Enhancements to the Picture Environment of LaTeX

Sunil Podar
Dept. of Computer Science
S.U.N.Y. at Stony Brook
Technical Report 86-17

Version 1.2: July 14, 1986.

Abstract

This document describes some new commands for the picture environment of LaTeX. Some of the picture drawing commands of LaTeX are very low-level. New higher-level commands are implemented and described here. These commands enhance the graphic capabilities of LaTeX and provide a friendlier and more powerful user interface than currently existent. Their implementation has been done with the aim of reducing the amount of manual calculations required to specify the layout of *objects*. With the addition of the commands described in this document,

it should be possible to draw more sophisticated pictures with lesser effort than was previously possible.

Enhancements to the Picture Environment of LaTeX

1 Introduction

LaTeX provides a reasonably powerful picture drawing capability. There are many useful commands provided although the user-interface has room for improvement. The commands described in this document aim to achieve a simpler and more powerful interface.

Most picture drawing commands require explicit specification of coordinates for every *object*. Although explicit coordinates is the basis of the picture environment, it is possible to provide higher level commands which reduce the amount of coordinates that need to be manually calculated. There are basically two approaches that can be taken in designing such commands:

- providing ability to specify a set of objects such that the entire set can be plotted by specifying one or two coordinate pairs; `\shortstack` command falls into this category.
- providing commands that do most of the computation internally and require simple coordinate pairs to be specified; `\multiput` command is one example of this approach.

The obvious advantage of having commands that fall into the above categories is that not only they are easier to specify initially, but any subsequent modification to the layout requires minimal recalculations. For instance, to modify the coordinates in a `\multiput` statement plotting n objects requires recalculation of at most 4 coordinates, whereas the equivalent `\put` statements may require upto $2n$ calculations and/or recalculations.

Another frequently used command, `\line` has severe limitations and drawbacks. The arguments that the `\line` command expects are very nonintuitive and requires extensive calculations — often the thought process in writing a `\line` command involves:

1. calculating the coordinates of the two end-points.
2. calculating the horizontal and vertical distance.
3. figuring out if the desired slope is available and if not then repeating steps 1 and 2 till a satisfactory slope is achieved.
4. translating above into an (x,y) pair for specifying a slope and a horizontal distance for specifying the length of the line.

Above mechanism is a cumbersome way of specifying a line. It also has the drawback that the length of the shortest line of different slopes that can be drawn is different; for instance, assuming `\unitlength=1pt`, `\line(1,6){10}` is the shortest line of the given slope that can be drawn; it is considerably longer than the available line segment of this slope — 60.8pt rather than about 11pt. It should be emphasized that this is a drawback of only the implementation of the `\line` command and is not an inherent limitation. This report describes a few line drawing commands all of which overcome such a drawback, while providing a simpler syntax. They all take, as arguments, only the coordinates of the end-points, thus eliminating all other steps involved in specifying a line; it also seems to be a natural way of perceiving a line in an environment where all the work is done in terms of coordinates.

A few new commands are developed and described in this report. They provide a simpler syntax and a higher-level user-interface. Also some of the commands permit one to plot objects that were previously cumbersome or difficult to plot. All existing commands still remain accessible. With the new commands it should now be possible to make pictures with less effort and make more sophisticated pictures than was possible earlier.

2 Commands

Following commands are described here:

`\multiputlist`	`\dottedline`	`dottedjoin environment`	`\jput`
`\matrixput`	`\dashline`	`dashjoin environment`	`\picsquare`
`\grid`	`\drawline`	`drawjoin environment`	`\putfile`

All the examples in the following sections have been plotted with `\unitlength = 1mm`.

2.1 \multiputlist

`SYNOPSIS:`

`\multiputlist(`x`,`y`)(`Δx`,`Δy`)[tbrl]{`*item1*`,`*item2*`,`*item3*`,`$\ldots$`,`*itemN*`}`

This command is a variation of the regular LaTeX command `\multiput`. The `\multiput` command permits one to put the *same* object at regularly spaced coordinates. Often one wishes to put *different* objects at coordinates that have regular increments – `\multiputlist` command can be used in those cases. This command enables one to specify a collection of objects with a single command thus simplifying the task of calculating coordinates. All those objects may also be plotted separately using `\put` commands, but any future revision of those coordinates may involve lot of manual work. This command also encourages certain regularity and symmetry in laying out various objects in a picture.

In the `\multiputlist`, as the coordinates are incremented, the objects to be put are picked up from the *list of items*, i.e., first item in first position, second item in second position, and so on. For example, numbers along the X-axis in a graph may be plotted by simply specifying:

```
\multiputlist(0,0)(10,0){1.00,1.25,1.50,1.75,2.00}
```

This is almost equivalent to the sequence:

```
\put(0,0){1.00}
\put(10,0){1.25}
\put(20,0){1.50}
\put(30,0){1.75}
\put(40,0){2.00}
```

The difference is that each *item* is put in a `\makebox(0,0)[tbrl]{...}` kind of construction which allows the specification of the reference point of the box containing the item. The `[tbrl]` is optional and its absence makes the item centered at the specified coordinate. Note that `\put` command does not have such an option.

The objects in the *list* can be virtually anything including any `\makebox`, `\framebox`, math characters, etc. This command can be usefully employed in a situation where a variety of objects are to be put at coordinates that have a regular increment along the x-axis and the y-axis.

Few comments about `\multiputlist` command:

- Individual items have to be grouped in `{}` if they contain ","s.
- In the list of items, blanks are not ignored (of course, consecutive blanks are coalesced into one, as always). For a list of items longer than a line of input, put a % at the end in order to nullify the newline if a blank is not intended to be a part of the item.
- Specifying individual items in a list format provides a powerful mechanism for specifying a variety of objects in a single command. Moreover, often real numbers need to be plotted and it is nontrivial to generate real numbers or otherwise handle them in TeX; they need to be explicitly specified as *objects* in the desired format. The `\multiputlist` command somewhat simplifies such a task.

- The implementation of `\multiputlist` uses two macros derived from the ones given in the TeXbook, namely, `\lop` and `\lopoff` for list-manipulation.

