A Ti*k*Z tutorial: Generating graphics in the spirit of TEX

Andrew Mertz & William Slough

Abstract

Ti*k*Z is a system which can be used to specify graphics of very high quality. For example, accurate placement of picture elements, use of TEX fonts, ability to incorporate mathematical typesetting, and the possibility of introducing macros can be viewed as positive factors of this system. The syntax uses an amalgamation of ideas from METAFONT, METAPOST, PSTricks, and SVG, allowing its users to "program" their desired graphics. The latest revision to Ti*k*Z introduces many new features to an already feature-packed system, as evidenced by its 560-page user manual. Here, we present a tutorial overview of this system, suitable for both beginning and intermediate users of Ti*k*Z.

1 Introduction

PGF, an acronym for "portable graphics format", is a TEX macro package intended for the creation of publication-quality graphics [16]. The use of PGF requires its users to adopt a relatively low-level approach to the specification of a graphical image. To sidestep the associated verboseness of this level, a front-end named Ti*k*Z is also available for use. The syntax of Ti*k*Z borrows ideas from METAFONT, METAPOST, PSTricks, and SVG: its ultimate aim is to simplify the task of specifying graphics.

Users seeking authoritative documentation of Ti*k*Z are well advised to consult its thorough reference manual [15]. Those with an interest in discovering the possibilities of this system may wish to peruse the *TEXample* website [3], a repository with many examples of graphics created with Ti*k*Z accompanied by the associated code. First-time users ready to "take the plunge" with Ti*k*Z may benefit from introductory-level information found in [11] and [18], for example.

Our current purpose is to expand on our earlier treatment of the use of Ti*k*Z. In the intervening years since [11] appeared, a number of developments have taken place. For example, many new capabilities — such as the inclusion of a mathematics engine — are now available within Ti*k*Z. Another interesting development is the appearance of "third-party" packages which extend Ti*k*Z to specialized domains, such as the creation of combinatorial graphs or electrical circuits. A third development is the appearance of other software, such as dynamic geometry systems, which can export in PGF and/or Ti*k*Z formats.

```
\documentclass{article}
...
\usepackage{tikz}
% Optional libraries:
\usetikzlibrary{arrows, automata}
...
\begin{document}
   ...
   \begin{tikzpicture}
   ...
   \end{tikzpicture}
   ...
\end{document}
```

Listing 1: Layout of a document which uses Ti*k*Z.

2 Some Ti*k*Z fundamentals

Ti*k*Z provides support for various input formats, including plain TEX, LATEX, and ConTEXt. The requirements for each of these are fairly similar, so we focus on just one of these, LATEX, for simplicity.

Listing 1 illustrates the layout for a LATEX document containing a number of Ti*k*Z-generated graphics. In the preamble, the `tikz` package is specified, optionally followed by one or more Ti*k*Z libraries. Each graphic to be generated is specified within a `tikzpicture` environment.

Exactly which libraries are needed depend ons the requirements of the images being generated. For the simplest diagrams, no library is required. Other situations which utilize features from the various Ti*k*Z libraries require their explicit mention. For example, in listing 1, the `arrows` and `automata` libraries are referenced to gain access to a variety of arrow tips and obtain the ability to draw finite-state automata.

Specifications for the desired graphic appear within the `tikzpicture` environment. One of the simplest commands available is the `\draw` command which, when coupled with the `--` operator, joins points with straight line segments. This syntax is inspired by METAFONT and METAPOST. Figure 1 shows how a diamond can be drawn with a single `\draw` command, joining the four points on the x and y axes one unit from the origin. Since no dimensional units are provided, the default, one centimeter, is used. The `cycle` keyword is shorthand for the first point on the path. The `\fill` command used here fills the interior of a circle centered at $(0, 0)$ with a one point radius.

In this first example, Cartesian coordinates, illustrated in Figure 2, have been used. An alternate approach is to use polar coordinates, as shown in Figure 3. Angles are specified in degrees, although the inclusion of an `r` suffix can be used to indicate radian

```
\begin{tikzpicture}
  \draw (1,0) -- (0,1)
        -- (-1,0) -- (0,-1) -- cycle;
 \fill (0,0) circle (1pt);
\end{tikzpicture}
```

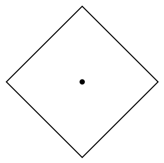


Figure 1: Drawing a diamond with a closed path, using points specified with the usual Cartesian coordinate system. The origin is shown with a filled circle.

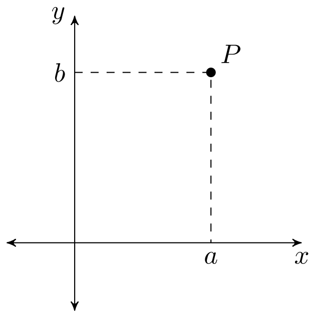


Figure 2: Using Cartesian coordinates, the point P is denoted (a, b).

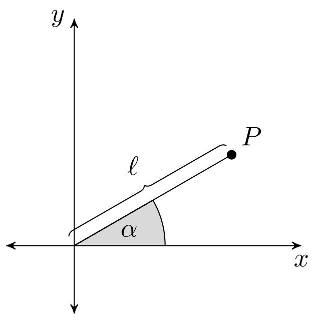


Figure 3: Using polar coordinates, the point P is denoted $(\alpha : \ell)$.

measure. Thus, the point with Cartesian coordinates $(0, -1)$ can be denoted in Ti*k*Z in a number of ways: `(0,-1)`, `(270:1)`, and `(3/2 * pi r:1)`. In the last case, in addition to specifying radian measure we are making use of the arithmetic expression capabilities within Ti*k*Z to compute the value of $\frac{3}{2}\pi$.

A variety of options can influence the outcome of the `\draw` command. These options control such things as the pen color and width, whether or not to fill or shade the interior, and what line style is to be used — solid or dashed, for instance. These options are enclosed within square brackets and serve to modify the `\draw` command. Figure 4 provides an example of three `\draw` commands with a few options in effect. Multiple options, separated by commas, may appear.

So far, we have seen how the Ti*k*Z `--` operator can be used to draw line segments. There are other operators, including `grid`, `circle`, `rectangle`, and `arc`, which can be used to draw other shapes. For `grid` and `rectangle`, two opposing points of the desired shape are given. A circle is obtained with `circle`, which takes a center point and the radius; an ellipse requires a center point and two radii. A circular arc is specified by giving a starting point and three values: two angles and a radius. Starting from the given point, an arc with the specified radius which sweeps between the two angles is drawn. Figure 5 shows a few examples of these operators.

