LaTeX Workshop 5:
Making Graphics with Ti*k*Z

Dan Parker and David Schwein*

19 October 2014

Ti*k*Z is a package for TeX/LaTeX for drawing figures, diagrams, and images.[1] It's comprehensive, and extremely powerful, and capable of doing pretty much anything you could want it to. My goal for this next hour is to show you some of the basics of Ti*k*Z, so that you can get started making figures. Before I begin, I'll try and convince you that's its worth knowing, by showing you some of the figures you can create with it. (See the examples file on the LaTeX workshop website.)

Ok, let's get started.

How do you use Ti*k*Z?

First, include the package: `\usepackage{tikz}`.

Now, to actually create a figure, begin the environment '`tikzpicture`'. If you're going to be putting this figure into a larger document, you'll probably want to surround with a 'figure' environment so that it floats appropriately, and so you can give it a caption and label.

One thing that makes Ti*k*Z a little confusing at first is that it has an entirely different syntax than LaTeX, and the Ti*k*Z code you'll be writing won't necessarily look like anything you've seen in LaTeX before.

Once you have the environment set up, you start drawing. The basic way to draw something in Ti*k*Z is with a command like:

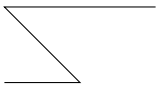
```
\draw (0, 0) -- (1, 0) -- (0, 1) -- (1, 1) -- (2, 1);
```

Commands will usually start with '`\draw`' and then have a series of coordinates with instructions between them, ending with a semicolon. Ti*k*Z has a nice shorthand for creating pictures with a single command. I can write '`\tikz \draw ...;`' instead of beginning and ending an environment. We'll use this shorthand to spare typing whenever possible. Here's an example:

`\tikz \draw (0, 0) -- (1, 0) -- (0, 1) -- (1, 1) -- (2, 1);` produces

*These notes are based on a set of notes by Spencer Gordon.

[1] See the Ti*k*Z documentation, at `www.texample.net/media/pgf/builds/pgfmanualCVS2012-11-04.pdf`.



and is exactly the same as

```
\begin{tikzpicture}
\draw (0, 0) -- (1, 0) -- (0, 1) -- (1, 1) -- (2, 1);
\end{tikzpicture}
```

You can read the previous command as saying 'Draw a path starting at (0, 0) then connected by a straight line to (1, 0) then connect that with a straight line to (1, 0)', and so on ...

The coordinates here are in an implicit cartesian plane, as familiar from high school algebra and calculus. The only difference is that the grid is implicit, and not displayed with your figure.

Two interesting things to observe: First, the default coordinate system is has the positive x-axis pointing right, and the positive y-axis pointing up, and one unit corresponds to one centimeter. If you want to work with different units, you can simply add the units to your coordinates, like so: '`(2in,0.3pt)`'. It's pretty straightforward to resize everything after you've made an image, so it's probably a good idea to ignore absolute sizes of things when creating an image, and only afterward, try to make things fit in a certain space.

Second, Ti*k*Z takes care of computing the size of your figure and making sure that your entire figure is visible in the '`tikzpicture`' so you also don't have to worry about any part of your figure getting cut off.

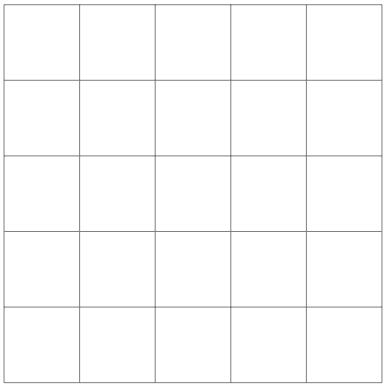
Finally, for those of you who don't have experience with computer graphics programming, you'll have to get used to thinking in terms of coordinates when creating your diagrams, which might be a challenge at first, but Ti*k*Z can help you out.

Another available command that will almost certainly be useful when planning out a figure is '`\grid`'.

Here's an example:

```
\tikz \draw[help lines] (0, 0) grid (5, 5);
```

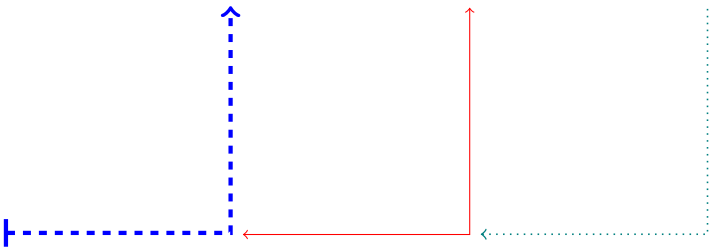
Let's take a look at what this generates.



