

# Appendix A

## Construction of Fourth-Order Central Differences

Assuming a uniform spacing of  $\delta$  between sample points, we seek an approximation of the derivative of a function at  $x_0$  which falls midway between two sample point. Taking the Taylor series expansion of the function at the four sample points nearest to  $x_0$  yields

$$f\left(x_0 + \frac{3\delta}{2}\right) = f(x_0) + \frac{3\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{3\delta}{2}\right)^2 f''(x_0) + \frac{1}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0) + \dots, \quad (\text{A.1})$$

$$f\left(x_0 + \frac{\delta}{2}\right) = f(x_0) + \frac{\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0) + \frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots, \quad (\text{A.2})$$

$$f\left(x_0 - \frac{\delta}{2}\right) = f(x_0) - \frac{\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0) - \frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots, \quad (\text{A.3})$$

$$f\left(x_0 - \frac{3\delta}{2}\right) = f(x_0) - \frac{3\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{3\delta}{2}\right)^2 f''(x_0) - \frac{1}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0) + \dots \quad (\text{A.4})$$

Subtracting (A.3) from (A.2) and (A.4) from (A.1) yields

$$f\left(x_0 + \frac{\delta}{2}\right) - f\left(x_0 - \frac{\delta}{2}\right) = \delta f'(x_0) + \frac{2}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots \quad (\text{A.5})$$

$$f\left(x_0 + \frac{3\delta}{2}\right) - f\left(x_0 - \frac{3\delta}{2}\right) = 3\delta f'(x_0) + \frac{2}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0) + \dots \quad (\text{A.6})$$

The goal now is to eliminate the term containing  $f'''(x_0)$ . This can be accomplished by multiplying (A.5) by 27 and then subtracting (A.6). The result is

$$27f\left(x_0 + \frac{\delta}{2}\right) - 27f\left(x_0 - \frac{\delta}{2}\right) - f\left(x_0 + \frac{3\delta}{2}\right) + f\left(x_0 - \frac{3\delta}{2}\right) = 24\delta f'(x_0) + O(\delta^5). \quad (\text{A.7})$$

Solving for  $f'(x_0)$  yields

$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \frac{9}{8} \frac{f\left(x_0 + \frac{\delta}{2}\right) - f\left(x_0 - \frac{\delta}{2}\right)}{\delta} - \frac{1}{24} \frac{f\left(x_0 + \frac{3\delta}{2}\right) - f\left(x_0 - \frac{3\delta}{2}\right)}{\delta} + O(\delta^4). \quad (\text{A.8})$$

A.378 *APPENDIX A. CONSTRUCTION OF FOURTH-ORDER CENTRAL DIFFERENCES*

The first term on the right-hand side is the contribution from the sample points nearest  $x_0$  and the second term is the contribution from the next nearest points. The highest-order term not shown is fourth-order in terms of  $\delta$ .

## Appendix B

# Generating a Waterfall Plot and Animation

Assume we are interested in plotting multiple snapshots of one-dimensional data. A waterfall plot displays all the snapshots in a single image where each snapshot is offset slightly from the next. On the other hand, animations display one image at a time, but cycle through the images quickly enough so that one can clearly visualize the temporal behavior of the field. Animations are a wonderful way to ascertain what is happening in an FDTD simulation but, since there is no way to put an animation on a piece of paper, waterfall plots also have a great deal of utility.

We begin by discussing waterfall plots. First, the data from the individual frames must be loaded into MATLAB. The m-file for a function to accomplish this is shown in Program B.1. This function, `readOneD()`, reads each frame and stores the data into a matrix—each row of which corresponds to one frame. `readOneD()` takes a single argument corresponding to the base name of the frames. For example, issuing the command `z = readOneD('sim');` would create a matrix `z` where the first row corresponded to the data in file `sim.0`, the second row to the data in file `sim.1`, and so on.

Note that there is a `waterfall()` function built into MATLAB. One could use that to display the data by issuing the command `waterfall(z)`. However, the built-in command is arguably overkill. Its hidden-line removal and colorization slow the rendering and do not necessarily aide in the visualization of the data.

The plot shown in Fig. 3.9 did not use MATLAB's `waterfall()` function. Instead, it was generated using a function called `simpleWaterfall()` whose m-file is shown in Program B.2. This command takes three arguments. The first is the matrix which would be created by `readOneD()`, the second is the vertical offset between successive plots, and the third is a vertical scale factor.

