Appendix 1 Graphical and Mathematical Software

Successful numerical model building depends on your ability to express algorithms in a form that a computer can understand, and on how well you can see the model's behavior (and perhaps hear or even feel it, Section A1.2.6). Good *visualization* is essential; notable discoveries have come first in simulations and then later been understood analytically. Fortunately, there are a number of good popular tools that help with these tasks. The most important distinction is between compiled languages with subroutine libraries, and interactive environments. The great virtue of the latter is the ease with which problems can be specified and solved, but the penalty can be a slowdown by 1–2 orders of magnitude over the intrinsic hardware speed. Obtaining performance at the limits of processors and output devices usually requires lower-level control. The interactive environments also make it easy to create impressive graphics, but once again for demanding tasks more direct control can be needed. This appendix starts by reviewing important programming languages and libraries, covers some of the interactive environments, and then briefly introduces four useful and ubiquitous standards that can be used for graphics: Postscript, X Windows, OpenGL, and Java. Sample programs are provided to draw and animate lines, images, and surfaces. These short programs are intended to show the minimal code needed to produce output; they leave out some of the steps that are not needed for such simple tasks, but that are part of the full specification for well-behaved program. They can be used as templates for other applications, and to provide some guidance in navigating through the (large) language reference manuals.

Along with worrying about a program's efficiency, don't lose track of the efficiency of the algorithms that it uses, and of your own common sense. Advanced programming tools can serve as effective aids in the production of nonsense. They do not substitute for a good understanding of how an algorithm scales, and of how its results can be verified, as essential ingredients in the modeling process (see the introduction to Part Two).

A1.1 MATH PACKAGES

A1.1.1 Programming Environments

A1.1.1.1 Languages

The most important distinction among computer languages for modeling is how far they remove you from the hardware. The fastest code possible comes from learning the instruction set of a processor and hand-coding assembly language. However, this is very slow to write and difficult to understand, and so is almost never done for general-purpose computing. *Compiled* languages let you express your ideas more naturally and then turn them into reasonably efficient machine instructions. *Interpreted* languages can be designed to match particular domains (such as mathematics), and so can allow algorithms to be expressed still more rapidly and conveniently, but usually pay a penalty in producing the least efficient code. The overall goal is to reduce the time to develop and then execute a program. If it takes months to write a program that executes in minutes, it's worth looking for a better language; conversely, if you're going to wait weeks for the output from your program then you should make sure that it's as efficient as possible. It's important to remember that abstraction alone is not a desirable goal unless it helps to make the entire development process more efficient; many graceful programs have ground to a halt from excessive parameter passing.

The two primary compiled languages used for mathematical computing are *C* and *Fortran*. Fortran is much older; it was developed at IBM in 1954 as the first commerical high-level language, and for many years it was equivalent to scientific computing. That is what everyone used, and what every computer supported. C was developed later at Bell Labs (the language reference was first published in 1978), originally as the language to write the UNIX operating system and its utilities. C was not intended for mathematics and so some important features from Fortran are missing such as the `complex` data type. However, C grew to replace Fortran for most computing on workstations. There are two reasons for this: it had richer abstractions for data structures and flow control, and it continues to be the language most closely associated with operating systems and so is usually the first and best-supported language on any platform. The exception to this is supercomputers, where Fortran lives on because of the large existing base of programs and programmers. A more modern compiled language that is also used for numerical analysis is *ADA* (`http://www.acm.org/sigada`), which was standardized in 1983 and then 1995 for writing portable, large-scale, real-time applications and has a devoted following drawn by its purity (as well as its significant military/industrial investment).

Both Fortran and C have important extensions that address some of their original limitations. The successor to C is *C++*, which provides much more flexibility in defining operators and objects that can be inherited, modified, and reused. The new version of Fortran is *Fortran 90* (named after the year of the IEEE committee that developed it), and the closely related *High Performance Fortran* (*HPF*). These add more modern data types and flow control, and add important syntactical conveniences that let many explicit loops be replaced by operators on more complex data structures (such as saying `A=B+C`, where `A`, `B`, and `C` are arrays). This helps the programmer, because the code is simpler to read and write, and the compiler, because it makes the parallelism explicit.

C is used on everything from microcontrollers to supercomputers and so there is an enormous range of commerical and open-source compilers. The best known is *GCC* from the GNU project; this is perhaps the most thoroughly tested and debugged compiler of any kind because so many people from around the world have contributed to its development. GCC is available at `http://www.gnu.org/software/gcc` (along with front-ends for most of the other languages discussed here, and cross-compilers targeting an enormous range of platforms).

Starting with the first recorded bug removed from a program (a moth trapped in Harvard's Mark II Aiken Relay Calculator in 1945), the tools used to fix computer programs have been at least as important as the tools used to write them. The debugging counterpart to GCC is *GDB* (`http://www.gnu.org/software/gdb`), which can be called from within graphical environments such as *DDD* (`http://www.gnu.org/software/ddd`) that interactively show a program's variables and instructions. From there, it's possible to use a full *Integrated Development Environment* (*IDE*) such as *KDevelop* (`http://www.kdevelop.org`) and *Visual Studio* (`http://msdn.microsoft.com/vstudio`); in addition to compilation and debugging these provide aids for prototyping user interfaces, preparing packages for distribution, and managing large projects.

Interpreted languages are rarely used for serious mathematical computing. Interpreted implementations of popular languages such as *Tcl/Tk* (`http://www.tcl.tk`), *Python* (`http://www.python.org`), and *BASIC* (`http://msdn.microsoft.com/vbasic`), are typically too slow to be useful for anything but the simplest problems, and floating point math can be truly painful in *LISP*. They are used for rapid prototyping, though, permitting easy integration of sophisticated interface elements and operating system features within a mature coding environment. These can be particularly convenient for building complex user interfaces linked to compiled code. But one interesting long-standing exception is *APL*, which has a curious place in the history of computer languages. It was originally developed to be a more coherent replacement for conventional mathematical notation, and happened to be useful as a programming language [Iverson, 1962; Orth, 1976]. Therefore, it is ideally suited for expressing algorithms. It is possible to write one (cryptic) line of APL that expresses the same thing as many lines of conventional code. Because, unlike C, it is better matched to the minds of algorithm writers rather than compiler writers, it has languished with relatively poor support. Although it is still available (both as APL and through its descendent *J*, `http://www.acm.org/sigapl`), its greatest impact may be via its many fans on the evolution of languages that have incorporated some of its best features, including Fortran 90, *Mathematica*, and *S-PLUS* plus along with its open-source variant *R* (`http://www.r-project.org`).

One final language with great promise (and equally great hype) that merges many of the strengths of these other languages is *Java*, covered in Section A1.2.4.

A1.1.1.2 Subroutine Libraries

There are many algorithms, such as matrix inversion, that are difficult to write well but that are used routinely (pun not intended). For this reason there are a number of collections of useful mathematical subroutines.

Perhaps the best-known and most widely used are those found in *Numerical Recipes* (`http://www.nr.com`). This marvelous collection of about 400 routines covers everything from `addint` to `zroots`, and is available for C, C++, and Fortran (77 and 90), along with versions from older languages and under development for newer ones. The routines are carefully documented and explained in the accompanying text [Press *et al.*, 2007], and because the source code is available they are easily modified and incorporated into other programs. Their best feature is that they reflect the working experience of a number of leading scientists rather than the theoretical opinions of a few numerical analysts and so replace rigor with relevance.

Among the major commerical subroutine libraries available are *NAG* (`http://www.nag.co.uk`) and *IMSL* (`http://www.vni.com`) . These are much more expensive than Numerical Recipes, and because they do not release the source code they are closer to black boxes, hence they are less useful for anything other than production applications which might need the extra support and algorithm sophistication.

Finally, *Netlib* (`http://www.netlib.org`) houses a broad collection of specialized routines that are freely available, as well as related information such as benchmarks on many machines. At its heart are the *Basic Linear Algebra Subprograms* (*BLAS*, `http://www.netlib.org/blas`), which were written for early supercomputers and now form the core of a number of mathematical packages, including Java numerical class libraries (`http://math.nist.gov/javanumerics`), and the *MATLAB* environment.

A1.1.2 Interactive Environments

Interactive environments all provide a programming language, useful built-in primitives, and some kind of graphical capabilities. The best can compete with the performance of much more laboriously-written compiled programs, and add features such as animation, sonification, and symbolic math.

