circumcenter, incenter, and orthocenter.

More details can be found here:
https://artofproblemsolving.com/wiki/index.php/Asymptote:_Macros_and_Packages

---

Pascal's Triangle

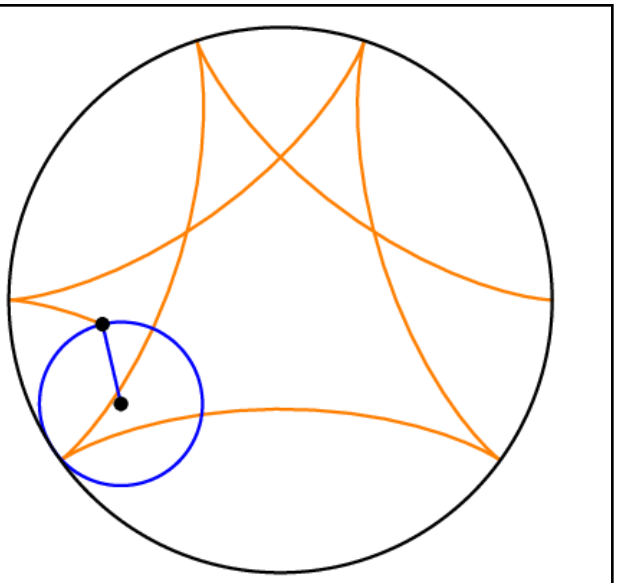
```
[asy]
usepackage("amsmath");
usepackage("amssymb");

unitsize (0.6cm);

int n, k;

for (n = 0; n <= 5; ++n) {
for (k = 0; k <= n; ++k) {
  label("$\binom{" + string(n) + "}{"
+ string(k) + "}$", n*(-1,-1) +
k*(2,0));
}}
[/asy]
```



You can use LaTeX in Asymptote, but certain commands, like `\binom` require importing the packages `amsmath` and/or `amssymb`.

Parametric Graphs

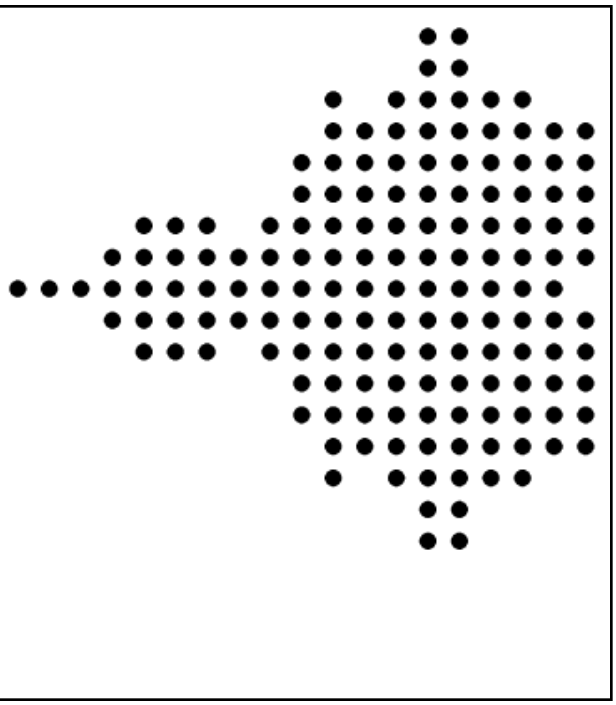
```
[asy]
unitsize(2 cm);

pair hypopair (real t) {
  int p, q;
  p = 3;
  q = 10;
  return (((1 - p/q)*cos(t) +
p/q*cos((q - p)/p*t), (1 - p/q)*sin(t)
- p/q*sin((q - p)/p*t)));
}

int p, q;
real t, tmax;
pair O;

p = 3;
q = 10;
tmax = 10;
O = ((1 - p/q)*cos(tmax), (1 -
p/q)*sin(tmax));

draw(graph(hypopair,0,tmax), orange);
draw(Circle(O,p/q), blue);
draw(O--hypopair(tmax), blue);
draw(Circle((0,0),1));

dot(O);
dot(hypopair(tmax));
[/asy]
```



In Asymptote, we can draw graphs defined parametrically.

