Reprint from
The *Mathematica* Conference
June, 1992   Boston, MA

876.21.10.1992

Chapter 3 **Graphics Output**

This chapter discusses the steps which are carried out when a *Mathematica* graphic is rendered. It first shows how the plotting commands construct primitives and produce a particular type of graphic. Then we see how the primitives are rendered into PostScript. This is followed with a discussion of the ways in which hardcopy output can be obtained. The chapter ends with a description of the PostScript generated by *Mathematica*. This last section is an advanced topic which many readers may wish to skip.

3.1 Graphics Commands and Objects

We have seen how *Mathematica* can produce a number of different types of graphical objects. These include two-dimensional graphics, three-dimensional graphics and special types which represent images such as contour diagrams and simple surfaces. The special types exist for efficiency reasons.

`Graphics`	two-dimensional pictures
`SurfaceGraphics`	simple three-dimensional surfaces
`Graphics3D`	general three-dimensional pictures
`ContourGraphics`	contour diagrams
`DensityGraphics`	bitmaps
`GraphicsArray`	arrays of other pictures
`Sound`	sounds

Graphical objects in *Mathematica*.

Built-in commands are provided to construct all these objects. As examples we have seen how `Plot` and `ListPlot` can produce two-dimensional pictures and how `Plot3D` produces a three-dimensional image. For all the graphical objects listed above there is a number of commands which will generate them.

Command	Object
`Plot, ParametricPlot, ListPlot`	`Graphics`
`Plot3D, ListPlot3D`	`SurfaceGraphics`
`ContourPlot, ListContourPlot`	`ContourGraphics`
`DensityPlot, ListDensityPlot`	`DensityGraphics`
`ParametricPlot3D`	`Graphics3D`
`Play, ListPlay`	`Sound`

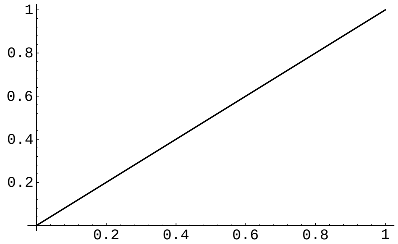
Graphics commands and the objects they produce.

Each command works by constructing the appropriate object. For example `Plot` builds a `Graphics` object. As an aside one should note the distinction between the generic use of the term graphics and the particular use of the *Mathematica* command `Graphics`. The latter is the *Mathematica* expression which holds two-dimensional graphics.

As can be seen there are many easy ways provided to generate all these different types of pictures. Here we shall look in more detail at a typical command, `Plot`. We can see the actual result, the *Mathematica* expression which is returned, by using the *Mathematica* command `InputForm`; the drawing of the picture is a side effect of the evaluation which returned this object.

This draws a picture.

```
In[1]:= Plot[ x, {x,0,1}]
```



This is the result returned to *Mathematica* by the `Plot` command. The primitives which are found in a result like this are described in the next chapter.

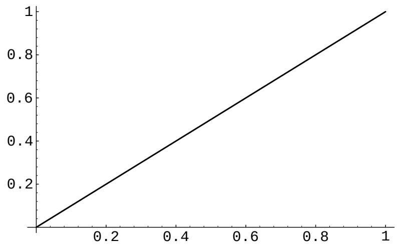
```
In[2]:= InputForm[%]

Out[2]//InputForm=
  Graphics[{{Line[{{0., 0.},
       {0.04166666666666666, 0.04166666666666666},
       {0.0833333333333333, 0.0833333333333333},
       {0.125, 0.125}, {0.1666666666666666,
        0.1666666666666666},
       {0.2083333333333333, 0.2083333333333333}, {0.25, 0.25},
       {0.2916666666666666, 0.2916666666666666},
       {0.3333333333333333, 0.3333333333333333},
       {0.375, 0.375}, {0.4166666666666666,
        0.4166666666666666},
       {0.4583333333333333, 0.4583333333333333}, {0.5, 0.5},
       {0.5416666666666666, 0.5416666666666666},
       {0.5833333333333333, 0.5833333333333333},
       {0.625, 0.625}, {0.6666666666666666,
        0.6666666666666666},
       {0.7083333333333333, 0.7083333333333333}, {0.75, 0.75},
       {0.7916666666666666, 0.7916666666666666},
       {0.833333333333333, 0.833333333333333}, {0.875, 0.875},
       {0.916666666666667, 0.916666666666667},
       {0.958333333333333, 0.958333333333333}, {1., 1.}}]}},
    {PlotRange -> Automatic, AspectRatio -> GoldenRatio^(-1),
     DisplayFunction :> $DisplayFunction,
     ColorOutput -> Automatic, Axes -> Automatic,
     AxesOrigin -> Automatic, PlotLabel -> None,
     AxesLabel -> None, Ticks -> Automatic, GridLines -> None,
     Prolog -> {}, Epilog -> {}, AxesStyle -> Automatic,
     Background -> Automatic, DefaultColor -> Automatic,
     DefaultFont :> $DefaultFont, RotateLabel -> True,
     Frame -> False, FrameStyle -> Automatic,
     FrameTicks -> Automatic, FrameLabel -> None,
     PlotRegion -> Automatic}]
```

