Computational Physics

Richard Fitzpatrick

Professor of Physics

The University of Texas at Austin

Contents

6 The Diffusion Equation   **218**

1   Introduction

1.1   Intended Audience

These set of lecture notes are designed for an upper-division undergraduate course on computational physics.

1.2   Major Sources

The sources which I have consulted most frequently whilst developing course material are as follows:

C/C++ PROGRAMMING:

Software engineering in C, P.A. Darnell, and P.E. Margolis (Springer-Verlag, New York NY, 1988).

The C++ programming language, 2nd edition, B. Stroustrup (Addison-Wesley, Reading MA, 1991).

Schaum's outline: Programming with C, 2nd edition, B. Gottfried (McGraw-Hill, New York NY, 1996).

Schaum's outline: Programming with C++, 2nd edition, J.R. Hubbard (McGraw-Hill, New York NY, 2000).

NUMERICAL METHODS AND COMPUTATIONAL PHYSICS:

Computational physics, D. Potter (Wiley, New York NY, 1973).

Numerical recipes in C: the art of scientific computing, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press, Cambridge UK, 1992).

Computational physics, N.J. Giordano (Prentice-Hall, Upper Saddle River NJ, 1997).

Numerical methods for physics, 2nd edition, A.L. Garcia (Prentice-Hall, Upper Saddle River NJ, 2000).

PHYSICS OF BASEBALL:

>   The physics of baseball, R.K. Adair (Harper & Row, New York NY, 1990).

>   The physics of sports, A.A. Armenti, Jr., Ed. (American Institute of Physics, New York NY, 1992).

CHAOS:

>   Chaos in a computer-animated pendulum, R.L. Kautz, Am. J. Phys. **61**, 407 (1993).

>   Nonlinear dynamics and chaos, S.H. Strogatz, (Addison-Wesley, Reading MA, 1994).

>   Chaos: An introduction to dynamical systems, K.T. Alligood, T.D. Sauer, and J.A. Yorke, (Springer-Verlag, New York NY, 1997).

1.3   Purpose of Course

The purpose of this course is demonstrate to students how computers can enable us to both broaden and deepen our understanding of physics by vastly increasing the range of mathematical calculations which we can conveniently perform.

1.4   Course Philosophy

My approach to computational physics is to write self-contained programs in a high-level scientific language—*i.e.,* either FORTRAN or C/C++. Of course, there are many other possible approaches, each with their own peculiar advantages and disadvantages. It is instructive to briefly examine the available options.

1.5   Programming Methodologies

Basically, there are *three* possible methods by which we could perform the numerical calculations which we are going to encouter during this course.

Firstly, we could use a mathematical software package, such as MATHEMATICA[1], MAPLE[2] or MATLAB.[3] The main advantage of these packages is that they facilitate the very rapid coding up of numerical problems. The main disadvantage is that they produce executable code which is *interpreted*, rather than *compiled*. Compiled code is translated directly from a high-level language into machine code instructions, which, by definition, are platform dependent—after all, an Intel x86 chip has a completely different instruction set to a Power-PC chip. Interpreted code is translated from a high-level language into a set of meta-code instructions which are platform independent. Each meta-code instruction is then translated into a fixed set of machine code instructions which is peculiar to the particular hardware platform on which the code is being run. In general, interpreted code is nowhere near as efficient, in terms of computer resource utilization, as compiled code: *i.e.*, interpreted code run *a lot slower* than equivalent compiled code. Thus, although MATHEMATICA, MAPLE, and MATLAB are ideal environments in which to perform relatively *small* calculations, they are not suitable for full-blown research projects, since the code which they produce generally runs far too slowly.

Secondly, we could write our own programs in a high-level language, but use calls to pre-written, pre-compiled routines in commonly available subroutine libraries, such as NAG,[4] LINPACK,[5] and ODEPACK,[6] to perform all of the real numerical work. This is the approach used by the majority of research physicists.

Thirdly, we could write our own programs—completely from scratch—in a high-level language. This is the approach used in this course. I have opted *not* to use pre-written subroutine libraries, simply because I want students to develop the ability to think for themselves about scientific programming and numerical techniques. Students should, however, realize that, in many cases, pre-written library routines offer solutions to numerical problems which are pretty hard to improve upon.

---

[1]See http://www.wolfram.com

[2]See http://www.maplesoft.com

[3]See http://www.mathworks.com

[4]See http://www.nag.com

[5]See http://www.netlib.org

[6]*ibid.*

1.6 Scientific Programming Languages

What is the best high-level language to use for scientific programming? This, unfortunately, is a highly contentious question. Over the years, literally hundreds of high-level languages have been developed. However, few have stood the test of time. Many languages (*e.g.*, Algol, Pascal, Haskell) can be dismissed as ephemeral computer science fads. Others (*e.g.*, Cobol, Lisp, Ada) are too specialized to adapt for scientific use. Let us examine the remaining options:

FORTRAN 77: FORTRAN was the first high-level programming language to be developed: in fact, it predates the languages listed below by decades. Before the advent of FORTRAN, all programming was done in assembler code! Moreover, FORTRAN was specifically designed for scientific computing. Indeed, in the early days of computers *all* computing was scientific in nature— *i.e.*, physicists and mathematicians were the original computer scientists! FORTRAN's main advantages are that it is very straightforward, and it interfaces well with most commonly available, pre-written subroutine libraries (since these libraries generally consist of compiled FORTRAN code). FORTRAN's main disadvantages are all associated with its relative antiquity. For instance. FORTRAN's control statements are fairly rudimentary, whereas its input/output facilities are positively paleolithic.

FORTRAN 90: This language is a major extension to FORTRAN 77 which does away with many of the latter language's objectionable features. In addition, many "modern" features, such as dynamic memory allocation, are included in the language for the first time. The major disadvantage of this language is the absence of an inexpensive compiler. There seems little prospect of this situation changing in the near future.

C: This language was originally developed by computer scientists to write operating systems. Indeed, all UNIX operating systems are written in C. C is, consequently, an extremely flexible and powerful language. Amongst its major advantages are its good control statements and excellent input/output facilities. C's main disadvantage is that, since it was not specifically written to be a scientific language, some important scientific features (*e.g.*, complex

arithmetic) are missing. Although C is a high-level language, it incorporates many comparatively low-level features, such as pointers (this is hardly surprisingly, since C was originally designed to write operating systems). The low-level features of C—in particular, the rather primitive implementation of arrays—sometimes make scientific programming more complicated than need be the case, and undoubtedly facilitate programming errors. On the other hand, these features allow scientific programmers to write *extremely* efficient code. Since efficiency is generally the most important concern in scientific computing, the low-level features of C are, on balance, advantageous.

C++: This language is a major extension of C whose main aim is to facilitate object-orientated programming. Object-orientation is a completely different approach to programming than the more traditional procedural approach: it is particularly well suited to large projects involving many people who are each writing different segments of the same code. However, object-orientation represents a large, and somewhat unnecessary, overhead for the type of straightforward, single person programming tasks considered in this course. Note, however, that C++ incorporates some non-object-orientated extensions to C which are extremely useful.

Of the above languages, we can immediately rule out C++, because object-orientation is an unnecessary complication (at least, for our purposes), and FORTRAN 90, because of the absence of an inexpensive compiler. The remaining options are FORTRAN 77 and C. I have chosen to use C (augmented by some of the useful, non-object-orientated features of C++) in this course, simply because I find the archaic features of FORTRAN 77 too embarrassing to teach students in the 21st century.

2   Scientific Programming in C

2.1   Introduction

As we have already mentioned, C is a flexible, extremely powerful, high-level programming language which was initially designed for writing operating systems and system applications. In fact, all UNIX operating systems, as well as most UNIX applications (*e.g.*, text editors, window managers, *etc.*) are written in C. However, C is also an excellent vehicle for scientific programming, since, almost by definition, a good scientific programming language must be powerful, flexible, and high-level. Having said this, many of the features of C which send computer scientists into raptures are not particularly relevant to the needs of the scientific programmer. Hence, in the following, we shall only describe that subset of the C language which is really necessary to write scientific programs. It may be objected that our cut-down version of C bears a suspicious resemblance to FORTRAN. However, this resemblance is hardly surprising. After all, FORTRAN is a high-level programming language which was specifically designed with scientific computing in mind.

As discussed previously, C++ is an extension of the C language whose main aim is to facilitate object-orientated programming. The object-orientated features of C++ are superfluous to our needs in this course. However, C++ incorporates some new, non-object-orientated features which are extremely useful to the scientific programmer. We shall briefly discuss these features towards the end of this section. Finally, we shall describe some prewritten C++ classes which allow us to incorporate complex arithmetic (which is not part of the C language), variable size arrays, and graphics into our programs.

2.2   Variables

Variable names in C can consist of letters and numbers in any order, except that the first character must be a letter. Names are *case sensitive*, so upper- and lowercase letters are not interchangeable. The underscore character (_) can also be

included in variable names, and is treated as a letter. There is no restriction on the length of names in C. Of course, variable names are not allowed to clash with keywords that play a special role in the C language, such as `int`, `double`, `if`, `return`, `void`, *etc*. The following are examples of valid variable names in C:

```
x   c14   area   electron_mass   TEMPERATURE
```

The C language supports a great variety of different data types. However, the two data types which occur most often in scientific programs are *integer*, denoted `int`, and *floating-point*, denoted `double`. (Note that variables of the most basic floating-point data type `float` are not generally stored to sufficient precision by the computer to be of much use in scientific programming.) The data type (`int` or `double`) of every variable in a C program must be declared *before* that variable can appear in an executable statement.

Integer constants in C are denoted, in the regular fashion, by strings of arabic numbers: *e.g.*,

```
0   57   4567   128933
```

Floating-point constants can be written in either regular or scientific notation: *e.g.*,

```
0.01   70.456   3e+5   .5067e-16
```

Strings are mainly used in scientific programs for data input and output purposes. A string consists of any number of consecutive characters (including blanks) enclosed in double quotation marks: *e.g.*,

```
"red"   "Austin TX, 78723"   "512-926-1477"
```

Line-feeds can be incorporated into strings via the *escape sequence* \n: *e.g.*,

```
"Line 1\nLine 2\nLine 3"
```

The above string would be displayed on a computer terminal as

```
Line 1
Line 2
Line 3
```

A *declaration* associates a group of variables with a specific data type. As mentioned previously, all variables must be declared before they can appear in executable statements. A declaration consists of a data type followed by one or more variable names, ending in a semicolon. For instance,

```
int     a, b, c;
double    acc, epsilon, t;
```

In the above, `a`, `b`, and `c` are declared to be integer variables, whereas `acc`, `epsilon`, and `t` are declared to be floating-point variables.

A type declaration can also be used to assign initial values to variables. Some examples of how to do this are given below:

```
int     a = 3, b = 5;
double    factor = 1.2E-5;
```

Here, the integer variables `a` and `b` are assigned the initial values 3 and 5, respectively, whereas the floating-point variable `factor` is assigned the initial value 1.2×10^{-5}.

Note that there is no restriction on the length of a type declaration: such a declaration can even be split over many lines, so long as its end is signaled by a semicolon. However, all declaration statements in a program (or program segment) must occur *prior* to the first executable statement.

2.3   Expressions and Statements

An *expression* represents a single data item—usually a number. The expression may consist of a single entity, such as a constant or variable, or it may consist of some combination of such entities, interconnected by one or more *operators*. Expressions can also represent logical conditions which are either true or false. However, in C, the conditions true and false are represented by the integer values 1 and 0, respectively. Several simple expressions are given below:

```
a + b
x = y
```

```
t = u + v
x <= y
++j
```

The first expression, which employs the *addition operator* (+), represents the sum of the values assigned to variables `a` and `b`. The second expression involves the *assignment operator* (=), and causes the value represented by `y` to be assigned to `x`. In the third expression, the value of the expression (`u + v`) is assigned to `t`. The fourth expression takes the value `1` (true) if the value of `x` is less than or equal to the value of `y`. Otherwise, the expression takes the value `0` (false). Here, `<=` is a *relational operator* that compares the values of `x` and `y`. The final example causes the value of `j` to be increased by `1`. Thus, the expression is equivalent to

```
j = j + 1
```

The *increment* (by unity) operator `++` is called a *unary* operator, because it only possesses one operand.

A *statement* causes the computer to carry out some definite action. There are three different classes of statements in C: *expression statements*, *compound statements*, and *control statements*.

An expression statement consists of an expression followed by a semicolon. The execution of such a statement causes the associated expression to be evaluated. For example:

```
a = 6;
c = a + b;
++j;
```

The first two expression statements both cause the value of the expression on the right of the equal sign to be assigned to the variable on the left. The third expression statement causes the value of `j` to be incremented by `1`. Again, there is no restriction on the length of an expression statement: such a statement can even be split over many lines, so long as its end is signaled by a semicolon.

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression

statements, compound statements, or control statements. Unlike expression statements, compound statements do *not* end with semicolons. A typical compound statement is shown below:

```
{
  pi = 3.141593;
  circumference = 2. * pi * radius;
  area = pi * radius * radius;
}
```

This particular compound statement consists of three expression statements, but acts like a single entity in the program in which it appears.

A *symbolic constant* is a name that substitutes for a sequence of characters. The characters may represent either a number or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. Symbolic constants are usually defined at the beginning of a program, by writing

```
#define   NAME   text
```

where `NAME` represents a symbolic name, typically written in upper-case letters, and `text` represents the sequence of characters that is associated with that name. Note that `text` does *not* end with a semicolon, since a symbolic constant definition is not a true C statement. In fact, during compilation, the resolution of symbolic names is performed (by the *C preprocessor*) before the start of true compilation. For instance, suppose that a C program contains the following symbolic constant definition:

```
#define   PI   3.141593
```

Suppose, further, that the program contains the statement

```
area = PI * radius * radius;
```

During the compilation process, the preprocessor replaces each occurrence of the symbolic constant `PI` by its corresponding text. Hence, the above statement becomes

```
area = 3.141593 * radius * radius;
```

Symbolic constants are particularly useful in scientific programs for representing constants of nature, such as the mass of an electron, the speed of light, *etc*. Since these quantities are fixed, there is little point in assigning variables in which to store them.

2.4 Operators

As we have seen, general expressions are formed by joining together constants and variables via various operators. Operators in C fall into five main classes: *arithmetic operators, unary operators, relational and logical operators, assignment operators*, and the *conditional operator*. Let us, now, examine each of these classes in detail.

There are *four* main arithmetic operators in C. These are:

```
addition          +
subtraction       -
multiplication    *
division          /
```

Unbelievably, there is no built-in exponentiation operator in C (C was written by computer scientists)! Instead, there is a *library function* (`pow`) which carries out this operation (see later).