The Mandelbrot Set

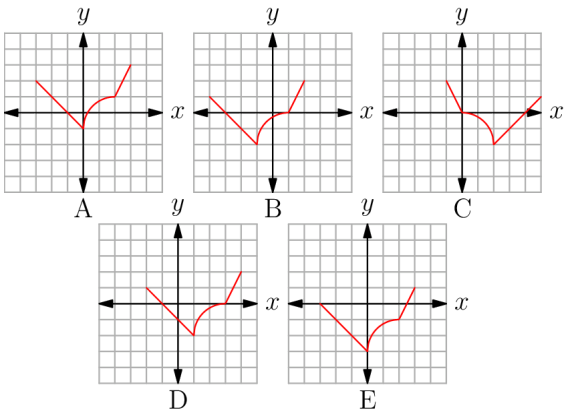
```
[asy]
unitsize (2 cm);

int n;
real x, y;
pair C, W, Z;

for (x = -2.0; x <= 1.0; x = x + 0.1)
{
for (y = -1.0; y <= 1.0; y = y + 0.1)
{
  n = 0;
  C = (x,y);
  Z = (x,y);
  while (abs(Z) < 2 && n <= 100) {
       W = Z^2 + C;
       Z = W;
       ++n;
  }
  if (abs(Z) < 2) {dot(C);}
}}
[/asy]
```



Asymptote treats pairs as complex numbers. In other words, if A and B are pairs, then A + B, A - B, A*B, and A/B are all computed as complex numbers.

Multiple Graphs

```
[asy]
unitsize(0.25 cm);

picture[] graf;
int i, n;

real func(real x) {
  real y;
  if (x >= -3 && x <= 0) {y = -2 - x;}
  if (x >= 0 && x <= 2) {y = sqrt(4 -
(x - 2)^2) - 2;}
  if (x >= 2 && x <= 3) {y = 2*(x -
2);}
  return(y);
}

for (n = 1; n <= 5; ++n) {
  graf[n] = new picture;
  for (i = -5; i <= 5; ++i) {

draw(graf[n],(i,-5)--(i,5),gray(0.7));
```

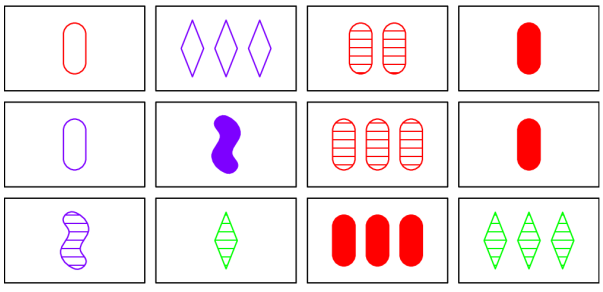


```
draw(graf[n],(-5,i)--(5,i),gray(0.7));
  }

draw(graf[n],(-5,0)--(5,0),Arrows(6));

draw(graf[n],(0,-5)--(0,5),Arrows(6));

  label(graf[n],"$x$", (5,0), E);
  label(graf[n],"$y$", (0,5), N);
}

draw(graf[1],shift(0,1)*graph(func,-3,
3),red);
draw(graf[2],shift(-1,0)*graph(func,-3
,3),red);
draw(graf[3],reflect((1,0),(1,1))*grap
h(func,-3,3),red);
draw(graf[4],shift(1,0)*graph(func,-3,
3),red);
draw(graf[5],shift(0,-1)*graph(func,-3
,3),red);

label(graf[1], "A", (0,-6));
label(graf[2], "B", (0,-6));
label(graf[3], "C", (0,-6));
label(graf[4], "D", (0,-6));
label(graf[5], "E", (0,-6));

add(graf[1]);
add(shift((12,0))*(graf[2]));
add(shift((24,0))*(graf[3]));
add(shift((6,-12))*(graf[4]));
add(shift((18,-12))*(graf[5]));
[/asy]
```

When drawing multiple graphs, using the `picture` element can come in handy.