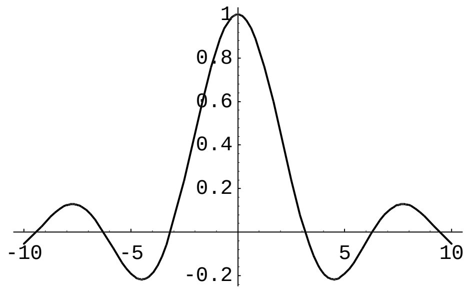
Symbolic math is the ability to do math as you would by manipulating symbols instead of numbers wherever possible. π remains π rather than 3.14159, $\int \sin = \cos$, and so forth. The first major symbolic math package was *Macsyma*, written at MIT in 1968. It languished when the government reduced funding for symbolic math (among other reasons), but after a commerical phase it has evolved into the *Maxima* open-source project (`http://maxima.sourceforge.net`). It is at first liberating to be able to type an expression into an environment like Maxima:

```
(C1) diff(sin(x)/x,x);
                              COS(x)   SIN(x)
(D1)                          ------ - ------
                                x         2
                                         x
```

and get an analytical answer. But it soon becomes clear that most problems that *look* hard really *are* hard and do not have closed-form solutions; integrating $\sin(x)/x$ just gives the *sine integral function* $\mathrm{Si}(x)$ which cannot be further simplified. The greatest strength of symbolic math environments is bookkeeping for calculations that are large but straightforward (such as analytical perturbation theory), along with interactive exploration of the behavior functions.

Maple is another symbolic math environment. It was originally developed at the University of Waterloo in Canada, and continuing work is also being done by the ETH in Zürich. Because of this ongoing academic grounding Maple has perhaps the strongest of the symbolic math engines (although people fight religiously about this). It is available

```
In[1]:=
    Plot[Sin[x]/x,{x,-10,10}]
```



```
Out[1]=
    -Graphics-
```

Figure A1.1. Output from Mathematica.

in a stand-alone environment (`http://www.maplesoft.com`), and is commonly used as a module in MATLAB or *Mathcad* (described below).

Mathematica (`http://www.wri.com`) is the creation of Stephen Wolfram, intended to meet an unfilled need that he saw for software that would reflect how people (including himself) do math. It is distinguished by allowing almost any style of programming. Symbolic and variable precision numerical calculations can be freely intermixed, and the language is equally adept at APL-like array processing, C-like coding, and LISP-like data structure manipulation. For this reason Mathematica is very common in physics research. Figure A1.1 shows a simple example, now plotting $\sin(x)/x$. Note that there is no need to specify where to plot the points; Mathematica is able to examine the function and decide how to sample it, including the division of 0/0 at the origin that has a limiting value of 1.

SymPy

Sage

MATLAB (`http://www.mathworks.com`) has a very different lineage. Cleve Moler was one of the authors of the *Linpack* and *Eispack* Fortran libraries (for solving linear systems of equations and eigenvalue problems, respectively, available from Netlib). MATLAB was originally written as an instructional interactive front end to the routines, and has grown into the most commonly used engineering math environment. The underlying language reflects its lineage from the early days of Fortran linear algebra, but its programming constructs and data structures are catching up with more modern programming styles. Many toolboxes are available for application domains such as optimization, neural networks, signal processing, and Maple for symbolic math, and there is an open-source version *Octave* (`http://www.octave.org`). To produce a $\sin(x)/x$ plot in MATLAB the input is

```
>> x = -10.5:.2:10.5;
>> plot(x,sin(x)./x)
```

and the output is shown in Figure A1.2. It is necessary to explicitly construct an array of points to be plotted, and to avoid the tricky point at the origin, but it is easy to express the construction of a vector and the parallel element-by-element operations on that vector.

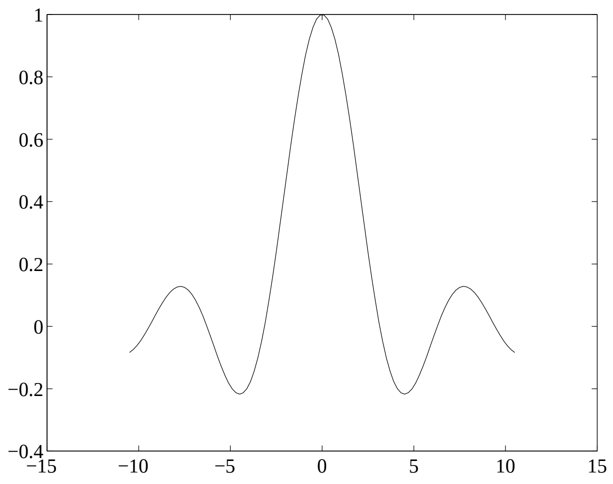


Figure A1.2. Output from MATLAB.

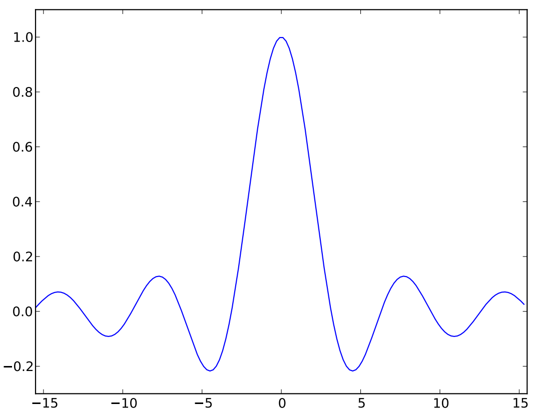


Figure A1.3. Output from matplotlib.

When it is possible to express an algorithm as such a vectorized construct in MATLAB the performance is within an order of magnitude of optimized compiled code, but if looping is needed then the interpretation overhead increases the performance penalty to 1–2 orders of magnitude.

Spreadsheets such as *Excel* (`http://www.microsoft.com/office/excel`) and *Gnumeric* (`http://www.gnome.org/projects/gnumeric`) were originally aimed at manipulating tables of numbers for financial analysis, but have grown into more general environments for working with all kinds of information. Because of this origin they're particularly useful for managing and annotating data, and because of their extensibility and popularity there are a large number of available plug-ins.

The last math environment to be mentioned is *Mathcad* (`http://www.mathsoft.com`). This has completely rethought the interface to do math on a computer, dispensing

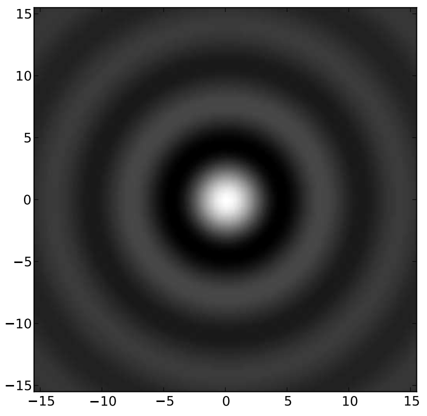


Figure A1.4. Output from matplotlib.

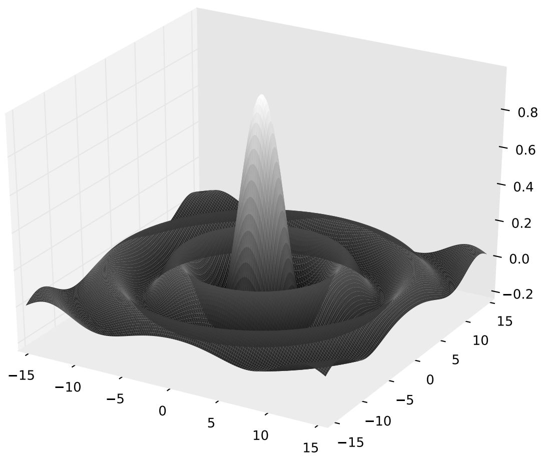


Figure A1.5. Output from matplotlib.

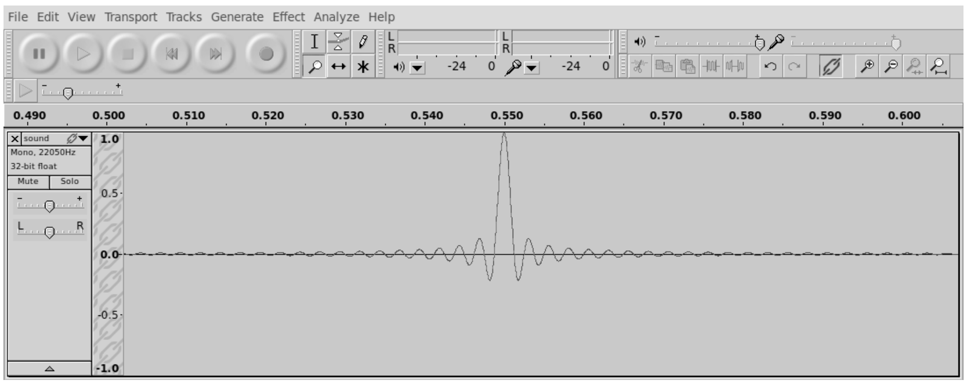


Figure A1.6. Output from matplotlib.