Given these two m-files, Fig. 3.9 was generated using the following commands:

```
z = readOneD('sim');
simpleWaterfall(z, 1, 1.9) % vertical offset = 1, scale factor = 1.9
xlabel('Space [spatial index]')
ylabel('Time [frame number]')
```

---

**Program B.1** `readOneD.m` MATLAB code to read one-dimensional data from a series of frames.

```
1 function z = readOneD(basename)
2 %readOneD(BASENAME) Read 1D data from a series of frames.
3 % [Z, dataLength, nFrames] = readOneD(BASENAME) Data
4 % is read from a series of data files all which have
5 % the common base name given by the string BASENAME,
6 % then a dot, then a frame index (generally starting
7 % with zero). Each frame corresponds to one row of Z.
8
9 % read the first frame and establish length of data
10 nFrames = 0;
11 filename = sprintf('%s.%d', basename, nFrames);
12 nFrames = nFrames + 1;
13 if exist(filename, 'file')
14     z = dlmread(filename, '\n');
15     dataLength = length(z);
16 else
17     return;
18 end
19
20 % loop through other frames and break out of loop
21 % when next frame does not exist
22 while 1
23     filename = sprintf('%s.%d',basename,nFrames);
24     nFrames = nFrames + 1;
25     if exist(filename, 'file')
26         zTmp = dlmread(filename, '\n');
27         if length(zTmp) ~= dataLength % check length matches
28             error('Frames have different sizes.')
29             break;
30         end
31         z = [z zTmp]; % append new data to z
32     else
33         break;
34     end
35 end
36
37 % reshape z to appropriate dimensions
38 z = reshape(z, dataLength, nFrames - 1);
39 z = z';
40
41 return;
```

---

**Program B.2** simpleWaterfall.m MATLAB function to generate a simple waterfall plot.

---

```

1 function simpleWaterfall(z, offset, scale)
2 %simpleWaterfall Waterfall plot from offset x-y plots.
3 % simpleWaterfall(Z, OFFSET, SCALE) Plots each row of z
4 % where successive plots are offset from each other by
5 % OFFSET and each plot is scaled vertically by SCALE.
6
7 hold off % release any previous plot
8 plot(scale * z(1, :)) % plot the first row
9 hold on % hold the plot
10
11 for i = 2:size(z, 1) % plot the remaining rows
12 plot(scale * z(i, :) + offset * (i - 1))
13 end
14
15 hold off % release the plot
16
17 return

```

---

A function to generate an animation of one-dimensional data sets is shown in Program B.3. There are multiple ways to accomplish this goal and thus one should keep in mind that Program B.3 is not necessarily the best approach for a particular situation. The function in Program B.3 is called `oneDmovie()` and it takes three arguments: the base name of the snapshots, and the minimum and maximum values of the vertical axis. The function uses a loop, starting in line 25, to read each of the snapshots. Each snapshot is plotted and the plot recorded as a frame of a MATLAB “movie” (see the MATLAB command `movie()` for further details). The `oneDmovie()` function returns an array of movie frames. As an example, assume the base name is “sim” and the user wants the plots to range from  $-1.0$  to  $1.0$ . The following commands would display the animation 10 times (the second argument to the `movie()` command controls how often the movie is repeated):

```

reel = oneDmovie('sim', -1, 1);
movie(reel, 10)

```

An alternative implementation might read all the data first, as was done with the waterfall plot, and then determine the “global” minimum and maximum values from the data itself. This would free the user from specify those value as `oneDmovie()` currently requires. Such an implementation is left to the interested reader.

---

**Program B.3** `oneDmovie.m` MATLAB function which can be used to generate an animation for multiple one-dimensional data sets. For further information on MATLAB movies, see the MATLAB command `movie`.

---

```

1 function reel = oneDmovie(basename, y_min, y_max)
2 % oneDmovie Create a movie from data file with a common base
3 % name which contain 1D data.

```

```
4 %
5 % basename = common base name of all files
6 % y_min    = minimum value used for all frames
7 % y_max    = maximum value used for all frames
8 %
9 % reel = movie which can be played with a command such as:
10 %      movie(reel, 10)
11 %      This would play the movie 10 times. To control the frame
12 %      rate, add a third argument specifying the desired rate.
13
14 % open the first frame (i.e., first data file).
15 frame = 1;
16 filename = sprintf('%s.%d', basename, frame);
17 fid = fopen(filename, 'rt');
18
19 % to work around rendering bug under Mac OS X see:
20 % <www.mathworks.com/support/solutions/
21 % data/1-VW0GM.html?solution=1-VW0GM>
22 figure; set(gcf, 'Renderer', 'zbuffer');
23
24 % provided fid is not -1, there is another file to process
25 while fid ~= -1
26     data=fscanf(fid, '%f');    % read the data
27     plot(data)                % plot the data
28     axis([0 length(data) y_min y_max]) % scale axes appropriately
29     reel(frame) = getframe;    % capture the frame for the movie
30
31     % construct the next file name and try to open it
32     frame = frame + 1;
33     filename = sprintf('%s.%d', basename, frame);
34     fid = fopen(filename, 'rb');
35 end
36
37 return
```