There are two differences between this draw command and the previous one. In the previous one, we used the '`--`' instruction to say draw a straight line between the previous and following coordinates. Here we are using the '`grid`' instruction to draw a grid in the rectangle with corners at $(0, 0)$ and $(5, 5)$.

The other new feature is the option given to the draw command, '`help lines`'. This is an example of a *style*, which is a collection of different options, each of which controls a parameter used when drawing, such as line width, line color, line style, coordinate transformations, line start and end decorations, and many many more. See the manual (available online for the full list.) A style is just a collection of settings that can be invoked at once. In this case, '`help lines`' just sets the line width to thin, and makes the line color a lighter gray.

To give you a sense of the possibilities available, the following figures all have the very same path drawn, just with different options.

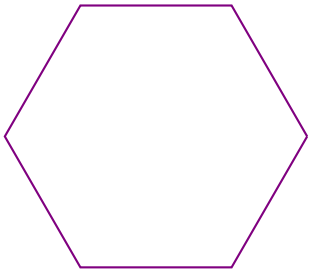


I'll let you play around with what you've seen in a second, but I just want to present two alternative ways of specifying coordinates.

First, polar coordinates. For example, I can write

```
\tikz \draw[color=violet, thick] (0:2) -- (60:2) -- (120:2) -- (180:2)
  -- (240:2) -- (300:2) -- (0:2);
```

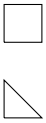
to draw a hexagon using polar coordinates. To use polar coordinates, you specify an angle (in degrees) followed by a `:`, followed by a radius. You can freely mix polar and cartesian coordinates in your figures.



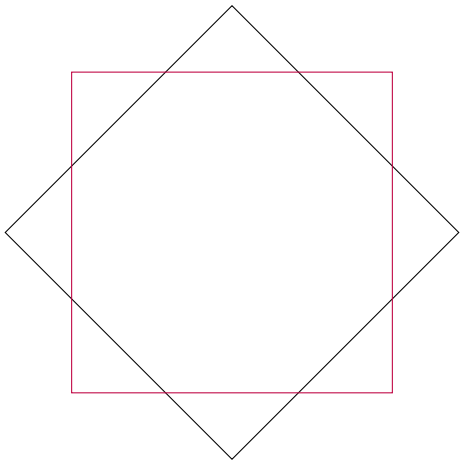
Finally, you can specify relative coordinates with a neat special syntax. If you want to say 'the point 2 units above the previous point, you can write: '`++(2, 0)`'. Here's an example. Let's say I want to draw a square 3 units above a triangle. Instead of translating all the coordinates in my head when drawing the square, I can just do the following:

```
\begin{tikzpicture}[scale=0.5]
\draw (0, 0) -- (0, 1) -- (1, 0) -- (0, 0);
\draw (0, 2) -> ++(1, 0) -> ++(0, 1) -> ++(-1, 0) -> ++(0, -1);
\end{tikzpicture}
```

This makes the figure



Ok, now an exercise. I would like you to try and recreate this figure:



Now that you have a decent grasp on the basic functionality needed to make line drawings, we'll move on to some more sophisticated functionality.

I can draw a curve as follows:

```
\tikz[scale=0.4] \draw (0, 0) to [out=135, in=100] (1, 1);
```

The `out` option specifies the angle at which the curve should leave the starting point and the `in` option specifies the angle at which the curve should arrive at the end point.

To draw a circle,

```
\tikz \draw[orange, semithick] (1, 1) circle [radius=0.5];
```

The first point is the center in this case.

To draw a rectangle,

```
\tikz[scale=0.4] \draw[brown] (0, 0) rectangle (4, 2);
```

Here the first point is one corner of the rectangle and the second point is the opposite corner of the rectangle.

There are many many more handy shortcuts that Ti*k*Z provides for common shapes and patterns, and you should take a look at the manual to find out more, but I would like to move on.

You may not be surprised at this point to learn that the draw command I've been using this whole time is actually shorthand for yet another command, which is the actual command underlying all these different path commands, namely '`\path`'.

The command I used for the circle just a minute ago was equivalent to the following command, which is exactly what Ti*k*Z expands the above command into.

```
\tikz \path[draw=orange, semithick] (1, 1) circle [radius=0.5];
```

Just like there is a '`draw`' option to the `path` command, there is also an option to `fill` a path, which can be given in the same way, so I can fill the same circle as above with:

```
\tikz \path[draw=orange, fill=green, thick] (1, 1) circle [radius=0.5];
```

I'm actually both filling and drawing this circle, and with two different colors. I can choose to just fill the circle by using a shortcut

```
\tikz \fill[green] (1, 1) circle [radius=0.5];
```

or by using the '`none`' value instead of color as the value for the '`draw`' option.