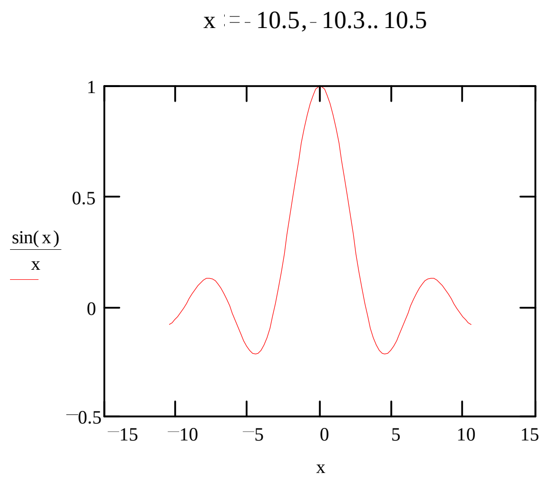


Figure A1.7. Output from Mathcad.

with any notion of a command line. Instead, the metaphor is an active piece of paper that can be written on anywhere. When valid expressions are written they are evaluated, and if they reference expressions anywhere else on the sheet they become linked and will update each other if any one of them is changed. Our sin(x)/x example in Mathcad is shown in Figure A1.7. The expression above the graph defines a vector, and then filling in expressions on axes drawn on the page evaluates them based on the definition of the vector. Changing the values in the vector would automatically update the plot.

A1.2 GRAPHICS TOOLS

A1.2.1 Postscript

PostScript was developed by John Warnock and Chuck Geschke (at Xerox's Palo Alto Research Center and then at Adobe Systems) as a page description language, motivated by the difficulty of using pixel-based approaches to communicate with increasingly high-resolution printers. The first release was in 1984. It is a true programming language, so that the processing needed to describe a page can be shared between the host and the printer, and it is *device-independent*, so that the same program can be used on any PostScript printer and will take advantage of its available capabilities (most importantly, the resolution). With the advent of *Display PostScript*, and, more modestly, good software interpreters such as *Ghostscript* and previewers such as *Ghostview* (available from `http://www.cs.wisc.edu/ghost`), PostScript can be used interactively as well as for producing hardcopy. One of the reasons for the success of Postscript is the careful balance it strikes between the generality of recognizing that describing a page is really a programming problem, and the specificity of making it easy to describe a page. Postscript is documented in [Taft & Walden, 1990].

PostScript comes in three flavors: ordinary PostScript, *Encapsulated PostScript* (*EPS*), and the *Portable Document Format* (*PDF*) aimed at online access. Most people use PostScript without thinking much about it, but there are many reasons to learn to write PostScript directly. It is easy to do, it allows you to bypass intervening programs and specify exactly what you want, and knowledge of PostScript is frequently useful for patching up anomalies in the output from other programs.

Regular PostScript is intended to be sent to a printer; Encapsulated PostScript is a set of conventions for PostScript figures to be included in other documents. EPS does not have a `showpage` command to eject a page, and it must have *DSC* (Document Structuring Convention) information such as the size of a bounding box that encloses the image (although it is good form for all PostScript programs to include this). Both types of files start with the characters `%!` to indicate that it is PostScript.

Our first example program, `psline`, shows how to write a PostScript file that sets up a coordinate system, sets the linewidth and darkness, and then draws a line (once again, the function sin(x)/x). The default coordinate system for PostScript has the origin in the lower-left corner of the page, and one unit is equal to 1/72 inch (a *printer's point*). This program is certainly longer and less transparent than `Plot[Sin[x]/x]`, but once you understand this program you will be able to write directly to any PostScript device.

```
/*
* psline.c
* (c) Neil Gershenfeld 9/1/97
* demonstrates drawing lines in PostScript by drawing
*    sin(k*x)/k*x
*/

#include <math.h>
#include <stdio.h>

int i;
float x,y;
FILE *outfile;

#define NPTS 1000
#define PAGE_WIDTH 8
#define PAGE_HEIGHT 10

main() {
   outfile = fopen("psline.eps","w");
   fprintf(outfile,"%%! psline output\n");
   fprintf(outfile,"%%%%BoundingBox: 0 0 %f %f\n",
      72.0*PAGE_WIDTH,72.0*PAGE_HEIGHT);
   fprintf(outfile,"gsave\n");
   fprintf(outfile,"/l {lineto} def\n");
   fprintf(outfile,"%f %f scale\n",72.0*PAGE_WIDTH*0.5,
      72.0*PAGE_HEIGHT*0.5);
   fprintf(outfile,"90 rotate\n");
   fprintf(outfile,"1 -1 translate\n");
   fprintf(outfile,"0.5 setgray 0.02 setlinewidth\n");
   x = (1.0 - NPTS)/NPTS;
   y = sin(50.0*x)/(50.0*x);
   fprintf(outfile,"%.3f %.3f moveto\n",x,y);
   for (i = 1; i < NPTS; ++i) {
      x = (2.0*i + 1.0 - NPTS)/NPTS;
      y = sin(50.0*x)/(50.0*x);
      fprintf(outfile,"%.3f %.3f l\n",x,y);
   }
   fprintf(outfile,"stroke\n");
   fprintf(outfile,"grestore\n");
   fclose(outfile);
}
```

On most UNIX systems it can be compiled by `cc psline.c -o psline -lm`. The output that it writes to the file `psline.eps` looks like:

```
%! psline output
%%BoundingBox: 0 0 576.000000 720.000000
gsave
/l {lineto} def
288.000000 360.000000 scale
90 rotate
1 -1 translate
0.5 setgray 0.02 setlinewidth
-0.999 -0.006 moveto
-0.997 -0.008 l
-0.995 -0.010 l
...
0.997 -0.008 l
0.999 -0.006 l
stroke
grestore
```

and sending this to a PostScript interpreter produces the output shown in Figure A1.8. Note that it needs a `showpage` command added at the end to produce a page if it is sent to a printer.

The second program, `psimage`, shows the use of the `image` command to produce a grayscale image of $\sin(r)/r$.

```
/*
* psimage.c
* (c) Neil Gershenfeld  9/1/97
* demonstrates PostScript images by drawing sin(r)/r
```

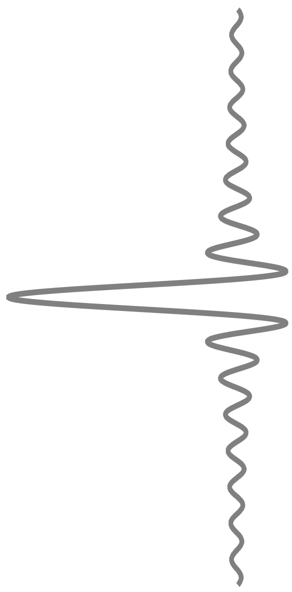


Figure A1.8. Output from `psline`.

```
*/

#include <math.h>
#include <stdio.h>

int i,j;
float x,y,r,z;
unsigned char grey;
FILE *outfile;

#define NPTS 100
#define PAGE_WIDTH 8.0

main() {
   outfile = fopen("psimage.eps","w");
   fprintf(outfile,"%%! psimage output\n");
   fprintf(outfile,"%%%%BoundingBox: 0 0 %f %f\n",
      72.0*PAGE_WIDTH,72.0*PAGE_WIDTH);
   fprintf(outfile,"gsave\n");
   fprintf(outfile,"%f %f scale\n",72.0*PAGE_WIDTH,
      72.0*PAGE_WIDTH);
   fprintf(outfile,"%d %d 8 [%d 0 0 %d 0 0] {<\n",
      NPTS,NPTS,NPTS,NPTS);
   for (i = 0; i < NPTS; ++i)
      for (j = 0; j < NPTS; ++j) {
        x = (2.0*i + 1.0 - NPTS)/NPTS;
        y = (2.0*j + 1.0 - NPTS)/NPTS;
```

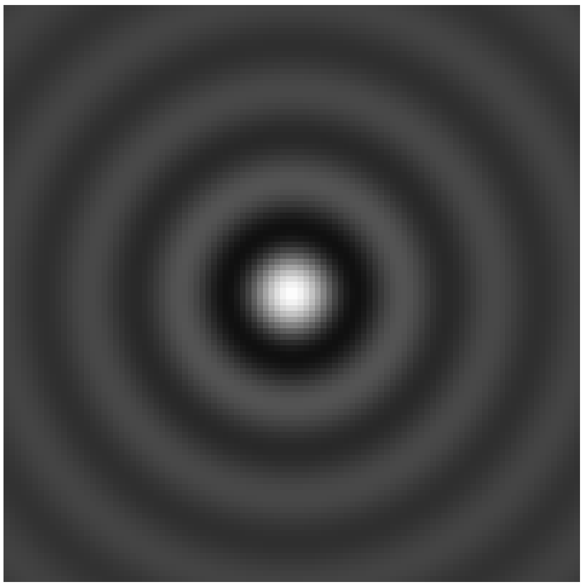


Figure A1.9. Output from psimage.

```
        r = 20.0*sqrt(x*x + y*y);
        z = sin(r)/r;
        grey = (unsigned char) (255.0 * (z + 0.3)/1.3);
        fprintf(outfile,"%.2x",grey);
   }
   fprintf(outfile,">} image\n");
   fprintf(outfile,"grestore\n");
   fclose(outfile);
}
```

Its output:

```
%! psimage output
%%BoundingBox: 0 0 576.000000 576.000000
gsave
576.000000 576.000000 scale
100 100 8 [100 0 0 100 0 0] {<
3c3e4041
...
44444443
>} image
grestore
```

is shown in Figure A1.9.