---

# Appendix C

## Rendering and Animating Two-Dimensional Data

The function shown below is MATLAB code that can be used to generate a movie from a sequence of binary (raw) files. The files (or frames) are assumed to be named such that they share a common base name then have a dot followed by a frame number. Here the frame number is assumed to start at zero. The function can have one, two, or three arguments. This first argument is the base name, the second is the value which is used to normalize all the data, and the third argument specifies the number of decades of data to display. Here the absolute value of the data is plotted in a color-mapped image. Logarithmic (base 10) scaling is used so that the value which is normalized to unity will correspond to zero on the color scale and the smallest normalized value will correspond, on the color scale, to the negative of the number of decades (e.g., if the number of decades were three, the smallest value would correspond to  $-3$ ). This smallest normalized value actually corresponds to a normalized value of  $10^{-d}$  where  $d$  is the number of decades. Thus the (normalized) values shown in the output varying from  $10^{-d}$  to 1. The default normalization and number of decades are 1 and 3, respectively.

---

**Program C.1** `raw2movie.m` MATLAB function to generate a movie given a sequence of raw files.

---

```
1 function reel = raw2movie(basename, z_norm, decades)
2 % raw2movie Creates a movie from "raw" files with a common base
3 %     name.
4 %
5 % The absolute value of the data is used together with
6 % logarithmic scaling. The user may specify one, two, or
7 % three arguments.
8 % raw2movie(basename, z_norm, decades) or
9 % raw2movie(basename, z_norm) or
10 % raw2movie(basename):
11 % basename = common base name for all files
12 % z_norm    = value used to normalize all frames, typically this
13 %           would be the maximum value for all the frames.
```

```
14 %           Default value is 1.
15 % decades = decades to be used in the display. The normalized
16 %           data is assumed to vary between 1.0 and 10^(-decades)
17 %           so that after taking the log (base 10), the values
18 %           vary between 0 and -decades. Default value is 3.
19 %
20 % return value:
21 % reel = movie which can be played with a command such as:
22 %           movie(reel, 10)
23 %           pcolor() is used to generate the frames.
24 %
25 % raw file format:
26 % The raw files are assumed to consist of all floats (in
27 % binary format). The first two elements specify the horizontal
28 % and vertical dimensions. Then the data itself is given in
29 % English book-reading order, i.e., from the upper left corner
30 % of the image and then scanned left to right. The frame number
31 % is assumed to start at zero.
32
33 % set defaults if we have less than three arguments
34 if nargin < 3, decades = 3; end
35 if nargin < 2, z_norm = 1.0; end
36
37 % open the first frame
38 frame = 0;
39 filename = sprintf('%s.%d', basename, frame);
40 fid = fopen(filename, 'rb');
41
42 if fid == -1
43     error(['raw2movie: initial frame not found: ', filename])
44 end
45
46 % to work around rendering bug under Mac OS X implementation.
47 figure; set(gcf, 'Renderer', 'zbuffer');
48
49 % provided fid is not -1, there is another file to process
50 while fid ~= -1
51     size_x = fread(fid, 1, 'single');
52     size_y = fread(fid, 1, 'single');
53
54     data = flipud(transpose(...
55         reshape(...
56             fread(fid, size_x * size_y, 'single'), size_x, size_y)...
57         ));
58
59     % plot the data
60     if decades ~= 0
```

```
61     pcolor(log10(abs((data + realmin) / z_norm)))
62     shading flat
63     axis equal
64     axis([1 size_x 1 size_y])
65     caxis([-decades 0])
66     colorbar
67 else
68     pcolor(abs((data + realmin) / z_norm))
69     shading flat
70     axis equal
71     axis([1 size_x 1 size_y])
72     caxis([0 1])
73     colorbar
74 end
75
76 % capture the frame for the movie (MATLAB wants index to start
77 % at 1, not zero, hence the addition of one to the frame)
78 reel(frame + 1) = getframe;
79
80 % construct the next file name and try to open it
81 frame = frame + 1;
82 filename = sprintf('%s.%d', basename, frame);
83 fid = fopen(filename, 'rb');
84
85 end
```

---



# Appendix D

## Notation

$c$	speed of light <i>in free space</i>
$N_{\text{freq}}$	index of spectral component corresponding to a frequency with discretization $N_\lambda$
$N_\lambda$	number of points per wavelength for a given frequency (the wavelength is the one pertaining to propagation in free space)
$N_L$	number of points per skin depth ( $L$ for loss)
$N_P$	number of points per wavelength at peak frequency of a Ricker wavelet
$N_T$	number of time steps in a simulation
$S_c$	Courant number ( $c\Delta_t/\Delta_x$ in one dimension)
$s_t$	Temporal shift operator
$s_x$	Spatial shift operator (in the $x$ direction)



# Appendix E

## PostScript Primer

### E.1 Introduction

PostScript was developed by Adobe Systems Incorporated and is both a page-description language and a programming language. Unlike a JPEG or GIF file which says what each pixel in an image should be, PostScript is “vector based” and thus a PostScript file specifies graphic primitives, such as lines and arcs. These primitives are described by various PostScript commands. The quality of the image described by a PostScript file will depend on the output device. For example, a laser printer with 1200 dots per inch will draw a better curved line than would a laser printer with 300 dots per inch (although both will typically produce very good output).