SET Cards

```
[asy]
import patterns;
add("hatch",hatch());
add("hatchred",hatch(1.5mm,W,red));
add("hatchpurple",hatch(1.5mm,W,purple
));
add("hatchgreen",hatch(1.5mm,W,green))
;
```



```
unitsize(1 cm);

real cardheight = 1.5, cardwidth =
2.5, cardspace = 0.2;
int i, j;
path card =
(0,0)--(cardwidth,0)--(cardwidth,cardh
eight)--(0,cardheight)--cycle;
path oval =
arc((0,0.25),0.2,0,180)--arc((0,-0.25)
,0.2,180,360)--cycle;
path diamond =
(0.2,0)--(0,0.5)--(-0.2,0)--(0,-0.5)--
cycle;
path squiggle =
(0.25,0.2){dir(90)}::(0,0.5){dir(160)}
::(-0.2,0.4){dir(250)}::(-0.1,0.15){di
r(270)}::(-0.25,-0.2){dir(270)}::(0,-0
.5){dir(340)}::(0.2,-0.4){dir(70)}::(0
.1,-0.15){dir(90)}::cycle;

// number: self-explanatory
// color: 1 = red, 2 = purple, 3 =
green
// shape: 1 = oval, 2 = diamond, 3 =
squiggle
// shading: 1 = none, 2 = shaded, 3 =
solid
void drawsetcard (int cardnumber, int
cardcolor, int cardshape, int
cardshading, pair cardcenter) {
  pen p;
  path shape;
  int i;

  if (cardcolor == 1) {p = red;}
  if (cardcolor == 2) {p = purple;}
  if (cardcolor == 3) {p = green;}
  if (cardshape == 1) {shape = oval;}
  if (cardshape == 2) {shape =
diamond;}
  if (cardshape == 3) {shape =
squiggle;}

  for (i = 0; i <= cardnumber - 1;
++i) {
      if (cardshading == 1) {
      draw(shift(cardcenter + ((1 -
cardnumber)*0.3 +
i*0.6,0))*(shape),p);
      }
      if (cardshading == 2) {
      draw(shift(cardcenter + ((1 -
cardnumber)*0.3 +
i*0.6,0))*(shape),p);
      if (cardcolor == 1) {
```

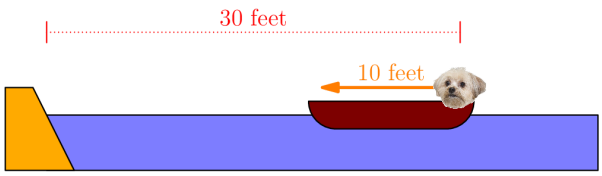
```
      fill(shift(cardcenter + ((1 -
cardnumber)*0.3 +
i*0.6,0))*(shape),pattern("hatchred"))
;
      }
      if (cardcolor == 2) {
      fill(shift(cardcenter + ((1 -
cardnumber)*0.3 +
i*0.6,0))*(shape),pattern("hatchpurple
"));
      }
      if (cardcolor == 3) {
      fill(shift(cardcenter + ((1 -
cardnumber)*0.3 +
i*0.6,0))*(shape),pattern("hatchgreen"
));
      }
      }
      if (cardshading == 3) {
      filldraw(shift(cardcenter + ((1
- cardnumber)*0.3 +
i*0.6,0))*(shape),p,p);
      }
  }
}

for (i = 0; i <= 3; ++i) {
for (j = 0; j <= 2; ++j) {
  draw(shift((i*cardwidth +
i*cardspace, j*cardheight +
j*cardspace))*(card));
}}

drawsetcard(1,2,3,2,(cardwidth/2,cardh
eight/2));
drawsetcard(1,3,2,2,(3*cardwidth/2 +
cardspace,cardheight/2));
drawsetcard(3,1,1,3,(5*cardwidth/2 +
2*cardspace,cardheight/2));
drawsetcard(3,3,2,2,(7*cardwidth/2 +
3*cardspace,cardheight/2));
drawsetcard(1,2,1,1,(cardwidth/2,3*car
dheight/2 + cardspace));
drawsetcard(1,2,3,3,(3*cardwidth/2 +
cardspace,3*cardheight/2 +
cardspace));
drawsetcard(3,1,1,2,(5*cardwidth/2 +
2*cardspace,3*cardheight/2 +
cardspace));
drawsetcard(1,1,1,3,(7*cardwidth/2 +
3*cardspace,3*cardheight/2 +
cardspace));
drawsetcard(1,1,1,1,(cardwidth/2,5*car
dheight/2 + 2*cardspace));
drawsetcard(3,2,2,1,(3*cardwidth/2 +
cardspace,5*cardheight/2 +
```

```
2*cardspace));
drawsetcard(2,1,1,2,(5*cardwidth/2 +
2*cardspace,5*cardheight/2 +
2*cardspace));
drawsetcard(1,1,1,3,(7*cardwidth/2 +
3*cardspace,5*cardheight/2 +
2*cardspace));
[/asy]
```

The `void` function type is useful when you want to draw objects by passing in arguments, but you don't want the function to actually return any value.