- `fun[x]` is evaluated for `x` varying from `xmin` to `xmax`.
- These points are placed in line primitives and given a head of `Graphics`.
- The `Graphics` object is passed to `Show`, which processes options and calls the `DisplayFunction` which typically causes PostScript to be emitted.
- The `Graphics` object is returned as the result.

The steps involved in evaluating `Plot[ fun[x], {x,xmin, xmax}]`.

It is `Show` which actually invokes the PostScript interpreter. We can pass the result back into `Show` again.

```
In[3]:= Show[ %]
```



We have seen some of the different uses of `Show` such as redisplaying graphics and changing options. Later we will see how it can combine pictures by carrying out appropriate conversions. However maybe the most important use of `Show` is to call the `DisplayFunction`.

All the graphics commands have the option `DisplayFunction`. This defaults to `$DisplayFunction` which is set to send the graphics to `$Display`. The latter will invoke the PostScript interpreter for the window system your computer is using.

Default setting of `DisplayFunction` option.

```
In[4]:= Options[ Plot, DisplayFunction]

Out[4]= {DisplayFunction :> $DisplayFunction}
```

Default value of `$DisplayFunction`.

```
In[5]:= $DisplayFunction

Out[5]= Display[$Display, #1] &
```

On your machine you may see something else. Here the PostScript interpreter is an executable called `motifps`.

```
In[6]:= InputForm[ $Display]

Out[6]//InputForm=
  {"!motifps  -title 'Mathematica Graphics:  Out[6]'"}
```

`Show[` *obj1*`..,` *obj2*`, ..` *opts*`..]`	combine pictures, process options and call `DisplayFunction`
`Display[` *stream*`,` *object*`]`	send PostScript representation of *object* `to` *stream*

Commands used to display *Mathematica* graphics.

Generally one does not want to change the `DisplayFunction`. However, there are reasons why you may wish to do this. You maybe want to write your own renderer and use *MathLink*. Then if you made the correct assignment to `DisplayFunction` all the commands like `Plot` and `Show` would just work. Another use is to set `DisplayFunction` to be `Identity`. This suppresses the production of PostScript. In certain situations this is very useful.

This produces no pictures. The result can be used in `GraphicsArray`.

```
In[7]:= Table[
          Plot[ x^n, {x,0,2}, DisplayFunction -> Identity],
             {n,1,4,1}]

Out[7]= {-Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

3.2 Hardcopy Output

We have seen how the *Mathematica* kernel can render a sequence of graphics primitives into PostScript. In this section we examine how to obtain hardcopy output from this stream. This is done by rendering the picture into PostScript with the command `Display`. We saw in the last section that by default `Display` is called automatically by `Show`. However we can call it directly with a top-level command and maybe send the PostScript somewhere other than to an interpreter. For example we can send it into a file.

This sends PostScript into the file `tmp.ps`. Here we use `Display` directly.

```
In[8]:= Display[ "tmp.ps", Graphics[ Line[ {{0,0},{1,1}}]]]

Show::gmed: No graphics output medium specified.

Out[8]= -Graphics-
```

Here are the contents of the file which resulted. Note that it is not standard PostScript.