The goal here is to show how one can easily generate PostScript files which convey information about an FDTD simulation. Thus we are more interested in the page-description aspects of PostScript rather than its programming capabilities. (There is a wonderful book and Web site by Bill Casselman that describe PostScript extremely well while illustrating a host of mathematical concepts. The book is entitled *Mathematical Illustrations: A Manual of Geometry and PostScript* which you can find at [www.math.ubc.ca/~cass/graphics/manual/](http://www.math.ubc.ca/~cass/graphics/manual/). It is well worth checking out.)

PostScript is a Forth-like language in that it uses what is known as postfix notation. If you have used an RPN (reverse Polish notation) calculator, you are familiar with postfix notation. You put arguments onto a “stack” and then select an operation which “pops” the arguments from the stack and operates on them. For example, to add 3 and 12 you would enter the following:

```
3
<ENTER>
12
+
```

When 3 is typed on the keypad, it is placed at the top of the stack. It is pushed onto the next stack location by hitting the ENTER key. When 12 is typed, it is put at the top of the stack. Hitting the plus sign tells the calculator you want to add the top two numbers on the stack, i.e., the 3 and 12. These numbers are popped (i.e., taken) from the stack, and the result of the addition (15) is placed at the top of the stack.

The PostScript language is much like this. Arguments are given before the operations. Giving arguments before operations facilitates the construction of simple interpreters. PostScript interpreters typically have the task of translating the commands in a PostScript file to some form of viewable graphics. For example, there are PostScript printers which translate (interpret) a PostScript file into a printed page. Most computers have PostScript interpreters which permit the graphics described in a PostScript file to be displayed on the screen. There are free PostScript interpreters available via the Web (you should do a search for GhostScript if you are in need of an interpreter).

## E.2 The PostScript File

A file which contains PostScript commands, which we will call a PostScript file, is a plain ASCII file which must start with “%!PS”. These characters are often referred to as a “magic word.” Magic words appear at the start of many computer files and identify the contents of the file. This %!PS magic word identifies the contents of the file as PostScript to the interpreter. (The names of PostScript file often end with the suffix .ps, but the interpreter does not care what the file name is.) The last command in a PostScript file is typically `showpage`. This command essentially tells the interpreter that all the commands have been given and the page (or screen image or whatever) should be rendered.

What comes between %!PS and `showpage` are the commands which specify how the page should appear. Before exploring some of these commands it is important to know that a PostScript interpreter, by default, thinks in terms of units of “points” which are not points in the geometric sense, but rather 1/72 of an inch. Points are a traditional unit used in the printing industry (thus a “12-point font” is one for which a typical capital letter is 12/72 of an inch high). A default “page” is 8.5 by 11 inches and thus 612 by 792 points. The origin is the lower left corner of the page.

## E.3 PostScript Basic Commands

The PostScript command `moveto` takes two arguments: the  $x$  and  $y$  coordinates to which the current point should be moved. You can think of the current point as akin to the point where the tip of a pen is moved. To define a line we can give the command `lineto`. `lineto` also takes two arguments: the  $x$  and  $y$  coordinates of the point to which the line should be drawn. In PostScript, after issuing the `lineto` command we have merely defined the path of the line—we have not actually drawn anything yet. You can think of this as the pen having drawn the line in invisible ink. We have to issue one more command to make the line visible, the `stroke` command.

A complete PostScript file (which we will identify as a “Program”) which draws a line from the point (100, 200) to the point (300, 600) is shown in Program E.1.

---

**Program E.1** PostScript commands to draw a single tilted line.

---

```
%!PS
100 200 moveto
```

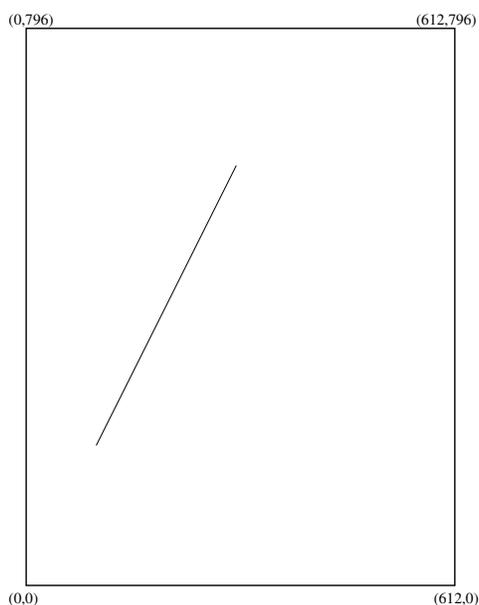


Figure E.1: Simple line rendered by the PostScript commands giving in Program E.1 and E.2. The surrounding box and corner labels have been added for the sake of clarity.

```
300 600 lineto
stroke
showpage
```

---

The image drawn by these commands is shown in Fig. E.1. The surrounding border and coordinate labels have been added for clarity. The only thing which would actually be rendered is the tilted line shown within the border.