Rather than go through any more small isolated pieces of functionality, I'm going to try and work through a larger example, and introduce some new concepts as needed.

Here's the diagram I want to create:

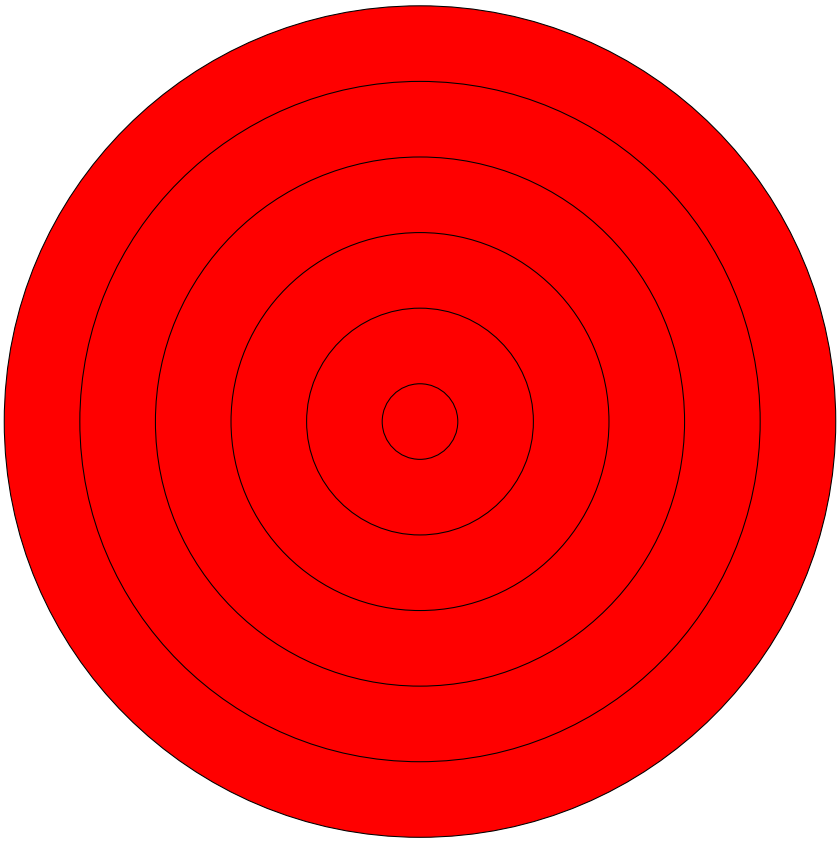
Looking at this, it seems clear that writing this by hand using the `\draw ... circle ...;` would be really annoying, and if I wanted to adjust it at all, I'd have to make a lot of changes so everything lined up. So Ti*k*Z offers a really nice feature, which will be familiar to those of you with some programming experience: For-loops. Here's the code I used to make this bullseye-like figure:

```
% Bullseye
\begin{tikzpicture}
\foreach \x in {5,4,...,0}{
  \draw[fill=red,draw=black] (0,0) circle[radius=\x+0.5];
  \draw[fill=white,draw=black] (0,0) circle[radius=\x];
};
\end{tikzpicture}
```

The interesting part is the loop, which begins with `\foreach \x in {5,4,...,0}`. This first part means, take whatever code I have below, and copy and paste it once for each item in the list going $5, 4, \ldots, 0$ and have `\x` be the current item in the list each time. Note that we used the value of `\x` in our for-loop to adjust the radius of each circle that we draw, and that we do simple arithmetic to make the red circle bigger than the white circle.

Here's a question for you: What would happen if we switched the order in which we drew the two circles?