The `\coordinate` command provides a handy mechanism to name a point. This is especially useful if the point is to be referenced more than once, since its definition is only needed once and referred to by name thereafter. Even in cases where only a single reference is needed, readability can be improved with the introduction of names. The Cartesian point (a, b) can be given the name `P` using the command

`\coordinate (P) at (`a, b`);`

Similarly,

`\coordinate (P) at (`$\alpha : \ell$`);`

names a point with polar coordinates. Once the coordinate P has been defined, it can appear in a subsequent `\draw` command as `(P)`, whereupon its defined value is used. For example, the diamond of Figure 1 can also be obtained with the code shown in Listing 2:

```
\begin{tikzpicture}
   % Define four points
   \coordinate (P0) at (1,0);
   \coordinate (P1) at (0,1);
   \coordinate (P2) at (-1,0);
   \coordinate (P3) at (0,-1);

   % Draw the diamond
   \draw (P0)--(P1)--(P2)--(P3)--cycle;
\end{tikzpicture}
```

Listing 2: Drawing a diamond using named points.

The `\node` command extends the idea of a coordinate by associating shapes, such as circles and rectangles, and labels with a specified location. For example,

```
\node (N) at
   (0,0) [draw, shape=circle] {$v_0$};
```

```
\begin{tikzpicture}
  \draw [thick, dotted]
     (1,0) -- (0,1) -- (-1,0) -- (0,-1) -- cycle;
  \draw [ultra thick]
     (0:1.5) -- (90:1.5) -- (180:1.5) -- (270:1.5) -- cycle;
  \draw [dashed, thick, color=gray]
     (0 r:2) -- (pi/2 r:2) -- (pi r:2) -- (3/2 * pi r:2) -- cycle;
\end{tikzpicture}
```

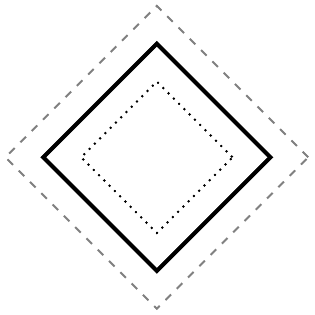


Figure 4: Drawing diamonds using Cartesian and polar coordinates, using angles specified with degrees and radians. Options to `\draw` have been introduced to change the style of the line segments.

```
\begin{tikzpicture}[scale=2/3]
   \draw (0,0) grid (4,4);
   \draw (2,2) circle (2);
   \draw (2,2) circle (1 and 2);
   \draw (0,0) rectangle (4,-1);
   \draw (0,4) arc (90:270:2);
   \fill (0,0) circle (3pt);
\end{tikzpicture}
```

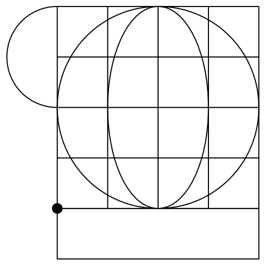


Figure 5: A sampling of Ti*k*Z path operators. As before, the origin is shown with a small filled circle. The `scale` option applied to the entire environment is used to resize the image.

defines a node named `N` which is to be placed at the origin along with a surrounding circular shape and an interior label of v_0. Like a coordinate, its name (in this case, `N`) can appear in subsequent commands. Since nodes have associated shape information, lines drawn between them don't extend to the center point, but stop at the perimeter of the shape.

If multiple nodes are defined within a graphic, it is convenient to use the `\tikzstyle` command to provide options which apply to all defined nodes. For example,

```
\tikzstyle{every node}=
   [draw,shape=circle]
```

indicates all subsequent nodes are to be drawn with a circular shape. This would allow our previous command to be abbreviated as:

```
\node (N) at (0,0) {$v_0$};
```

A complete example with nodes is shown in Figure 6.

Another capability of Ti*k*Z is the `\foreach` command, which provides a way to introduce looping actions. Listing 3 shows yet another way to obtain the diamond figure. A loop with four iterations, one for each edge of the diamond, is established. A subtlety with the parsing involved in this example requires curly braces to be used in order to group the expression corresponding to the mathematical entity $(i+1)\frac{\pi}{2}$.

```
\begin{tikzpicture}
   \foreach \i in {0,...,3}
   {
      \draw (\i * pi/2 r:1) --
            ({(\i + 1) * pi/2} r:1);
   }
\end{tikzpicture}
```

Listing 3: Drawing a diamond with a `\foreach` loop. Each iteration draws one edge of the diamond.

3 The mathematical engine

Ti*k*Z has access to a mathematical engine which provides arithmetic and relational operators, as well as a host of mathematical functions one typically encounters in a traditional programming language. The math engine can also be used independently of Ti*k*Z.

The arithmetic and relational operators are `+`, `-`, `*`, `/`, `^`, `<`, `==`, and `>`, which may appear within infix expressions in the usual manner. Here are the functions the Ti*k*Z mathematical engine supports, as of version 2.0:

`mod`	`max`	`min`	`abs`	`round`	`floor`
`ceil`	`exp`	`ln`	`pow`	`sqrt`	`veclen`
`pi`	`r`	`rad`	`deg`	`sin`	`cos`
`tan`	`sec`	`cosec`	`cot`	`asin`	`acos`
`atan`	`rnd`	`rand`			

```
\begin{tikzpicture}
   % Default actions for each node
   \tikzstyle{every node}=[draw, shape=circle];

   % Define and draw five nodes
   \node (v0) at   (0:0) {$v_0$};
   \node (v1) at   (0:2) {$v_1$};
   \node (v2) at  (90:2) {$v_2$};
   \node (v3) at (180:2) {$v_3$};
   \node (v4) at (270:2) {$v_4$};

   % Draw radial edges
   \draw (v0) -- (v1)  (v0) -- (v2)
         (v0) -- (v3)  (v0) -- (v4);
\end{tikzpicture}
```

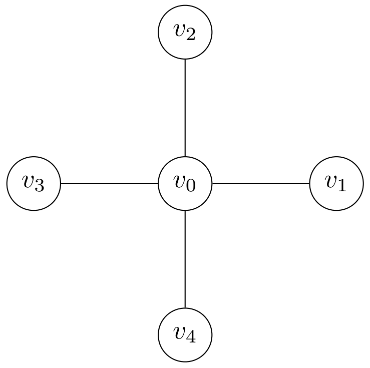


Figure 6: Using nodes.

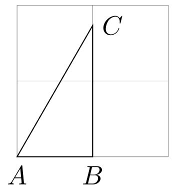


Figure 7: A 30–60–90 triangle.