Instead of using the command `lineto` to specify the point to which we want the line to be drawn, the command `rlineto` can be used where now the arguments specify the relative movement from the current point (hence the “r” for relative). The arguments of `rlineto` specify the relative displacement from the current point. In the commands in Program E.1, the line which was drawn went 200 points in the  $x$  direction and 400 points in the  $y$  direction from the starting point. Thus instead of writing `300 600 lineto`, one could obtain the same result using `200 400 rlineto`. PostScript does not care about whitespace and the percent sign is used to indicate the start of a comment (the interpreter ignores everything from the percent sign to the end of the line). Thus, another file which would also yield the output shown in Fig. E.1 is shown in Program E.2. (The magic word must appear by itself on the first line of the file.)

---

**Program E.2** PostScript commands to draw a single tilted line. Here the `rlineto` command is used. The resulting image is identical to the one produced by Program E.1 and is shown in Fig. E.1.

---

```
%!PS
% tilted line using the rlineto command
100 200 moveto 300 600 lineto stroke showpage
```

---

When creating graphics it is often convenient to redefine the origin of the coordinate system to a point which is more natural to the object being drawn. PostScript allows us to translate the origin to any point on the page using the `translate` command. This command also takes two arguments corresponding to the point in the current coordinate system where the new origin should be located. For example, let us assume we want to think in terms of both positive and negative coordinates. Therefore we wish to place the origin in the middle of the page. This can be accomplished with the command `306 396 translate`. The PostScript commands shown in Program E.3 demonstrate the use of the `translate` command.

---

**Program E.3** PostScript commands which first translate the origin to the center of the page and then draw four lines which “radiate away” from the origin. The corner labels show the corner coordinates after the translation of the origin to the center of the page.

---

```
%!PS
306 398 translate % translate origin to center of page
 100 100 moveto 50 50 rlineto stroke
-100 100 moveto -50 50 rlineto stroke
-100 -100 moveto -50 -50 rlineto stroke
 100 -100 moveto 50 -50 rlineto stroke
showpage
```

---

Program E.3 yields the results shown in Fig. E.2.

As you might imagine, thinking in terms of units of points ( $1/72$  of an inch) is not always convenient. PostScript allows us to scale the dimensions by any desired value using the `scale` command. In fact, one can use a different scale factor in both the  $x$  and the  $y$  directions and thus `scale` takes two arguments. However, we will stick to using equal scaling in both directions.

In the previous example, all the locations were specified in terms of multiples of 50. Therefore it might make sense to scale the dimensions by a factor of 50 (in both the  $x$  and  $y$  direction). This scaling should be done *after* the translation of the origin. We might anticipate that the commands shown in Program E.4 would render the same output as shown in Fig. E.2.

---

**Program E.4** PostScript file where the units are scaled by a factor of 50 in both the  $x$  and  $y$  dimensions.

---

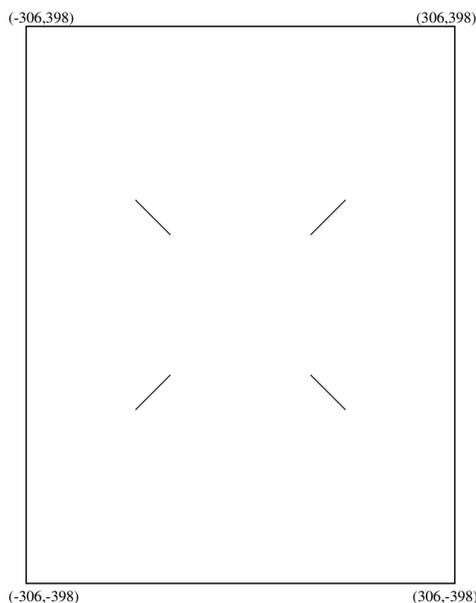


Figure E.2: Output rendered by Program E.3 which translates the origin to the center of the page. This output is also produced by Program E.5 and Program E.6.

```

%!PS
306 398 translate
50 50 scale % scale units in x and y direction by 50
 2  2 moveto 1  1 rlineto stroke
-2  2 moveto -1 1 rlineto stroke
-2 -2 moveto -1 -1 rlineto stroke
 2 -2 moveto 1 -1 rlineto stroke
showpage

```

---

However this file yields the output shown in Fig. E.3. Note that the lines which radiate from origin are now much thicker. In fact, they are 50 times thicker than they were in the previous image. By default, a line in PostScript has a thickness of unity i.e., one point. The `scale` command scaled the line thickness along with all the other dimensions so that now the line thickness is 50 points.