2.2 \matrixput

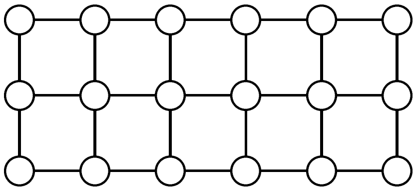
SYNOPSIS:

`\matrixput`(x,y)(Δx_1,Δy_1){n_1}(Δx_2,Δy_2){n_2}{*object*}

Above command is the two-dimensional equivalent of the regular LaTeX command `\multiput`. The `\matrixput` command is equivalent to:

`\multiput`(x,y)(Δx_2,Δy_2){n_2}{*object*}
`\multiput`($x + \Delta x_1$,$y + \Delta y_1$)(Δx_2,Δy_2){n_2}{*object*}
...
`\multiput`($x + n_1 \Delta x_1$,$y + n_1 \Delta y_1$)(Δx_2,Δy_2){n_2}{*object*}

However, it is more efficient to use `\matrixput` than the equivalent n_1 `\multiput` statements; first the objects along the dimension with larger index are saved in a box and subsequently the box is copied along the other dimension, resulting in a $O(n_1 + n_2)$ execution time rather than $O(n_1 * n_2)$ which would be the case with the equivalent `\multiput` statements. This command can be useful in making pictures where a pattern is repeated at regular intervals in two dimensions, such as certain kinds of transition diagrams. An illustration of the `\matrixput` command is presented below.



```
\matrixput(0,0)(10,0){6}(0,10){3}{\circle{4}}
\matrixput(2,0)(10,0){5}(0,10){3}{\line(1,0){6}}
\matrixput(0,2)(10,0){6}(0,10){2}{\line(0,1){6}}
```

Note: The `\matrixput` command does not restrict the Δx's and the Δy's to be zero. The *matrix* of *objects* can be "skewed", i.e., with nonzero Δx's and/or Δy's.

2.3 \grid

SYNOPSIS:

`\grid(`*width*`,`*height*`)(`Δ*width*`,`Δ*height*`)[`*initial-X-integer*`,`*initial-Y-integer*`]`

For example, the following are all valid commands:

```
\put(0,0){\grid(95,100)(9.5,10)}
\put(0,0){\grid(100,100)(10,5)[-10,0]}
\put(0,0){\tiny \grid(100,100)(5,5)[0,0]} % the numbers in \tiny font.
\put(50,50){\makebox(0,0){\tiny \grid(20,20)(4,4)}}
```

The `\grid` command makes a grid of size *width* units by *height* units where vertical lines are drawn at intervals of $\Delta width$ and horizontal lines at intervals of $\Delta height$. The major motivation for this command is that making a grid in the picture initially can be very useful when laying out pictures – it's like having a graph underneath the picture which can be eventually deleted or commented out. Moreover, one might actually want a grid as an object in its own right! Figure 1 (on page 14) presents an example of this command.

The *width* and *height* should be divisible by their respective Δ's, otherwise the grid will not be of correct dimensions. The numbers in `[ ]` at the end are optional. Their absence makes a simple grid with lines. Their presence makes a "numbered" grid with integers around the borders where the numbers put have the starting value as specified in `[. , .]` argument and are incremented by Δ*width* and Δ*height* respectively. If specified, then these starting numbers must be integers. The dimensions are all in units and do not have to be integers, although in most cases one will want integers only. There is an additional constraint when plotting a "numbered" grid — the "Δ"-dimensions have to be integers, since one cannot easily generate real numbers from within TEX. None of the errors of this kind are caught, hence, if the grid comes out funny, one of the above-mentioned conditions may have been violated.

The `\grid` command produces a box and thus needs to be `\put` at the required coordinates. The reference point of the grid is the bottom-left corner and the numbers along the borders, if any, do not affect the reference point. If it is desired to have another reference point, then the whole grid statement may be put in a `\makebox(0,0)[..]{...\grid...}` kind of construction.

2.4 \dottedline

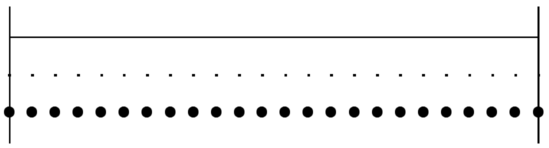
`SYNOPSIS:`

`\dottedline[`*optional dotcharacter*`]{`*dotgap in units*`}`$(x_1,y_1)(x_2,y_2)\ldots(x_n,y_n)$

The above command connects the specified points by drawing a dotted-line between each pair of coordinates. At least two points must be specified. The dotted line is drawn with inter-dot gap as specified in the second argument (in unitlengths). Note that since integral number of dots have to be plotted, the interdot-gap may not necessarily be exactly as specified, but very close. It really doesn't matter in visual appearance except when the length of dottedline is very small. By default, a little square (`\picsquare`, described later) is used as the dot, and can be changed by optionally specifying another character. The thickness of dots is governed by currently effective `\thinlines`, `\thicklines` or `\linethickness...` declaration when the default character is used. Note that some characters such as "*" in roman font do not come out centered, although most other characters do.

One can obtain a *solid line* by specifying a very small inter-dot gap. Since LaTeX provides for only finite number of slopes for drawing lines, this gives a general way of making lines with arbitrary slopes. However, if *solid lines* are made using above technique, there is a good chance TeX will run out of memory, hence it is suggested that this command be used only for "dotted" lines. Another, much more efficient, way of making solid lines is described later in the section on `\drawline`.

Each "dot" in the dottedline is plotted as a centered object, including those at the end points. Thus, a dottedline with a large-sized *dotcharacter* may appear to be longer although, technically speaking, correct. To clarify the point, below are three lines of equal length and, in the case of dottedlines, with equal spacing:



```
\put(0,10){\line(1,0){70}}
\dottedline{3}(0,5)(70,5)
\dottedline[$\bullet$]{3}(0,0)(70,0)
```

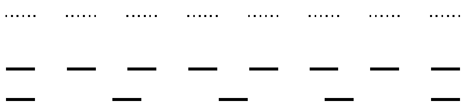
2.5 \dashline

SYNOPSIS:

`\dashline[`*stretch*`]{`*dash-length*`}[`*inter-dot-gap for dash*`]`$(x_1,y_1)(x_2,y_2)\ldots(x_n,y_n)$
where *stretch* is an integer between -100 and infinity.

The above command connects the specified points by drawing a dashline between each pair of coordinates. At least two points must be specified. A `\dashline` is a dashed line where each *dash* is constructed using a *dotted-line*[1]. The dash-length is the length of the *dash* and inter-dot-gap is the gap between each dot that is used to construct the dash, both in unitlengths.