An amusing question that is often asked is how to write a filter to examine a PostScript program and determine the number of pages that it will produce. If the program follows the DSC rules then it will include a comment that gives the number of pages it contains and so this is easy, but if it doesn't then this problem is insoluble: determining the number of pages produced would be equivalent to a solution to the *halting problem*, which Turing showed is impossible (page 135). This is the cost of using a universal language in a printer. In return, it's possible to do significant computing in PostScript directly. At one time there were more RISC processors in PostScript printers than in workstations; it's even possible to do 3D rendering directly in a PostScript program!

PDF was developed because reading online documents does not require computational universality, but it does rely on easy random access to arbitrary pages. The PDF file format has much more structure; there's a header, a body with separate and complete descriptions for each page, a cross-reference table pointing to the page locations in the document, and a trailer. This is much simpler for a viewer to read, but more work for a programmer to write; there are a number of utilites available for converting more easily-generated PostScript to PDF (including *ps2pdf*, and Adobe's *Acrobat*).

A1.2.2 X Windows

The great contribution of *X Windows* (originally developed at MIT in the mid 1980s) is that it decouples a computer's display from the programs that use it, so that any processor on a network can produce output on any display (as long as it has permission). For this reason it has emerged as the low-level graphics standard for workstations. Its equally great liabilities are that it is pixel-based rather than device-independent (like PostScript), so that a program needs to know about the display that it is using and the output will appear different on different displays, and since it is aimed at low-level graphics communications simple high-level tasks (such as drawing a line) require either many initial function calls or the use of a software toolkit. Nevertheless, X has become a lowest common denominator for network graphics and is available on most platforms, and so here again a modest investment in learning some basic routines will allow you to directly write very portable graphics programs.

The X program that manages a computer's display is called the *server*, and the programs associated with individual windows (such as a terminal emulator or a numerical model that produces graphics) are called *clients*. If I am using an X server running on a machine that has an Internet address of `mach.univ.edu`, then I can a run any program anywhere else on Internet and see the output if I tell the remote machine to `export DISPLAY=mach.univ.edu:0.0` (or an equivalent command). The first 0 is the number of the X server to connect to, and the second 0 is the number of the screen. Most commonly this is just 0.0, but a computer might be running more than one X server if, for example, there are *Virtual Network Computing* sessions (*VNC*, `http://www.tightvnc.com`, `http://www.realvnc.com`), and a *multi-headed* computer can independently drive more than one screen. Depending on how permissions are set on the server it may also be necessary to give it the command `xhost +`, which permits clients from any other system to access it.

The first program, `xline`, is similar to `psline`, but shows how it is possible to animate curves by repeatedly drawing and then erasing them (now showing the variation

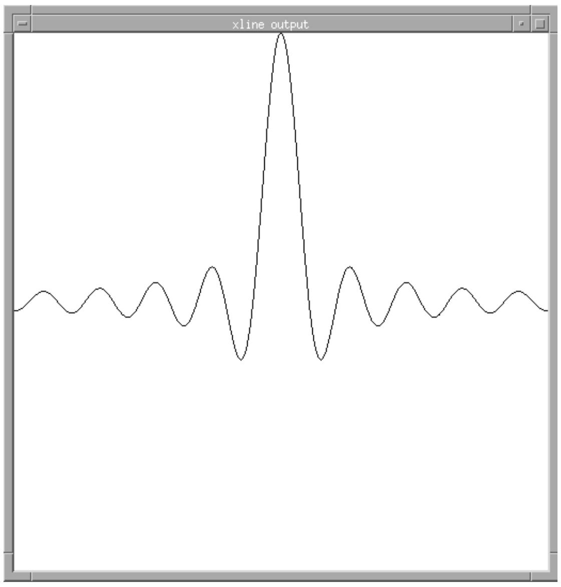


Figure A1.10. Snapshot of the output from `xline`.

of sin(kx)/(kx) with k). It is compiled by `gcc xline.c -o xline -lm -lX11`; Figure A1.10 shows the output. The default coordinate system for X has the origin at the upper-left corner, with y increasing down and x increasing to the right measured in pixel units. This is traditional in display graphics because of the raster pattern used in CRTs.

```
/*
* xline.c
* (c) Neil Gershenfeld  2/4/03
* demonstrates drawing X lines by animating sin(k*x)/k*x
*/

#include <X11/Xlib.h>
#include <math.h>
#include <unistd.h>

Display *D;
int     S,Loop,Point;
Window  W;
GC      Gc,GcRev;
XEvent  Event;
XPoint  PointBuf[1000],OldPointBuf[1000];
float   r;

#define WIDTH 500
#define HEIGHT 500
#define NPTS 500
```

```
main() {
   D = XOpenDisplay("");
   S = DefaultScreen(D);
   W = XCreateSimpleWindow(D, DefaultRootWindow(D),
       0, 0, WIDTH,HEIGHT, 1, WhitePixel(D,S), WhitePixel(D,S));
   XStoreName(D, W, "xline output");
   XMapRaised(D, W);
   Gc = XCreateGC (D, W, 0L, (XGCValues *) 0);
   XSetForeground(D, Gc, BlackPixel(D, S));
   XSetBackground(D, Gc, WhitePixel(D, S));
   XSetLineAttributes(D, Gc, 0, LineSolid, CapButt, JoinMiter);
   GcRev = XCreateGC (D, W, 0L, (XGCValues *) 0);
   XSetForeground(D, GcRev, WhitePixel(D, S));
   XSetBackground(D, GcRev, BlackPixel(D, S));
   XSetLineAttributes(D, GcRev, 0, LineSolid, CapButt, JoinMiter);
   for (Loop = 1; Loop <= NPTS; ++Loop) {
      for (Point = 0; Point < NPTS; ++Point) {
         r = 0.1 * (2.0*Point + 1.0 - NPTS)/NPTS;
         OldPointBuf[Point].x = PointBuf[Point].x;
         OldPointBuf[Point].y = PointBuf[Point].y;
         PointBuf[Point].x = Point;
         PointBuf[Point].y = (int) (NPTS * (1.0 - sin(Loop*r) /
            (Loop*r)) / 2.0);
         }
      XDrawLines(D, W, GcRev, OldPointBuf, NPTS, CoordModeOrigin);
      XDrawLines(D, W, Gc, PointBuf, NPTS, CoordModeOrigin);
      XFlush(D);
      usleep(100);
   }
}
```

The connection between an X client and server is *asynchronous*. For efficiency, drawing requests are buffered and sent in groups. The `XFlush` command instructs the client to send all pending requests to the server. A fast client can overwhelm a slow server; the `XSync` command instead makes sure that the server and client states are identical. It requires more communication to check this and so can slow down a program, but may be necessary for slow networks or servers.

The second program, `ximage`, shows the use of X images for 2D grayscale animation (Figure A1.11). It associates an image in the server with a data array in the client, updating their connection after each iteration. Historically, because the *Digital to Analog Converters* (*DAC*s) used in displays had limits on their speed and resolution, display drivers had to use a *colormap* to specify the color to be associated with each output bit pattern. This is increasingly not necessary; `ximage` assumes that the display can use a 24 bit *TrueColor* visual that directly assigns 8 bits each of a pixel value to its red, green,

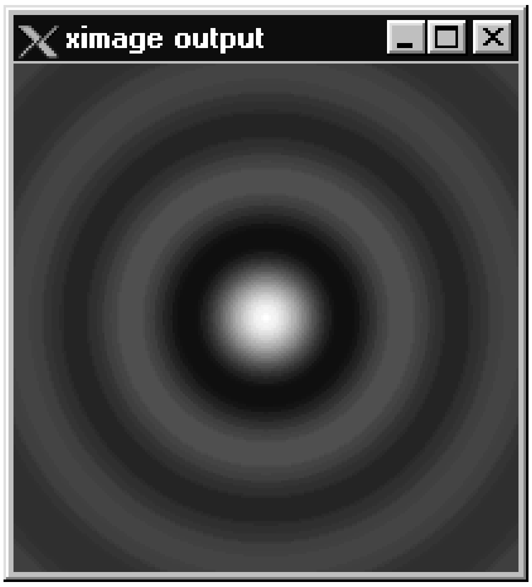


Figure A1.11. Snapshot of the output from `ximage`.