As shown earlier, coordinates can be specified using arithmetic. For example:

```
\draw (0,0) -- (360.0 / 7.0 * 3.0 : 1);
```

Coordinates can also be defined in terms of mathematical functions. For example, Figure 7 shows a 30–60–90 triangle where one of the coordinates has been defined as follows:

```
\coordinate [label=right:$C$] (C) at
   (1, {sqrt(3)});
```

Note that curly braces distinguish the case where parentheses are used mathematically and the case where they denote a named coordinate.

Points can also be computed in terms of other points. Basic calculations involving coordinates such as addition, subtraction, and scaling can be performed. For such coordinate calculations, the `calc` library is required:

```
\usetikzlibrary{calc}
```

The desired coordinate calculations are then enclosed within `$` symbols. Examples of these types of calculations are illustrated in Figure 8.

Coordinate calculations can also be used to compute points that are partway between two points. The coordinate calculation

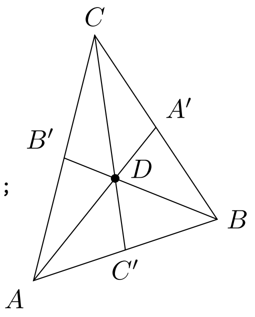
```
($(A)!0.25!(B)$)
```

becomes the point that is 25% of the way along the segment from `A` to `B`. Figure 9 shows examples of coordinate calculations being used to compute the medians of a triangle.

It is sometimes useful to be able to compute the distance between two points. Although Ti*k*Z does not currently have a direct way to do this, it is possible with the `let` operation and `veclen` function. The `let` operation allows coordinates to be defined that are available for just one path. Figure 10 demonstrates a simple use of the `let` operation; note that the macros used to name points must begin with a `p`. The `let` operation also allows extraction of the x and y components of a point. An example of this feature is given in Figure 11, which also uses the function `veclen` to compute the distance between two points.

For example, the centroid of the triangle of Figure 9 can be determined by finding the point of intersection of any two of its medians. This point, labeled D below, can be obtained with the following Ti*k*Z statement:

```
\coordinate (D) at
   (intersection of
   A--Aprime and C--Cprime);
```



4 A few Ti*k*Z libraries

Libraries, optionally loaded in the preamble section of a LaTeX document, extend the capabilities of Ti*k*Z and simplify some kinds of tasks. There are currently more than a dozen different libraries available, providing users with tools to create specific types of diagrams, such as finite-state automata, calendars,

```
\begin{tikzpicture}
   \draw[help lines] (0,0) grid (4,2);
   \coordinate[label=below:$A$] (A) at (2, 1);

   % Three points determined from a reference point
   \coordinate[label=above:$B$] (B) at ($2*(A)$);
   \coordinate[label=above:$C$] (C) at ($(A) + (-1,1)$);
   \coordinate[label=below:$D$] (D) at ($(A) - (-1,1)$);

   % Mark each point with a filled circle
   \fill (A) circle (2pt)  (B) circle (2pt)
         (C) circle (2pt)  (D) circle (2pt);
\end{tikzpicture}
```

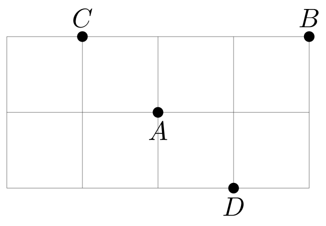


Figure 8: Using coordinate calculations.

```
\begin{tikzpicture}
   % Three vertices of a triangle
   \coordinate[label=below left:$A$] (A) at (0,0);
   \coordinate[label=right:$B$]      (B) at (3,1);
   \coordinate[label=above:$C$]      (C) at (1,4);

   % Find the midpoints
   \coordinate[label=above right:$A'$] (Aprime) at ($(B)!0.5!(C)$);
   \coordinate[label=above left:$B'$]  (Bprime) at ($(A)!0.5!(C)$);
   \coordinate[label=below:$C'$]       (Cprime) at ($(A)!0.5!(B)$);

   % Draw the triangle and its three medians
   \draw (A) -- (B) -- (C) -- cycle
         (A) -- (Aprime) (B) -- (Bprime) (C) -- (Cprime);
\end{tikzpicture}
```

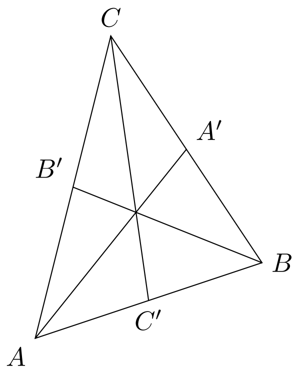


Figure 9: Using coordinate calculations to determine the medians of a triangle.

```
\begin{tikzpicture}
   \draw[help lines] (0,0) grid (2,2);
   \draw let \p1=(0,0), \p2=(1,2), \p3=(2,0) in
      (\p1) -- (\p2) -- (\p3);
\end{tikzpicture}
```

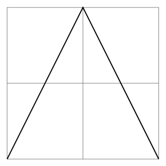


Figure 10: A simple use of the `let` operation.

```
\begin{tikzpicture}
   \coordinate [label=below:$A$] (A) at (0.5,0.75);
   \coordinate [label=above:$B$] (B) at (1,1.85);
   \draw (A) -- (B);
   \draw (A) let \p1 = ($(B) - (A)$) in circle ({veclen(\x1,\y1)});
\end{tikzpicture}
```

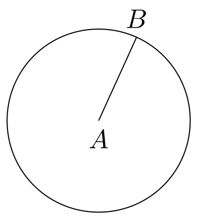


Figure 11: Using `let` to compute the distance between two coordinates.

mind maps, Petri nets, and entity-relationship diagrams. There is even one a bit more whimsical in nature, which typesets a do-it-yourself pattern for a dodecahedron, where the twelve faces can be given arbitrary content. In this section, we examine two of these libraries: `automata` and `mindmap`.

The `automata` library is used to create diagrams of finite-state automata and Turing machines. The details of one such automaton is given in Figure 12.