Diagram with Gizmo

```
[asy]
size( 300 ) ;

void boat( pair center , real length ,
real height , pen fill_pen ){
    filldraw(
shift(center)*(arc((-length/2+height,h
eight),height,180,270)--arc((length/2-
height,height),height,270,360)--cycle)
, fill_pen ) ;
}

string gizmo =
graphic("/var/www/cdn/school/crypt/001
64-d7fa6e1e888381c2d44abd3cb230b7e393b
a77c6/files/Gizmo.eps","width=1cm");

filldraw( box( (0,0) , (40,4) ) ,
lightblue ) ;
boat( (25,3) , 12 , 2 , brown ) ;
filldraw(
(2,0)--(-1,6)--(-3,6)--(-3,0)--cycle ,
yellow+brown ) ;
label( gizmo , (30,6) ) ;

path to_shore = (0,10)--(30,10) ;
add(pathticks( to_shore , 1, 0 , s=50
, red )) ; add(pathticks( to_shore ,
1, 1 , s=50 , red )) ;
draw( to_shore , red+dotted ) ; label(
"30 feet" , to_shore , N , red ) ;

draw( (30,6)--(20,6) , arrow=Arrow(8)
, orange+linewidth(1.5) ) ; label( "10
feet" , (30,6)--(20,6) , N , orange )
```



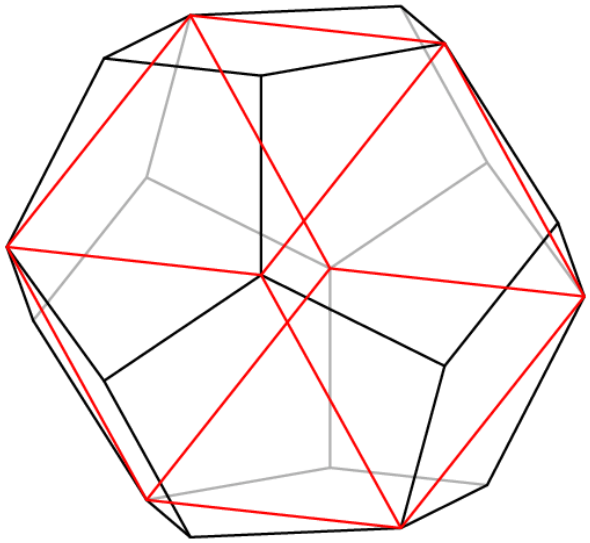
; [/asy]	

It is possible to include pictures inside an Asymptote diagram. The pictures must be in EPS (Encapsulated PostScript) format. If your picture is not a different format like .png or .jpg, there are plenty of web-based converters you can find just by searching "convert to eps" online. Or, if you're comfortable with it, you can use a tool like imagemagick from https://imagemagick.org/

Source: https://artofproblemsolving.com/crypt/document/164/8407

3D Geometry

```
[asy]
size(150);

import three;
currentprojection =
orthographic(0.1,0.8,0.3);

triple[] D;
real t, x;
int i, j;

t = (1 + sqrt(5))/2;

D[1] = (1,1,1);
D[2] = (1,1,-1);
D[3] = (1,-1,1);
D[4] = (1,-1,-1);
D[5] = (-1,1,1);
D[6] = (-1,1,-1);
D[7] = (-1,-1,1);
D[8] = (-1,-1,-1);
D[9] = (0,t,1/t);
D[10] = (0,t,-1/t);
D[11] = (0,-t,1/t);
D[12] = (0,-t,-1/t);
D[13] = (1/t,0,t);
D[14] = (1/t,0,-t);
D[15] = (-1/t,0,t);
D[16] = (-1/t,0,-t);
D[17] = (t,1/t,0);
D[18] = (t,-1/t,0);
D[19] = (-t,1/t,0);
D[20] = (-t,-1/t,0);

draw(D[13]--D[1]--D[17],gray(0.7));
draw(D[1]--D[9]--D[5],gray(0.7));
draw(D[15]--D[5]--D[19],gray(0.7));
```



```
draw(D[9]--D[10]--D[2],gray(0.7));
draw(D[10]--D[6],gray(0.7));
draw(D[3]--D[13]--D[15]--D[7]--D[20]--
D[19]--D[6]--D[16]--D[14]--D[2]--D[17]
--D[18]--cycle);
draw(D[3]--D[11]--D[7]);
draw(D[11]--D[12]--D[4]--D[18]);
draw(D[4]--D[14]);
draw(D[12]--D[8]--D[16]);
draw(D[8]--D[20]);
draw(D[13]--D[7]--D[12]--D[18]--cycle,
red);
draw(D[9]--D[19]--D[16]--D[2]--cycle,
red);
draw(D[13]--D[9], red);
draw(D[7]--D[19], red);
draw(D[12]--D[16], red);
draw(D[18]--D[2], red);
[/asy]
```

We can draw 3D diagrams in Asymptote. For some reason, `unitsize` doesn't work with 3D diagrams, so you must use the `size` command. Also, `opacity` doesn't work so well on our website.