It is poor programming practice to mix types in arithmetic expressions. In other words, the two operands operated on by the addition, subtraction, multiplication, or division operators should both be either of type `int` or type `double`. The value of an expression can be converted to a different data type by prepending the name of the desired data type, enclosed in parenthesis. This type of construction is known as a *cast*. Thus, to convert an integer variable `j` into a floating-point variable with the same value, we would write

```
(double) j
```

Finally, to avoid mixing data types when dividing a floating-point variable `x` by an integer variable `i`, we would write

```
x / (double) i
```

Of course, the result of this operation would be of type `double`.

The operators within C are grouped hierarchically according to their *precedence* (*i.e.*, their order of evaluation). Amongst the arithmetic operators, ∗ and / have precedence over + and -. In other words, when evaluating expressions, C performs multiplication and division operations prior to addition and subtraction operations. Of course, the rules of precedence can always be bypassed by judicious use of parentheses. Thus, the expression

```
a - b / c + d
```

is equivalent to the unambiguous expression

```
a - (b / c) + d
```

since division takes precedence over addition and subtraction.

The distinguishing feature of unary operators is that they only act on single operands. The most common unary operator is the *unary minus*, which occurs when a numerical constant, variable, or expression is preceded by a minus sign. Note that the unary minus is distinctly different from the arithmetic operator (-) which denotes subtraction, since the latter operator acts on two separate operands. The two other common unary operators are the *increment operator*, ++, and the *decrement operator*, --. The increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1. For example, --i is equivalent to i = i - 1. A *cast* is also considered to be a unary operator. Note that unary operators have precedence over arithmetic operators. Hence, - x + y is equivalent to the unambiguous expression (-x) + y, since the unary minus operator has precedence over the addition operator.

Note that there is a subtle distinction between the expressions a++ and ++a. In the former case, the value of the variable a is returned *before* it is incremented. In the latter case, the value of a is returned *after* incrementation. Thus,

```
b = a++;
```

is equivalent to

```
b = a;
a = a + 1;
```

whereas

```
b = ++a;
```

is equivalent to

```
a = a + 1;
b = a;
```

There is a similar distinction between the expressions a-- and --a.

There are four relational operators in C. These are:

```
less than                   <
less than or equal to       <=
greater than                >
greater than or equal to    >=
```

The precedence of these operators is lower than that of arithmetic operators.

Closely associated with the relational operators are the two *equality operators*:

```
equal to        ==
not equal to    !=
```

The precedence of the equality operators is below that of the relational operators.

The relational and equality operators are used to form logical expressions, which represent conditions that are either true or false. The resulting expressions are of type int, since true is represented by the integer value 1 and false by the integer value 0. For example, the expression i < j is true (value 1) if the value of i is less than the value of j, and false (value 0) otherwise. Likewise, the expression j == 3 is true if the value of j is equal to 3, and false otherwise.

C also possess two *logical operators*. These are:

```
&&         and
||         or
```

The logical operators act on operands which are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex expressions that are either true or false. The result of a *logical and* operation is only true if both operands are true, whereas the result of a *logical or* operation is only false if both operands are false. For instance, the expression (i >= 5) && (j == 3) is true if the value of i is greater than or equal to 5 *and* the value of j is equal to 3, otherwise it is false. The precedence of the *logical and* operator is higher than that of the *logical or* operator, but lower than that of the equality operators.

C also includes the unary operator ! that negates the value of a logical expression: *i.e.*, it causes an expression that is originally true to become false, and *vice versa*. This operator is referred to as the *logical negation* or *logical not* operator. For instance, the expression !(k == 4) is true if the value of k is not equal to 4, and false otherwise.

Note that it is poor programming practice to rely too heavily on operator precedence, since such reliance tends to makes C programs very hard for other people to follow. For instance, instead of writing

```
i + j == 3 && i * l >= 5
```

and relying on the fact that arithmetic operators have precedence over relational and equality operators, which, in turn, have precedence over logical operators, it is better to write

```
((i + j) == 3) && (i * l >= 5)
```

whose meaning is fairly unambiguous, even to people who cannot remember the order of precedence of the various operators in C.

The most common assignment operator in C is =. For instance, the expression

```
f = 3.4
```

causes the floating-point value 3.4 to be assigned to the variable f. Note that the assignment operator = and the equality operator == perform completely different functions in C, and should not be confused. Multiple assignments are permissible in C. For example,

```
i = j = k = 4
```

causes the integer value 4 to be assigned to i, j, and k, simultaneously. Note, again, that it is poor programming practice to mix data types in assignment expressions. Thus, the data types of the constants or variables on either side of the = sign should always match.

C contains four additional assignment operators: +=, -=, *=, and /=. The expression

```
i += 6
```

is equivalent to i = i + 6. Likewise, the expression

```
i -= 6
```

is equivalent to i = i - 6. The expression

```
i *= 6
```

is equivalent to i = i * 6. Finally, the expression

```
i /= 6
```

is equivalent to i = i / 6. Note that the precedence of assignment operators is below that of all the operators discussed previously.

Simple conditional operations can be carried out with the *conditional operator* (? :). An expression that makes use of the conditional operator is called a *conditional expression*. Such an expression takes the general form

```
expression 1  ?  expression 2  :  expression 3
```

If expression 1 is true (*i.e.*, if its value is nonzero) then expression 2 is evaluated and becomes the value of the conditional expression. On the other hand, if expression 1 is false (*i.e.*, if its value is zero) then expression 3 is evaluated and becomes the value of the conditional expression. For instance, the expression

```
(j < 5) ? 12 : -6
```

takes the value 12 if the value of j is less than 5, and the value -6 otherwise. The assignment statement

```
k = (i < 0) ? n : m
```

causes the value of n to be assigned to the variable k if the value of i is less than zero, and the value of m to be assigned to k otherwise. The precedence of the conditional operator is just above that of the assignment operators.

As we have already mentioned, scientific programs tend to be extremely resource intensive. Scientific programmers should, therefore, always be on the lookout for methods of speeding up the execution of their codes. It is important to realize that multiplication (*) and division (/) operations consume *considerably more* CPU time that addition (+), subtraction (-), comparison, or assignment operations. Thus, a simple rule of thumb for writing efficient code is to try to avoid redundant multiplication and division operations. This is particularly important for sections of code which are executed repeatedly: *e.g.*, code which lies within control loops. The classic illustration of this point is the evaluation of a polynomial. The most straightforward method of evaluating (say) a fourth-order polynomial would be to write something like:

```
p = c_0 + c_1 * x + c_2 * x * x + c_3 * x * x * x + c_4 * x * x * x * x
```

Note that the above expression employs *ten* expensive multiplication operations. However, this number can be reduced to *four* via a simple algebraic rearrangement:

```
p = c_0 + x * (c_1 + x * (c_2 + x * (c_3 + x * c_4)))
```

Clearly, the latter expression is far more computationally efficient than the former.

2.5   Library Functions

The C language is accompanied by a number of standard *library functions* which carry out various useful tasks. In particular, all input and output operations (*e.g.*, writing to the terminal) and all math operations (*e.g.*, evaluation of sines and cosines) are implemented by library functions.

In order to use a library function, it is necessary to call the appropriate *header file* at the beginning of the program. The header file informs the program of the name, type, and number and type of arguments, of all of the functions contained in the library in question. A header file is called via the preprocessor statement

```
#include   <filename>
```

where `filename` represents the name of the header file.

A library function is accessed by simply writing the function name, followed by a list of *arguments*, which represent the information being passed to the function. The arguments must be enclosed in parentheses, and separated by commas: they can be constants, variables, or more complex expressions. Note that the parentheses must be present even when there are no arguments.

The C *math library* has the header file `math.h`, and contains the following useful functions:

```
Function        Type        Purpose
--------        ----        -------

acos(d)         double      Return arc cosine of d (in range 0 to pi)
asin(d)         double      Return arc sine of d (in range -pi/2 to pi/2)
atan(d)         double      Return arc tangent of d (in range -pi/2 to pi/2)
atan2(d1, d2)   double      Return arc tangent of d1/d2 (in range -pi to pi)
cbrt(d)         double      Return cube root of d
cos(d)          double      Return cosine of d
cosh(d)         double      Return hyperbolic cosine of d
exp(d)          double      Return exponential of d
fabs(d)         double      Return absolute value of d
hypot(d1, d2)   double      Return sqrt(d1 * d1 + d2 * d2)
log(d)          double      Return natural logarithm of d
log10(d)        double      Return logarithm (base 10) of d
```

```
pow(d1, d2)      double       Return d1 raised to the power d2
sin(d)           double       Return sine of d
sinh(d)          double       Return hyperbolic sine of d
sqrt(d)          double       Return square root of d
tan(d)           double       Return tangent of d
tanh(d)          double       Return hyperbolic tangent of d
```

Here, `Type` refers to the data type of the quantity that is returned by the function. Moreover, `d`, `d1`, *etc.* indicate arguments of type `double`.

A program that makes use of the C math library would contain the statement

```
#include   <math.h>
```

close to its start. In the body of the program, a statement like

```
x = cos(y);
```

would cause the variable `x` to be assigned a value which is the cosine of the value of the variable `y` (both `x` and `y` should be of type `double`).

Note that math library functions tend to be *extremely expensive* in terms of CPU time, and should, therefore, only be employed when absolutely necessary. The classic illustration of this point is the use of the `pow()` function. This function assumes that, in general, it will be called with a *fractional* power, and, therefore, implements a full-blown (and very expensive) series expansion. Clearly, it is not computationally efficient to use this function to square or cube a quantity. In other words, if a quantity needs to be raised to a small, positive integer power then this should be implemented *directly,* instead of using the `pow()` function: *i.e.,* we should write `x * x` rather than `pow(x, 2)`, and `x * x * x` rather than `pow(x, 3)`, *etc.* (Of course, a properly designed exponentiation function would realize that it is more efficient to evaluate small positive integer powers by the direct method. Unfortunately, the `pow()` function was written by computer scientists!)

The C math library comes with a useful set of predefined mathematical constants:

```
Name          Description
```

```
----        -----------

M_PI        Pi, the ratio of a circle's circumference to its diameter.
M_PI_2      Pi divided by two.
M_PI_4      Pi divided by four.
M_1_PI      The reciprocal of pi (1/pi).
M_SQRT2     The square root of two.
M_SQRT1_2   The reciprocal of the square root of two
             (also the square root of 1/2).
M_E         The base of natural logarithms.
```

The other library functions commonly used in C programs will be introduced, as appropriate, during the remainder of this discussion.

2.6 Data Input and Output

Data input and output operations in C are carried out by the standard input/output library (header file: `stdio.h`) via the functions `scanf`, `printf`, `fscanf`, and `fprintf`, which read and write data from/to the terminal, and from/to a data file, respectively. The additional functions `fopen` and `fclose` open and close, respectively, connections between a C program and a data file. In the following, these functions are described in detail.

The `scanf` function reads data from standard input (usually, the terminal). A call to this function takes the general form

```
scanf(control_string, arg1, arg2, arg3, ...)
```

where `control_string` refers to a character string containing certain required formatting information, and `arg1`, `arg2`, *etc.*, are arguments that represent the individual input data items.

The control string consists of individual groups of characters, with one character group for each data input item. In its simplest form, each character group consists of a percent sign (%), followed by a set of *conversion characters* which indicate the type of the corresponding data item. The two most useful sets of conversion characters are as follows:

```
Character        Type
---------        ----

d                int
lf               double
```

The arguments are a set of variables whose types match the corresponding character groups in the control string. For reasons which will become apparent later on, *each variable name must be preceded by an ampersand* (&). Below is a typical application of the scanf function:

```
#include  <stdio.h>
. . .
int  k;
double  x, y;
. . .
scanf("%d %lf %lf", &k, &x, &y);
. . .
```

In this example, the scanf function reads an integer value and two floating-point values, from standard input, into the integer variable k and the two floating-point variables x and y, respectively.

The scanf function returns an integer equal to the number of data values successfully read from standard input, which can be fewer than expected, or even zero, in the event of a matching failure. The special value EOF (which on most systems corresponds to -1) is returned if an end-of-file is reached before any attempted conversion occurs. The following code snippet gives an example of how the scanf function can be checked for error-free input:

```
#include  <stdio.h>
. . .
int  check_input;
double  x, y, z;
. . .
check_input = scanf("%lf %lf %lf", &x, &y, &z);
if (check_input < 3)
 {
  printf("Error during data input\n");
   . . .
 }
. . .
```

27

See later for an explanation of the `if()` construct.

The `printf` function writes data to standard output (usually, the terminal). A call to this function takes the general form

```
printf(control_string, arg1, arg2, arg3, ...)
```

where `control_string` refers to a character string containing formatting information, and `arg1`, `arg2`, *etc.*, are arguments that represent the individual output data items.

The control string consists of individual groups of characters, with one character group for each output data item. In its simplest form, each character group consists of a percent sign (%), followed by a *conversion character* which controls the format of the corresponding data item. The most useful conversion characters are as follows:

```
Character       Meaning
---------       -------

d               Display data item as signed decimal integer
f               Display data item as floating-point number without exponent
e               Display data item as floating-point number with exponent
```

The arguments are a set of variables whose types match the corresponding character groups in the control string (*i.e.*, type `int` for `d` format, and type `double` for `f` or `e` format). In contrast to the `scanf` function, the arguments are *not* preceded by ampersands. Below is a typical application of the `scanf` function:

```
#include  <stdio.h>
. . .
int  k = 3;
double  x = 5.4, y = -9.81;
. . .
printf("%d %f %f\n", k, x, y);
. . .
```

In this example, the program outputs the values of the integer variable `k` and the floating-point variables `x` and `y` to the terminal. Executing the program produces the following output:

```
3 5.400000 -9.810000
%
```

Note that the purpose of the escape sequence \n in the control string is to generate a line-feed after the three data items have been written to the terminal.

Of course, the printf function can also be used to write a simple text string to the terminal: *e.g.,*

```
printf(text_string)
```

Ordinary text can also be incorporated into the control string described above.

An example illustrating somewhat more advanced use of the printf function is given below:

```
#include  <stdio.h>
. . .
int  k = 3;
double  x = 5.4, y = -9.81;
. . .
printf("k = %3d  x + y = %9.4f  x*y = %11.3e\n", k, x + y, x*y);
. . .
```

Executing the program produces the following output:

```
k =   3  x + y =   -4.4100  x*y =  -5.297e+01
%
```

Note that the final two arguments of the printf function are arithmetic expressions. Note, also, the incorporation of explanatory text into the control string.

The character sequence %3d in the control string indicates that the associated data item should be output as a signed decimal integer occupying a field whose width is *at least* 3 characters. More generally, the character sequence %nd indicates that the associated data item should be output as a signed decimal integer occupying a field whose width is *at least* n characters. If the number of characters in the data item is less than n characters, then the data item is preceded by

enough leading blanks to fill the specified field. On the other hand, if the data item exceeds the specified field width then additional space is allocated to the data item, such that the entire data item is displayed.

The character sequence `%9.4f` in the control string indicates that the associated data item should be output as a floating-point number, in non-scientific format, which occupies a field of at least 9 characters, and has 4 figures after the decimal point. More generally, the character sequence `%n.mf` indicates that the associated data item should be output as a floating-point number, in non-scientific format, which occupies a field of at least `n` characters, and has `m` figures after the decimal point.