By default, a solid looking dash is constructed, but by specifying an inter-dot-gap in the third argument, different looking dashes may be constructed. With a large inter-dot-gap (about >0.4mm), each dash will have the appearance of a little dotted line. One can create a variety of *dashlines* where each *dash* looks different. Here are a few sample dashlines:



```
\dashline{4}[0.7](0,18)(60,18)
\thicklines
\dashline{4}(0,11)(60,11)
\dashline[-30]{4}(0,7)(60,7)
```

The *stretch* in `[ ]` is an integer percentage and implies a certain "stretch" for positive values and "shrink" for negative values; it is optional and by default is "0" unless the default itself has been changed (described later). The number "0" signifies that a minimum number of dashes be put such that they are approximately equally spaced with the empty spaces between them. A +ve number means increase the number of dashes by *stretch* percent, and a −ve number means reduce by that percent. By reducing the number of dashes, the empty space between dashes is stretched while maintaining the symmetry. The lower limit on *stretch* is obviously -100 since at less than -100% reduction one essentially gets nothing. On the upper side, the number,

[1] for efficiency, in the case of horizontal and vertical dashlines, the dash is constructed using a rule.

theoretically, can be as large as infinity (barring arithmetic overflows) and the macro does not check for any upper bound; one should normally not require more than 100 percent increase (100 $\Rightarrow$ double the number of dashes) since that would essentially mean a "solid line" and it is more efficient to use the `\drawline` command for drawing such lines, as described later.

The idea behind the *stretch* percentage option is that if several dashed lines of different lengths are being drawn, then all the dashed lines with the same $-$ve or $+$ve *stretch* will have similar visual appearance, as might be desired if one were plotting a graph — one would like a particular "curve" to look the same between all the points on that curve. Also, it can be used to take any corrective actions, if the appearance of the default dashline does not meet one's approval.

The default *stretch* percentage can be changed by a `\renewcommand` on the parameter `\dashlinestretch` any time and it takes effect immediately. The argument is the integer percentage increase or reduction that will be applied to all `\dashline` commands except the ones in which the percentage is explicitly given using `[ ]` optional parameter. For example, all *dashlines* could be reduced by 50 percent by putting the following line *before* using any `\dashline` command:

```
\renewcommand{\dashlinestretch}{-50} % ONLY INTEGERS PERMITTED.
```

An explicit argument to the `\dashline` command in `[ ]` overrides any default values, so for instance, after the above declaration, if a dashline with "0" stretch was desired, then one would simply say:

```
\dashline[0]{...}(x1,y1)(x2,y2)  % where "0" implies no stretch or shrink
```

A note about dashlines of small length. All dashlines always have a dash beginning at the first coordinate and another ending at the second coordinate, which implies that a minimum of two dashes are plotted. For small lines (or larger lines with accordingly larger sized dashes) the dash-length is reduced as much as necessary to meet above conditions; in such cases, if necessary, the $-$ve stretch arguments are ignored. Such dashlines usually do not have an acceptable appearance, and may either be omitted or be plotted separately as a dottedline or a dashline with a small dash-length.

2.6 \drawline

`SYNOPSIS:`

`\drawline[`*stretch*`]`$(x_1, y_1)(x_2, y_2)\ldots(x_n, y_n)$

where *stretch* is an integer between -100 and infinity.

The above command connects the specified points by drawing a line between each pair of coordinates using line segments of the closest slope available in the fonts. At the minimum two points must be specified. Since there are only finite number of slopes available in the line segment fonts, some lines appear jagged. A `\drawline` can be thick or thin depending on the `\thinlines` or `\thicklines` declaration in effect; these are the only two thicknesses available for such lines. This is also the most efficient, in terms of memory and cpu usage, way of drawing lines of arbitrary slopes.

The *stretch* parameter has properties similar to those described earlier in the context of dashlines. It is again a percentage and implies a certain "stretch" or "shrink"; it is optional and by default is "0" unless the default itself has been changed (described later). The same rules apply to the range of the *stretch* value. In this case, the number "0" signifies that a minimum number of dashes be put such that the line appears solid and each dash "connected" at the ends. By reducing the number of dashes by specifying a $-$ve *stretch*, one effectively gets a dashed line. On the other hand, by specifying a +ve *stretch*, more dashes will be used in constructing the line, giving a less jagged appearance.

A parameter, namely, `\drawlinestretch`, has been provided for `\drawline`'s and its usage is identical to `\dashlinestretch` described earlier in the context of `\dashline`.

A limitation of drawing lines using line-segment fonts is that the length of segments is fixed and is not user-controllable. If explicit control over the line-segment length is desired, then `\dashline` may be used. If the length of the line to be drawn is smaller than the length of available line segment, then a solid line is constructed using `\dottedline` with dots being very close; the thickness of the *line* thus constructed is chosen appropriately. Note that in such a case, only a solid line can be constructed between the two points, i.e., dashed appearance can not be given to such small lines, and any $-$ve stretch is ignored.

2.7 The join environments

SYNOPSIS:

`\jput(`x`,`y`){`*object*`}`

`\begin{dottedjoin}[`*optional dotcharacter*`]{`*inter-dot-gap*`}`
`.....                dottedlines drawn here for each \jput`
`statement.`
`\end{dottedjoin}`

`\begin{dashjoin}[`*stretch*`]{`*dash-length*`}[`*inter-dot-gap for dash*`]`
`.....                dashlines drawn here for each \jput`
`statement.`
`\end{dashjoin}`

`\begin{drawjoin}[`*stretch*`]`
`.....                drawlines drawn here for each \jput`
`statement.`
`\end{drawjoin}`

Three environments, corresponding to the three kinds of lines described earlier, are also provided. They are `dottedjoin`, `dashjoin` and `drawjoin`. All the three environments use yet another new command `\jput`[2] (join and put) which is identical to the regular `\put` command of LaTeX except that it behaves differently when in any of the three environments.

All *objects* put using a `\jput` command within the scope of any of the three environments are, in addition to being plotted, joined by lines of the respective kind; in other words, a line of the specified kind is drawn between *points* plotted using `\jput` statement in the order they are encountered; a *point* refers to the x and y coordinates specified in the `\jput` statement. Consecutive `\jput` statements are assumed to define adjacent points — hence, the input should be accordingly ordered. Moreover, the plotted point should be in a `\makebox(0,0){...}` (except, of course, centered *objects* such as `\circle` and `\circle*`) if it is to be centered on the specified coordinate; without it the object's bottom-left corner will be at the specified coordinate. Each instance of any of the three join environments defines a separate "curve" hence every set of points belonging to different "curves" should be enclosed in separate join environments.

All the parameters, optional and mandatory, other than the coordinates that go along with the line drawing commands, may be specified after the `\begin{...join}` command as its arguments. Currently effective default values are used when not specified in `[]`, and may be changed anytime using the `\renewcommand` as discussed previously.