For each state of an automaton, the following attributes need to be specified: where on the page it should appear, what text is needed for the state name, and whether or not it is an initial and/or an accepting state. For the current example, this infor-

```
\usetikzlibrary{automata, positioning}
...
\begin{tikzpicture}[>=latex, shorten >=1pt,
                    node distance=0.75in, on grid, auto]
   % Vertices of automaton
   \node[state, initial]   (q0)               {$q_0$};
   \node[state]            (q1) [right=of q0] {$q_1$};
   \node[state, accepting] (q2) [right=of q1] {$q_2$};
   \node[state]            (q3) [above=of q1] {$q_3$};

   % Edges of automaton
   \path[->] (q0) edge [loop below] node {0} (q0)
             (q0) edge              node {1} (q1)
             (q1) edge [loop above] node {1} (q1)
             (q1) edge [bend left]  node {0} (q2)
             (q2) edge [bend left]  node {0} (q1)
             (q2) edge [bend right] node[swap] {1} (q3)
             (q3) edge [bend right] node[swap] {1} (q0)
             (q3) edge [loop above] node {0} (q3);
\end{tikzpicture}
```

$$\begin{tikzcd}
 & q_3 \arrow[loop above, "0"] \arrow[dl, bend right, "1"'] & \\
\text{start} \arrow[r] & q_0 \arrow[loop below, "0"] \arrow[r, "1"] & q_1 \arrow[loop above, "1"] \arrow[r, bend left, "0"] & q_2 \arrow[l, bend left, "0"] \arrow[ul, bend right, "1"']
\end{tikzcd}$$

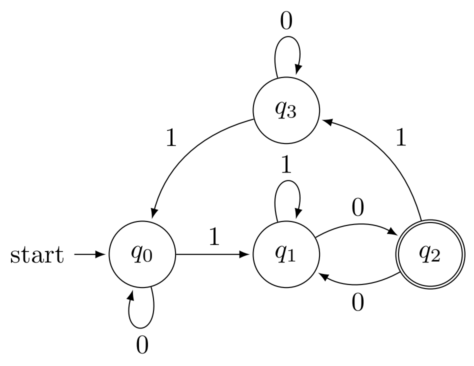


Figure 12: A finite automaton drawn with the Ti*k*Z `automata` library.

mation is provided as a sequence of `\node` commands. Information about initial and accepting states is provided as options to this command; the text appears as an argument. Layout on the page is grid based: the positions of states of the automaton are indicated relative to the location of others.

Each edge of the automaton has a source and destination node, but also requires additional information about typesetting the figure. In particular, edges can be drawn as a straight edge or with a bend; also, the edge label can appear on either "side" of its associated edge. (Imagine "driving" along an edge in the indicated direction. The edge label is either on your left or your right.)

Referring again to Figure 12, note that edge information is given in one extended `\path` command. By default, edges are drawn as straight edges, but this behavior can be modified with `bend` options. The `swap` option changes the default position of the edge label, from left to right.

Options provided to the `tikzpicture` environment specify such things as the type of arrowhead desired, the spacing of nodes on the grid, and the amount of space to leave between an arrowhead and a circular state.

The high-quality output made possible by the `automata` library can be used in conjunction with other software concerned with formal languages and automata theory. One such system is JFLAP [13], which allows users to draw and simulate automata. The creation of state diagrams with JFLAP is easily accomplished with mouse and keyboard interactions. We have implemented a translation utility [9] which converts automata stored in the JFLAP file format to Ti*k*Z format, providing an avenue for typeset output while freeing the user from knowing the underlying Ti*k*Z language details.

The `mindmap` library provides support for drawing hierarchical structures consisting of multicolored, filled shapes with text and annotations. Figure 13 shows an example of a diagram created with this library.

In our example, there is one root node, *bicycle*, with three children: *tandem bicycle*, *mountain bicycle*, and *road bicycle*. The latter child, in turn, has two children of its own. Observe how this hierarchical information is conveyed within the Ti*k*Z code: one root, with three children, and two further child nodes. The `grow` option provides an angle indicating where the child node should appear relative to its parent. For this example, we scaled the entire diagram by 65% to adjust the size of the circles. For this to be done correctly, the `transform shape` option is a crucial requirement.

5 CircuiTi*k*Z

CircuiTi*k*Z [12] is intended for drawing electrical networks and is based on Ti*k*Z, as the name implies. It is inspired by `circuit-macros` [1], a system based on the m4 macro language.[*] However, unlike that system, all processing is performed within the context of Ti*k*Z, so circuits can be edited directly in the TEX source document.

[*] As an aside, `circuit-macros` is capable of producing PGF output.

```
\usetikzlibrary{mindmap}
...
\begin{tikzpicture}[scale=0.65]
\path[mindmap, concept color=black, text=white, transform shape]
   node[concept] {bicycle}
      child[grow=230, concept color=blue!80!black] {
         node[concept]{road bicycle}
            child[grow=-120] {
               node[concept]{time trial bicycle}
            }
            child[grow=-60] {
               node[concept]{road racing bicycle}
            }
      }
      child[grow=180, concept color=red!80!black] {
         node[concept]{mountain bicycle}
      }
      child[grow=120, concept color=red!80!black] {
         node[concept]{tandem bicycle}
      };
\end{tikzpicture}
```

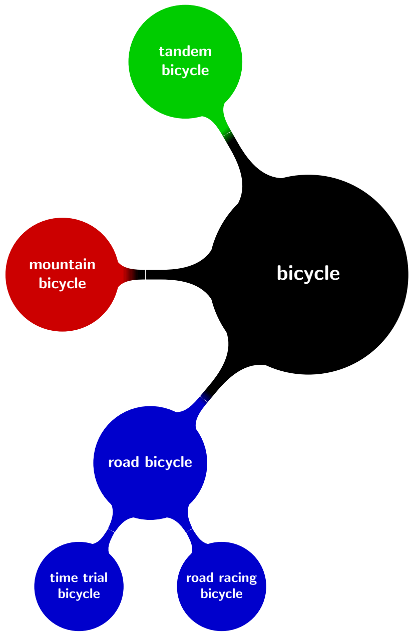


Figure 13: Example output of the `mindmap` library.

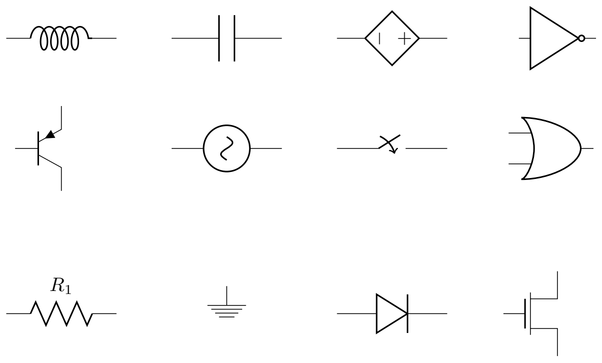


Figure 14: A sampling of circuit symbols available from CircuiTi*k*Z.

Figure 14 shows some of the symbols which are available within CircuiTi*k*Z. Symbols for both American and European electrical conventions are available.