Finally, the character sequence `%11.3e` in the control string indicates that the associated data item should be output as a floating-point number, in scientific format, which occupies a field of at least 11 characters, and has 3 figures after the decimal point. More generally, the character sequence `%n.me` indicates that the associated data item should be output as a floating-point number, in scientific format, which occupies a field of at least `n` characters, and has `m` figures after the decimal point.

The `printf` function returns an integer equal to the number of printed characters, or a negative value if there was an output error.

When working with a *data file*, the first step is to establish a *buffer area*, where information is temporarily stored whilst being transferred between the program and the file. This buffer area allows information to be read or written to the data file in question more rapidly than would otherwise be possible. A buffer area is established by writing

```
FILE *stream;
```

where `FILE` (upper-case letters required) is a special structure type that establishes a buffer area, and `stream` is the identifier of the created buffer area. Note that a buffer area is often referred to as an input/output *stream*. The meaning of the asterisk (`*`) that precedes the identifier of the stream, in the above statement, will become clear later on. It is, of course, possible to establish multiple input/output streams (provided that their identifiers are distinct).

A data file must be opened and attached to a specific input/output stream before it can be created or processed. This operation is performed by the function `fopen`. A typical call to `fopen` takes the form

```
stream = fopen(file_name, file_type);
```

where `stream` is the identifier of the input/output stream to which the file is to be attached, and `file_name` and `file_type` are character strings that represent the name of the data file and the manner in which the data file will be utilized, respectively. The `file_type` string must be one of the strings listed below:

```
file_type          Meaning
---------          -------


"r"         Open existing file for reading only
"w"         Open new file for writing only (Any existing file
             will be overwritten)
"a"         Open existing file in append mode. (Output will be
             appended to the file)
```

The `fopen` function returns the integer value `NULL` (which on most systems corresponds to zero) in the event of an error.

A data file must also be closed at the end of the program. This operation is performed by the function `fclose`. The syntax for a call to `fclose` is simply

```
fclose(stream);
```

where `stream` is the name of the input/output stream which is to be deattached from a data file. The `fclose` function returns the integer value `0` upon successful completion, otherwise it returns the special value `EOF`.

Data can be read from an open data file using the `fscanf` function, whose syntax is

```
fscanf(stream, control_string,  arg1,  arg2,  arg3,  ...)
```

Here, `stream` is the identifier of the input/output stream to which the file is attached, and the remaining arguments have exactly the same format and meaning

as the corresponding arguments for the `scanf` function. The return values of `fscanf` are similar to those of the `scanf` function.

Likewise, data can be written to an open data file using the `fprintf` function, whose syntax is

```
fprintf(stream, control_string, arg1, arg2, arg3, ...)
```

Here, `stream` is the identifier of the input/output stream to which the file is attached, and the remaining arguments have exactly the same format and meaning as the corresponding arguments for the `printf` function. The return values of `fprintf` are similar to those of the `printf` function.

An example of a C program which outputs data to the file "data.out" is given below:

```
#include  <stdio.h>
. . .
int  k = 3;
double  x = 5.4, y = -9.81;
FILE *output;
. . .
output = fopen("data.out", "w");
if (output == NULL)
 {
  printf("Error opening file data.out\n");
  . . .
 }
. . .
fprintf(output, "k = %3d  x + y = %9.4f  x*y = %11.3e\n", k, x + y, x*y);
. . .
fclose(output);
. . .
```

On execution, the above program will write the line

```
k =   3  x + y =   -4.4100  x*y =  -5.297e+01
```

to the data file "data.out".

2.7   Structure of a C Program

The syntax of a complete C program is given below:

```
. . .
int main()
{
 . . .
 return 0;
}
. . .
```

The meaning of the statements `int main()` and `return` will become clear later on. Preprocessor statements (*e.g.*, `#define` and `#include` statements) are conventionally placed *before* the `int main()` statement. All executable statements must be placed *between* the `int main()` and `return` statements. Function definitions (see later) are conventionally placed *after* the `return` statement.

A simple C program (`quadratic.c`) that calculates the real roots of a quadratic equation using the well-known quadratic formula is listed below.

```
/* quadratic.c */
/*
  Program to evaluate real roots of quadratic equation

     2
  a x  + b x + c = 0

  using quadratic formula

                2
  x = ( -b +/- sqrt(b - 4 a c) ) / (2 a)
*/

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

int main()
{
  double a, b, c, d, x1, x2;

  /* Read input data */
```

```
  printf("\na = ");
  scanf("%lf", &a);
  printf("b = ");
  scanf("%lf", &b);
  printf("c = ");
  scanf("%lf", &c);

  /* Perform calculation */
  d = sqrt(b * b - 4. * a * c);
  x1 = (-b + d) / (2. * a);
  x2 = (-b - d) / (2. * a);

  /* Display output */
  printf("\nx1 = %12.3e   x2 = %12.3e\n", x1, x2);

  return 0;
}
```

Note the use of *comments* (which are placed between the delimiters /∗ and ∗/) to first explain the function of the program and then identify the program's major sections. Note, also, the use of *indentation* to highlight the executable statements. When executed, the above program produces the following output:

```
a = 2
b = 4
c = 1

x1 =   -2.929e-01   x2 =   -1.707e+00
%
```

Of course, the 2, 4, and 1 were entered by the user in response to the programs's prompts.

It is important to realize that there is more to writing a complete computer program than simply arranging the individual declarations and statements in the right order. Attention should also be given to making the program and its output as *readable* as possible, so that the program's function is *immediately obvious* to other people. This can be achieved by judicious use of indentation and whitespace, as well as the inclusion of comments, and the generation of clearly labeled output. It is hoped that this approach will be exemplified by the example programs used in this course.

2.8 Control Statements

The C language includes a wide variety of powerful and flexible control statements. The most useful of these are described in the following.

The `if-else` statement is used to carry out a logical test and then take one of two possible actions, depending on whether the outcome of the test is true or false. The `else` portion of the statement is optional. Thus, the simplest possible `if-else` statement takes the form:

```
if (expression) statement
```

The expression must be placed in parenthesis, as shown. In this form, the statement will only be executed if the expression has a nonzero value (*i.e.*, if `expression` if true). If the expression has a value of zero (*i.e.*, if `expression` is false) then the statement will be ignored. The statement can be either simple or compound.

The program `quadratic.c`, listed previously, is incapable of dealing correctly with cases where the roots are complex (*i.e.*, $b^2 < 4\,a\,c$), or cases where $a = 0$. It is good programming practice to test for situations which fall outside the domain of validity of a program, and produce some sort of error message when these occur. An amended version of `quadratic.c` which uses `if-else` statements to reject invalid input data is listed below.

```
/* quadratic1.c */
/*
  Program to evaluate real roots of quadratic equation

     2
  a x   + b x + c = 0

  using quadratic formula

                2
  x = ( -b +/- sqrt(b - 4 a c) ) / (2 a)

  Program rejects cases where roots are complex
  or where a = 0.
*/

#include <stdio.h>
```

```c
#include <math.h>
#include <stdlib.h>

int main()
{
  double a, b, c, d, e, x1, x2;

  /* Read input data */
  printf("\na = ");
  scanf("%lf", &a);
  printf("b = ");
  scanf("%lf", &b);
  printf("c = ");
  scanf("%lf", &c);

  /* Test for complex roots */
  e = b * b - 4. * a * c;

  if (e < 0.)
   {
    printf("\nError: roots are complex\n");
    exit(1);
   }

  /* Test for a = 0. */
  if (a == 0.)
   {
    printf("\nError: a = 0.\n");
    exit(1);
   }

  /* Perform calculation */
  d = sqrt(e);
  x1 = (-b + d) / (2. * a);
  x2 = (-b - d) / (2. * a);

  /* Display output */
  printf("\nx1 = %12.3e   x2 = %12.3e\n", x1, x2);

  return 0;
}
```

Note the use of indentation to highlight statements which are conditionally exe-
cuted (*i.e.*, statements within an `if-else` statement). The standard library func-
tion call `exit(1)` (header file: `stdlib.h`) causes the program to abort with an

error status. Execution of the above program for the case of complex roots yields the following output:

```
a = 4
b = 2
c = 6

Error: roots are complex
%
```

The general form of an `if-else` statement, which includes the `else` clause, is

```
if (expression)  statement 1  else  statement 2
```

If the expression has a non-zero value (*i.e.*, if `expression` is true) then `statement 1` is executed. Otherwise, `statement 2` is executed. The program listed below is an extended version of the previous program `quadratic.c` which is capable of dealing with complex roots.

```
/* quadratic2.c */
/*
  Program to evaluate all roots of quadratic equation

     2
  a x  + b x + c = 0

  using quadratic formula

               2
  x = ( -b +/- sqrt(b - 4 a c) ) / (2 a)

  Program rejects cases where a = 0.
*/

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

int main()
{
  double a, b, c, d, e, x1, x2;
```

```
  /* Read input data */
  printf("\na = ");
  scanf("%lf", &a);
  printf("b = ");
  scanf("%lf", &b);
  printf("c = ");
  scanf("%lf", &c);

  /* Test for a = 0. */
  if (a == 0.)
   {
    printf("\nError: a = 0.\n");
    exit(1);
   }

  /* Perform calculation */
  e = b * b - 4. * a * c;

  if (e > 0.) // Test for real roots
   {
    /* Case of real roots */
    d = sqrt(e);
    x1 = (-b + d) / (2. * a);
    x2 = (-b - d) / (2. * a);
    printf("\nx1 = %12.3e   x2 = %12.3e\n", x1, x2);
   }
  else
   {
    /* Case of complex roots */
    d = sqrt(-e);
    x1 = -b / (2. * a);
    x2 = d / (2. * a);
    printf("\nx1 = (%12.3e, %12.3e)   x2 = (%12.3e, %12.3e)\n",
           x1, x2, x1, -x2);
   }
  return 0;
}
```

Note the use of an `if-else` statement to deal with the two alternative cases of real and complex roots. Note also that the C compiler ignores all characters on a line which occur after the `//` construct.[7] Hence, this construct can be used to comment individual lines in a program. The output from the above program for

---

[7] Strictly speaking, this is a C++ extension to the C language. However, most modern C compilers now accept this comment style.

the case of complex roots looks like:

```
a = 9
b = 2
c = 2

x1 = (  -1.111e-01,    4.581e-01)   x2 = (  -1.111e-01,   -4.581e-01)
%
```

The `while` statement is used to carry out looping operations, in which a group of statements is executed repeatedly until some condition is satisfied. The general form of a `while` statement is

```
while  (expression)  statement
```

The statement is executed repeatedly, as long as the expression is nonzero (*i.e.*, as long as `expression` is true). The statement can be either simple or compound. Of course, the statement must include some feature that eventually alters the value of the expression, thus providing an escape mechanism from the loop.

The program listed below (`iteration.c`) uses a `while` statement to solve an algebraic equation via iteration, as explained in the initial comments.

```
/* iteration.c */
/*
   Program to solve algebraic equation

    5      2
   x   + a x   - b = 0

   by iteration. Easily shown that equation must have at least
   one real root. Coefficients a and b are supplied by user.

   Iteration scheme is as follows:

                 2  0.2
   x    = ( b - a x  )
    n+1            n

   where x_n is nth iteration. User must supply initial guess for x.
   Iteration continues until relative change in x per iteration is
```

```
   less than eps (user supplied) or until number of iterations exceeds
   NITER. Program aborts if (b - a x*x) becomes negative.
*/

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

/* Set max. allowable no. of iterations */
#define NITER 30

int main()
{
  double a, b, eps, x, x0, dx = 1., d;
  int count = 0;

  /* Read input data */
  printf("\na = ");
  scanf("%lf", &a);
  printf("b = ");
  scanf("%lf", &b);
  printf("eps = ");
  scanf("%lf", &eps);

  /* Read initial guess for x */
  printf("\nInitial guess for x = ");
  scanf("%lf", &x);
  x0 = x;

  while (dx > eps)  // Start iteration loop: test for convergence
   {
    /* Check for too many iterations */
    ++count;
    if (count > NITER)
     {
      printf("\nError: no convergence\n");
      exit(1);
     }

    /* Reject complex roots */
    d = b - a * x * x;
    if (d < 0.)
     {
      printf("Error: complex roots - try another initial guess\n");
      exit(1);
```

```
   }

   /* Perform iteration */
   x = pow(d, 0.2);
   dx = fabs( (x - x0) / x );
   x0 = x;

   /* Output data on iteration */
   printf("Iter = %3d   x = %8.4f   dx = %12.3e\n", count, x, dx);
   }
  return 0;
}
```

The typical output from the above program looks like:

```
a = 3
b = 10
eps = 1.e-6

Initial guess for x = 1
Iter =    1   x =    1.4758   dx =    3.224e-01
Iter =    2   x =    1.2823   dx =    1.509e-01
Iter =    3   x =    1.3834   dx =    7.314e-02
Iter =    4   x =    1.3361   dx =    3.541e-02
Iter =    5   x =    1.3595   dx =    1.720e-02
Iter =    6   x =    1.3483   dx =    8.350e-03
Iter =    7   x =    1.3537   dx =    4.056e-03
Iter =    8   x =    1.3511   dx =    1.969e-03
Iter =    9   x =    1.3524   dx =    9.564e-04
Iter =   10   x =    1.3518   dx =    4.644e-04
Iter =   11   x =    1.3521   dx =    2.255e-04
Iter =   12   x =    1.3519   dx =    1.095e-04
Iter =   13   x =    1.3520   dx =    5.318e-05
Iter =   14   x =    1.3519   dx =    2.583e-05
Iter =   15   x =    1.3520   dx =    1.254e-05
Iter =   16   x =    1.3520   dx =    6.091e-06
Iter =   17   x =    1.3520   dx =    2.958e-06
Iter =   18   x =    1.3520   dx =    1.436e-06
Iter =   19   x =    1.3520   dx =    6.975e-07
%
```

When a loop is constructed using a `while` statement, the test for the continuation of the loop is carried out at the *beginning* of each pass. Sometimes, however, it is desirable to have a loop where the test for continuation takes place at the

end of each pass. This can be accomplished by means of a `do-while` statement. The general form of a `do-while` statement is

```
do  statement  while  (expression);
```

The statement is executed repeatedly, as long as the expression is true. Note, however, that the statement is always executed at least once, since the test for repetition does not take place until the end of the first pass through the loop. The statement can be either simple or compound, and should, of course, include some feature that eventually alters the value of the expression.