and blue components; (a display's supported modes can be shown by the `xdpyinfo` command).

```
/*
* ximage.c
* (c) Neil Gershenfeld  2/4/03
* demonstrates drawing X images by animating sin(k*x)/k*x
*/

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <math.h>

#define WIDTH 200
#define HEIGHT 200
#define NPTS 200
#define NCOLORS 256
#define DEPTH 24

Display  *D;
Visual   *V;
int      S,Loop,Point,i,j;
Window   W;
GC       Gc;
XImage   *I;
```

```
XVisualInfo Info;
char     Data[HEIGHT][WIDTH][4];
float    x,y,r,z;

main() {
   D = XOpenDisplay("");
   S = DefaultScreen(D);
   if (XMatchVisualInfo(D, S, DEPTH, TrueColor, &Info) == 0) {
      printf("Display does not support %d bit TrueColor Visual\n",DEPTH);
      return;
   }
   V = Info.visual;
   W = XCreateSimpleWindow(D, DefaultRootWindow(D), 0, 0, WIDTH, HEIGHT, 0, 0, 0);
   XStoreName(D, W, "ximage output");
   XMapRaised(D, W);
   Gc = XCreateGC (D, W, 0L, (XGCValues *) 0);
   I = XCreateImage(D, V, DEPTH, ZPixmap, 0, (char *) Data, WIDTH, HEIGHT, 8, 0);
   for (Loop = 1; Loop <= NPTS; ++Loop) {
      for (i = 0; i < WIDTH; ++i)
         for (j = 0; j < HEIGHT; ++j) {
            x = (2.0*i + 1.0 - WIDTH)/WIDTH;
            y = (2.0*j + 1.0 - HEIGHT)/HEIGHT;
            r = Loop*20.0*sqrt(x*x + y*y)/NPTS;
            z = sin(r)/r;
            Data[j][i][0] = (unsigned char) (NCOLORS * (z + 0.3)/1.3);
            Data[j][i][1] = (unsigned char) (NCOLORS * (z + 0.3)/1.3);
            Data[j][i][2] = (unsigned char) (NCOLORS * (z + 0.3)/1.3);
         }
      XPutImage(D, W, Gc, I, 0, 0, 0, 0, WIDTH, HEIGHT);
   }
}
```

X Windows is documented and available at `http://www.x.org` and `http://www.xfree86.org`. In addition to the often-cryptic documentation, O'Reilly and Associates publish a good comprehensive series of X reference manuals [Nye, 1992].

A1.2.3 OpenGL

X Windows is a windowing system, and it is oriented towards 2D graphics. *OpenGL* is the converse: it is not a windowing system, and it is designed for 3D graphics. It has descended from Silicon Graphics' proprietary IRIS GL standard which ran on their graphics computers. IRIS GL was difficult to port to other platforms; OpenGL introduced a number of incompatibilities with IRIS GL in order to make it much more portable and usable. It runs within whatever windowing system is native to the operating system; versions are available for many platforms. OpenGL is now a mature and widely used 3D graphics programming standard, documented in [Woo *et al.*, 1997; Kempf &

Frazier, 1997]. Implementation information is available at `http://www.opengl.org`, and *Mesa* is an open-source version (`http://www.mesa3d.org`).

Writing an ordinary OpenGL program requires mastering not only its calls but also the native windowing system on each platform you want to use to manage putting up a window and getting user input. This chore is significantly simplified by the *GLUT* toolkit, which hides these details and hence makes it straightforward to write portable GL programs. GLUT is available on all of the platforms that have GL; it's documented at `http://www.opengl.org/developers/documentation/glut.html` .

Here's a first example of a GLUT program, which can be compiled on a machine with the libraries by `gcc glsurf.c -o glsurf -lm -lglut`, with output shown in Figure A1.12:

```
/*
* glexample.c
* (c) Neil Gershenfeld  8/30/97
* draw a sphere and a cube with GLUT
*/

#include <GL/glut.h>

void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glTranslatef(-.3,0,0);
  glutSolidSphere(0.5,50,50);
  glTranslatef(.5,0,0);
  glutSolidCube(1.0);
  glPopMatrix();
  glFlush();
  }

void mouse(int button, int state, int x, int y) {
   exit(0);
}

void main(int argc, char **argv) {
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutCreateWindow("GLUT example");
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
```

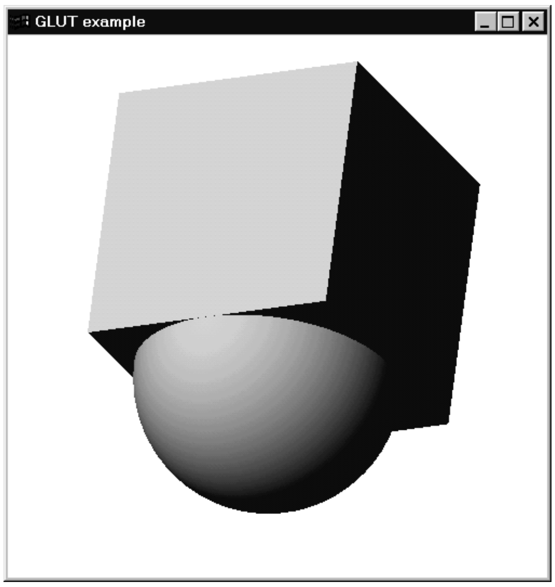


Figure A1.12. Output from glexample.

```
   glClearColor(1.0,1.0,1.0,1.0);
   glMatrixMode(GL_PROJECTION);
   glRotatef(-140.0,1.0,1.0,0.0);
   glutMainLoop();
}
```

GL is based on a state machine model: successive function calls update the state of a virtual machine that renders the scene. The main routine puts up a window, registers a routine named display to be called whenever the display needs updating (for example, when the window is raised) and a routine mouse to be called when there is mouse input, turns on lighting, sets the viewpoint, and then starts running a loop to process events. The display routine clears the window (to the white background set in main), draws a sphere and a cube displaced relative to the current coordinate system, and flushes any pending display requests. The mouse routine simply exits when a mouse is clicked in the window.

Now here's a more complex example:

```
/*
* glsurf.c
* (c) Neil Gershenfeld  2/4/03
* example of GL and GLUT, drawing sin(kr)/kr surface
*/

#include <GL/glut.h>
#include <math.h>
```

```
#include <unistd.h>

void normal(GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,
            GLfloat,GLfloat,GLfloat,GLfloat*,GLfloat*,GLfloat*);

#define NGRID 50
#define KMIN 0.0
#define KMAX 20.0
float k=0.0, dk=0.2;

#define r(x,y) (k*sqrt(x*x+y*y))
#define height(x,y) (sin(r(x,y))/r(x,y))

GLfloat x[NGRID][NGRID],y[NGRID][NGRID],z[NGRID][NGRID];

void display(void) {
   int i,j;
   GLfloat nx,ny,nz;
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBegin(GL_QUADS);
   for (i = 0; i < (NGRID-1); ++i) {
      for (j = 0; j < (NGRID-1); ++j) {
         normal(x[i][j],y[i][j],z[i][j],
                x[i+1][j],y[i+1][j],z[i+1][j],
                x[i][j+1],y[i][j+1],z[i][j+1],
                &nx,&ny,&nz);
         glNormal3f(nx,ny,nz);
         glVertex3f(x[i][j],y[i][j],z[i][j]);
         glVertex3f(x[i+1][j],y[i+1][j],z[i+1][j]);
         glVertex3f(x[i+1][j+1],y[i+1][j+1],z[i+1][j+1]);
         glVertex3f(x[i][j+1],y[i][j+1],z[i][j+1]);
         }
      }
  glEnd();
  glFlush();
  }

void idle(void) {
   GLfloat rx,ry,rz;
   int i,j;
   if ((k > KMAX) | (k < KMIN))
      dk = -dk;
   k += dk;
   for (i = 0; i < NGRID; ++i)
      for (j = 0; j < NGRID; ++j) {
         x[i][j] = 2.0*((float) j + 0.5)/NGRID - 1.0;
         y[i][j] = 2.0*((float) i + 0.5)/NGRID - 1.0;
         z[i][j] = height(x[i][j],y[i][j]);
 }
   glMatrixMode(GL_MODELVIEW);
   rx = rand();
   ry = rand();
   rz = rand();
   glRotatef(1.0,rz,ry,rz);
   glutSwapBuffers();
   glutPostRedisplay();
   usleep(100);
   }

void mouse(int button, int state, int x, int y) {
   exit(0);
   }

void normal(GLfloat x1, GLfloat y1, GLfloat z1,
            GLfloat x2, GLfloat y2, GLfloat z2,
            GLfloat x3, GLfloat y3, GLfloat z3,
            GLfloat *xn, GLfloat *yn, GLfloat *zn) {
   *xn = (y2-y1)*(z3-z1) - (z2-z1)*(y3-y1);
   *yn = (z2-z1)*(x3-x1) - (x2-x1)*(z3-z1);
   *zn = (x2-x1)*(y3-y1) - (y2-y1)*(x3-x1);
   }

void main(int argc, char **argv) {
   GLfloat matl_ambient[] = {.25, .22, .06, 1.0};
   GLfloat matl_diffuse[] = {.35, .31, .09, 1.0};
   GLfloat matl_specular[] = {.80, .72, .21, 1.0};
   GLfloat light_ambient[] = {0.9, 0.9, 0.9, 1.0};
   GLfloat light_diffuse[] = {0.8, 0.8, 0.8, 1.0};
   GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat light_position[] = {0,0,1, 1.0};

   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("GLUT sin(kr)/kr example");
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
   glutIdleFunc(idle);
   glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,matl_ambient);
   glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,matl_diffuse);
   glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,matl_specular);
   glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS, 95.0);
   glEnable(GL_LIGHTING);
   glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
```

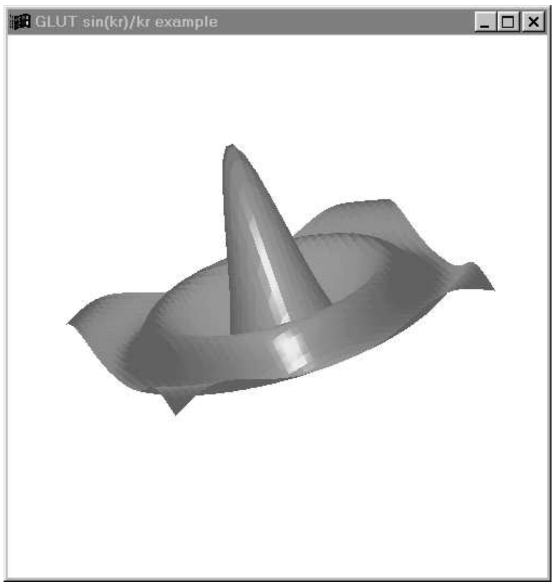


Figure A1.13. Snapshot of the output from `glsurf`.