Figure 15 shows how one might draw a diagram for an RLC circuit. As can be seen, its specification uses Cartesian coordinates and a sequence of `\draw` commands. Most of these commands specify a starting and ending point, along with a symbol (such as a resistor) to draw midway between these two points. Observe how electrical connections are not explicitly described as filled circles, but instead use the option `*-*` to indicate the connections at both ends. The grid was included as an aid to understand how the diagram was constructed. The origin for this diagram appears at the lower left corner.

6 Combinatorial graphs via tkz-graph

Combinatorial graphs, as opposed to graphs of functions, are the structures studied in the branch of mathematics known as graph theory. Drawing such graphs is an application made-to-order for Ti*k*Z. However, several other packages, `tkz-graph` and `tkz-berge` [8], are specialized for this task and provide simplifications.

Figure 16 provides a glimpse of the possibilities with the `tkz-berge` package. This package, named in honor of the mathematician Claude Berge, is primarily intended for drawing the well-known graphs in the field. Rather than explore the details of this package, we instead turn our attention to `tkz-graph` which can be used to draw arbitrary graphs.

Figure 17 illustrates a five-vertex graph drawn in three different styles. One attractive feature of this package is that it is very easy to switch from one style to another, primarily by stating the desired style as an option. As a minor complication, when vertex labels appear outside their respective vertices, as in the middle graph of Figure 17, additional information about relative location must be supplied.

Figure 18 provides the details needed to draw this graph in the "normal" style, information conveyed in the `\GraphInit` command. Mathematically, it is unimportant where each vertex appears on the page. However, in order to draw the graph, a location is needed for each of the vertices. Of the various ways allowed to specify these locations, we have chosen to use a Cartesian coordinate system. Each `\Vertex`

```
\usepackage[american]{circuitikz}
...
\begin{circuitikz}
   \draw [help lines] (0,0) grid (6,4);
   \draw (0,0) to [V=V] (0,4);
   \draw (0,4) to (6,4);
   \draw (1,4) node[above] {$I \rightarrow$};
   \draw (6,4) to [C=C] (6,0);
   \draw (4,4) to [L=L, *-*] (4,0);
   \draw (2,4) to [R=R, *-*] (2,0);
   \draw (6,0) to (0,0);
\end{circuitikz}
```

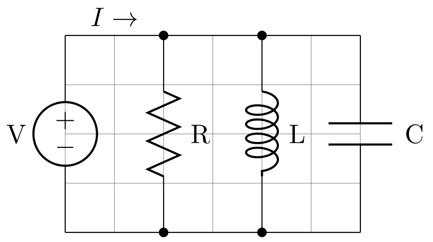


Figure 15: An RLC circuit drawn with CircuiTi*k*Z.

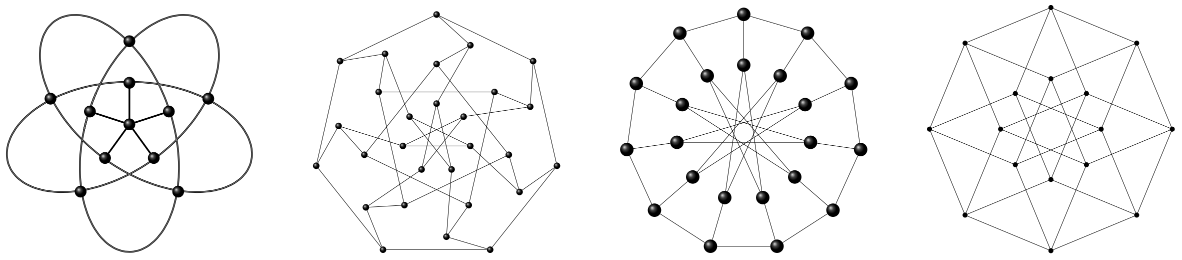


Figure 16: A brief gallery of graphs drawn by `tkz-berge`. Images created by Alain Matthes.

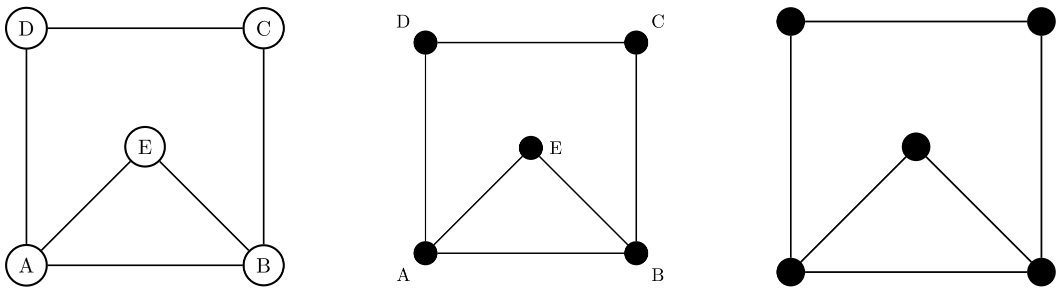


Figure 17: An undirected graph drawn with `tkz-graph` in three different styles: normal, classic, and simple.

command introduces a desired location and a vertex name. Each edge of the graph is specified with an `\Edge` command.

To change the drawing so vertex labels are omitted simply requires a modification to the `GraphInit` command:

```
\GraphInit{vstyle=Simple}
```

To change the appearance of the graph so vertex labels appear next to their respective vertices, two changes are required. First, the vertex style is changed:

```
\GraphInit{vstyle=Classic}
```

To specify each label position, another option is included in each `Vertex` command. So, for example, since vertex C is in the northeast corner relative to its vertex, we use a specification of 45°:

```
\Vertex[x=4, y=4, Lpos=45] {C}
```

A small syntactic detail is worth noting here: in Ti*k*Z, commands are terminated with a semicolon, but no semicolons are required with `tkz-graph`.

The addition of just one line to our example,

```
\tikzset{EdgeStyle/.style={post}}
```

yields a directed graph, as shown in Figure 19. Including the `label` option to each `Edge` command provides a way to describe a weighted, directed graph, as illustrated in Figure 20.