The program listed below is a marginally improved version of the previous program (`iteration.c`) which uses a `do-while` loop to test for convergence at the end (as opposed to the beginning) of each iteration loop.

```
/* iteration1.c */
/*
   Program to solve algebraic equation

    5      2
   x   + a x   - b = 0

   by iteration. Easily shown that equation must have at least
   one real root. Coefficients a and b are supplied by user.

   Iteration scheme is as follows:

                 2  0.2
   x     = ( b - a x  )
    n+1            n

   where x_n is nth iteration. User must supply initial guess for x.
   Iteration continues until relative change in x per iteration is
   less than eps (user supplied) or until number of iterations exceeds
   NITER. Program aborts if (b - a x*x) becomes negative.
*/

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

/* Set max. allowable no. of iterations */
```

```
#define NITER 30

int main()
{
  double a, b, eps, x, x0, dx, d;
  int count = 0;

  /* Read input data */
  printf("\na = ");
  scanf("%lf", &a);
  printf("b = ");
  scanf("%lf", &b);
  printf("eps = ");
  scanf("%lf", &eps);

  /* Read initial guess for x */
  printf("\nInitial guess for x = ");
  scanf("%lf", &x);
  x0 = x;

  do   // Start iteration loop
   {
    /* Check for too many iterations */
    ++count;
    if (count > NITER)
     {
      printf("\nError: no convergence\n");
      exit(1);
     }

    /* Reject complex roots */
    d = b - a * x * x;
    if (d < 0.)
     {
      printf("Error: complex roots - try another initial guess\n");
      exit(1);
     }

    /* Perform iteration */
    x = pow(d, 0.2);
    dx = fabs( (x - x0) / x );
    x0 = x;

    /* Output data on iteration */
    printf("Iter = %3d   x = %8.4f   dx = %12.3e\n", count, x, dx);
```

```
  } while (dx > eps);   // Test for convergence

  return 0;
}
```

The output from the above program is essentially identical to that from the program `iteration.c`.

The `while` and `do-while` statements are particularly well suited to looping situations in which the number of passes through the loop *is not* known in advance. Conversely, situations in which the number of passes through the loop *is* known in advance are often best dealt with using a `for` statement. The general form of a `for` statement is

```
for  (expression 1;  expression 2;  expression 3)  statement
```

where `expression 1` is used to initialize some parameter (called an *index*) that controls the looping action, `expression 2` represents a condition that must be true for the loop to continue execution, and `expression 3` is used to alter the value of the parameter initially assigned by `expression 1`. When a `for` statement is executed, `expression 2` is evaluated and tested at the *beginning* of each pass through the loop, whereas `expression 3` is evaluated at the *end* of each pass.

The program listed below uses a `for` statement to evaluate the factorial of a non-negative integer.

```
/* factorial.c */
/*
   Program to evaluate factorial of non-negative
   integer n supplied by user.
*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int n, count;
  double fact = 1.;
```

```
/* Read in value of n */
printf("\nn = ");
scanf("%d", &n);

/* Reject negative value of n */
if (n < 0)
 {
  printf("\nError: factorial of negative integer not defined\n");
  exit(1);
 }

/* Calculate factorial */
for (count = n; count > 0; --count) fact *= (double) count;

/* Output result */
printf("\nn = %5d   Factorial(n) = %12.3e\n", n, fact);

return 0;
}
```

The typical output from the above program is shown below:

```
n = 6

n =     6    Factorial(n) =    7.200e+02
%
```

The statements which occur within `if-else`, `while`, `do-while`, or `for` statements can themselves be control statements, giving rise to the possibility of nested `if-else` statements, conditionally executed loops, nested loops, *etc.* When dealing with nested control statements, it is vital to adhere religiously to the syntax rules described above, in order to avoid confusion.

2.9 Functions

We have seen that C supports the use of predefined library functions which are used to carry out a large number of commonly occurring tasks. However, C also allows programmers to define their own functions. The use of programmer-defined functions permits a large program to be broken down into a number of

smaller, self-contained units. In other words, a C program can be *modularized* via the sensible use of programmer-defined functions. In general, modular programs are far easier to write and debug than monolithic programs. Furthermore, proper modularization allows other people to grasp the logical structure of a program with far greater ease than would otherwise be the case.

A *function* is a self-contained program segment that carries out some specific, well-defined task. Every C program consists of one or more functions. One of these functions must be called `main`. Execution of the program always begins by carrying out the instructions contained in `main`. Note that if a program contains multiple functions then their definitions may appear in *any order*. The same function can be accessed from several different places within a program. Once the function has carried out its intended action, control is returned to the point from which the function was accessed. Generally speaking, a function processes information passed to it from the calling portion of the program, and returns a *single value*. Some functions, however, accept information but do not return anything.

A function definition has two principal components: the *first line* (including the *argument declarations*), and the so-called *body* of the function.

The first line of a function takes the general form

```
data-type  name(type 1  arg 1,  type 2  arg 2,  ..., type n  arg n)
```

where `data-type` represents the data type of the item that is returned by the function, `name` represents the name of the function, and `type 1`, `type 2`, `...`, `type n` represent the data types of the arguments `arg 1`, `arg 2`, `...`, `arg n`. The allowable data types for a function are:

```
int      for a function which returns an integer value
double   for a function which returns an floating-point value
void     for a function which does not return any value
```

The allowable data types for a function's arguments are `int` and `double`. Note that the identifiers used to reference the arguments of a function are *local*, in the sense that they are not recognized outside of the function. Thus, the argument

names in a function definition *need not* be the same as those used in the segments of the program from which the function was called. However, the corresponding data types of the arguments must always match.

The body of a function is a compound statement that defines the action to be taken by the function. Like a regular compound statement, the body can contain expression statements, control statements, other compound statements, *etc.* The body can even access other functions. In fact, it can even access itself—this process is known as *recursion*. In addition, however, the body must include one or more `return` statements in order to return a value to the calling portion of the program.

A `return` statement causes the program logic to return to the point in the program from which the function was accessed. The general form of a `return` statement is:

```
return  expression;
```

This statement causes the value of `expression` to be returned to the calling part of the program. Of course, the data type of `expression` should match the declared data type of the function. For a `void` function, which does not return any value, the appropriate `return` statement is simply:

```
return;
```

A maximum of *one* expression can be included in a `return` statement. Thus, a function can return a maximum of one value to the calling part of the program. However, a function definition can include multiple `return` statements, each containing a different expression, which are conditionally executed, depending on the program logic.

Note that, by convention, the `main` function is of type `int` and returns the integer value `0` to the operating system, indicating the error-free termination of the program. In its simplest form, the `main` function possesses *no* arguments. The library function call `exit(1)`, employed in previous example programs, causes the execution of a program to abort, returning the integer value `1` to the operating system, which (by convention) indicates that the program terminated with an error status.

The program segment listed below shows how the previous program `factorial.c` can be converted into a function `factorial(n)` which returns the factorial (in the form of a floating-point number) of the non-negative integer n:

```
double factorial(int n)
{
  /*
     Function to evaluate factorial (in floating-point form)
     of non-negative integer n.
  */

  int count;
  double fact = 1.;

  /* Abort if n is negative integer */
  if (n < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (count = n; count > 0; --count) fact *= (double) count;

  /* Return value of factorial */
  return fact;
}
```

A function can be accessed, or *called*, by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments then an empty pair of parentheses must follow the name of the function. The function call may be part of a simple expression, such as an assignment statement, or it may be one of the operands within a more complex expression. The arguments appearing in a function call may be expressed as constants, single variables, or more complex expressions. However, both the number and the types of the arguments must match those in the function definition.

The program listed below (`printfact.c`) uses the function `factorial()`, described above, to print out the factorials of all the integers between 0 and 20:

```c
/* printfact.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>
#include <stdlib.h>

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

double factorial(int n)
{
  /*
     Function to evaluate factorial (in floating-point form)
     of non-negative integer n.
  */

  int count;
  double fact = 1.;

  /* Abort if n is negative integer */
  if (n < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (count = n; count > 0; --count) fact *= (double) count;

  /* Return value of factorial */
  return fact;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int main()
{
  int j;

  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
    printf("j = %3d    factorial(j) = %12.3e\n", j, factorial(j));
```

```
   return 0;
}
```

Note that the call to `factorial()` takes place inside a complex expression (*i.e.*, a `printf()` function call). Note also that the argument of `factorial()` has a different name (but the same data type) in the two sections of the program (*i.e.*, in the `main()` function and the `factorial()` function). The output from the above program looks like:

```
j =    0    factorial(j) =    1.000e+00
j =    1    factorial(j) =    1.000e+00
j =    2    factorial(j) =    2.000e+00
j =    3    factorial(j) =    6.000e+00
j =    4    factorial(j) =    2.400e+01
j =    5    factorial(j) =    1.200e+02
j =    6    factorial(j) =    7.200e+02
j =    7    factorial(j) =    5.040e+03
j =    8    factorial(j) =    4.032e+04
j =    9    factorial(j) =    3.629e+05
j =   10    factorial(j) =    3.629e+06
j =   11    factorial(j) =    3.992e+07
j =   12    factorial(j) =    4.790e+08
j =   13    factorial(j) =    6.227e+09
j =   14    factorial(j) =    8.718e+10
j =   15    factorial(j) =    1.308e+12
j =   16    factorial(j) =    2.092e+13
j =   17    factorial(j) =    3.557e+14
j =   18    factorial(j) =    6.402e+15
j =   19    factorial(j) =    1.216e+17
j =   20    factorial(j) =    2.433e+18
%
```

Ideally, function definitions should always *precede* the corresponding function calls in a C program. This requirement can usually be satisfied by judicious ordering of the various functions which make up a program, but, almost inevitably, restricts the location of the `main()` function to the *end* of the program. Hence, if the order of the two functions in the above program [*i.e.*, `factorial()` and `main()`] were *swapped* then an error message would be generated on compilation, since an attempt would be made to call `factorial()` prior to its definition. Unfortunately, for the sake of logical clarity, most C programmers prefer to place

the `main()` function at the *beginning* of their programs. After all, `main()` is always the first part of a program to be executed. In such situations, function calls [within `main()`] are bound to precede the corresponding function definitions: fortunately, however, compilation errors can be avoided by using a construct known as a *function prototype*.

Function prototypes are conventionally placed at the beginning of a program (*i.e.*, before the `main()` function) and are used to inform the compiler of the name, data type, and number and data types of the arguments, of all user-defined functions employed in the program. The general form of a function prototype is

```
data-type  name(type 1,  type 2,  ...,  type n);
```

where `data-type` represents the data type of the item returned by the referenced function, `name` is the name of the function, and `type 1, type 2,..., type n` are the data types of the arguments of the function. Note that it is not necessary to specify the names of the arguments in a function prototype. Incidentally, the function prototypes for predefined library functions are contained within the associated header files which must be included at the beginning of every program which uses these functions.

The program listed below is a modified version of `printfact.c` in which the `main()` function is the first function to be defined:

```
/* printfact1.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>
#include <stdlib.h>

/* Prototype for function factorial() */
double factorial(int);

int main()
{
  int j;
```

```
  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
    printf("j = %3d    factorial(j) = %12.3e\n", j, factorial(j));

  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

double factorial(int n)
{
  /*
     Function to evaluate factorial (in floating-point form)
     of non-negative integer n.
  */

  int count;
  double fact = 1.;

  /* Abort if n is negative integer */
  if (n < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (count = n; count > 0; --count) fact *= (double) count;

  /* Return value of factorial */
  return fact;
}
```

Note the presence of the function prototype for `factorial()` prior to the defi-
nition of `main()`. This is needed because the program calls `factorial()` before
this function has been defined. The output from the above program is identical
to that from `printfact.c`.

It is generally considered to be good programming practice to provide function
prototypes for *all* user-defined functions accessed in a program, whether or not
they are strictly required by the compiler.[8] The reason for this is fairly simple.

---

[8]Function prototypes are a requirement for all user-defined functions in C++ programs.

If we provide a prototype for a given function then the compiler can carefully compare each use of the function, within the program, with this prototype so as to determine whether or not we are calling the function properly. In the absence of a prototype, an incorrect call to a function (*e.g.*, using the wrong number of arguments, or arguments of the wrong data type) can give rise to run-time errors which are difficult to diagnose.

When a single value is passed to a function as an argument then the value of that argument is simply *copied* to the function. Thus, the argument's value can subsequently be altered within the function but this *will not* affect its value in the calling routine. This procedure for passing the value of an argument to a function is called *passing by value*.

Passing an argument by value has advantages and disadvantages. The advantages are that it allows a single-valued argument to be written as an expression, rather than being restricted to a single variable. Furthermore, in cases where the argument is a variable, the value of this variable is *protected* from alterations which take place within the function. The main disadvantage is that information cannot be transferred back to the calling portion of the program via arguments. In other words, passing by value is a strictly *one-way* method of transferring information. The program listed below, which is another modified version of `printfact.c`, illustrates this point:

```
/* printfact2.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>
#include <stdlib.h>

/* Prototype for function factorial() */
double factorial(int);

int main()
{
  int j;

  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
```

53

```
   printf("j = %3d    factorial(j) = %12.3e\n", j, factorial(j));

  return 0;
}


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

double factorial(int n)
{
  /*
     Function to evaluate factorial (in floating-point form)
     of non-negative integer n.
  */

  double fact = 1.;

  /* Abort if n is negative integer */
  if (n < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (; n > 0; --n) fact *= (double) n;

  /* Return value of factorial */
  return fact;
}
```

Note that the function `factorial()` has been modified such that its integer argument n is also used as the index in a `for` statement. Thus, the value of this argument is *modified* within the function. Nevertheless, the output of the above program is identical to that from `printfact.c`, since the modifications to n are *not* passed back to the main part of the program. Note, incidentally, the use of a null initialization expression in the `for` statement appearing in `factorial()`.

2.10   Pointers

One of the main characteristics of a scientific program is that *large amounts* of numerical information are exchanged between the various functions which make up the program. It is generally most convenient to pass this information via the *argument lists*, rather than the *names*, of the these functions. After all, only one number can be passed via a function name, whereas scientific programs generally require far more than one number to be passed during a function call. Hence, the functions employed in scientific programs generally return no values via their names (*i.e.*, they tend to be of data type `void`) but possess large strings of arguments. There is one obvious problem with this approach. Namely, a `void` function which passes all of its arguments by value is incapable of returning any information to the program segment from which it was called. Fortunately, there is a way of getting around this difficulty: we can pass the arguments of a function by *reference*, rather than by *value*, using constructs known as *pointers*. This allows the *two-way* communication of information via arguments during function calls. Pointers are discussed in the following.

Suppose that `v` is a variable in a C program which represents some particular data item. Of course, the program stores this data item at some particular location in the computer's memory. The data item can thus be accessed if we know its location, or *address*, in computer memory. The address of `v`'s memory location is determined by the expression `&v`, where `&` is a unary operator known as the *address operator*.

Suppose that we assign the address of `v` to another variable `pv`. In other words,

```
pv = &v
```

This new variable is called a *pointer* to `v`, since it points to the location where `v` is stored in memory. Remember, however, that `pv` represents `v`'s address, and *not* its value.

The data item represented by `v` (*i.e.*, the data item stored at `v`'s memory location) can be accessed via the expression `*pv`, where `*` is a unary operator, called the *indirection operator*, which only operates on pointer variables. Thus, `*pv`

and v both represent the same data item. Furthermore, if we write pv = &v and
u = *pv then both u and v represent the same value.