The primary motivation for designing the join environments is for use in plotting graphs and joining different curves by different looking lines. It is not necessary that the `\jput` statements put some object; if the object is null then one gets only lines — in such a case it is much simpler to use the

[2]could have redefined the `\put` statement; `\jput` behaves identically to `\put` when not in any join environment.

respective line drawing command directly.

2.8 \picsquare

`\picsquare` is a simple macro that gives a little square dot with its center as the reference point. The size of the square is dependent on the currently effective `\thinlines`, `\thicklines` or `\linethickness...` declaration. Most of the commands described earlier that plot little dots, use this macro[3]. It has been provided primarily to be used in conjunction with `\putfile` command described below. Only `\picsquare` has been made accessible to the user.

2.9 \putfile

`\putfile{`*filename*`}{`*object*`}`

The command `\putfile` is similar to the `\put` command except that the x and y coordinates required by the `\put` command are read from an external file and the same *object* is plotted at each of those coordinates.

The motivation behind this command is that TEX does not have the capability to do floating point calculations which would be required if one wished to plot any parametric curve other than straight lines. Coordinates for such curves can be easily generated by programs in other languages and subsequently a "dotted" curve can be plotted via TEX or LATEX. Even if coordinates for certain curves could be generated from within TEX, it is much more efficient to use other languages — eventually only the coordinates of the points are required. For instance, one can use the Unix[4] facility *spline* to generate smooth curves with equidistant "dots".

[3]The `\dottedline` macro actually uses another similar macro `\picsquare@bl`, which gives an identical square, but with the bottom-left corner as the reference point.

[4] *Unix* is a trademark of AT&T.

Format of the External File: The external file of coordinates must have "x y" pairs, one pair on each line, with a space between them. Also, it is suggested that some extension such as "`.put`" be used for such data files to distinguish them from regular text files in which case it must be explicitly specified in the first argument so that TEX doesn't look for a "`.tex`" extension.

The "`%`" character remains valid as a comment character and such lines are ignored. However, there should be at least one space after the second entry if a comment is on the same line as data since `%` eats up the newline.

For example, to plot a smooth curve along a set of coordinates, one may undertake the following steps:

1. have a file of "x y" coordinates for original data points, say, `datafile`.
2. run the command (for Unix systems): `spline -200 datafile > data.put`
3. in a picture environment in a LATEX file, put the command:
   `\putfile{data.put}{\picsquare}`
   (see previous section for explanation of `\picsquare`).

3 General Comments

A few remarks about efficiency and quirks:

- In most of the above commands, simply typing a `[]` for optional arguments with *nothing* as the value will either cause an error or will be interpreted as a null value; hence a `[]` should not be typed if an optional argument is not meant to be specified.
- If too many "dots" are to be plotted in one picture, it is suggested that a character other than the default be used — about 40–50% more

dots can be plotted in a picture using a period (.) or a `\bullet` (•) in various sizes, rather than the default `\picsquare`, although the latter seems to have a better visual appearance. The use a `\picsquare` also enables one to have a better control over the thickness of dots and lines.

A note on efficiency: when specifying a font or a fontsize for a character it is more efficient to say:
`{\tiny \dottedline[$\bullet$]{2}(0,0)(40,30)(80,10)}`, rather than
`\dottedline[\tiny $\bullet$]{2}(0,0)(40,30)(80,10)`.
In the latter case, `\tiny` macro gets invoked for *each* instance of the dotcharacter `$\bullet$` as the dottedline is plotted.

- If it is not very important as to how accurately spaced a dashed line appears, then it is suggested that `\drawline` command with a −ve stretch be used instead of `\dashline`, since the former is much more cpu- and memory-efficient.

- `\dottedline` and `\dashline` come out much too thin with `\thinlines`. Moreover, the thicker the `\dashline`, fewer "dots" are required to construct dashes resulting in lesser memory and cpu usage. Thus, it is recommended that they be plotted with `\thicklines` in effect, or with a linethickness of about 1–2pt.

- In the case of `\drawline`, any explicit linethickness declarations (i.e. using `\linethickness` command) are ignored. The only applicable declarations are `\thinlines` and `\thicklines` since line-segment fonts are available in only two thicknesses.

Above commands are available in the picture environment only since they use many of the LaTeX's predefined picture commands. Extensive use of some of the internal macros and variables of LaTeX has been made for efficiency sake, even though that makes these macros vulnerable to future revisions of LaTeX.

The dottedline macro gets complicated because TeX does not have any builtin facility for floating point calculations or for calculating square-roots or trigonometric functions. The inter-dot-gap in a dottedline has to be treated as the actual distance between two dots along the "hypotenuse" and not its projected distance along x-axis or y-axis, since the latter interpretation would result in a different *real* inter-dot-gap for different slopes; it would be incorrect if we were joining points on a graph. The dootedline macro treats the inter-dot-gap as the actual distance between two dots and draws the various segments of the "curve" with this distance fixed. The macro accomplishes this by estimating the actual length of the line and the number of segments of the specified distance that will fit between the two end-points; a macro, namely, `\sqrtandstuff` calculates this square-root. Some algebraic relations are used in estimating this square-root and are described in appendix A.

Beware, if far too many dots are put in one picture, LaTeX will run out of memory (box full), so be kind to it. For instance, by reducing the inter-dot-gap to about 0.3mm in the case of a `\dottedline`, one can get essentially a solid line, but that would mean a LOT of dots and it may run out of memory.

If many lines using above-mentioned macros are drawn, then a `\clearpage` ought to be put at judicious places in the document so as to tell LaTeX not to keep those figures floating – LaTeX sometimes keeps entire figures in memory while trying to figure out how and where to lay them and it can frequently run out of memory. A `\clearpage` may prevent running out of memory and may reduce execution times. In case of such a memory-full error message, a `\clearpage` in the region where the error occurred should be attempted first and if that does not help then the number of "dots" in the picture will have to be reduced.

A word about `\drawline` is in order. LaTeX's `\line` command takes an ordered pair of integers to specify the slope of the line where the numbers are between -6 and 6 such that the least common divisor is 1. For the `\drawline` command, the given arbitrary slope has to be mapped to the pair of integers representing the *closest* available slope. Another macro, `\lineslope` is used to accomplish this task. The macro `\lineslope` takes two arguments, the base and the height of the triangle whose hypotenuse represents the line to be drawn and returns the ordered pair of integers representing the closest slope; using a line segment of that slope, a jagged line between the two specified end-points is then constructed. More details can be found in the macro file `epic.sty`.