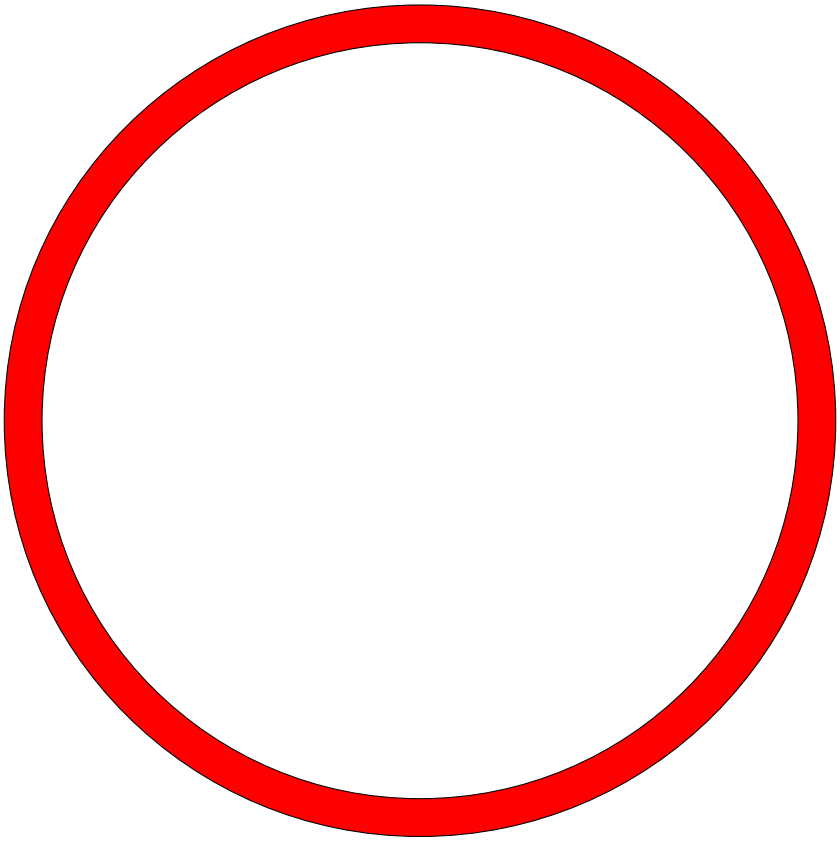
Here's the answer:



What happens is the red circles are drawn after the white circles, and go right on top of the white circles, so the white circles can't be seen.

Another question: What would happen if we went back to the original order for drawing the circles, but instead of counting down, counted up?

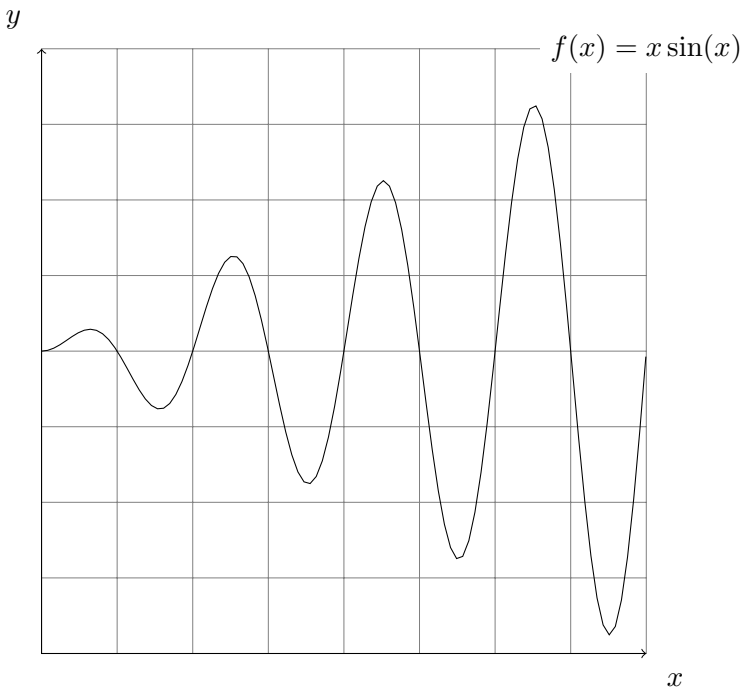
Here's the answer:



The last two circles cover all of the previous circles, so they can't be seen.

Basically, in Ti*k*Z, every new part of a drawing goes on top of everything else, so you need to make sure to pay attention to the order in which you draw things to ensure that they don't overwrite each other incorrectly. Sometimes, this is actually very handy, since you'll want to draw something on top of part of your drawing, and it's nice to not have to worry about specifying what's behind what, when all you want is newer stuff to be on top of older stuff.

Ok, I'll do one more example, and then give you exercises to work on. Here's what I want to produce:



I make this diagram with a few commands. First, I use

```
\node[label=above left:$y$] (topLeft) at (0, 2) {};
\node[label=below right:$x$] (botRight) at (2, 0) {};
```

This defines two new coordinates, which can be referred to by `(topLeft)` and `(botRight)` respectively, and adds labels to the image with positions relative to the coordinates. The `{}` contains any text associated with the node. In this case, we don't want the nodes themselves to have text.

Generally, nodes are like coordinates in that they have a location and can be referred to later, but they can also have their own appearance and text content, though I'm not using either of those features in this case.