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_specular);
glLightfv(GL_LIGHT0,GL_POSITION,light_position);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
glEnable(GL_NORMALIZE);
glClearColor(1.0,1.0,1.0,1.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.5,1.5,-1.5,1.5,-1.5,1.5);
glutMainLoop();
}
```

This program draws $\sin(kr)/kr$ as a 3D surface, animates it as k is changed, and slowly rotates the orientation from which it is viewed. The `main` routine now does a bit more work, defining the properties of the materials and lights in the scene (`glsphere.c` used the defaults for these). It also registers a new routine, `idle`, that is called whenever there are CPU cycles free. This is what does the work of calculating the animation. It updates the surface, swaps the buffer being displayed with the one being calculated, and asks for a redisplay. The `display routine` draws the surface with quadrilaterals, and calls another routine to calculate the normals of the quadrilaterals (which GL needs to determine the shading). Figure A1.13 shows a frame of the output.

GL is a low-level procedural language for describing 3D graphics. *Open Inventor* is built on top of GL; it is a high-level language for describing scenes. There is a file format associated with Open Inventor, and the *VRML* 3D extension (pronounced "vermul") to the World Wide Web is based on this. A good reference for Inventor is [Wernecke, 1994].

A1.2.4 Java

Even though code written with GLUT is portable, it still needs to be recompiled for each platform on which it is run. *Java* takes portability a big step further by using machine-independent byte codes so that there's no need for recompilation. It started as a project at Sun to develop a programming language for appliances, and grew from there to encompass the whole Internet. Development environments are available from Sun (`http://java.sun.com`), commercial vendors, and open-source projects (`http://www.blackdown.org`).

The language is completely *object oriented*, so that instead of writing conventional procedural programs a Java program consists of the definition of a set of *classes* that contain data structures and *methods* associated with them (a method is the object-oriented programming name for a function). This style of programming makes it much easier to create new programs by specifying differences from old ones, and to handle programming abstractions. For example, many routines with the same name could be defined that differ in the expected data type. The object orientation of Java is similar to that of C++, but it dispenses with many aspects of C++ that reflect the C legacy or that have proved themselves to be rich sources of bugs. Unfortunately the single aspect of C++ most useful for mathematical programming was dropped because it was deemed to fit into the latter category. In C++ dyadic operators can be overloaded, so that for example $A + B$ can work for matrices, complex numbers, members of a group, and so forth. In Java this is not possible; overloading is restricted to functions.

The Jave byte codes are interpreted by a Java *virtual machine* for a particular platform. This introduces some performance overhead, although that can be minimized through the use of *Just-In-Time* (*JIT*) compilation that generates native instructions on the fly. Java programs can be run as stand-alone applications, and more importantly as *applets* that are downloaded from a Web server and then executed in a local Web browser (in theory at least: the browser must have access to the system libraries used to develop the applet, which can be a challenge given the pace of Java's evolution). X Windows sends a description of a window over the Net; Java sends the whole program. This does mean that the processor time available to an applet will depend on the scheduling priorities of the browser as well as the operating system.

The first example program shows how the Java 2D graphics routines can be used to animate the $sin(kx)/kx$ curve as an applet. It makes use of *Java Foundation Classes* (*JFC*), which includes *AWT*, the *Abstract Windowing Toolkit* that contains the 2D classes (as well as earlier graphical routines that don't coexist well with them), and the *SWING* user interface components. Running `javac JavaLine.java` compiles this and produces a binary byte code file `JavaLine.class`. An *HTML* (HyperText Markup Language) file containing the following instructions:

```
<html>
```

```
<body>
<applet code="JavaLine.class" width=500 height=500>
</applet>
</body>
</html>
```

will load this class and execute it (in a Web browser, or in a stand-alone interpreter such as `appletviewer`). Since Java programs almost always are run with many other processes, time-sharing is very important. This example uses *threads* to periodically execute the program to update the display and then go back to sleep. The Java interpreter calls the `start` method to set the applet up, the `run` method to execute it, the `stop` method to halt it, and the `paintComponent` method to redraw the window.

```
//
// JavaLine.java
// (c) Neil Gershenfeld  2/6/03
// demonstrates Java by animating sin(k*x)/k*x
//

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class JavaLine extends JApplet implements Runnable {
   Thread T;
   final int NPTS = 500;
   final int NSTEPS = 100;
   boolean stopped = false;
   int point,step;
   int x[] = new int[NPTS];
   int y[] = new int[NPTS];

   class LinePanel extends JPanel {
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         super.setBackground(Color.white);
         Graphics2D g2d = (Graphics2D) g;
         GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD,NPTS);
         path.moveTo(x[0],y[0]);
         for (point = 1; point < (NPTS-1); ++point) {
            path.lineTo(x[point],y[point]);
         }
         g2d.draw(path);
      }
   }
   public void init() {
      Container C = getContentPane();
      C.add(new LinePanel());
   }

   public void start() {
      if (T == null) {
         T = new Thread(this);
         T.start();
      }
   }

   public void stop() {
      stopped = true;
   }

   public void run() {
      double r;
      while (stopped == false) {
         for (step = 1; step < NSTEPS; ++step) {
            for (point = 0; point < (NPTS-1); ++point) {
               r = 100 * (step*(point+0.5-NPTS/2))/(NPTS*NSTEPS);
               x[point] = point;
               y[point] = (int) ((NPTS/2) - (NPTS/2)*Math.sin(r)/r);
            }
            repaint();
            try {Thread.sleep(10);}
            catch (InterruptedException e) { }
         }
      }
   }
}
```

The next example, `JavaImage`, shows the use of images to animate the $\sin(kr)/kr$ example. This time it is defined as a stand-alone Java program by including a `main` method, so that in addition to being runnable as an applet it can also be executed as an application with the `java` command). There are severe security restrictions on what applets can and cannot do (such as read or write files); these do not apply to applications.

```
//
// JavaImage.java
// (c) Neil Gershenfeld  2/6/03
// demonstrates Java by animating sin(k*r)/k*r
//
```

```java
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;

import java.awt.geom.*;

public class JavaImage extends JApplet implements Runnable {

   static final int NPTS = 200;
   static final int NSTEPS = 100;
   static final int XBORDER = 5;
   static final int YBORDER = 20;
   boolean stopped = false;
   Thread T;
   WritableRaster R;
   BufferedImage I;

   public static void main(String args[]) {
      JavaImage J = new JavaImage();
      JFrame F = new JFrame("JavaImage");
      F.setSize(NPTS+XBORDER,NPTS+YBORDER);
      J.init();
      F.getContentPane().add(J);
      F.setVisible(true);
      J.start();
   }

   class ImagePanel extends JPanel {
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         Graphics2D g2d = (Graphics2D) g;
         g2d.drawImage(I, 0, 0, this);
      }
   }

   public void init() {
      Container C = getContentPane();
      C.add(new ImagePanel());
      I = new BufferedImage(NPTS,NPTS,BufferedImage.TYPE_INT_RGB);
      R = I.getRaster();
   }

   public void start() {
      if (T == null) {
         T = new Thread(this);
         T.start();
      }
   }

   public void stop() {
      stopped = true;
   }

   public void run() {
      double xr, yr, r;
      int gray;
      int step;
      while (stopped == false) {
         for (step = 1; step < NSTEPS; ++step) {
            for (int i = 0; i < NPTS; i++) {
               for (int j = 0; j < NPTS; j++) {
                  xr = (2.0*i + 1.0 - NPTS)/NPTS;
                  yr = (2.0*j + 1.0 - NPTS)/NPTS;
                  r = step* 20.0 * Math.sqrt(xr*xr + yr*yr)/NSTEPS;
                  gray = (int) (255*(0.5+Math.sin(r)/r)/1.5);
                  int[] color = {gray, gray, gray};
                  R.setPixel(i,j,color);
               }
            }
            repaint();
            try {Thread.sleep(10);}
            catch (InterruptedException e) { };
         }
      }
   }
}
```

Java has 3D classes that are similar to Open Inventor in how it describes scenes, but there are also a number of Java bindings for OpenGL routines, running either as pure Java (such as jGL) or calling GL system libraries (like GL4Java).

A1.2.5 HTML

HTML5
   Canvas
   SVG
   WebGL

A1.2.6 *ification

This Appendix has been discussing *visualization*, producing and interacting with images generated from models and data. Beyond using printers and screens,
   3D holovideo worn Jaron whole room CAVE

this can be done in *immersive* with head-mounted displays or rooms with multiple video projectors and input tracking that

(roughly proportional to their cost) in conveying

within the

Humans use all of their senses to understand the physical world, and the same skills can carry over into mathematical worlds. *Sonification* is the serious study of the aural perception of data; for example, playing