As noted earlier, the command `\jput` behaves identically as `\put` when not in any of the join environments. The author considered obliterating the `\put` command too radical a step. Also, there should have been a command `\jputfile` corresponding to the `\jput` command (like the `\putfile` command) but that was considered unnecessary since typically the number of coordinates plotted in a join environment would be an order less than what might be the case with `\putfile` and can be easily typed explicitly in the document using `\jput` commands. However, if it is desired to have all the `\put` commands treated as though they were `\jput`, the following declaration may be used:

```
\let\put\jput
```

Above declaration will make all the `\put` commands be treated as `\jput`; in particular, `\putfile` command would then behave as though it were a `\jputfile` when in any of the join environments. However, it is suggested that such "tricks" be used with care.

Finally, commands to plot vectors of arbitrary slopes have not been implemented. One way to plot them is to plot a line, and subsequently plot a `\vector` of appropriate slopes and length zero at the required place.

Following pages contain some examples. The test-sample picture for `\drawline` command (Figure 2) is also about the maximum amount of objects that one can put in one picture. Older versions of TEX and LATEX may not be able to print pictures of this size.

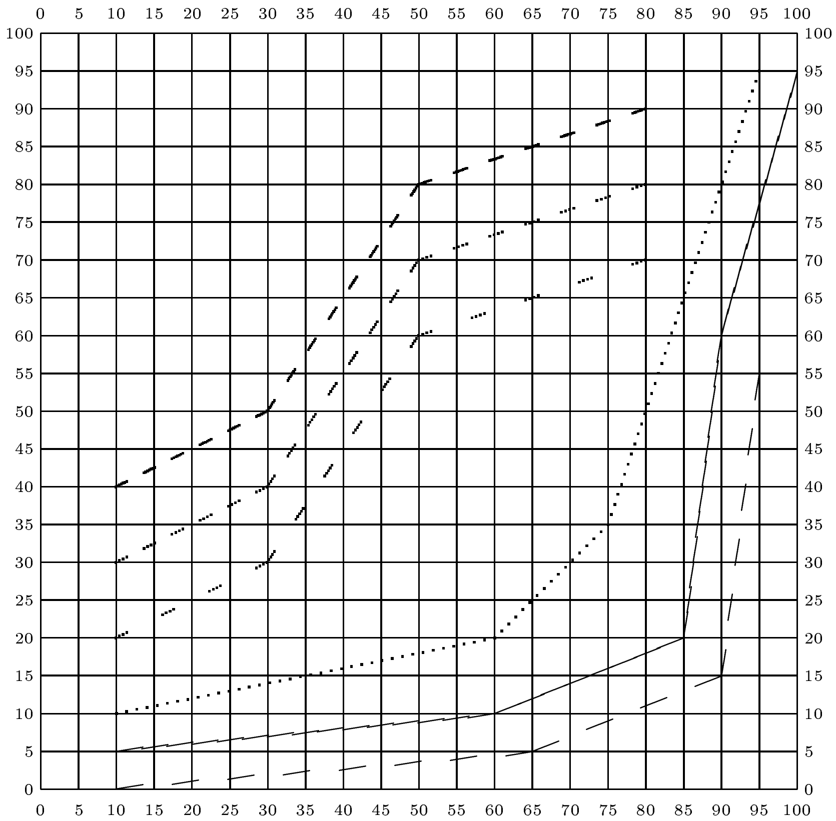


Figure 1: An Example of Various Line Drawing Commands

```
%\newcommand{\plotchar}{\makebox(0,0){\large $\otimes$}}
\unitlength = 1mm
\begin{picture}(100,100)(0,0)
\put(0,0){\tiny \grid(100,100)(5,5)[0,0]}
\drawline(10,5)(60,10)(85,20)(90,60)(100,95)
\drawline[-50](10,0)(65,5)(90,15)(95,55)
\thicklines
\dottedline{1.4}(10,10)(60,20)(75,35)(95,95)
\dashline{2}(80,90)(50,80)(30,50)(10,40)
\dashline{2}[0.5](80,80)(50,70)(30,40)(10,30)
\dashline[-30]{2}[0.5](80,70)(50,60)(30,30)(10,20)
```

```
\end{picture}
```