The simple program listed below illustrates some of the points made above:

```
/* pointer.c */
/*
   Simple illustration of the action of pointers
*/

#include <stdio.h>

main()
{
  int u = 5;
  int v;
  int *pu;        // Declare pointer to an integer variable
  int *pv;        // Declare pointer to an integer variable

  pu = &u;        // Assign address of u to pu
  v = *pu;        // Assign value of u to v
  pv = &v;        // Assign address of v to pv

  printf("\nu = %d  &u = %X  pu = %X  *pu = %d", u, &u, pu, *pu);
  printf("\nv = %d  &v = %X  pv = %X  *pv = %d\n", v, &v, pv, *pv);

  return 0;
}
```

Note that pu is a pointer to u, whereas pv is a pointer to v. Incidentally, the con-
version character X, which appears in the control strings of the above printf()
function calls, indicates that the associated data item should be output as a *hex-
adecimal number*—this is the conventional method of representing an address in
computer memory. Execution of the above program yields the following output:

```
u = 5  &u = BFFFFA24  pu = BFFFFA24  *pu = 5
v = 5  &v = BFFFFA20  pv = BFFFFA20  *pv = 5
%
```

In the first line, we see that u represents the value 5, as specified in its declaration
statement. The address of u is determined automatically by the compiler to be
BFFFFA24 (hexadecimal). The pointer pu is assigned this value. Finally, the value

to which `pu` points is 5, as expected. Similarly, the second line shows that `v` also represents the value 5. This is as expected, since we assigned the value `*pu` to `v`. The address of `v` is BFFFFA20. Of course, `u` and `v` have different addresses.

The unary operators `&` and `*` are members of the same precedence group as the other unary operators (*e.g.*, `++` and `--`). The address operator (`&`) can only act upon operands which possess a unique address, such as ordinary variables. Thus, the address operator *cannot* act upon arithmetic expression, such as `2 * (u + v)`. The indirection operator (`*`) can only act upon operands which are pointers.

Pointer variables, like all other variables, must be declared before they can appear in executable statements. A pointer declaration takes the general form

```
data-type  *ptvar;
```

where `ptvar` is the name of the pointer variable, and `data-type` is the data type of the data item towards which the pointer points. Note that an asterisk must always precede the name of a pointer variable in a pointer declaration.

Referring to Sect. 2.6, we can now appreciate that the mysterious asterisk which appears in the declaration of an input/output stream, *e.g.*,

```
FILE  *stream;
```

is there because `stream` is a pointer variable (pointing towards an object of the special data type `FILE`). In fact, `stream` points towards the beginning of the associated input/output stream in memory.

Pointers are often passed to a function as arguments. This allows data items within the calling part of the program to be accessed by the function, altered within the function, and then passed back to the calling portion of the program in altered form. This use of pointers is referred to as passing arguments by *reference*, rather than by value.

When an argument is passed by value, the associated data item is simply copied to the function. Thus, any alteration to the data item within the function is not passed back to the calling routine. When an argument is passed by

reference, however, the *address* of the associated data item is passed to the function. The contents of this address can be freely accessed by both the function and the calling routine. Furthermore, any changes made to the data item stored at this address are recognized by both the function and the calling routine. Thus, the use of a pointer as an argument allows the *two-way* communication of information between a function and its calling routine.

The program listed below, which is yet another modified version of `printfact.c`, uses a pointer to pass back information from a function to its calling routine:

```
/* printfact3.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>
#include <stdlib.h>

/* Prototype for function factorial() */
void factorial(int, double *);

int main()
{
  int j;
  double fact;

  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
   {
    factorial(j, &fact);
    printf("j = %3d    factorial(j) = %12.3e\n", j, fact);
   }
  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void factorial(int n, double *fact)
{
  /*
     Function to evaluate factorial *fact (in floating-point form)
     of non-negative integer n.
```

```
  */

  *fact = 1.;

  /* Abort if n is negative integer */
  if (n < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (; n > 0; --n) *fact *= (double) n;

  return;
}
```

The output from this program is again identical to that from `printfact.c`. Note that the function `factorial()` has been modified such that there is no data item associated with its name (*i.e.*, the function is of data type `void`). However, the argument list of this function has been extended such that there are now *two* arguments. As before, the first argument is the value of the positive integer `n` whose factorial is to be evaluated by the function. The second argument, `fact`, is a *pointer* which passes back the factorial of `n` (in the form of a floating-point number) to the main part of the program. Incidentally, the compiler knows that `fact` is a pointer because its name is proceeded by an *asterisk* in the argument declaration for `factorial()`. Of course, in the body of the function, reference is made to `*fact` (*i.e.*, the value of the data item stored in the memory location towards which `fact` points) rather than `fact` (*i.e.*, the address of the memory location towards which `fact` points). Note that a `void` function, which returns no value, can only be called via a statement consisting of the function name followed by a list of its arguments (in parentheses and separated by commas). Thus, the function `factorial()` is called in the main part of the program via the statement

```
factorial(j, &fact);
```

This statement passes the integer value `j` to `factorial()`, which, in turn, passes back the value of the factorial of `j` via its second argument. Note that since the second argument is passed by reference, rather than by value, it is written `&fact` (*i.e.*, the address of the memory location where the floating-point value `fact` is stored) rather than `fact` (*i.e.*, the value of the floating-point variable `fact`). Note, finally, that the function prototype for `factorial()` takes the form

```
void factorial(int, double *);
```

Here, the *asterisk* after `double` indicates that the second argument is a *pointer* to a floating-point data item.

We can now appreciate that the mysterious ampersands which must precede variable names in `scanf()` calls: *e.g.,*

```
scanf("%d %lf %lf", &k, &x, &y);
```

are not so mysterious, after all. `scanf()` is a function which returns data to its calling routine via its arguments (excluding its first argument, which is a control string). Hence, these arguments must be passed to `scanf()` by *reference*, rather than by *value*, otherwise they would be unable to pass information back to the calling routine. It follows that we must pass the *addresses* of variables (*e.g.,* &k) to `scanf()`, rather than the *values* of these variables (*e.g.,* k). Note that since the `printf()` function does not return any information to its calling routine via its arguments, there is no need to pass these arguments by reference—passing by value is fine. This explains why there are no ampersands in the argument list of a `printf()` function.

A pointer to a function can be passed to another function as an argument. This allows one function to be transferred to another, as though the first function were a variable. This is very useful in scientific programming. Imagine that we have a routine which numerically integrates a general one-dimensional function. Ideally, we would like to use this routine to integrate more than one specific function. We can achieve this by passing (to the routine) the name of the function to be integrated as an argument. Thus, for example, we can use the *same* routine to integrate a polynomial, a trigonometric function, or a logarithmic function.

Let us refer to the function whose name is passed as an argument as the *guest function*. Likewise, the function to which this name is passed is called the *host function*. A pointer to a guest function is identified in the host function definition by an entry of the form

```
data-type  (*function-name)(type 1,  type 2, ...)
```

in the host function's argument declaration.[9] Here, `data-type` is the data type
of the guest function, `function-name` is the local name of the guest function in
the host function definition, and `type 1`, `type 2`, `...` are the data types of the
guest function's arguments. The pointer to the guest function also requires an
entry of the form

```
data-type  (*)(type 1,  type 2,  ...)
```

in the argument declaration of the host function's prototype. The guest function
can be accessed within the host function definition by means of the indirection
operator. To achieve this, the indirection operator must precede the guest func-
tion name, and both the indirection operator and the guest function name must
be enclosed in parenthesis: *i.e.,*

```
(*function-name)(arg 1,  arg 2,  ...)
```

Here, `arg 1`, `arg 2`,`...` are the arguments passed to the guest function. Finally,
the name of a guest function is passed to the host function, during a call to the
latter function, via an entry like

```
function-name
```

in the host function's argument list.

    The program listed below is a rather silly example which illustrates the passing
of function names as arguments to another function:

```
/* passfunction.c */
/*
  Program to illustrate the passing of function names as
  arguments to other functions via pointers
*/

#include <stdio.h>

/* Function prototype for host fun. */
void cube(double (*)(double), double, double *);
```

---

[9]The parenthesis around `*function-name` are very important:
`data-type *function-name(type 1, type 2, ...)` is interpreted by the compiler as a reference to a function
which returns a pointer to type `data-type`, rather than a pointer to a function which returns type `data-type`.

```c
double fun1(double);     // Function prototype for first guest function
double fun2(double);     // Function prototype for second guest function

int main()
{
  double x, res1, res2;

  /* Input value of x */
  printf("\nx = ");
  scanf("%lf", &x);

  /* Evaluate cube of value of first guest function at x */
  cube(fun1, x, &res1);

  /* Evaluate cube of value of second guest function at x */
  cube(fun2, x, &res2);

  /* Output results */
  printf("\nx = %8.4f   res1 = %8.4f   res2 = %8.4f\n", x, res1, res2);

  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void cube(double (*fun)(double), double x, double *result)
{
  /*
    Host function: accepts name of floating-point guest function
    with single floating-point argument as its first argument,
    evaluates this function at x (the value of its second argument),
    cubes the result, and returns final result via its third argument.
  */

  double y;

  y = (*fun)(x);          // Evaluate guest function at x
  *result = y * y * y;    // Cube value of guest function at x

  return;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
double fun1(double z)
{
  /*
    First guest function
  */

  return 3.0 * z * z - z;
}


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

double fun2(double z)
{
  /*
    Second guest function
  */

  return 4.0 * z  - 5.0 * z * z * z;
}
```

In the above program, the function `cube()` accepts the name of a guest function
(with one argument) as its first argument, evaluates this function at x (the value
of its second argument, which is ultimately specified by the user), cubes the re-
sult, and then passes the final result back to the main part of the program via its
third argument (which, of course, is a pointer). The two guest functions, `fun1()`
and `fun2()`, whose names are passed to `cube()`, are both simple polynomials.
The output from the above program looks like:

```
x = 2

x =   2.0000   res1 = 1000.0000   res2 = -32768.0000
%
```

2.11   Global Variables

We have seen that a general C program is basically a collection of functions. Fur-
thermore, the variables used by these functions are *local* in scope: *i.e.,* a variable
defined in one function is not recognized in another. The main method of trans-
ferring data from one function to another is via the argument lists in function

calls. Arguments can be passed in one of two different manners. When an argument is passed by *value* then the value of a local variable (or expression) in the calling routine is copied to a local variable in the function which is called. When an argument is passed by *reference* then a local variable in the calling routine shares the same memory location as a local variable in the function which is called: hence, a change in one variable is automatically mirrored in the other. However, there is a *third* method of transferring information from one function to another. It is possible to define variables which are *global* in extent: such variables are recognized by *all* the functions making up the program, and have the *same* values in all of these functions.

The C compiler recognizes a variable as global, as opposed to local, because its declaration is located *outside* the scope of any of the functions making up the program. Of course, a global variable can only be used in an executable statement *after* it has been declared. Hence, the natural place to put global variable declaration statements is *before* any function definitions: *i.e.*, right at the beginning of the program. Global variables declarations can be used to initialize such variables, in the regular manner. However, the initial values must be expressed as constants, rather than expressions. Furthermore, the initial values are only assigned once, at the beginning of the program.

The program listed below is yet another version of `printfact.c`, in which communication between the two sections of the program takes place entirely via global variables:

```
/* printfact4.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>
#include <stdlib.h>

/* Prototype for function factorial() */
void factorial();

/* Global variable declarations */
int j;
```

```
double fact;

int main()
{
  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
   {
    factorial();
    printf("j = %3d    factorial(j) = %12.3e\n", j, fact);
   }
  return 0;
}


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void factorial()
{
  /*
     Function to evaluate factorial (in floating-point form)
     of non-negative integer j. Result stored in variable fact.
  */

  int count;

  /* Abort if j is negative integer */
  if (j < 0)
   {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
   }

  /* Calculate factorial */
  for (count = j, fact = 1.; count > 0; --count) fact *= (double) count;

  return;
}
```

The output from the above program is identical to that from `printfact.c`. Observe that the global variable `j` is used to pass the integer value whose factorial is to be calculated from the main part of the program to the function `factorial()`, whilst the global variable `fact` is used to pass the calculated factorial back to the main part of the program. Incidentally, note the use of multiple initialization statements (separated by commas) in the `for` statement appearing in

`factorial()`.

Global variables should be used *sparingly* in scientific programs (or any other type of program), since there are inherent dangers in their employment. An alteration in the value of a global variable within a given function is carried over into all other parts of the program. Unfortunately, such an alteration can sometimes happen inadvertently, as the side-effect of some other action. Thus, there is the possibility of the value of a global variable changing unexpectedly, resulting in a subtle programming error which can be *extremely difficult* to track down, since the offending line could be located *anywhere* in the program. Similar errors involving local variables are much easier to debug, since the scope of local variables is far more limited than that of global variables.

2.12 Arrays

Scientific programs very often deal with multiple data items possessing common characteristics. In such cases, it is often convenient to place the data items in question into an *array,* so that they all share a common name (*e.g.,* x). The individual data items can be either integers or floating-point numbers. However, they all must be of the *same* data type.

In C, an element of an array (*i.e.,* an individual data item) is referred to by specifying the array name followed by one or more *subscripts,* with each subscript enclosed in square brackets. All subscripts must be nonnegative integers. Thus, in an n-element array called x, the array elements are `x[0]`, `x[1]`, `...`, `x[n-1]`. Note that the first element of the array is `x[0]` and *not* `x[1]`, as in other programming languages.

The number of subscripts determines the *dimensionality* of an array. For example, `x[i]` refers to an element of a one-dimensional array, x. Similarly, `y[i][j]` refers to an element of a two-dimensional array, y, *etc.*

Arrays are declared in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification (which specifies the number of elements). For a one-dimensional array, the size is specified by

a positive integer constant, enclosed in square brackets. The generalization for multi-dimensional arrays is fairly obvious. Several valid array declarations are shown below:

```
int j[100];
double x[20];
double y[10][20];
```

Thus, j is a 100-element integer array, x is a 20-element floating point array, and y is a 10x20 floating-point array. Note that variable size array declarations, *e.g.*,

```
double a[n];
```

where n is an integer variable, are *illegal* in C.

It is sometimes convenient to define an array size in terms of a symbolic constant, rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size can be altered by simply changing the value of the symbolic constant. This approach is used in many of the example programs employed in this course.

Like an ordinary variable, an array can be either *local* or *global* in extent, depending on whether the associated array declaration lies inside or outside, respectively, the scope of any of the functions which constitute the program. Both local and global arrays can be initialized via their declaration statements.[10] For instance,

```
int j[5] = {1, 3, 5, 7, 9};
```

declares j to be a 5-element integer array whose elements have the initial values j[0]=1, j[1]=3, *etc.*

Single operations which involve entire arrays are *not* permitted in C. Thus, if x and y are similar arrays (*i.e.*, the same data type, dimensionality, and size) then assignment operations, comparison operations, *etc.* involving these two arrays

---

[10]Prior to the ANSI standard, to which the GNU C compiler adheres, local arrays could not be initialized via their declaration statements.

must be carried out on an element by element basis. This is usually accomplished within a loop (or within nested loops, for multi-dimensional arrays).