7 Two-dimensional constructions via tkz-2d

The `tkz-2d` package [7] is a collection of macros intended to simplify the construction of diagrams in a two dimensional Cartesian coordinate system. It has particular strengths in the realm of geometric constructions, as it provides higher level abstractions

```
\usepackage{tkz-graph}
...
\begin{tikzpicture}
   % Initialize tkz-graph
   \GraphInit[vstyle=Normal]

   % Vertices
   \Vertex[x=0, y=0] {A}
   \Vertex[x=4, y=0] {B}
   \Vertex[x=4, y=4] {C}
   \Vertex[x=0, y=4] {D}
   \Vertex[x=2, y=2] {E}

   % Edges
   \Edge(A)(B)  \Edge(B)(C)
   \Edge(C)(D)  \Edge(D)(A)
   \Edge(A)(E)  \Edge(E)(B)
\end{tikzpicture}
```

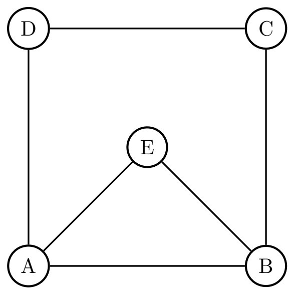


Figure 18: Undirected graph drawn with `tkz-graph`.

```
\usepackage{tkz-graph}
\usetikzlibrary{arrows}
...
\begin{tikzpicture}
   % Initialize tkz-graph
   \GraphInit[vstyle=Normal]
   \tikzset{EdgeStyle/.style={post}}

   % Vertices
   ... same as above ...
\end{tikzpicture}
```

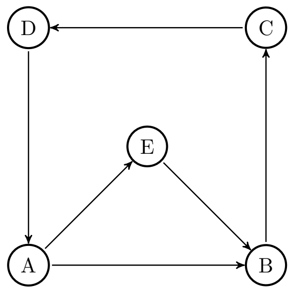


Figure 19: A directed graph drawn with `tkz-graph`.

```
\usepackage{tkz-graph}
\usetikzlibrary{arrows}
...
\begin{tikzpicture}
   % Initialize tkz-graph
   \GraphInit[vstyle=Normal]
   \tikzset{EdgeStyle/.style={post}}

   % Vertices
   \Vertex[x=0, y=0] {A}
   \Vertex[x=4, y=0] {B}
   \Vertex[x=4, y=4] {C}
   \Vertex[x=0, y=4] {D}
   \Vertex[x=2, y=2] {E}

   % Edges
   \Edge[label=$10$](A)(B)   \Edge[label=$5$](B)(C)
   \Edge[label=$20$](C)(D)   \Edge[label=$8$](D)(A)
   \Edge[label=$30$](A)(E)   \Edge[label=$16$](E)(B)
\end{tikzpicture}
```

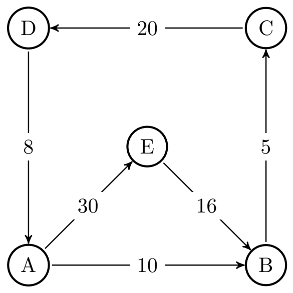


Figure 20: A directed graph drawn with `tkz-graph`.

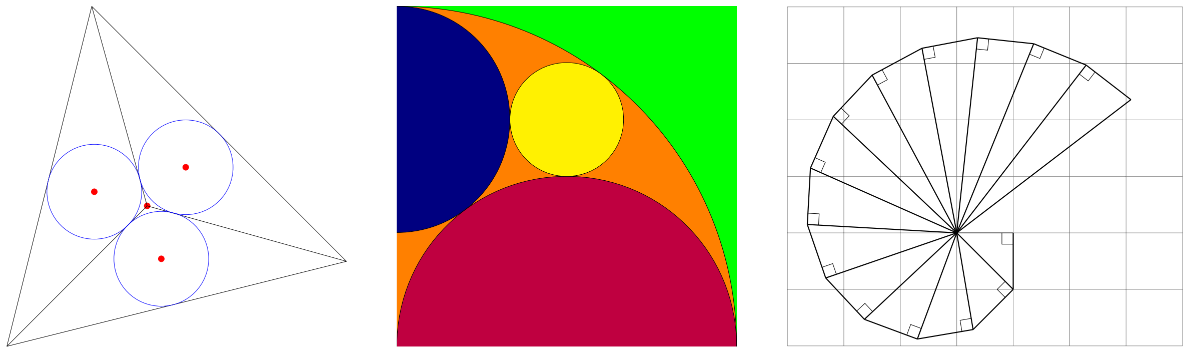


Figure 21: A brief gallery of pictures drawn by `tkz-2d`. Images created by Alain Matthes.

```
\begin{tikzpicture}
   % Initialize tkz-2d
   \tkzInit

   % Define and label two points, A and B, and a segment joining them
   \tkzPoint[pos=left](1,1){A}
   \tkzPoint[pos=right](4,2){B}
   \tkzSegment[style=thick](A/B)

   % Construct two circles, each with radius AB
   \tkzCircle(A,B)
   \tkzCircle(B,A)

   % Find and label the intersection points
   % C and D of the two circles
   \tkzInterCC(A,A,B)(B,B,A){C}{D}
   \tkzDrawPoint[pos=above left](C)
   \tkzDrawPoint(D)

   % Draw the remaining sides of the equilateral triangle
   \tkzPolySeg(A,C,B)

   % Draw a perpendicular bisector
   \tkzMidPoint(A,B){E}
   \tkzSegment[style=dashed](C/D)

   % Mark the 90 degree angle
   \tkzRightAngle(A/E/C)
\end{tikzpicture}
```

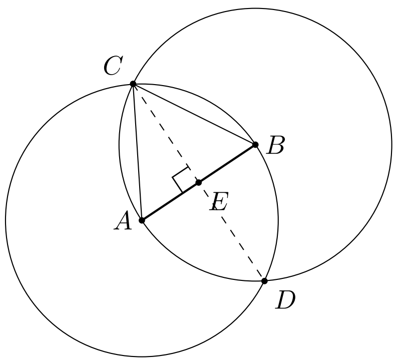


Figure 22: A construction due to Euclid, expressed with `tkz-2d`.

compared to those available to the Ti*k*Z user. The illustrations shown in Figure 21 provide a glimpse of the possibilities afforded by this package.

For an example of `tkz-2d`, we consider a geometric construction due to Euclid. In this construction, a line segment is given and the goal is to construct an equilateral triangle, one side of which is the given segment. A similar construction appears as a tutorial in the Ti*k*Z manual, although we feel the approach made possible by `tkz-2d` is more natural.

Figure 22 provides the full details of the construction. We begin by introducing two points, specifying the position and label information, using the `\tkzPoint` command. The two circles are drawn with the `\tkzCircle` command, given the center and implied radius. The most complicated statement shown here is `\tkzInterCC` which computes the intersection points of the two circles, storing their coordinates in `C` and `D`. The remainder of the construction follows easily from `tkz-2d` primitives.