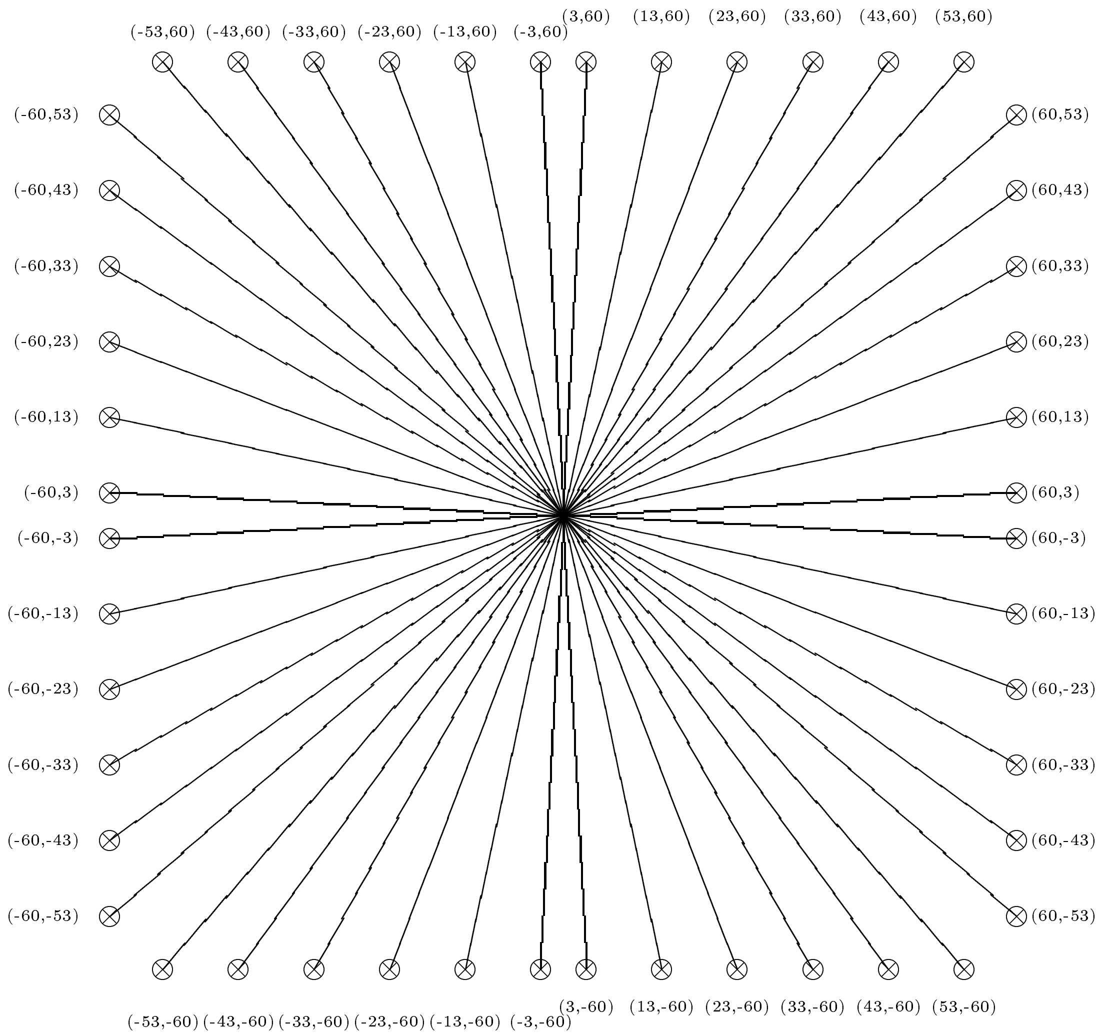


Figure 2: Test Sample: Lines of various slopes with `thinlines`

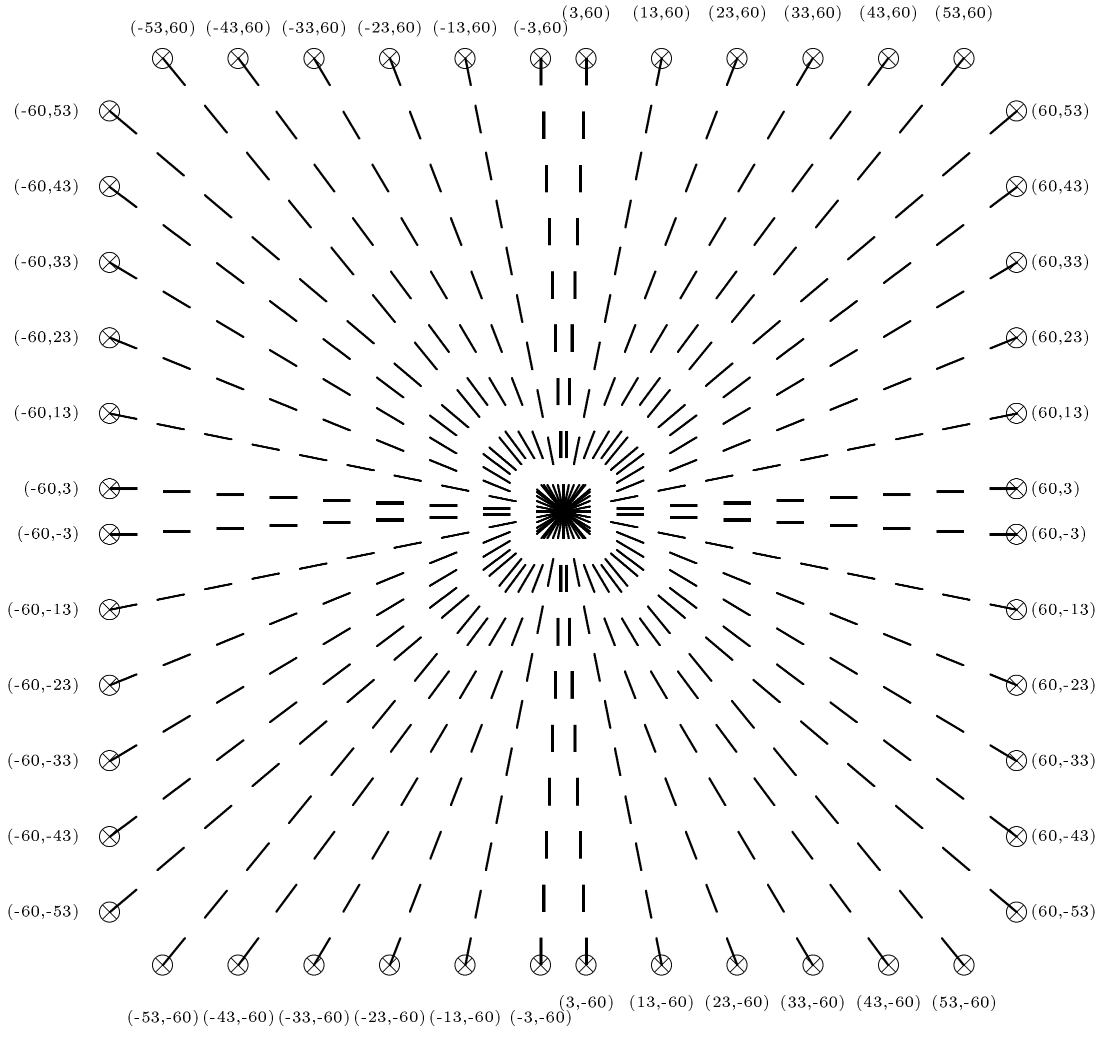


Figure 3: Test Sample: Dashed lines of various slopes using `\drawline` command with `linethickness=1pt` and `drawlinestretch = -50`

```
%Following commands were used to produce the graph on the next page.

\newcommand{\plotcharms}{\makebox(0,0){$\otimes$}}
\newcommand{\plotcharscs}{\circle{1.5}}
\newcommand{\plotcharcs}{\makebox(0,0){$\diamond$}}
\begin{figure}
\begin{center}
\begin{picture}(140,160)(-130,-10)
\linethickness{0.4mm}
\put(-130,0){\vector(1,0){140}}
\put(-130,0){\vector(0,1){150}}
\thicklines
\multiput(0,-1)(-10,0){14}{\line(0,1){2}}
\multiput(-131,0)(0,10){15}{\line(1,0){2}}
\multiputlist(0,-4)(-20,0){0,50,100,150,200,250,300} %numbers along X-axis
\multiputlist(-132,20)(0,20)[r]{10,20,30,40,50,60,70} %numbers along Y-axis
\put(-60,-10){\makebox(0,0){Interarrival Times (msec.)}}
\put(-141,75){\makebox(0,0){\shortstack{%
N\\o\\r\\m\\a\\l\\i\\z\\e\\d\\[3ex]L\\i\\f\\e\\t\\i\\m\\e\\s}}}
\thinlines
\put(-120,150){\makebox(0,0)[tl]{\fbox{\shortstack[l]{
 {\makebox(4,2)[lb]{\put(2,1){\plotcharms}}}: Message Switching\\[0.5mm]
 {\makebox(4,3)[lb]{\put(2,1){\plotcharscs}}}: Staged Circuit Switching\\[0.5mm]
 {\makebox(4,3)[lb]{\put(2,1){\plotcharcs}}}: Circuit Switching\\[0.5mm]
 {\makebox(2,0)[b]{}}
 }}}}
%
\begin{dottedjoin}{2}
\thicklines
\jput(-120.00000,  34.44896){\plotcharms}
\jput( -60.00000,  35.55244){\plotcharms}
\jput( -40.00000,  36.57292){\plotcharms}
\jput( -30.00000,  37.71716){\plotcharms}
\jput( -20.00000,  40.15218){\plotcharms}
\jput( -12.00000,  48.16034){\plotcharms}
\jput(  -8.00000,  67.75840){\plotcharms}
\jput(  -7.60000,  74.27934){\plotcharms}
\jput(  -7.20000,  83.02326){\plotcharms}
\end{dottedjoin}
%
\begin{dashjoin}{2}