Although we have only given integer dimensions so far, as far as PostScript is concerned all values are actually real numbers (i.e., floating-point numbers). We can control the line thickness with the `setlinewidth` command which takes a single argument. If we want the line thickness still to be one point, the line thickness should be set to the inverse of the scale factor, i.e.,  $1/50 = 0.02$ . Also, it is worth noting that the `stroke` command does not have to be given after each drawing command. We just have to ensure that it is given before the end of the file (or before the line style changes to something else). Thus, a PostScript file which scales the dimensions by 50 and produces the same output as shown in Fig. E.2 is shown in Program E.4

---

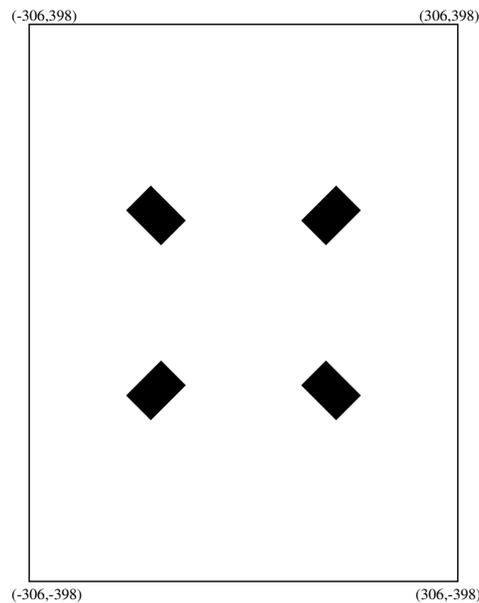


Figure E.3: Output rendered by Program E.4 which scales the units by 50.

**Program E.5** PostScript file where the units are scaled by a factor of 50 and the line thickness is corrected to account for this scaling. Note that a single `stroke` command is given.

---

```

%!PS
306 398 translate
50 50 scale
0.02 setlinewidth % correct line thickness to account for scaling
 2 2 moveto 1 1 rlineto
-2 2 moveto -1 1 rlineto
-2 -2 moveto -1 -1 rlineto
 2 -2 moveto 1 -1 rlineto stroke
showpage

```

---

PostScript permits the use of named variables and, as we shall see, named procedures. This is accomplished using the `def` command which takes, essentially, two arguments: the first being the literal string which is the variable or procedure name and the second being the value or procedure (where the procedure would be enclosed in braces). A literal string is a backslash character followed by the string. For example, the following sets the variable `scalefactor` to 50:

```
/scalefactor 50 def
```

After issuing this command, we can use `scalefactor` in place of 50 everywhere in the file.

The PostScript language includes a number of mathematical functions. One can add using `add`, subtract using `sub`, multiply using `mul`, and divide using `div`. Each of these functions

takes two arguments consistent with an RPN calculator. To calculate the inverse of 50, one could issue the following command:

```
1 50 div
```

This places 1 on the stack, then 50, and then divides the two. The result, 0.02, remains at the top of the stack.

The program shown in Program E.6 uses the `def` and `div` commands and is arguably a bit cleaner and better self-documenting than the one shown in Program E.5. Program E.6 also produces the output shown in Fig. E.2.

---

**Program E.6** PostScript file which uses the `def` command to define a scale-factor which is set to 50. The inverse of the scale-factor is obtained by using the `div` command to divide 1 by the scale-factor.

---

```
%!PS
306 398 translate
% define "scalefactor" to be 50
/scalefactor 50 def
% scale x and y directions by the scale factor
scalefactor scalefactor scale
% set line width to inverse of the scale factor
1 scalefactor div setlinewidth
 2 2 moveto 1 1 rlineto
-2 2 moveto -1 1 rlineto
-2 -2 moveto -1 -1 rlineto
 2 -2 moveto 1 -1 rlineto stroke
showpage
```

---

The `arc` command takes five arguments: the  $x$  and  $y$  location of the center of the arc, the radius of the arc, and the angles (in degrees) at which the arc starts and stops. For example, the following command would draw a complete circle of radius 0.5 about the point (2, 2):

```
2 2 0.5 0 360 arc stroke
```

Let us assume we wish to draw several circles, each of radius 0.5. We only wish to change the center of the circle. Rather than specifying the `arc` command each time with all its five arguments, we can use the `def` command to make the program more compact. Consider the program shown in Program E.7. Here the `def` command is used to say that the literal `circle` is equivalent to `0.5 0 360 arc stroke`, i.e., three of the arguments are given to the `arc` command—one just has to provide the two missing arguments which are the  $x$  and  $y$  location of the center of the circle. The output produced by this program is shown in Fig. E.4.