The program listed below is a simple illustration of the use of arrays in C. The program reads a list of numbers, entered by the user, into a one-dimensional array, `list`, and then calculates the average of these numbers. The program also calculates and outputs the deviation of each number from the average.

```c
/* average.c */
/*
  Program to calculate the average of n numbers and then
  compute the deviation of each number from the average

  Code adapted from "Programming with C", 2nd Edition, Byron Gottfreid,
  Schaum's Outline Series, (McGraw-Hill, New York NY, 1996)
*/

#include <stdio.h>
#include <stdlib.h>

#define NMAX 100

int main()
{
  int n, count;
  double avg, d, sum = 0.;
  double list[NMAX];

  /* Read in value for n */
  printf("\nHow many numbers do you want to average? ");
  scanf("%d", &n);

  /* Check that n is not too large or too small */
  if ((n > NMAX) || (n <= 0))
   {
    printf("\nError: invalid value for n\n");
    exit(1);
   }

  /* Read in the numbers and calculate their sum */
  for (count = 0; count < n; ++count)
   {
    printf("i = %d  x = ", count + 1);
    scanf("%lf", &list[count]);
```

```
     sum += list[count];
    }

  /* Calculate and display the average */
  avg = sum / (double) n;
  printf("\nThe average is %5.2f\n\n", avg);

  /* Calculate and display the deviations about the average */
  for (count = 0; count < n; ++count)
    {
     d = list[count] - avg;
     printf("i = %d  x = %5.2f  d = %5.2f\n", count + 1, list[count], d);
    }
  return 0;
}
```

Note the use of the symbolic constant `NMAX` to specify the size of the array `list`, and, hence, the maximum number of values which can be averaged. The typical output from the above program looks like:

```
How many numbers do you want to average? 5
i = 1   x = 4.6
i = 2   x = -2.3
i = 3   x = 8.7
i = 4   x = 0.12
i = 5   x = -2.7

The average is   1.68

i = 1   x =    4.60   d =   2.92
i = 2   x =  -2.30   d = -3.98
i = 3   x =   8.70   d =   7.02
i = 4   x =   0.12   d = -1.56
i = 5   x =  -2.70   d = -4.38
%
```

It is important to realize that an array name in C is essentially a *pointer* to the first element in that array.[11] Thus, if x is a one-dimensional array then the address of the first array element can be expressed as either `&x[0]` or simply x. Moreover,

---

[11]An array name is not exactly equivalent to a pointer, because a pointer can point to any address in the computer memory—this address can even be changed—whereas an array name is constrained to always point towards the address of its associated first data item.

the address of the second array element can be written as either `&x[1]` or `(x+1)`. In general, the address of the `(i+1)`th array element can be expressed as either `&x[i]` or `(x+i)`. Incidentally, it should be understood that `(x+i)` is a rather special type of expression, since `x` represents an address, whereas `i` represents an integer quantity. The expression `(x+i)` actually specifies the address of the array element which is `i` memory locations offset from the address of the first array element (C, of course, stores all elements of an array both contiguously and in order in the computer memory). Hence, `(x+i)` is a *symbolic representation* of an address, rather than an arithmetic expression.

Since `&x[i]` and `(x+i)` both represent the address of the `(i+1)`th element of the array `x`, it follows that `x[i]` and `*(x+i)` must both represent the contents of that address (*i.e.*, the value of the `(i+1)`th element). In fact, the latter two terms are *completely interchangeable* in C programs.

For the moment, let us concentrate on one-dimensional arrays. An entire array can be passed to a function as an argument. To achieve this, the array name must appear by itself, without brackets or subscripts, as an argument within the function call. The corresponding argument in the function definition must be declared as an array. In order to do this, the array name is written followed by an empty pair of square brackets. The size of the array is not specified. In a function prototype, an array argument is specified by following the data type of the argument by an empty pair of square brackets.

Since, as we have seen, an array name is essentially a pointer, it is clear that when an array is passed to a function it is passed by *reference*, and not by *value*. Hence, if any of the array elements are altered within the function then these alterations *are* recognized in the calling portion of the program. Likewise, if an array (rather than an individual array element) appears in the argument list of a `scanf()` function then it should *not* be preceded by the address operator (`&`), since an array name already is an address. The reason why arrays in C are always passed by reference is fairly obvious. In order to pass an array by value, it is necessary to copy the value of *every* element. On the other hand, to pass an array by reference it is only necessary to pass the address of the first element. Clearly, for large arrays, passing by reference is far more efficient than passing by

value.

The program listed below is yet another version of `printfact.c`, albeit a far more efficient one than any of those listed previously. In this version, the factorials of all the non-zero integers up to 20 are calculated in one fell swoop, by the function `factorial()`, using the recursion relation

$$(n+1)! = (n+1)\,n!\tag{2.1}$$

The factorials are stored as elements of the array `fact[]`, which is passed as an argument from `factorial()` to the main part of the program.

```c
/* printfact5.c */
/*
  Program to print factorials of all integers
  between 0 and 20
*/

#include <stdio.h>

/* Function prototype for factorial() */
void factorial(double []);

int main()
{
  int j;
  double fact[21];            // Declaration of array fact[]

  /* Calculate factorials */
  factorial(fact);

  /* Output results */
  for (j = 0; j <= 20; ++j)
    printf("j = %3d    factorial(j) = %12.3e\n", j, fact[j]);

  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void factorial(double fact[])
{
  /*
    Function to calculate factorials of all integers
```

```
      between 0 and 20 (in form of floating-point
      numbers) via recursion formula

      (n+1)! = (n+1) n!

      Factorials returned in array fact[0..20]
   */

   int count;

   fact[0] = 1.;                    // Set 0! = 1

   /* Calculate 1! through 20! via recursion */
   for (count = 0; count < 20; ++count)
      fact[count+1] = (double)(count+1) * fact[count];

   return;
}
```

The output from the above program is identical to that from `printfact.c`.

It is important to realize that there is *no array bound checking* in C. If an array
x is declared to have 100 elements then the compiler will reserve 100 contigu-
ous, appropriately sized, slots in computer memory on its behalf. The contents
of these slots can be accessed via expressions of the form `x[i]`, where the inte-
ger i should lie in the range 0 to 99. As we have seen, the compiler interprets
`x[i]` to mean the contents of the memory slot which is i slots along from the
beginning of the array. Unfortunately, expressions such as `x[100]` or `x[1000]` are
interpreted in a like manner, leading the compiler to instruct the executable to
access memory slots which lie off the end of the memory block reserved for x.
Obviously, accessing elements of an array which do not exist is going to produce
some sort of error. Exactly what sort of error is very difficult to say—the program
may crash, it may produce absurdly incorrect output, it may produce plausible
but incorrect output, it may even produce correct output—it all depends on ex-
actly what information is being stored in the memory locations surrounding the
block of memory reserved for x. This type of error can be *extremely difficult* to
debug, since it may not be immediately apparent that something has gone wrong
when the program is executed. It is, therefore, the programmer's responsibility
to ensure that *all* references to array elements lie within the declared bounds of

the associated arrays.

Let us now discuss multi-dimensional arrays in more detail. The elements of a multi-dimensional array are stored contiguously in a block of computer memory. In scanning across this block, from its start to its end, the order of storage is such than the last subscript of the array varies *most rapidly* whilst the first varies *least rapidly*. For instance, the elements of the two-dimensional array x[2][2] are stored in the order: x[0][0], x[0][1], x[1][0], x[1][1]. The elements of a multi-dimensional array can only be addressed if the program is explicitly told the size of the array in its second, third, *etc.* dimensions. It is, therefore, not surprising to learn that when a multi-dimensional array is passed to a function, as an argument, then the associated argument declaration within the function definition *must* include *explicit size declarations* in all of the subscript positions *except the first*. The same is true for a multi-dimensional array argument appearing in a function prototype.

2.13   Character Strings

The basic C character data type is called char. A character string in C can be represented symbolically as an array of data type char. For instance, the declaration

```
char word[20] = "four";
```

initializes the 20-element character array word, and then stores the character string "four" in it. The resulting elements of word are as follows:

```
word[0] = 'f'   word[1] = 'o'   word[2] = 'u'   word[3] =  'r'   word[4] = '\0'
```

with the remaining elements undefined. Here, 'f' represents the character "f", *etc.*, and '\0' represents the so-called *null character* (ASCII code 0), which is used in C to signal the termination of a character string. The null character is automatically added to the end of any character string enclosed in double quotes. Note that, since all character strings in C must be terminated by the (invisible) null character, it takes a character array of size at least n+1 to store an n-letter string.

As with arrays of numbers, the name of a character array is essentially equivalent to a pointer to the first element of that array. Hence, `word[i]` and `*(word + i)` both refer to the same character in the character array `word`. Note, however, that the name of a character array is not a true pointer, since the address to which it points cannot be changed. Of course, we can always represent a character array using a true pointer. Consider the declaration

```
char *word = "four";
```

Here, `word` is declared to be a pointer to a `char` which points towards the first element of the character string `'f' 'o' 'u' 'r' '\0'`. Unlike the name of a character array, a true pointer to a `char` can be redirected. Thus,

```
char *word =  "four";
. . .
word = "five";
```

is legal, whereas

```
char word[20] =  "four";
. . .
word = "five";
```

is illegal. Note that, in the former example, the addresses of the first elements of the strings "four" and "five" are probably different. Of course, the contents of a character array can always be changed, element by element—it is just the address of the first element which must remain constant. Thus,

```
char word[20] =  "four";
. . .
word[0] = 'f';
word[1] = 'i';
word[2] = 'v';
word[3] = 'e';
word[4] = '\0';
```

is perfectly legal.

Note, finally, that a character string can be printed via the `printf()` function by making use of a `%s` entry in its control string: *e.g.,*

```
printf("word = %s\n", word);
```

Here, the second argument, `word`, can either be the name of a character array or a true pointer to the first element of a character string.

2.14   Multi-File Programs

In a program consisting of many different functions, it is often convenient to place each function in an individual file, and then use the `make` utility to compile each file separately and link them together to produce an executable.

There are a few common-sense rules associated with multi-file programs. Since a given file is initially compiled *separately* from the rest of the program, all symbolic constants which appear in that file must be defined at its start. Likewise, all referenced library functions must be accompanied by the appropriate references to header files. Also, any referenced user-defined functions must have their prototypes at the start of the file. Finally, all global variables used in the file must be declared at its start. This usually means that definitions for common symbolic constants, header files for common library functions, prototypes for common user-defined functions, and declarations for common global variables will appear in *multiple* files. Note that a given global variable can only be initialized in *one* of its declaration statements, which is regarded as the *true declaration* of that variable [conventionally, the true declaration appears in the file containing the function `main()`]. Indeed, the other declarations, which we shall term *definitions*, must be preceded by the keyword `extern` to distinguish them from the true declaration.

As an example, let us take the program `printfact4.c`, listed previously, and break it up into multiple files, each containing a single function. The files in question are called `main.c` and `factorial.c`. The listings of the two files which make up the program are as follows:

```
/* main.c */
/*
  Program to print factorials of all integers
```

```
  between 0 and 20
*/

#include <stdio.h>

/* Prototype for fucntion factorial() */
void factorial();

/* Global variable declarations */
int j;
double fact;

int main()
{
  /* Print factorials of all integers between 0 and 20 */
  for (j = 0; j <= 20; ++j)
    {
      factorial();
      printf("j = %3d    factorial(j) = %12.3e\n", j, fact);
    }
  return 0;
}
```

and

```
/* factorial.c */
/*
   Function to evaluate factorial (in floating point form)
   of non-negative integer j. Result stored in variable fact.
*/

#include <stdio.h>
#include <stdlib.h>

/* Global variable definitions */
extern int j;
extern double fact;

void factorial()
{
  int count;

  /* Abort if j is negative integer */
  if (j < 0)
```

```
  {
    printf("\nError: factorial of negative integer not defined\n");
    exit(1);
  }

/* Calculate factorial */
for (count = j, fact = 1.; count > 0; --count) fact *= (double) count;

return;
}
```

Note that all library functions and user-defined functions referenced in each file
are declared (either via a header file or a function prototype) at the start of that
file. Note, also, the distinction between the global variable *declarations* in the file
`main.c` and the global variable *definitions* in the file `factorial.c`.

2.15   Command Line Parameters

The `main()` function may optionally possess special arguments which allow pa-
rameters to be passed to this function from the operating system. There are two
such arguments, which are conventionally called `argc` and `argv`. The former ar-
gument, `argc`, is an integer which is set to the number of parameters passed to
`main()`, whereas the latter argument, `argv`, is an array of pointers to character
strings which contain these parameters. In order to pass one or more parameters
to a C program when it is executed from the operating system, the parameters
must follow the program name on the command line: *e.g.,*

```
% program-name  parameter_1  parameter_2  parameter_3 ... parameter_n
```

The program name will be stored in the first item in `argv`, followed by each of
the parameters. Hence, if the program name is followed by n parameters there
will be n + 1 entries in `argv`, ranging from `argv[0]` to `argv[n]`. Furthermore,
`argc` will be automatically set equal to n + 1.

   The program listed below is a simple illustration of the use of command line
parameters: it simply echoes all of the parameters passed to it.

```
/* repeat.c */
```

```
/*
  Program to read and echo data from command line
*/

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

  for (i = 1; i < argc; i++) printf("%s ", argv[i]);
  printf("\n");

  return 0;
}
```

Assuming that the executable is called `repeat`, the typical output from this program is as follows:

```
% repeat The quick brown fox jumped over the lazy hounds
The quick brown fox jumped over the lazy hounds
%
```

Suppose that one or more of the parameters passed to a given program are *numbers*. As we have seen, these numbers are actually passed as character strings. Hence, before they can be employed in calculations, they must be converted into either type `int` or type `double`. This can be achieved via the use of the functions `atoi()` and `atof()` (the appropriate header file for these functions is `stdlib.h`). Thus, `int atoi(char *ptr)` converts a string pointed to by `ptr` into an `int`, whereas `double atof(char *ptr)` converts a string pointed to by `ptr` into an `double`. The program listed below illustrates the use of the `atof()` function: it reads in a number passed as a command line parameter, interprets it as a temperature in degrees Fahrenheit, converts it to degrees Celsius, and then prints out the result.

```
/* ftoc.c */
/*
  Program to convert temperature in Fahrenheit input
  on command line to temperature in Celsius
*/

#include <stdlib.h>
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
  double deg_f, deg_c;

  /* If no parameter passed to program print error
     message and exit */
  if (argc < 2)
    {
      printf("Usage: ftoc temperature\n");
      exit(1);
    }

  /* Convert first command line parameter to double */
  deg_f = atof(argv[1]);
  /* Convert from Fahrenheit to Celsius */
  deg_c = (5. / 9.) * (deg_f - 32.);

  printf("%f degrees Fahrenheit equals %f degrees Celsius\n",
         deg_f, deg_c);

  return 0;
}
```

Assuming that the executable is called `ftoc`, the typical output from this program is as follows:

```
% ftoc 98
98.000000 degrees Fahrenheit equals 36.666667 degrees Celsius
%
```

2.16   Timing

The header file `time.h` defines a number of library functions which can be used to assess how much CPU time a C program consumes during execution. The simplest such function is called `clock()`. A call to this function, with no argument, will return the amount of CPU time used so far by the calling program. The time is returned in a special data type, `clock_t`, defined in `time.h`. This time must be divided by `CLOCKS_PER_SEC`, also defined in `time.h`, in order to covert it into

seconds. The ability to measure how much CPU time a given code consumes is useful in scientific programming: *e.g.*, because it allows the effectiveness of the various available compiler optimization flags to be determined. Optimization usually (but not always!) speeds up the execution of a program. However, over aggressive optimization can often slow a program down again.