```
In[9]:= !!tmp.ps

%!
%%Creator: Mathematica
%%AspectRatio: .61803
MathPictureStart
%% Graphics
/Courier findfont 5.5  scalefont  setfont
% Scaling calculations
0.02381 0.952381 0.014715 0.588604 [
[ 0 0 0 0 ]
[ 1 .61803  0 0 ]
] MathScale
% Start of Graphics
1 setlinecap
1 setlinejoin
newpath
[ ] 0 setdash
0 g
p
P
0 0 m
1 0 L
1 .61803  L
0 .61803  L
closepath
clip
newpath
.004  w
.02381  .01472  m
.97619  .60332  L
s
% End of Graphics
MathPictureEnd
```

Thus `Display` can send a PostScript form of a *Mathematica* graphic to a file. However, this PostScript is not suitable to be sent to a printer. This is because various PostScript procedure definitions must be added. The methods which are available to work with *Mathematica* PostScript depend upon whether your front end supports Notebooks or not.

PostScript from Front Ends with Notebooks

On a Notebook front end it is very easy to print a picture. One simply pulls down the menu item for printing and everything works in the typical way for your particular type of computer. On a NeXT you see a NeXT printer panel and options, on a Macintosh you see a Macintosh printer panel and so on. You set up the printer you want and you print. The PostScript header is added automatically.

Alternatively, you may wish to make a PostScript file that can be added into some other application program. This involves writing an Encapsulated PostScript file, an EPSF. To do this, you select the image in question, copy it to the Clipboard and choose the menu item for Clipboard conversion. One of the options will be for an EPSF. Selecting this will make the appropriate file.

PostScript from Front Ends without Notebooks

On front ends which do not support Notebooks the situation is more complex. One of the major reasons for this is that the operating system on Notebook front ends does a large amount of work. On front ends without Notebooks the operating system typically provides fewer facilities.

To print a graphic one uses the command `PSPrint`. This will add header information and send the result to your default printer. On Unix things look like this.

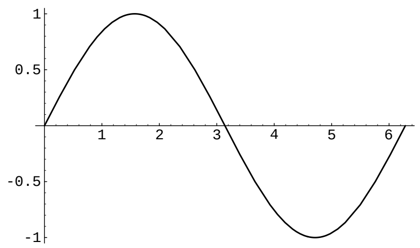
This is the definition for `PSPrint`.

In[9]:= **??PSPrint**

```
PSPrint[-graphics-] sends graphics to a printer.
Attributes[PSPrint] = {Protected}
PSPrint[x_] := (Display["!psfix | lpr", x]; x)
```

This makes a plot.

In[9]:= **Plot[Sin[x], {x,0,2Pi}]**



`PSPrint` returns the graphics object.

In[10]:= **PSPrint[%]**

Out[10]=-Graphics-

psfix

On Unix systems `PSPrint` calls `Display`, sends the result to an executable called `psfix` and pipes the result to the printer. The executable `psfix` is a shell script with a number of options. The result of applying `psfix` to the output stream is standard PostScript which can be sent to a printer or some other PostScript device. It should be noted that `psfix` is not available on DOS systems.

-width x	width of image
-height y	height of image
-lmarg x	left margin
-rmarg x	right margin
-bmarg y	bottom margin
-tmarg y	top margin
-land	print in landscape mode (rotated)
-epsf	produce EPSF
-stretch	do not maintain aspect ratio
-stringfix	adjust non-fixed width fonts

Some options for `psfix`.

rasterps

On DOS versions of *Mathematica* it is not possible to write a script such as `psfix`. Even on systems for which it exists it is possible that one requires functionality beyond that supplied by `psfix`. Maybe one wants to support a non-PostScript printer or to make an EPSF which contains a special rendering of the picture, a so-called preview image. For this the binary `rasterps` is provided. It is a PostScript interpreter which can render into a number of different formats. For DOS versions of *Mathematica* prior to Version 2.1 the functionality of `rasterps` was provided by a binary called `printps`.

-w x	image width in pixels
-h y	image height in pixels
-xdpi nx	number of dots per inch horizontal
-ydpi ny	number of dots per inch vertical
-format device	format type (see next table)

Some options for `rasterps`.

ps	PostScript
eps	Encapsulated PostScript
psimask	PostScript imagemask format
hplj.75	HP LaserJet at 75 dpi
hplj.100	HP LaserJet at 100 dpi
hplj.150	HP LaserJet at 150 dpi
hplj.300	HP LaserJet at 300 dpi
epsonfx	Epson FX series printer at 72 dpi
epsonlq	Epson LQ series printer at 180 dpi
toshp3	Toshiba P3 series printer at 180 dpi
ibmpro	IBM ProPrinter series printer at 60 x 72 dpi
ibmpro24	IBM ProPrinter24 series printer at 180 dpi
xbitmap	X bitmap file

Some of the possible format types for `rasterps`.

We can use `rasterps` to support other types of printer from within *Mathematica*. This is a *Mathematica* function which uses `rasterps` to print to a HP LaserJet.