We can use the website asymphost.xyz to produce interactive 3D diagrams in our scripts and homework. Examples can be found here:
https://artofproblemsolving.com/crypt/document/411/1539 (search for "square tiles")
https://artofproblemsolving.com/crypt/collection/57/homework/17 (problem 38659)

(When using asymphost.xyz for interactive 3D diagrams, remember to uncheck "Delete after 1 Hour" and check "Modify HTML for sandboxed embedding".)

Code for the Platonic Solids can be found here:
https://artofproblemsolving.com/crypt/document/57/20641

Other examples of 3D diagrams can be found in the following documents:
https://artofproblemsolving.com/crypt/document/186/197
https://artofproblemsolving.com/crypt/document/186/199
https://artofproblemsolving.com/crypt/document/186/202
https://artofproblemsolving.com/crypt/document/186/787
https://artofproblemsolving.com/crypt/document/186/788

https://artofproblemsolving.com/crypt/beamer/213/11949
https://artofproblemsolving.com/crypt/beamer/213/12014
https://artofproblemsolving.com/crypt/beamer/213/12117
https://artofproblemsolving.com/crypt/beamer/213/12159

https://artofproblemsolving.com/crypt/beamer/214/11938

https://artofproblemsolving.com/crypt/beamer/307/11764

https://artofproblemsolving.com/crypt/collection/307/homework/28 (Problem 41741)

Self-intersections

Asymptote has many nice intersection-counting functions, but none of them count how many times a path crosses itself. Here's a quick and dirty one I (= KB) wrote:

```
real[][] selfintersections(path g, real fuzz=0.1) {
  real[][] meets;
  for (real[] meet: intersections(g, g)) {
    if (abs(meet[1] - meet[0]) >= fuzz)
      meets.push(meet);
  }
  return meets;
}
```

It counts how many times g intersects a copy of itself, and then only keeps the intersections that happen at different times along g. The parameter fuzz is how far away two times have to be to count as a non-trivial self-intersection; I've found that it can be as small as 0.000000001 without really affecting the outcome.

Helpful functions

String scalar multiplication

Returns a string made up of n copies of s, as with Python string multiplication.

```
string operator *(string s, int n) {
  string ss = "";
  for (int i: sequence(n)) {
    ss += s;
  }

  return ss;
}
```

Random angle and vector generation

Returns a random angle in [0,360). Arbitrarily many arguments can be given, in which case they are interpreted as quadrants from which to choose the angle.

```
real randangle(...real[] quadrants) {
  if (quadrants.length==0) {
    return 360*unitrand();
  }

  return map(new real(real q) { return 90*((q-1)+unitrand()); },
quadrants)[rand(0,quadrants.length-1)];
}
```

Returns a random unit vector as a pair. Arbitrarily many arguments can be given, in which case they are interpreted as quadrants from which to choose the angle.

```
pair randdir(... real[] quadrants) {
  if (quadrants.length==0) {
    return dir(randangle());
  }

  return dir(randangle(...quadrants));
}
```

Getting the center and radius of a circle

These are for computing the center and radius of a circle that was drawn using a function such as `incircle` or `circumcircle`, both of which are defined in `olympiad.asy`.

```
pair center(path c) {
  return midpoint(point(c, 0)--midpoint(c));
}

real radius(path c) {
  return length(center(c)-point(c, 0));
}
```

Converting Asymptote units to centimeters

Asymptote diagrams are drawn in PostScript units, I (KB) think. The function `tocm` below takes a length in Asymptote and converts it to a length in cm.

```
real tocm(picture p=currentpicture, real l) {
  return l*p.xunitsize/1cm;
}
```

This is useful for things like the `underbrace` function below, which draws a LaTeX \underbrace of the desired length in Asymptote units.

```
Label underbrace(picture p=currentpicture, real l, string s="") {
  return "$\underbrace{\hspace{"+(string)tocm(p,
l)+"cm}}_{\textstyle\text{"+s+"}}$";
}
```

(For comparison, previously the only way to get an underbrace to be of the desired length was to eyeball its length in cm, and then iteratively make incremental edits and recompile.)