The program listed below illustrates the simple use of the clock() function. The program compares the CPU time required to raise a double to the fourth power via a direct calculation and via a call to the pow() function. Actually, both operations are performed a million times and the elapsed CPU time is then divided by a million.

```c
/* timing.c */
/*
  Program to test operation of clock() function
*/

#include <time.h>
#include <math.h>
#define N_LOOP 1000000

int main()
{
  int i;
  double a = 11234567890123456.0, b;
  clock_t time_1, time_2;

  time_1 = clock();
  for (i = 0; i < N_LOOP; i++) b = a * a * a * a;
  time_2 = clock();
  printf ("CPU time needed to evaluate a*a*a*a:    %f microsecs\n",
          (double) (time_2 - time_1) / (double) CLOCKS_PER_SEC);

  time_1 = clock();
  for (i = 0; i < N_LOOP; i++) b = pow(a, 4.);
  time_2 = clock();
  printf ("CPU time needed to evaluate pow(a, 4.): %f microsecs\n",
          (double) (time_2 - time_1) / (double) CLOCKS_PER_SEC);

  return 0;
}
```

The typical output from this program is as follows:

```
CPU time needed to evaluate a*a*a*a:    0.190000 microsecs
CPU time needed to evaluate pow(a, 4.): 1.150000 microsecs
%
```

Clearly, evaluating a fourth power using the `pow()` function is a lot more expensive than the direct calculation. Hence, as has already been mentioned, the `pow()` function should *not* be used to raise floating point quantities to small integer powers.

2.17   Random Numbers

A large class of scientific calculations (*e.g.*, so-called Monte Carlo calculations) require the use of random variables. A call to the `rand()` function (header file `stdlib.h`), with no arguments, returns a fairly good approximation to a random integer in the range 0 to `RAND_MAX` (defined in `stdlib.h`). The `srand()` function sets its argument, which is of type `int`, as the seed for a new sequence of numbers to be returned by `rand()`. These sequences are *repeatable* by calling `srand()` with the same seed value. If no seed value is provided, the `rand()` function is automatically seeded with the value 1. It is common practice in C programming to seed the random number generator with the number of seconds elapsed since 00:00:00 UTC, January 1st, 1970. This number is returned, as an integer, via a call to the `time (NULL)` function (header file: `<time.h>`). Seeding the generator in this manner ensures that a different set of random numbers is generated automatically each time the program is run.

The program listed below illustrates the use of the `rand()` function to construct a pseudo-random variable, x, which is uniformly distributed in the range 0 to 1. The program calculates 10^7 values of x, and then evaluates the mean and variance of these values.

```
/* random.c */
/*
  Program to test operation of rand() function
*/

#include <stdlib.h>
```

81

```c
#include <stdio.h>
#include <time.h>
#define N_MAX 10000000

int main()
{
  int i, seed;
  double sum_0, sum_1, mean, var, x;

  /* Seed random number generator */
  seed = time(NULL);
  srand(seed);

  /* Calculate mean and variance of x: random number uniformly
     distributed in range 0 to 1 */
  for (i = 1, sum_0 = 0., sum_1 = 0.; i <= N_MAX; i++)
    {
      x = (double) rand() / (double) RAND_MAX;

      sum_0 += x;
      sum_1 += (x - 0.5) * (x - 0.5);
    }
  mean = sum_0 / (double) N_MAX;
  var = sum_1 / (double) N_MAX;

  printf("mean(x) = %12.10f  var(x) = %12.10f\n", mean, var);

  return 0;
}
```

The typical output from this program is as follows:

```
mean(x) = 0.5000335261   var(x) = 0.0833193874
%
```

As is easily demonstrated, the theoretical mean and variance of x are 1/2 and 1/12, respectively. It can be seen that the values returned by the program agree with these theoretical values to five decimal places, which is all that can be expected with only 10^7 calls.

2.18   C++ Extensions to C

In this subsection, we shall briefly discuss some of the useful, non-object-orientated extensions to C introduced in the C++ language. Files containing source code which incorporates C++ elements should be distinguished from files containing plain C code via the extension `.cpp`.

In C, all local variables within a function must be declared *before* the first executable statement. In C++, this restriction is relaxed: a local variable can be declared (almost) *anywhere* in a function, provided that this declaration occurs prior to the variable's first use. The following code snippet illustrates the use of this new feature:

```
. . .
for (int i = 0; i < MAX; i++)
 {
  . . .
 }
. . .
```

Observe that the index `i` of the `for` loop is now declared at the start of the loop, instead of at the start of the function containing the loop. This is far more convenient, and also makes the code easier to read (since we no longer have to skip back to the declarations at the start of the function to check that `i` is an `int`). Note, however, that the variable `i` is only defined over the extent of the loop (*i.e.*, between the curly braces). Any attempt to reference `i` outside the loop will result in a compilation error. In general, when a variable is declared in C++ its *scope* (*i.e.*, range of definition) extends from its declaration to the closing curly brace which terminates the current program block. Program blocks are functions, loops, conditionally executed compound statements, *etc.*, and are delineated by curly braces. There are a number of restrictions to this new method of variable declaration. Variables *cannot* be declared within conditional statements, in the second or third expressions of `for` loops, or in function calls.

In C, we have seen that in order to pass an argument to a function in such a manner that changes made to this argument within the function are passed back to the calling routine, we must actually pass a *pointer* to the argument. This procedure, which is called passing by *reference*, is illustrated in the code snippet

listed below:

```
. . .
void square(double, double *);
. . .
int main()
{
  . . .
  double arg, res;
  square(arg, &res);
  . . .
  return 0;
}
. . .
void square(double x, double *y)
{
  *y = x * x;
  return;
}
```

Here, the second argument to `square()` is returned to `main()` in modified form. This argument must, therefore, be passed as a pointer. Consequently, we must write `&res`, rather that `res`, when calling `square()`, and we must refer to the argument as `*y`, rather than `y`, in the function itself. After a while, all these ampersands and asterisks can become a little tedious! C++ introduces a new method of passing by reference which somewhat less involved. This new method is illustrated in the following:

```
. . .
void square(double, double &);
. . .
int main()
{
  . . .
  double arg, res;
  square(arg, res);
  . . .
  return 0;
}
. . .
void square(double x, double &y)
{
  y = x * x;
```

```
  return;
}
```

Here, the second argument to `square()` is again passed by reference. However, this is now indicated by prepending an ampersand to the variable name in the function declaration. A corresponding ampersand appears in the associated function prototype. Note that we do not need to explicitly pass a pointer to the second argument when calling `square()` (this is done behind the scenes): *i.e.*, we write `res`, rather than `&res`, when calling `square()`. Likewise, we do not have to explicitly deference the argument in the function itself (this is also done behind the scenes): *i.e.*, we refer to the argument by its regular local name, `y`, as opposed to `*y`, within the function.

Functions are used in C programs to avoid having to repeat the same block of code in many different places in the source. The use of functions also renders a code easier to read and maintain. However, there is a price to pay for the convenience of functions. When a regular function is called in an executable, the program jumps to the block of memory in which the compiled function code is stored, and then jumps back to its original position in memory space when the function returns. Unfortunately, the large jumps in memory space associated with a function call take up a non-negligible amount of CPU time. Indeed, the overhead associated with making function calls often discourages scientific programmers from writing small functions, even when it may be desirable to do so. C++ provides a way out of this dilemma, via the use of the new keyword `inline`. An *inline function* looks like a normal function when it is used, but is compiled in a different manner. Calling an inline function from several different locations in a code does not result in multiple calls to a single function. Instead, the code for the inline function is inserted into the program code by the compiler wherever the function is used.

Inline functions are only useful for *small* functions. The disadvantage of inserting the code for a large function multiple times into the code for a typical program easily outweighs the small gain in performance obtained by the elimination of standard function calls. The break-even point for inline functions is usually about *three* executable lines.

To inline a function, a programmer adds the keyword `inline` at the start of the function's definition. For example:

```
inline double square(double x)
{
  return x * x;
}
```

Because the body of an inline function must be known before the compiler can insert it into the program code, wherever the function is used, we must define such a function prior to its first use—a prototype declaration is not enough. It is common practice to define inline functions at the same location in source code files that the prototypes for regular (*i.e.*, outline) functions are placed.

Variable size array declarations of the form

```
void function(n)
{
  int n;
  double x[n];
  . . .
}
```

are *illegal* in C, which is extremely inconvenient. In C++, such declarations are enabled via the use of the new keywords `new` and `delete`. Thus, the C++ implementation of the above code snippet takes the form:

```
void function(n)
{
  . . .
  int n;
  double *x = new double[n];
  . . .
  x[i] = . . .
  . . .
  . . .
  delete x[];
  . . .
}
```

Note that `x` is actually declared as a pointer, rather than a standard array. The declaration `new double[n]` reserves a memory block which is just large enough to store `n` doubles, and then returns the address of the start of this block. The line `delete x[]` frees up the block of memory associated with the array `x` when it is no longer needed (note that this is not done automatically). Unfortunately, the `new` and `delete` keywords cannot be used to make variable size multi-dimensional arrays.

2.19   Complex Numbers

As we have already mentioned, the C language definition does not include complex arithmetic—presumably because the square root of minus one is not a concept which crops up very often in systems programming! Fortunately, this rather serious deficiency—at least, as far as the scientific programmer is concerned—is remedied in C++. The program listed below illustrates the use of the C++ complex class (header file `complex.h`) to perform complex arithmetic using doubles:

```
/* complex.cpp */
/*
  Program to test out C++ complex class
*/

#include <complex.h>
#include <stdio.h>

/* Define complex double data type */
typedef complex<double> dcomp;

int main()
{
  dcomp i, a, b, c, d, e, p, q, r; // Declare complex double variables
  double x, y;

  /* Set complex double variable equal to complex double constant */
  i = dcomp (0., 1.);
  printf("\ni = (%6.4f, %6.4f)\n", i);

  /* Test arithmetic operations with complex double variables */
  a = i * i;
  b = 1. / i;
```

```
  printf("\ni*i = (%6.4f, %6.4f)\n", a);
  printf("1/i = (%6.4f, %6.4f)\n", b);

  /* Test mathematical functions using complex double variables */
  c = sqrt(i);
  d = sin(i);
  e = pow(i, 0.25);
  printf("\nsqrt(i) = (%6.4f, %6.4f)\n", c);
  printf("sin(i) = (%6.4f, %6.4f)\n", d);
  printf("i^0.25 = (%6.4f, %6.4f)\n", e);

  /* Test complex operations */
  p = conj(i);
  q = real(i);
  r = imag(i);
  printf("\nconj(i) = (%6.4f, %6.4f)\n", p);
  printf("real(i) = %6.4f\n", q);
  printf("imag(i) = %6.4f\n", r);

  return 0;
}
```

The typical output from this program is as follows:

```
i = (0.0000, 1.0000)

i*i = (-1.0000, 0.0000)
1/i = (0.0000, -1.0000)

sqrt(i) = (0.7071, 0.7071)
sin(i) = (0.0000, 1.1752)
i^0.25 = (0.9239, 0.3827)

conj(i) = (0.0000, -1.0000)
real(i) = 0.0000
imag(i) = 1.0000
%
```

The program first of all defines the complex double type `dcomp`. Variables of this type are then declared, set equal to complex constants, employed in arithmetic expressions, used as the arguments of mathematical functions, *etc.*, in much the same manner that we would perform similar operations in C with variables of

type `double`. Note the special functions `conj()`, `real()`, and `imag()`, which take the complex conjugate of, find the real part of, and find the imaginary part of a complex variable, respectively.

2.20   Variable Size Multi-Dimensional Arrays

Multi-dimensional arrays crop up in a wide variety of different applications in scientific programming. Moreover, it is very common for the sizes of the arrays employed in a scientific code to vary from run to run, depending on the particular values of the input parameters. For instance, in a standard fluid code the array sizes depend on the number of grid points, which, in turn, depends on the requested accuracy. Thus, a crucial test of the suitability of a programming language for scientific purposes is its ability to deal with variable size, multi-dimensional arrays in a convenient manner. Unfortunately, C fails this test rather badly, since variable size matrix declarations of the form

```
void function(a, m, n)
{
  int m, n;
  double a[m][n];
  . . .
```

are *illegal* (unlike in FORTRAN 77, where variable size matrix declarations are perfectly acceptable). Indeed, this unfortunate (and quite unnecessary!) omission in the C language definition is often put forward as a reason why C should *not* be used for scientific purposes. Fortunately, we can get around the absence of variable size multi-dimensional arrays in C by making use of a freely available C++ package called the Blitz++ library—see `http://www.oonumerics.org/blitz/`.