To give a node a label, use the `label` option and provide a location relative to the node, and then the label text. I find it pretty confusing to talk about node labels, since nodes usually contain text themselves, and are often used as labels, but remember that labels are always associated with a node, while nodes can be inserted pretty much anywhere in a path.

Next, I draw the grid using the grid command, specifying that I want horizontal lines to be drawn at every `ystep` units, and vertical lines to be drawn at every `xstep` units, and that I want them to be drawn with very thin, gray lines.

Now, I draw the axes with a single path, starting from the center of the `topLeft` node, and going to the origin, then the center of the `botRight` node.

The next command actually draws the curve. For this to work, you'll need to add the command `\usetikzlibrary{calc}` to the preamble. This tells Ti*k*Z to load the `calc` library.

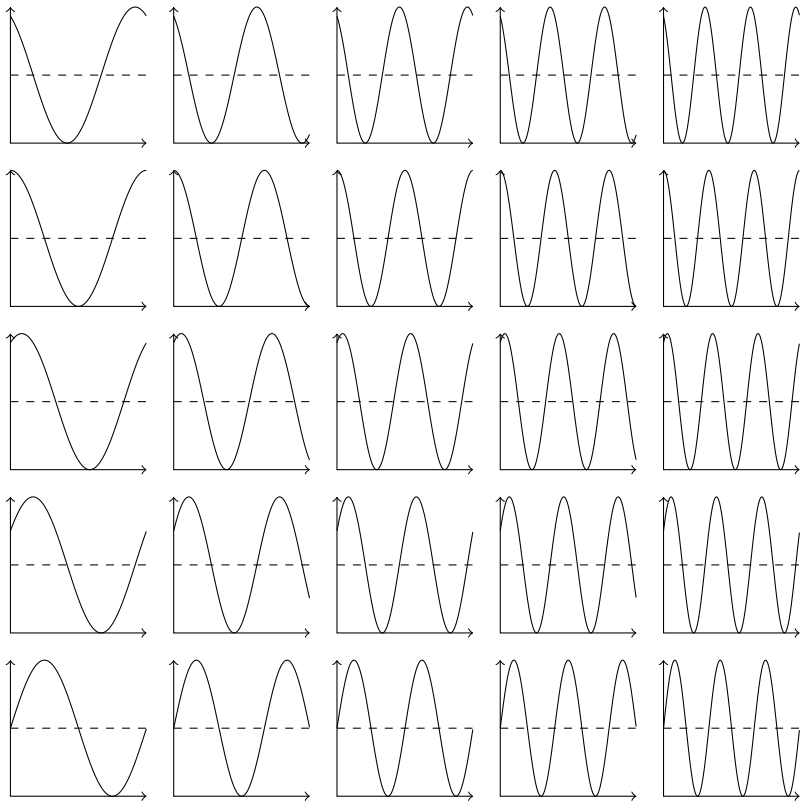
```
\draw[domain=0:2,fill=none,samples=100,draw=black]
  plot (\x, {\x*sin(2*360*\x)/2 + 1});
```

This command uses the plot path instruction, which creates a path from the plot of a function. I specify the function with `(\x, {\x*sin(2*360*\x)/2 + 1})`. Note that you have to use `\x` as your variable in the plot instruction unless you use an option to select a different variable name. Also, the function must be surrounded by curly braces, and can use any of many builtin functions. In this example we only use sine, but there are many builtins, which can be found in the documentation.

Two other things to note: We use the `domain` option to define the domain of coordinates over which we are plotting the function. If you want to plot a function over a domain that's different from the coordinates in which you putting the plot, you'll have to use some other options to shift things around, but this will work for the time being.

The way Ti*k*Z actually plots the function is by taking samples of the function value at various points in the domain and interpolating the missing values, and then trying to smooth out the curve, so the more complex your function the more samples you'll need to get something that looks right. In this case, I use the `samples` option to specify that Ti*k*Z should use 100 samples.

I don't have time to explain this in its entirety, but at this point you should be almost able to create some pretty cool complex figures like:



```
\begin{tikzpicture}[scale=1.8]
\foreach \x in {0,1,...,4} {
  \foreach \y in {0,1,...,4} {
    \draw[shift={+(1.2*\x,1.2*\y)}, <->] (1, 0) -- (0, 0) -- (0, 1);
    \draw[shift={+(1.2*\x,1.2*\y)}, dashed] (0, 0.5) -- (1, 0.5);
    \draw[shift={+(1.2*\x, 1.2*\y)},
      variable=\t,domain=0:1,smooth,samples=100]
plot (\t, {sin(360*\t*(\x/2+1)+30*\y)/2 + 0.5});
  };
};
\end{tikzpicture}
```