---

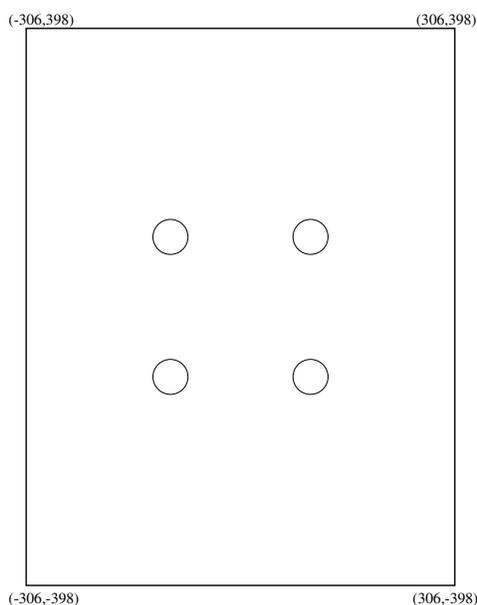


Figure E.4: Output rendered by Program E.7.

**Program E.7** PostScript file which renders the output shown in Fig. E.4.

---

```
%!PS
306 398 translate
/scalefactor 50 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/circle {0.5 0 360 arc stroke} def
 2 2 circle
-2 2 circle
-2 -2 circle
 2 -2 circle
showpage
```

---

In addition to `stroke`-ing a path, PostScript allows paths to be `fill`-ed using the `fill` command. So, instead of drawing a line around the perimeter of the circles shown in Fig. E.4, one can obtain filled circles by issuing the `fill` command instead of the `stroke` command. Program E.8 and the corresponding output shown in Fig. E.5 illustrate this.

**Program E.8** PostScript file which defines a `stroke`-ed and `fill`-ed circle. The corresponding output is shown in Fig. E.5.

---

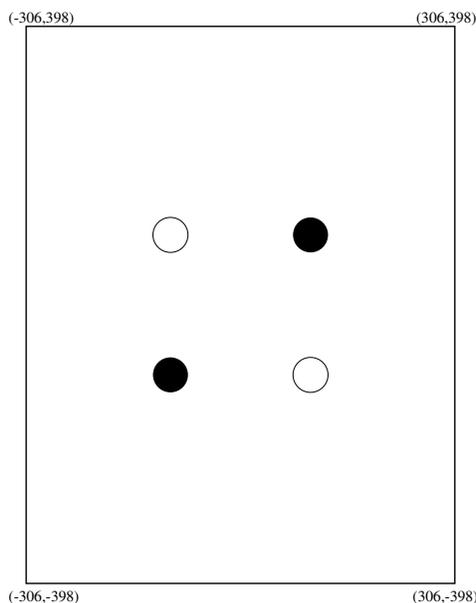


Figure E.5: Output rendered by Program E.8.

```

%!PS
306 398 translate
/scalefactor 50 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/circle {0.5 0 360 arc stroke} def
/circlef {0.5 0 360 arc fill} def
 2 2 circlef
-2 2 circle
-2 -2 circlef
 2 -2 circle
showpage

```

---

The PostScript commands we have considered are shown in Table E.1.

Instead of using an editor to write a PostScript file directly, we can use another program to generate the PostScript file for us. Specifically, let us consider a C program which generates a PostScript file. This program is supposed to demonstrate how one could use PostScript to display a particular aspect of an FDTD grid. For example, let us assume we are using a  $TM^z$  grid which is 21 cells by 21 cells. There is a PEC cylinder with a radius of 5 which is centered in the grid. We know that the  $E_z$  nodes which fall within the cylinder should be set to zero and this zeroing operation would be done with a for-loop. However, precisely which nodes are being set to zero? The code shown in Program E.9 could be incorporated into an FDTD program. This code produces the PostScript output file `grid.ps` which renders the image shown in Fig. E.6. The first several lines of the file `grid.ps` are shown in Fragment E.10.

Command	Description
$x\ y$ moveto	move current point to $(x, y)$
$x\ y$ lineto	draw a line from current point to $(x, y)$
$\delta_x\ \delta_y$ rlineto	from current point draw a line over $\delta_x$ and up $\delta_y$
$x\ y$ translate	translate the origin to the point $(x, y)$
$s_x\ s_y$ scale	scale $x$ and $y$ coordinates by $s_x$ and $s_y$
stroke	“apply ink” to a previously defined path
fill	fill the interior of a previously defined path
$w$ setlinewidth	set the line width to $w$
$d_1\ d_2$ div	calculate $d_1/d_2$ ; result is placed at top of stack
$x_c\ y_c\ r\ a_1\ a_2$ arc	draw an arc of radius $r$ centered at $(x_c, y_c)$ starting at angle $a_1$ and ending at angle $a_2$ (degrees)
<i>literal</i> { <i>definition</i> } def	define the <i>literal</i> string to have the given <i>definition</i> ; braces are needed if the definition contains any whitespace

Table E.1: An assortment of PostScript commands and their arguments.