```
%
% sinsound.m
% (c) Neil Gershenfeld  2/10/03
% play a modulated sine wave
%
NPTS = 100000;
DT1 = 0.1;
DT2 = 0.001;
FREQ = 48000;
NPLOT = pi/DT2;
t = 0.5 + (-NPTS/2):(NPTS/2);
t1 = t*DT1;
t2 = t*DT2;
x = sin(t1).*sin(t2)./t2;
sound(x,FREQ);
plot(x);
```

conveys a very different

haptic interfaces). audio

STL

stereolithography

DXF IGES

Wendy interact 3D

community studies

Human-Computer Interaction (*HCI*, `http://www.acm.org/sigchi`).

olfaction

GNU GSL library

Blender Python surface

OpenMP

A1.3 PROBLEMS

(A1.1) Write a program to directly produce PostScript output for the trajectory of a ball bouncing on a floor, assuming that at each reflection the ball's kinetic energy decreases by a constant fraction.

(A1.2) Write an X program to animate the motion of a bouncing ball.

(A1.3) Write a OpenGL program to animate the motion of a bouncing ball.

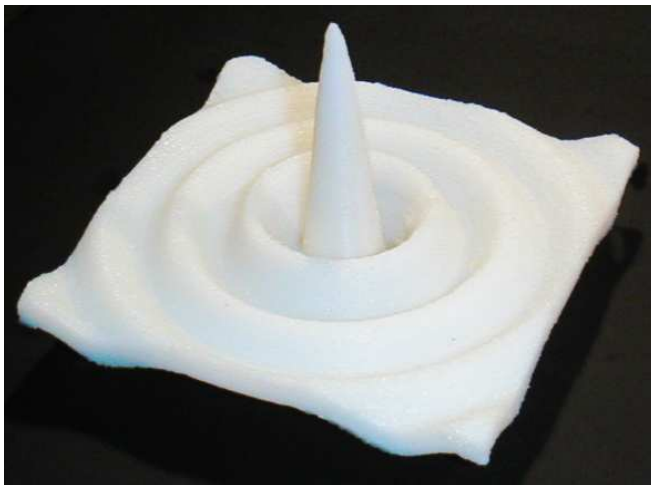


Figure A1.14. 3D-printed $\sin(x)/x$ surface.

(A1.4) Write a Java program to animate the motion of a bouncing ball.
(A1.5) Repeat these problems in as many different interactive environments as you can.

Appendix 2 Network Programming

It is much easier to find problems that exceed available computational resources than the converse. The obvious solution, using a faster computer, is far from universally applicable, if for no other reason than the approaching physical limits on scalar processors [Gershenfeld, 2000]. But the time required for a computation can be reduced by increasing the space, using more than one processor in parallel. Although this is done in dedicated parallel computers, a more accessible and scalable alternative is to network commodity computers [Ridge *et al.*, 1997]. For problems that don't require too much interprocessor communication, a building (or even an entire planet [Korpela *et al.*, 2001]) full of frequently-idle workstations can become a very effective supercomputer. This appendix introduces the techniques needed to distribute programs over computers on networks, starting with an introduction to network programming and ending with a peek at higher-level parallel computing languages.

A2.1 OSI, TCP/IP, AND ALL THAT

The *ISO* (International Standards Organization) develops international standards in many areas; the US member organization is *ANSI*, the American National Standards Institute. ISO adopted *OSI*, the Open Systems Interconnection model, in 1984 as a general framework for communications that recognizes seven layers (Table A2.1). This was originally intended to result in explicit standards for protocols at each layer. That grand vision ended up being settled more in the marketplace than in the standards committees, but this still remains a useful guide for understanding networks.

Layers 1 and 2 refer to the physical transmission medium, and the logical encoding of signals in the physical medium. Layer 3 is responsible for routing a message from its origin to its destination, possibly across many physical networks (*internetworking*). Layer 4 takes care of managing an end-to-end link to make sure that both ends agree to communicate and are satisfied with the result. The higher layers provide the interface to users and their programs.

Perhaps the most important network protocol is *IP* (Internet Protocol), which was developed as part of *ARPANET* and has since become the foundation of the *Internet*. An IP *packet* or *datagram* consists of data, an address, and some extra administrative information. The maximum size of a packet is hardware-dependent, but is guaranteed to be at least 576 bytes. A machine that generates an IP packet, which can range from a microcontroller to a supercomputer, need not know how to reach the final address –

Table A2.1. *OSI communications layers.*

layer	name	example
7	application	Telnet
		FTP
6	presentation	
5	session	
4	transport	TCP
		UDP
3	network	IP
		PPP
2	data link	ethernet
		V.32/42
1	physical	coaxial cable
		twisted pair

it just has to know how to put the packet onto the local network. From there, it gets passed between machines and across routers and gateways until it reaches its destination. Internet addresses are of the form 192.0.34.161, and can have a name registered to go along with them (in this case, `www.internic.net`, a directory of Internet machines). In addition to an Internet address, a packet also needs to have specified a *port*. This is an integer address that the sending and receiving program agree in advance to use. Many port numbers have special meanings and must be avoided; in particular, most of the ports below $\sim$ 1000 are reserved for system functions (such as `telnet` and `ftp`; on a UNIX system the known ports are listed in `/etc/services`).

TCP (Transmission Control Protocol), and *UDP* (User Datagram Protocol), are two options for managing communications with IP packets. Both wrap extra information around the packets. TCP takes a large message and splits it into pieces that are small enough to fit in packets, and it makes sure that all of the packets are correctly received and reassembled by the recipient. UDP is *connectionless*, so any receiver can accept a packet from any sender without an advance agreement to communicate, and it is *unreliable*, so no guarantee is made that a packet is received. This apparent liability is not as bad as it sounds because it means that there is much less communications overhead and so the bandwidth and latency are better. If a lower network layer is taking care of error control and the network is not overloaded, then UDP can actually be quite reliable. Almost all communications of any kind on the Internet are transported in TCP/IP or UDP/IP packets. The next section shows how to use UDP packets. For more information, see [Tanenbaum, 1988] for a very readable introduction to computer networks.

A2.2 SOCKET I/O

The most common programming approach for sending and receiving IP packets is through *sockets*, developed at Berkeley in the early 1980s. Setting up a socket requires defining the protocol (such as TCP or UDP), address, and port. Once that is done, the program simply writes to, or reads from, the socket to exchange data with the remote machine. Socket programming started in UNIX (see [Stevens, 1990] for more information about UNIX network programming), and has since expanded to most platforms.

Here is a simple program that opens a socket and writes a string from its command line to a remote machine:

```
/*
* sendudp.c
* (c) Neil Gershenfeld 9/1/97
*
* synatx: sendudp hostname string
*
* send this string to hostname in a udp packet on port 6543
*
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

main(argc, argv)
   int argc;
   char *argv[]; {

   struct hostent *IPAddress;
   struct sockaddr_in Remote, Local;
   int Port = 6543, UDPtr, IPSize;
   char Address[100];

   IPAddress = gethostbyname(argv[1]);
   if (IPAddress == NULL) {
      printf("%s isn't in the name server\n",argv[1]);
      return; }
   strcpy(Address, inet_ntoa(*((struct in_addr *)
      *(IPAddress->h_addr_list))));
   bzero((char *) &Local, sizeof(Local));
   bzero((char *) &Remote, sizeof(Remote));
   Remote.sin_family = AF_INET;
   Remote.sin_addr.s_addr = inet_addr(Address);
   Remote.sin_port = htons(Port);
   Local.sin_family = AF_INET;
   Local.sin_addr.s_addr = htonl(INADDR_ANY);
   Local.sin_port = htons(Port);
   UDPtr = socket(AF_INET,SOCK_DGRAM,0);
   if (UDPtr < 0) {
      puts("Can't create the local socket");
      return; }
   if (bind(UDPtr, (struct sockaddr *)
      &Local, sizeof(Local)) < 0) {
      puts("Can't bind local address");
      return; }
   IPSize = sendto(UDPtr, argv[2], strlen(argv[2]), 0,
      (struct sockaddr *) &Remote, sizeof(Remote));
   if (IPSize < 0) {
      puts("sendto error");
      return; }
   close(UDPtr);
   printf("Sent %d bytes to %s\n",IPSize, Address);
   }
```

and this program listens for the packet:

```
/*
* recvudp.c
* (c) Neil Gershenfeld 9/11/94
*
* waits for an incoming udp packet on port 6543
* and print it out
*
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#define BufferSize 100

main(argc, argv)
   int argc;
   char *argv[]; {

   struct hostent *IPAddress;
   struct sockaddr_in Remote, Local;
   int Port = 6543, UDPtr, IPSize, RemoteSize;
   char Buffer[BufferSize];
   UDPtr = socket(AF_INET,SOCK_DGRAM,0);
   if (UDPtr < 0) {
      puts("Can't create the local socket");
      return; }
   bzero((char *) &Local, sizeof(Local));
   Local.sin_family = AF_INET;
   Local.sin_addr.s_addr = htonl(INADDR_ANY);
   Local.sin_port = htons(Port);
   if (bind(UDPtr, (struct sockaddr *)
      &Local, sizeof(Local)) < 0) {
      puts("UDPRecv: Can't bind local address");
      return; }
   IPSize = recvfrom(UDPtr, Buffer, BufferSize, 0,
      (struct sockaddr *) &Remote, &RemoteSize);
   if (IPSize < 0) {
      puts("UDPRecv: recvfrom error");
      return; }
   close(UDPtr);
   printf("Received %d bytes: %s\n",IPSize, Buffer);
   }
```

Both programs compile with `gcc prog.c -o prog`. A packet that is sent:

```
rho.media.mit.edu> sendudp media.mit.edu "test message"
Sent 12 bytes to 18.85.13.107
```

will be displayed at the other end of the socket:

```
media-lab.media.mit.edu> recvudp
Received 12 bytes: test message
```

(where `recvudp` most of course be started before `sendudp`).

A2.3 PARALLEL PROGRAMMING

Using sockets it is possible to manually split a program into multiple pieces that can execute on available machines and communicate their results to each other. This has been done with great success for particular important problems, such as applying many machines around the Internet to the problem of finding prime factors of a cryptographic key (which is a very parallelizable algorithm). This kind of programming is something like writing assembly language: you must decide exactly when and how messages get passed. An important level of abstraction above that is *MPI*, the Message Passing Interface (`http://www.netlib.org/mpi`). This standard is widely supported on many platforms: the same MPI program can run whether the processors are housed in a supercomputer or in workstations in a building. It still requires explicitly identifying when messages need to be passed in a parallel algorithm, but it hides the details of socket programming and includes routines to handle more advanced functions such as multiway communications and implementing effective connectivity topologies.

In Appendix 3 we will look at the series representation

$$\pi \approx \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \tag{A2.1}$$

as a simple benchmark task. The following program shows how to use MPI to parallelize it:

```
/*
* mpipi.c
* (c) Neil Gershenfeld  9/1/97
* use MPI to evaluate pi by summation
*/

#include <stdio.h>
#include <mpi.h>

void main(int argc, char** argv) {
   int rank,nproc,tag,i,istart,iend,N;
   double sum,pi;

   tag = 0;
   sum = 0.0;
   N = 1000000;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
   if (rank == 0) {
      MPI_Reduce(&sum,&pi,1,
         MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
      printf("Using %d processes, pi = %f\n",nproc,pi);
   } else {
      istart = 1 + (rank-1)*N/(nproc-1);
      iend = rank*N/(nproc-1);
      for (i = istart; i <= iend; ++i)
         sum += 0.5/((i-0.75)*(i-0.25));
      MPI_Reduce(&sum,&pi,1,
         MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
      }
   MPI_Finalize();
   }
```

The same program is run on all the processors being used. After initializing MPI the first two calls determine how many processes are being run, and the process number of each program. Then all of the copies with a process number not equal to 0 add up part of the sum based on their process number, and make an MPI call that asks to send the results to process 0 and add them up. Process 0 takes a different branch, waiting to receive the sum and then printing it out. Asking this to be run with six processors gives the output

```
% mpirun -np 6 mpitest.exe

Using 6 processes, pi = 3.141592
Process [0] exited with status [0]
Process [1] exited with status [0]
Process [2] exited with status [0]
Process [3] exited with status [0]
Process [4] exited with status [0]
Process [5] exited with status [0]
MPIRUN exited with status [0]
```

MPI has taken care of all of the message passing, hiding it behind a few simple calls. Because this program is so easy to parallelize, the time to run it will decrease roughly linearly with the number of processors used.

The most convenient parallel programming of all would be to have a compiler rather than the programmer do the work of dividing a program among multiple processors. The style of parallel programming used in MPI is *coarse-grained*: a relatively small number of powerful processors execute complex programs. Although there have been attempts to automate this kind of programming they have not been very successful, which is not surprising given that it is a difficult task even for a skilled programmer. A *fine-grained* parallel computer is one that has a large number of processors each doing a simple job [Hillis, 1985; Hillis, 1987]. Loops that do the same operation on many elements, such as adding corresponding components of two vectors, are easy to recognize and automatically parallelize on such a machine. This is particularly true in a language like Fortran 90 that can express such operations without loops. However, the popularity of such machines has been limited in part by the remarkable improvement in scalar processing speeds, and in part because many complex problems don't naturally map onto such an architecture. These are active research questions; in the meantime a language like MPI provides a convenient way to make good use of available resources to speed up a program.

Appendix 3 Benchmarking

Benchmarking is a subject that receives both too little and too much attention. Too little, because knowing the relative speeds of machines, languages, and algorithms can have an enormous impact on your ability to obtain timely results. Too much, because tests that may have little bearing on practical problems can dominate manufacturers' advertising and ultimately users' purchase decisions.

Amid all of the hype, a simple recurring truth is that the best benchmark is a problem that you are interested in. An early standard was the *Linpack* set of subroutines, which have been run on an enormous range of machines (see the listing at `http://www.top500.org`). Because there is a great deal of specialized structure in these routines, some aggressive compilers started to have switches that recognized them and used carefully hand-tuned assembly code to appear faster on this benchmark. To prevent that, as well as to cover a much broader range of applications, an industry-wide group has defined a suite of test problems called the *SPEC* benchmark (`http://www.specbench.org`). This is a comprehensive set of programs covering many types of numerical algorithms. Where it's available, it's a reliable guide to machine speed. However, the suite may not be available for a particular machine that you're interested in, and it is not freely accessible. For this reason it's useful to have a simple test program that can provide a rough order-of-magnitude estimate of speed.

I've found it convenient to use a series expansion of π,

$$\begin{aligned}\pi &= 4\tan^{-1}(1) \\ &\approx \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \quad .\end{aligned}$$

Summing this series requires five floating-point operations per step (ignoring the overhead for iterating the loop), providing an estimate of the computational speed by measuring the time taken to sum it. This is usually reported in millions of floating-point operations per second, called *megaflops* or *Mflops*. The total time should be linear in the number of terms used once N is large enough (there is always some overhead associated with starting and finishing execution). And since the correct answer is known, it is easy to check the validity and precision of the result.

Furthermore, it's instructive to write the series two ways, one as a reduction over a scalar term

$$\pi(N) = \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \quad , \tag{A3.1}$$

Table A3.1. *Selected execution speeds to sum a series expansion of π.*

computer	*speed (Mflops)*	*program version*
Connection Machine CM-2 (32K)	851	scalar
DEC XP1000	207	scalar
IBM ES/9000	148	scalar
	16.2	array
1 GHz Pentium III	134	scalar
	143	array
Cray Y-MP4/464	118	scalar
	10.0	array
DEC AlphaStation 500/500	70.0	scalar
	69.4	array
HP 735	18.2	scalar
	14.1	array
SUN Sparc 10/40	9.86	scalar
	7.97	array
200 MHz Pentium Pro	13.3	scalar
90 MHz Pentium	6.33	scalar
Sun SPARCStation 2	4.50	scalar
	0.34	array
Sun SPARCStation 1	1.21	scalar
	1.01	array
DEC VAX 8650	1.72	scalar
25 MHz 486	0.70	scalar
33 MHz 386/387	0.35	scalar
25MHz 486/87	0.23	scalar
Sun 3/60 (68020)	0.036	scalar
	0.036	array
12.5 MHz 286	0.034	scalar
10 MHz 8088	0.0011	scalar

and the other recursively in terms of an array

$$\pi(N) = \pi(N-1) + \frac{0.5}{(N-0.75)(N-0.25)} \quad . \tag{A3.2}$$

While these are formally equivalent, the extra storage required by the latter can inhibit schemes for caching and parallelization and hence probe the relative strengths of computer architectures and compilers.

Table A3.1 shows some sample speeds (for compiled C), using the machine's native floating-point precision. It is NOT in any way a thorough characterization of these machines, but it is an easily-generated estimate that is typically surprisingly close to much more careful benchmarks.

The single most remarkable feature of this table is that it spans roughly six orders of magnitude. That's the difference between an algorithm taking one day and 3000 years, about the duration of recorded history. For some big problems, literally the fastest way to solve them can be to wait for a faster computer to be built. Notice also how the array version of the program (iterating over $N = 2 \times 10^6$ points) drops the speed of the vector architectures down by an order of magnitude, without slowing the scalar machines. For some problems what might naively appear to be a slower computer can be preferable.