```
In[11]:= HPPrint[ g_] :=
    Display["!rasterps -format hplj.75 | lpr -Phpl", g]
```

If the type of printer which you have is not supported by `rasterps` you can use the `psimask` format option. This makes a bitmap of the picture encoded in a simple PostScript way. You will have to discover how to make your printer render a bitmap and write some translation routine for it. This is a simpler task than writing a PostScript interpreter for your printer. The Technical Support Services at Wolfram Research can give guidance on this.

3.3 *Mathematica* PostScript

We have seen how the *Mathematica* command `Display` renders primitives into PostScript. Here we examine the PostScript which is generated. It is assumed that the reader has some knowledge of PostScript.

First it is important to understand that the *Mathematica* kernel does not produce PostScript which can be sent directly to a standard PostScript interpreter. This is because it contains references to special procedures. These procedures are understood by the PostScript interpreters supplied with *Mathematica*. To send the stream to some other interpreter it is necessary to add PostScript definitions for these special operators. When one is printing from *Mathematica* this will happen automatically. For Unix versions of *Mathematica* this header is present in a shell script which is called psfix. This adding of the headers is described in the previous section.

This writes PostScript into the file tmp.ps.

```
In[12]:= Display["tmp.ps", Graphics[ Line[ {{0,0}, {1,1}}]]]

Show::gmed: No graphics output medium specified.

Out[12]= -Graphics-
```

In the file tmp.ps few of the statements look like standard PostScript commands. There are a couple of reasons for this. First a number of *Mathematica* specific operators, all of which start with `M`, are present. Another difference is that common PostScript procedures have been given abbreviated names.

stroke	s	fill	F
gsave	p	grestore	P
lineto	L	moveto	m
setgray	g	setrgbcolor	r
setcmykcolor	k	setlinewidth	w
curveto	C		

Abbreviated forms for common PostScript operators.

Looking at the PostScript stream in more detail we can see a number of stages.

```
%!                                              identifier as PostScript
%%Creator:  Mathematica                         comments
%%AspectRatio:  .61803
MathPictureStart                                setup initial coordinates
%% Graphics
/Courier findfont 5.5 scalefont setfont         default font
% Scaling calculations
0.02381 0.952381 0.014715 0.588604 [
[ 0 0 0 0 ]                                     <0,0> will be shown
[ 1 .61803 0 0 ]                                <1, .61803> will be shown
] MathScale                                     calculate scaling factors
% Start of Graphics
1 setlinecap                                    various line rendering parameters
1 setlinejoin
newpath
[ ] 0 setdash
0 g                                             background color
p                                               critical region end
P
0 0 m
1 0 L
1 .61803 L
0 .61803 L
closepath
clip                                            clip to show only PlotRange
newpath
.004 w                                          sets the linewidth
.02381 .01472 m                                 our line starts here
.97619 .60332 L                                 our line goes to here
s                                               draw our line
% End of Graphic
MathPictureEnd                                  end the graphic
```

The structure of a PostScript file.

■ *xbias xscale ybias yscale* [*arr1 arr2* ...] `MathScale`

MathScale arguments.

`MathScale` is the most complex procedure in this process. The first four arguments show the mapping between user and display coordinates. This mapping is

$$(x_{disp}, y_{disp}) \;=\; (x_{user} \times x_{scale} + x_{bias}, y_{user} \times y_{scale} + y_{bias})$$

These variables are used by Notebook front ends to display points in a graphic. The last argument given to `MathScale` is an array of at least two arrays. These give values of display coordinates emitted in the critical region. Thus the scaling onto the paper is determined to include these. In the example here the display coordinates (0,0) and (0, 0.61803) will both be visible.

[*x y* 0 0]	the point (*x*,*y*) in display coordinates should be visible
[*x y dx dy*]	the point (*x y*) in display coordinates + (*dx dy*) in original coordinates should be visible (used for text)

Values for `MathScale` arrays.

User PostScript

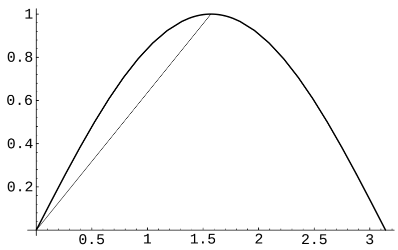
Since we now have an understanding of the coordinate systems which are used we can make more use of the primitive `PostScript` for adding raw PostScript into the output. There are tools to allow one to write raw PostScript in user coordinates.

MBeginOrig	enter user coordinates
MEndOrig	leave user coordinates

Tools for working with raw `PostScript` primitive.

This uses the `Epilog` option to add some `Graphics` primitives. In this case a `PostScript` primitive.

```
In[13]:= Plot[ Sin[x], {x,0,Pi}, Epilog ->
      PostScript["MBeginOrig 0 setlinewidth
             0 0 moveto 1.5708 1 lineto
             stroke MEndOrig"]]
```



By emitting raw PostScript in this way you can construct your own graphics primitive. The chapter on graphics programming had an example of exactly this. It is to be noted that the use of the primitive `PostScript` is very definitely an advanced feature. Using it is quite easy to corrupt the PostScript transformation matrices or the PostScript stack itself and make a result which cannot be rendered.