---

**Program E.9** C program which generates a PostScript file. The file draws either a cross or a filled circle depending on whether a node is outside or inside a circular boundary, respectively. The rendered image is shown in Fig. E.6.

---

```

1  /* C program to generate a PostScript file which draws a cross
2   * if a point is outside of a circular boundary and draws a
3   * filled circle if the point is inside the boundary.
4   */
5
6  #include <stdio.h>
7  #include <math.h>
8
9  int is_inside_pec(double x, double y);
10
11 int main() {
12     int m, n;
13
14     FILE *out;
15
16     out = fopen("grid.ps", "w"); // output file is "grid.ps"
17
18     /* header material for PostScript file */
19     fprintf(out, "%!PS\n"
20             "306 396 translate\n"
21             "/scalefactor 20 def\n"
22             "scalefactor scalefactor scale\n"
23             "1 scalefactor div setlinewidth\n"

```

```

24     "/cross {moveto\n"
25     "         -.2 0 rmoveto .4 0 rlineto\n"
26     "         -.2 -.2 rmoveto 0 .4 rlineto stroke} def\n"
27     "/circle {.2 0 360 arc fill} def\n"
28     );
29
30 for (m=-10; m<=10; m++)
31   for (n=-10; n<=10; n++)
32     if (is_inside_pec(m,n)) {
33       fprintf(out,"%d %d circle\n",m,n);
34     } else {
35       fprintf(out,"%d %d cross\n",m,n);
36     }
37
38 fprintf(out,"showpage\n");
39
40 return 0;
41 }
42
43 /* Function returns 1 if point (x,y) is inside a circle (or on
44 * the perimeter of circle) and returns 0 otherwise.
45 */
46 int is_inside_pec(double x, double y) {
47   double radius = 5.0;
48
49   return x*x + y*y <= radius*radius;
50 }

```

---

**Fragment E.10** First several lines of the file `grid.ps` which is produced by Program E.9.

---

```

%!PS
306 396 translate
/scalefactor 20 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/cross {moveto
        -.2 0 rmoveto .4 0 rlineto
        -.2 -.2 rmoveto 0 .4 rlineto stroke} def
/circle {.2 0 360 arc fill} def
-10 -10 cross
-10 -9 cross
-10 -8 cross
-10 -7 cross

```

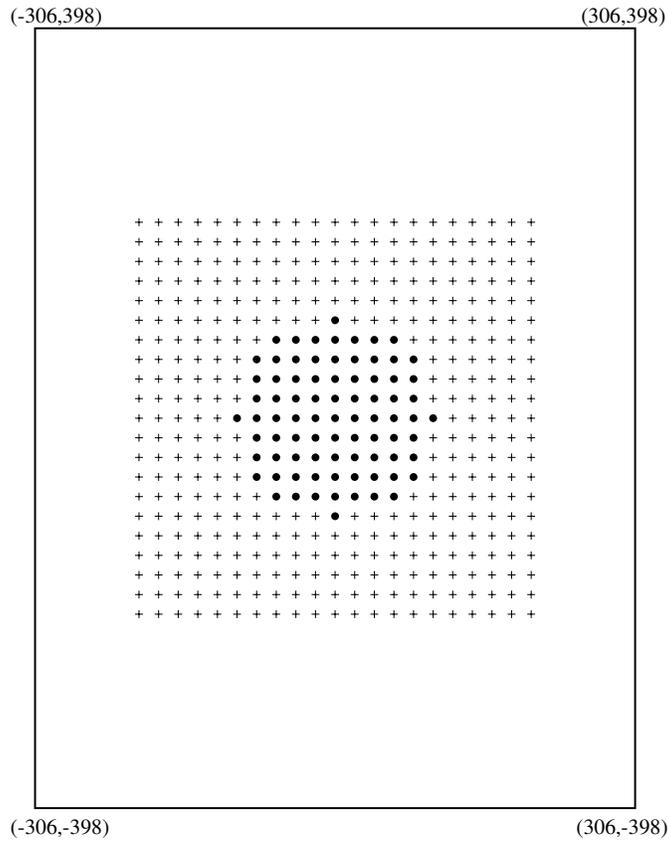


Figure E.6: Grid depiction rendered by the file `grid.ps` which is produced by Program E.9. Crosses corresponds to nodes which are outside a circular boundary of radius 5. Filled circles correspond to nodes inside the boundary (or identically on the perimeter of the boundary).

-10 -6 cross

·  
·  
·

---