\jput(-120.00000,  15.01202){\plotcharscs}
\jput( -60.00000,  15.95818){\plotcharscs}
\jput( -40.00000,  17.15990){\plotcharscs}
\jput( -30.00000,  18.16152){\plotcharscs}
\jput( -20.00000,  20.32388){\plotcharscs}
\jput( -12.00000,  27.05212){\plotcharscs}
\jput(  -8.00000,  41.58512){\plotcharscs}
\jput(  -7.60000,  45.3435){\plotcharscs}
\jput(  -7.20000,  51.52414){\plotcharscs}
\end{dashjoin}
%
\begin{drawjoin}
\jput(-120.00000,  15.17960){\plotcharcs}
\jput( -80.00000,  16.71960){\plotcharcs}
\jput( -60.00000,  18.29430){\plotcharcs}
```

```
\jput( -56.00000,  19.81980){\plotcharcs}
\jput( -52.00000,  20.31963){\plotcharcs}
\jput( -48.00000,  50.24912){\plotcharcs}
\jput( -44.00000,  56.96844){\plotcharcs}
\end{drawjoin}
\end{picture}
\end{center}
\caption[]{A real-life example of a graph}
\end{figure}
```

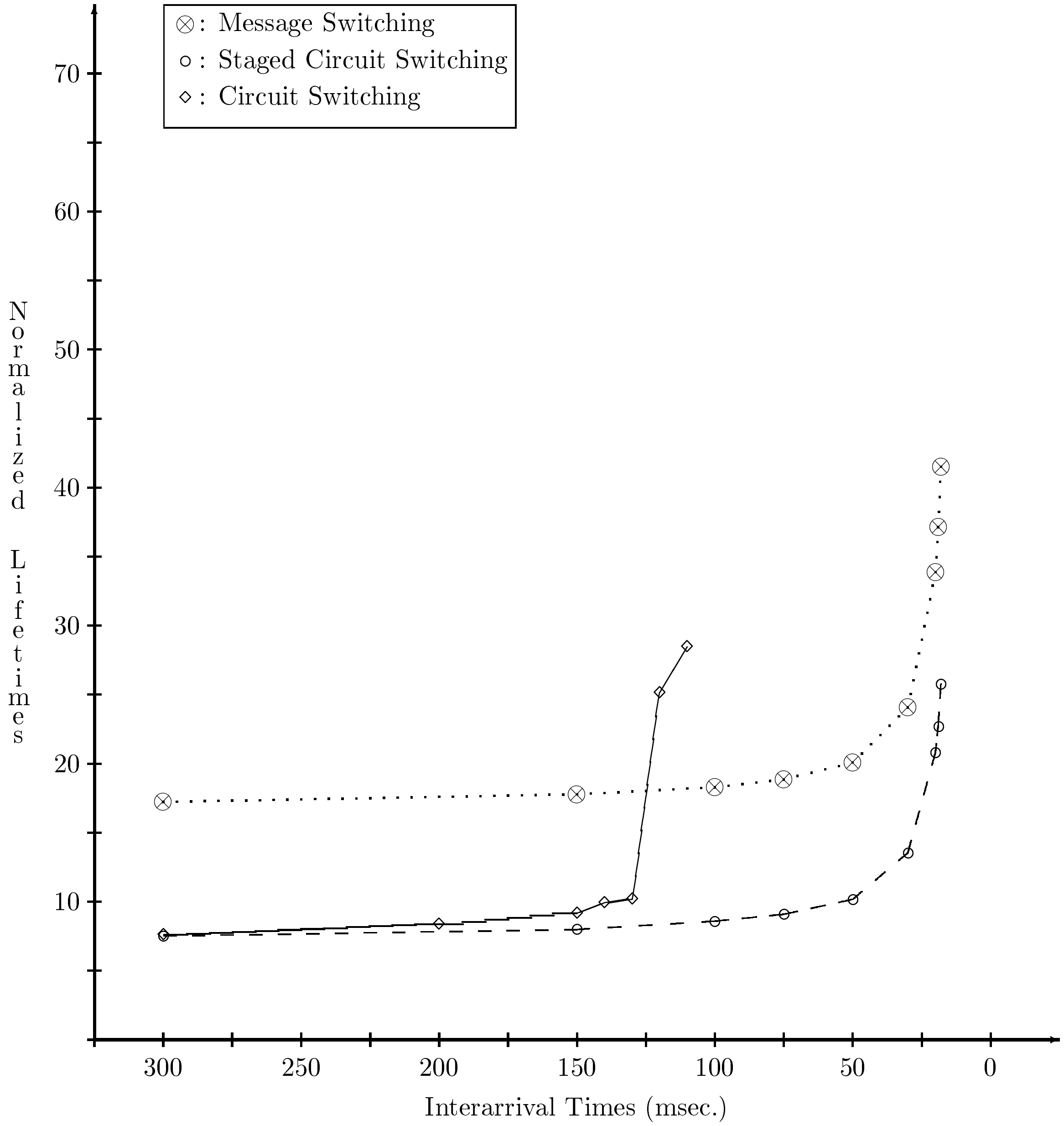


Figure 4: A real-life example of a graph

4 Installation and Usage of the Package

This package of new commands for the picture environment has been implemented as a documentstyle option "`epic`". To include these commands, "epic" should be added as an option in the `\documentstyle` command, e.g.:

```
\documentstyle[epic]{article}
```

For the above option to work, one of the following will have to be done prior to its use:

1. A copy of the macro file `epic.sty` be put in the standard place for such macros (typically `/usr/lib/tex/macros`), or

2. A copy of `epic.sty` be put in some other directory, and the path declared in the environment variable TEXINPUTS; e.g. for C-shell on unix systems, put a command similar to the following in the ".cshrc" file:

   ```
   setenv TEXINPUTS .:/usr/lib/tex/macros:/users/podar/texlib
   ```

Above environment variable is the directory search path for files specified in an `\input` or an `\openin` command.