A second example of the `tkz-2d` package is shown in Figure 23. Observe that the starred form

```
\begin{tikzpicture}[scale=1/2] % scaled to half-size
   % Initialize tkz-2d
   \tkzInit

   % Define three points of a 3-4-5 right triangle
   \tkzPoint*(0,0){C}
   \tkzPoint*(4,0){A}
   \tkzPoint*(0,3){B}

   % Draw the three sides of the triangle
   \tkzPolygon(C,A,B)

   % Create a 4 by 4 grid and move it down
   \begin{scope}[yshift=-4cm]
      \tkzGrid(0,0)(4,4)
   \end{scope}

   % Create a 3 by 3 grid and move it left
   \begin{scope}[xshift=-3cm]
      \tkzGrid(0,0)(3,3)
   \end{scope}

   % Create a 5 by 5 grid for the hypotenuse
   \begin{scope}[yshift=3cm, rotate=-atan(3/4)]
      \tkzGrid(0,0)(5,5)
   \end{scope}
\end{tikzpicture}
```

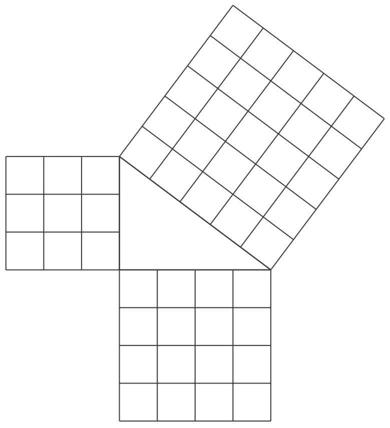


Figure 23: An illustration of the Pythagorean theorem drawn with `tkz-2d`.

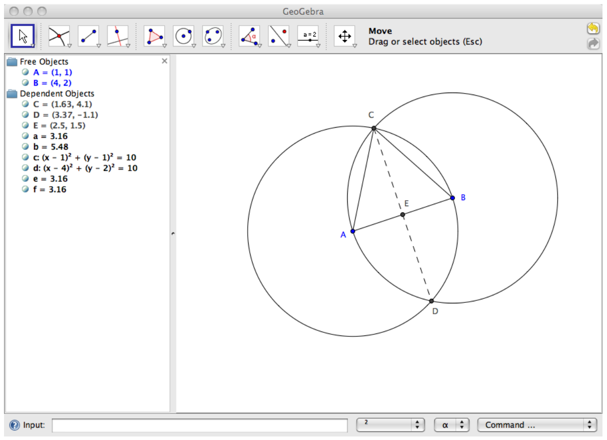
of the `\tkzPoint` command used here causes a point to be defined, but no corresponding label is drawn. Another interesting aspect of this example is the use of the Ti*k*Z `scope` environment to limit the effect of the shift and rotation operations applied to each of the three grids.

A few comments about these examples. Like `tkz-graph`, semicolons are not needed to terminate commands. Unlike Ti*k*Z, the use of spaces to separate arguments within `tkz-2d` commands is *not* allowed, an unfortunate requirement in our opinion. Finally, it is possible to blend "pure" Ti*k*Z commands with `tkz-2d` commands.

8 GeoGebra and Ti*k*Z

GeoGebra [4] is a software system intended for mathematics education. Although not limited to geometry, GeoGebra is an example of an interactive geometry system. In these systems, geometric constructions can be performed using fundamental objects such as points, lines and circles. After the construction is complete, it can be modified by dragging points or moving sliders, while preserving the key geometric relationships that defined the construction.

Figure 24 shows a GeoGebra session involving the geometric construction considered in Section 7. In contrast to specifying such a construction with



```
Free Objects
   A = (1, 1)
   B = (4, 2)
Dependent Objects
   C = (1.63, 4.1)
   D = (3.37, -1.1)
   E = (2.5, 1.5)
   a = 3.16
   b = 5.48
   c: (x - 1)² + (y - 1)² = 10
   d: (x - 4)² + (y - 2)² = 10
   e = 3.16
   f = 3.16
```

Figure 24: Screen image of a GeoGebra session.

Ti*k*Z commands, this construction was driven by the menus provided by the GUI presented by GeoGebra, involving construction choices such as "circle with center through point", "intersect two objects", and "segment between two points". In brief, GeoGebra provides for geometric constructions in a menu-driven approach, unlike the language-based approaches of Ti*k*Z and `tkz-2d`.

A relatively new feature of GeoGebra is its ability to export to the PGF/Ti*k*Z format. Once a construction is complete, it can be exported to Ti*k*Z format with a simple menu choice. The resulting `tikzpicture` environment can be placed in a TEX document. This feature provides an avenue for producing high quality graphics output without the overhead of learning Ti*k*Z language details. There are currently four different output formats supported: LATEX, plain TEX, ConTEXt, and the LATEX `beamer` class. (See [10] and [17], for example, for information about Beamer.) With the `beamer` format, the geometric construction is formatted for a "play-by-play" presentation, since pauses are inserted after each key construction step.

The GeoGebra software is free — it is open source and licensed under the GNU General Public License. Since it is based on Java, it runs on many different computer systems.

9 PDF and SVG output

For many users of Ti*k*Z, the overall aim is to generate several desired graphics, merging them with text to produce one document. However, there are situations where one simply wants to generate a collection of graphics images in PDF format, one per file.

Stand-alone PDF can be generated through the use of the `preview` package [5] and `pdftk`, the PDF toolkit [14], free software licensed under the GNU GPL and available for many computer systems. A schematic of the work flow needed to produce stand-alone PDF is shown in Figure 25.

With an appropriately constructed LATEX file, `pdflatex` will generate a PDF file, where each page consists of a tightly cropped image obtained from a `tikzpicture` environment. The PDF toolkit has a "burst" mode which can then be used to obtain the desired files, one image per file.

Listing 4 illustrates how to use the `preview` package for this purpose. For some situations, producing tightly cropped graphics is a bit too aggressive. However, the `preview` package conveniently allows the amount of cropping to be specified. In this example, we have specified a two point margin around the edges of the graphic by setting the length `\PreviewBorder`.

SVG [6], *scalable vector graphics*, is a format intended for use on the World Wide Web, in large measure due to its ability to obtain scalable graphical rendering within a browser. One of the supported output formats of PGF, and therefore Ti*k*Z, is SVG. Unfortunately, as is explained in the Ti*k*Z manual, there are some serious restrictions on the types of Ti*k*Z pictures which can be converted.