5 Concluding Remarks

The implementation of the new commands for the picture environment has been done with the LaTeX version 2.09 and TeX version 2. They have also been tested to work with LaTeX version 2.08. These commands may not work with earlier versions of TeX and LaTeX.

Most of the commands have been tested fairly thoroughly. No major revisions are anticipated in the near future, except, of course, bug fixes. The author welcomes any comments, constructive or otherwise, suggestions

for improvements, any ideas for possible future revisions and, of course, bugs. It is also requested that he be informed of any significant changes or modifications made to these macros.

All the help and encouragement from colleagues in the Dept. of Computer Science at SUNY at Stony Brook is gratefully acknowledged; in particular, Soumitra Sengupta's and Divyakant Agrawal's criticisms (often constructive), help with proofreading the numerous versions of this report and general moral support were critical to the completion of this project and are thankfully acknowledged.

Author's address:

USMAIL:	Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, N.Y. 11794
CSNET:	podar@sbcs.csnet
ARPA:	podar%suny-sb.csnet@csnet-relay.arpa
UUCP:	{allegra, hocsd, philabs, ogcvax}!sbcs!podar

References

[1] D. E. Knuth, "The TEXbook", Addison-Wesley Publishing Co., 1984.

[2] L. Lamport, "LATEX: A Document Preparation System", Addison-Wesley Publishing Co., 1986.

Appendix A Estimating Pythagorean Square-root

For the line drawing commands described in the main sections of this document, we need to estimate the Pythagorean square-root in order to determine the length of the line (along its slope). More precisely, we need to estimate the number of segments of a given length needed to draw a line. TEX does not provide for floating point calculations, and thus there are no direct means of calculating the above square-root. Most standard numerical techniques are iterative and would be too slow when used with TEX for lack of floating point calculations, and in particular, real division, since calculation of such a square-root is needed very frequently.

A simple non-iterative formula for estimating the square-root is derived and described below.

Problem: Given a and b, to find $c = \sqrt{a^2+b^2}$ using only operations in $\{+, -, *, /\}$.

We can get very tight bounds on the square-root as follows. Without loss of generality, let $a \geq b$. We seek a simple n such that:

$$\sqrt{a^2+b^2} \geq a + \frac{b}{n}$$

Squaring both sides, we have

$$\begin{array}{lrcl} \Leftrightarrow & a^2+b^2 & \geq & a^2 + \dfrac{b^2}{n^2} + \dfrac{2ab}{n} \\ \Leftrightarrow & (1-\dfrac{1}{n^2})b^2 & \geq & \dfrac{2ab}{n} \\ \Leftrightarrow & \dfrac{b}{a} & \geq & \dfrac{2n}{(n^2-1)} \\ \text{or} & (\dfrac{b}{a})n^2 - 2n - (\dfrac{b}{a}) & \geq & 0 \end{array}$$

¿From the quadratic equation above, we finally get an expression for n,

$$n = \frac{2 \pm \sqrt{4 + 4(\frac{b}{a})^2}}{\frac{2b}{a}} = \frac{1 \pm \sqrt{1 + (\frac{b}{a})^2}}{\frac{b}{a}}$$

Only the +ve root interests us since n has to be positive. Note that the term under the root is bounded above and below (since $\frac{b}{a} \leq 1$):

$$1 \leq \sqrt{1 + (\frac{b}{a})^2} \leq \sqrt{2}$$

Hence, we have two values for n,

$$n_l \;=\; \frac{1+1}{\frac{b}{a}} \;=\; \frac{2a}{b}; \qquad\qquad n_u \;=\; \frac{1+\sqrt{2}}{\frac{b}{a}} \;=\; \frac{(1+\sqrt{2})a}{b}$$

which finally gives us a lower and an upper bound for c, the Pythagorean square-root,

$$a + \frac{b^2}{(1+\sqrt{2})a} \;\leq\; c \;\leq\; a + \frac{b^2}{2a}$$

These are very tight bounds. Denoting the lower bound as c_l and upper one c_u, below are some numerical results (c = exact square-root):

a	b	c	c_l	c_u
100.0	100.0	141.4213	141.4213	150.0
100.0	80.0	128.0642	126.5096	132.0
30.0	20.0	36.0555	35.5228	36.6667

With the above bounds, one can do a linear interpolation to get exact values. In our case, since it is not required to be *extremely* accurate, for estimating the square-root in the line drawing commands, we simply take the midpoint of the two bounds. For small numbers, which is expected to be the case most of the time, the error is very small.

With some algebra, we get the mid-point estimate of c,

$$c = \frac{c_l + c_u}{2} = a + \frac{b^2 * (3+\sqrt{2})}{a * 4 * (1+\sqrt{2})} = a + \frac{0.457\, b^2}{a} \qquad (a \geq b)$$

The macro `\sqrtandstuff` uses the above formula for estimating the number of points (for `\dottedline` macro) and number of segments (for `\dashline` macro). The `\sqrtandstuff` macro, instead of calculating the length of the line, directly calculates the *number* of segments of a given length. For example, to draw a dotted line from (x_1, y_1) to (x_2, y_2) with the inter-dot-gap as d, we estimate the number of dots n using the following expression,

$$n = \frac{\Delta x}{d} + \frac{0.457\,(\frac{\Delta y}{d})^2}{\frac{\Delta x}{d}} \qquad \Delta x = |x_2 - x_1| \text{ and } \Delta y = |y_2 - y_1|$$

assuming $\Delta x \geq \Delta y$ (otherwise they may be inter-changed).

Note that since divisions in TEX are integer-divisions, it is simpler to deal in "number of segments" rather than actual lengths (e.g. in the expression above, $\frac{\Delta x}{d}$ = number of segments along X-axis).

Caveat: The approach presented here for estimation of Pythagorean square-root is an independent effort by the author. It may already exist in the literature — the author is neither aware of it nor has he made any serious attempts at uncovering it.