Happily, there are alternate routes for producing SVG output which do not suffer from these restric-

```
\documentclass{article}
% Use TikZ and any associated libraries
\usepackage{tikz}
\usetikzlibrary{arrows, automata}

\usepackage[tightpage, active]{preview}
\setlength{\PreviewBorder}{2pt}
\PreviewEnvironment{tikzpicture}

\begin{document}

   \begin{tikzpicture}  % First picture
   ...
   \end{tikzpicture}

   \begin{tikzpicture}  % Second picture
   ...
   \end{tikzpicture}
   ...                  % Other pictures
\end{document}
```

Listing 4: Using the `preview` package to generate tightly-cropped graphics images, one per page.

tions. In fact the technique, shown in Figure 26, is very similar to that for the production of stand-alone PDF. The key difference lies in the conversion of PDF to SVG, which can be accomplished with `pdf2svg` [2], a free utility.

10 Summary

Ti*k*Z is a very capable system which integrates vector graphics with TEX. Since its inception roughly four years ago, it has continued to evolve, gaining new capabilities and features. Moreover, a variety of other programs which can export to Ti*k*Z and/or PGF formats provides some evidence of its acceptance and popularity in the TEX world. Although not shown in our examples, Ti*k*Z's support for color allows for very compelling graphics, especially in conjunction with Beamer documents.

For some specialized domains, such as graph theory and theory of computing, there are relatively simple techniques which utilize Ti*k*Z to produce graphics that meet or exceed the quality of figures found in textbooks and journals in those areas.

References

[1] Dwight Aplevich. M4 macros for electric circuit diagams in LATEX documents. `http://mirror.ctan.org/graphics/circuit_macros/doc/CMman.pdf`.

[2] David Barton. `pdf2svg`. `http://www.cityinthesky.co.uk/pdf2svg.html`.

[3] Kjell Magne Fauske. TEXample.net: Ample resources for TEX users. `http://www.texample.net/`.

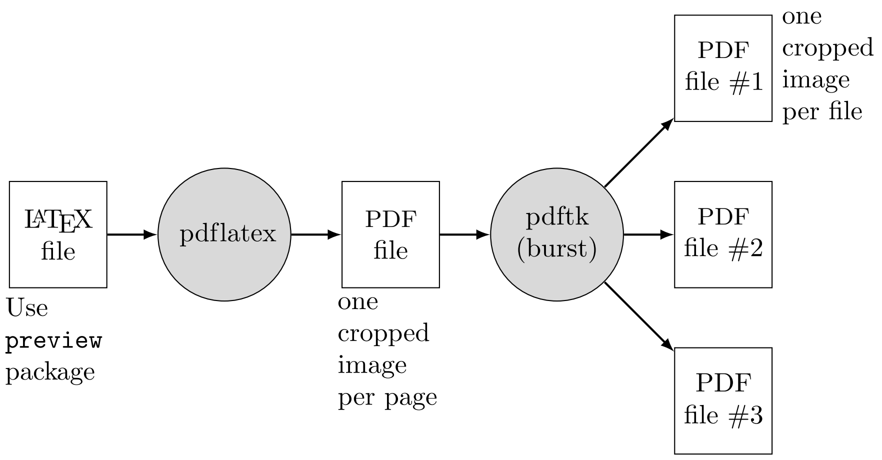


Figure 25: Processing a LaTeX file to obtain stand-alone PDF output.

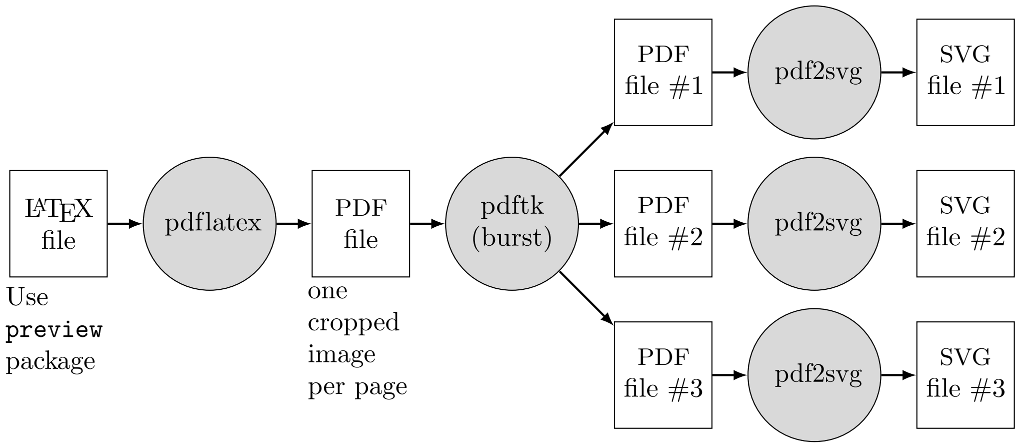


Figure 26: Processing a LaTeX file to obtain stand-alone SVG output.

[4] Markus Hohenwarter. GeoGebra. `http://www.geogebra.org/cms/`.

[5] David Kastrup. The preview package for LaTeX. `http://mirror.ctan.org/macros/latex/contrib/preview/`.

[6] Chris Lilley and Doug Schepers. W3C for the SVG working group. `http://www.w3.org/Graphics/SVG/`.

[7] Alain Matthes. `tkz-2d`. `http://altermundus.fr/pages/download.html`.

[8] Alain Matthes. `tkz-graph` and `tkz-berge`. `http://altermundus.com/pages/tikz.html`.

[9] Andrew Mertz and William Slough. `jflap2tikz`. `http://mirror.ctan.org/graphics/jflap2tikz`.

[10] Andrew Mertz and William Slough. Beamer by example. *TUGboat*, 26:68–73, 2005.

[11] Andrew Mertz and William Slough. Graphics with PGF and Ti*k*Z. *TUGboat*, 28:91–99, 2007.

[12] Massimo Redaelli. CircuiTi*k*Z. `http://home.dei.polimi.it/mredaelli/circuitikz/`.

[13] Susan H. Rodger. JFLAP. `http://www.jflap.org/`.

[14] Sid Stewart. `pdftk`. `http://www.accesspdf.com/pdftk/`.

[15] Till Tantau. PGF & Ti*k*Z. `http://mirror.ctan.org/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf`.

[16] Till Tantau. PGF and Ti*k*Z — graphic systems for TeX. `http://sourceforge.net/projects/pgf/`.

[17] Till Tantau. User's guide to the beamer class. `http://latex-beamer.sourceforge.net`.

[18] Zofia Walczac. Graphics in LaTeX using Ti*k*Z. *TUGboat*, 20:176–179, 2008.

◇ Andrew Mertz & William Slough
Department of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920
`aemertz (at) eiu dot edu`,
`waslough (at) eiu dot edu`