The program listed below illustrates the use of the Blitz++ library. The program adds together two matrices whose dimensions and elements are input by the user, and then prints out the result.

```
/* addmatrix.c */
/*
  Program to add two variable dimension matrices input by user
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <blitz/array.h>

using namespace blitz;

/* Function prototypes */
void readin(Array<double,2>);
void writeout(Array<double,2>);
void addmatrices(Array<double,2>, Array<double,2>, Array<double,2>);

int main()
{

  int n, m;

  /* Input number of rows and columns */
  printf("\nPlease input number of rows, n, and number of columns, m: ");
  scanf("%d %d", &n, &m);

  /* Check that n, m are positive integers */
  if (n <= 0 || m <= 0)
    {
      printf("\nError: invalid values for n and/or m\n");
      exit(1);
    }

  /* Array declarations */
  Array<double,2> A(n, m), B(n, m), C(n, m);

  /* Read in elements of A, row by row */
  printf("\nReading in elements of A:\n");
  readin(A);

  /* Read in elements of B, row by row */
  printf("\nReading in elements of B:\n");
  readin(B);

  /* Write out elements of A, row by row */
  printf("\nWriting out elements of A:\n");
  writeout(A);

  /* Write out elements of B, row by row */
  printf("\n\nWriting out elements of B:\n");
  writeout(B);
```

```
  /* Add matrices A and B */
  addmatrices(A, B, C);

  /* Write out matrix C = A + B, row by row */
  printf("\n\nWriting out elements of C = A + B:\n");
  writeout(C);
  printf("\n");

  return 0;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
  Read in elements of matrix M, row by row
*/

void readin(Array<double,2> M)
{
  int n = M.extent(0);
  int m = M.extent(1);

  for (int i = 0; i < n; i++)
    {
      printf("\nRow %d: ", i + 1);
      for (int j = 0; j < m; j++) scanf("%lf", &M(i, j));
    }
  return;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
  Write out elements of matrix M, row by row
*/

void writeout(Array<double,2> M)
{
  int n = M.extent(0);
  int m = M.extent(1);

  for (int i = 0; i < n; i++)
    {
      printf("\nRow %d: ", i + 1);
```

```c
        for (int j = 0; j < m; j++) printf("%7.2f ", M(i, j));
    }
  return;
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
    Add matrices M and N and store result in matrix P
*/

void addmatrices(Array<double,2> M, Array<double,2> N, Array<double,2> P)
{
  int n = M.extent(0);
  int m = M.extent(1);

  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      P(i, j) = M(i, j) + N(i, j);

   return;
}
```

Typical output from the program looks like

```
Please input number of rows, n, and number of columns, m: 3 3

Reading in elements of A:

Row 1: 1 4 5

Row 2: 6 7 8

Row 3: 3 6 0

Reading in elements of B:

Row 1: 1 6 0

Row 2: 4 7 8

Row 3: 4 8 8
```

```
Writing out elements of A:

Row 1:    1.00    4.00    5.00
Row 2:    6.00    7.00    8.00
Row 3:    3.00    6.00    0.00

Writing out elements of B:

Row 1:    1.00    6.00    0.00
Row 2:    4.00    7.00    8.00
Row 3:    4.00    8.00    8.00

Writing out elements of C = A + B:

Row 1:    2.00   10.00    5.00
Row 2:   10.00   14.00   16.00
Row 3:    7.00   14.00    8.00
```

The header file for the Blitz++ library is called `blitz/array.h`. Moreover, any program file which makes use of Blitz++ must include the cryptic line `using namespace blitz;` before the first call, otherwise compilation errors will ensue. The declaration `Array<double,2> A(n, m)` declares a two-dimensional n by `m` array of doubles. The generalization to an array of integers, or a higher dimension array, is fairly obvious. Note that the `i`, `j` element of matrix `A` is refered to simply as `A(i,j)`. The function call `M.extent(0)` returns the size of array `M` in its first dimension. Likewise, `M.extent(1)` returns the size of array `M` in its second dimension, *etc.* More details of the operation of Blitz++ can be found by reading the extensive documentation which accompanies this library. Unfortunately, the Blitz++ library slows down compilation considerably since it makes use of some very advanced templating features of the C++ language.

2.21   The CAM Graphics Class

There are a myriad of useful, prewritten C++ classes which are freely available on the web—for more details see `http://www.trumphurst.com/cpplibsx.html`. In this Subsection, we shall discuss just one of these—namely, the *CAM graphics*

class,[12] whose purpose is to enable a C++ program to generate simple line plots. This class is freely available from the following URL: `http://www.math.ucla.edu/-~anderson/CAMclass/CAMClass.html`

The CAM graphics class actually generates a PostScript[13] file. Postscript is a programming language that describes the appearance of a printed page. It was developed by Adobe in 1985, and has become an industry standard for printing and imaging. All major printer manufacturers make printers that can interpret PostScript. A PostScript file is conventionally identified via a `.ps` suffix. The schematic code listed below illustrates the basic use of the CAM graphics class:

```
. . .
#include <gprocess.h> // Header file for CAM graphic class
. . .
CAMgraphicsProcess Gprocess;                // declare a graphics process
CAMpostScriptDriver Pdriver("filename.ps"); // declare a PostScript driver
Gprocess.attachDriver(Pdriver);             // attach driver to process
. . .
Gprocess.frame();                           // "frame" the first plot
. . .
Gprocess.frame();                           // "frame" the second plot
. . .
. . .
Gprocess.frame();                           // "frame" the last plot
Gprocess.detachDriver();                    // detach the driver
. . .
```

The header file for the class is called `gprocess.h`. The procedure for generating a plot is to first declare a graphics process, then declare a PostScript driver and attach this to a PostScript file—`filename.ps`, in the above example—and, finally, attach this driver to the process. A PostScript file can contain multiple pictures, or *frames*. Each frame is terminated by a call to `Gprocess.frame()`. Finally, the driver is detached, which has the effect of closing the PostScript file.

The program listed below uses the CAM graphics class to plot the curve $y = \sin^2 x$ for x in the range -2π to 2π.

```
/* camgraph1.cpp */
```

---

[12] The CAM graphics class is copyrighted to its author, Prof. Chris Anderson, Department of Mathematics, UCLA, 1998.

[13] PostScript is a registered trademark of Adobe Systems Incorporated.

```
/*
  Illustration of use of CAM graphics class to create simple line plot

  Program plots y = sin^2 x versus x in range -2 PI to +2 PI

  Program adapted from gpsmp1.cpp by Chris Anderson, UCLA 1996
*/

#include <gprocess.h>
#include <math.h>
#include <stdlib.h>

double func(double);

int main()
{
  int N_points = 400;
  double x_start = -2. * M_PI;
  double x_end = 2. * M_PI;
  double delta_x = (x_end - x_start) / ((double) N_points - 1.);

  double *x = new double[N_points];
  double *y = new double[N_points];

  for (int i = 0; i < N_points; i++)
    {
      x[i] = x_start + (double) i * delta_x;
      y[i] = func(x[i]);
      x[i] /= M_PI;
    }

  { // This brace used to limit scope of Gprocess
    CAMgraphicsProcess Gprocess;                // declare a graphics process
    CAMpostScriptDriver Pdriver("graph1.ps");   // declare a PostScript driver
    Gprocess.attachDriver(Pdriver);             // attach driver to process

    Gprocess.setAxisRange(-2., 2., -2., 2.);    // set plotting ranges
    Gprocess.title("y = sin(x*x)");             // label the plot
    Gprocess.labelX("x / PI");
    Gprocess.labelY("y");

    Gprocess.plot(x, y, N_points);              // do the plotting

    Gprocess.frame();                           // "frame" the plot
```

95

```
    Gprocess.detachDriver();                    // detach the driver
  } // This brace calls the destructor for Gprocess:
    // without it the system() call would hang up

  delete[] x;
  delete[] y;

  system("gv graph1.ps");  // display plot on screen

  return 0;
}


double func(double x)
{
  return sin(x*x);
}
```

The command `Gprocess.plot(x, y, n)` plots the n values of vector y against the n values of vector x as a solid curve. The command `Gprocess.setAxisRange-(x_low, x_high, y_low, y_high)` sets the range of plotting. Finally, the commands `Gprocess.title("title")`, `Gprocess.labelX("x_label")`, and `Gprocess.-labelY("y_label")` label the plot, the x-axis, and the y-axis, respectively. Incidentally, the UNIX function call `system("gv graph1.ps")` is used to pass the command `gv graph1.ps` to the operating system. On execution, this command displays the contents of `graph1.ps` on the screen. The graph written in the file `graph1.ps` is shown in Fig 1

The program shown below illustrates some of the more advanced features of the CAM graphics class:

```
/* camgraph2.cpp */
/*
  Illustration of use of CAMgraphics class to create more advanced line plots

  Program plots three trigonometric functions versus x in range
   -2 PI to +2 PI using different plot styles and different
   line styles

  Program adapted from gpsmp2.cpp by Chris Anderson, UCLA 1996
*/
```
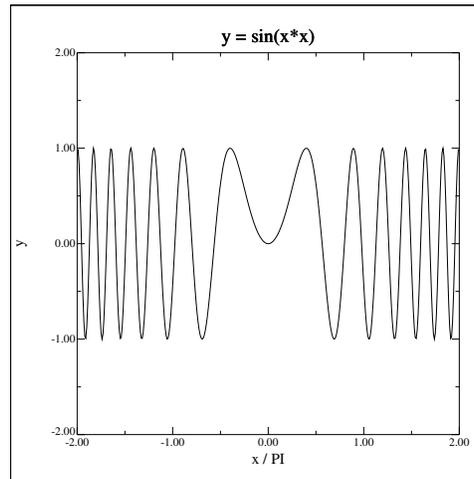
Figure 1: *An example plot generated by the CAM graphics class.*

```
#include <gprocess.h>
#include <math.h>
#include <stdlib.h>

double fun1(double);
double fun2(double);
double fun3(double);

int main()
{
  int N_points = 100;
  double x_start = -2. * M_PI;
  double x_end = 2. * M_PI;
  double delta_x = (x_end - x_start) / ((double) N_points - 1.);

  double *x  = new double[N_points];
  double *y1 = new double[N_points];
  double *y2 = new double[N_points];
  double *y3 = new double[N_points];

  for (int i = 0; i < N_points; i++)
    {
      x[i] = x_start + (double) i * delta_x;
      y1[i] = fun1(x[i]);
      y2[i] = fun2(x[i]);
      y3[i] = fun3(x[i]);
      x[i] /= M_PI;
    }
```

```
  {
    CAMgraphicsProcess Gprocess;               // declare a graphics process
    CAMpostScriptDriver Pdriver("graph2.ps"); // declare a PostScript driver
    Gprocess.attachDriver(Pdriver);            // attach driver to process

    /* First frame;  using different plot "styles" */
    Gprocess.setAxisRange(-2., 2., -2., 2.);           // set plotting ranges
    Gprocess.title("Plots Using Different Plot Styles");// label the plot
    Gprocess.labelX("x / PI");
    Gprocess.labelY("y");

    Gprocess.plot(x, y1, N_points);                 // solid line (default)
    Gprocess.plot(x, y2, N_points, '+');        // + markers
    Gprocess.plot(x, y3, N_points, '+', 2);     // + markers and solid line

    Gprocess.frame();                               // "frame" the plot

    /* Second frame; using different plot line "styles" */
    Gprocess.setAxisRange(-2., 2., -2., 2.);           // set plotting ranges
    Gprocess.title("Plots Using Different Line Styles");// label the plot
    Gprocess.labelX("x / PI");
    Gprocess.labelY("y");

    Gprocess.plot(x, y1, N_points);           // solid line (default)
    Gprocess.setPlotDashPattern(1);
    Gprocess.plot(x, y2, N_points);           // dashed line
    Gprocess.setPlotDashPattern(4);
    Gprocess.plot(x, y3, N_points);           // dashed-dot line

    Gprocess.frame();                         // "frame" the plot

    Gprocess.detachDriver();                  // detach the driver
  }

  delete[] x;
  delete[] y1;
  delete[] y2;
  delete[] y3;

  system("gv graph2.ps"); // display plots on screen

  return 0;
}
```

```
double fun1(double x)
{
  return sin(x);
}

double fun2(double x)
{
  return cos(x);
}

double fun3(double x)
{
  return cos(2.*x);
}
```

The command `Gprocess.plot(x, y, n, '+')` plots the n values of vector y against the n values of vector x as a set of points, each indicated by a '+' character. The command `Gprocess.plot(x, y, n, '+', 2)` does the same, but also connects the points with a solid line. The fourth argument of this command is an integer code which determines the plot style. The various options are as follows: 0 - curve; 1 - points; 2 - curve and points. The command `Gprocess.setPlot-DashPattern(n)` sets the line style. The argument is again an integer code. The various options are: 0 - solid; 1 - dash; 2 - double-dash; 4 - dash-dot; 5 - dash-double-dot; 6 - dots.

The graphs written in the first and second frames of `graph2.ps` are shown in Figs. 2 and 3, respectively.
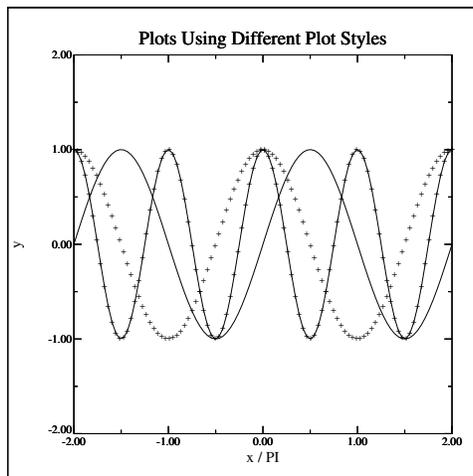
Figure 2: *An example plot generated by the CAM graphics class.*
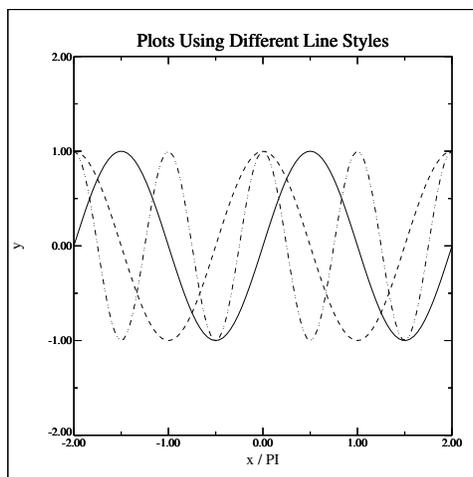
Figure 3: *An example plot generated by the CAM graphics class.*

3   Integration of ODEs

3.1   Introduction

In this section, we shall discuss the standard numerical techniques used to inte-
grate systems of ordinary differential equations (ODEs). We shall then employ
these techniques to simulate the trajectories of various different types of baseball
pitch.

By definition, an ordinary differential equation, or o.d.e., is a differential equa-
tion in which all dependent variables are functions of a *single* independent vari-
able. Furthermore, an nth-order o.d.e. is such that, when it is reduced to its
simplest form, the highest order derivative it contains is nth-order.

According to Newton's laws of motion, the motion of any collection of rigid
objects can be reduced to a set of second-order o.d.e.s. in which time, t, is the
common independent variable. For instance, the equations of motion of a set of
n interacting point objects moving in 1-dimension might take the form:

$$\frac{d^2 x_j}{dt^2} = \frac{F_j(x_1, ..., x_n, t)}{m_j} \tag{3.1}$$

for $j = 1$ to n, where x_j is the position of the jth object, m_j is its mass, *etc.*
Note that a set of n second-order o.d.e.s can always be rewritten as a set of $2n$
first-order o.d.e.s. Thus, the above equations of motion can be rewritten:

$$\frac{dx_j}{dt} = v_j, \tag{3.2}$$

$$\frac{dv_j}{dt} = \frac{F_j(x_1, ..., x_n, t)}{m_j}. \tag{3.3}$$

for $j = 1$ to n. We conclude that a general knowledge of how to numerically solve
a set of coupled first-order o.d.e.s would enable us to investigate the behaviour
of a wide variety of interesting dynamical systems.

3.2 Euler's Method

Consider the general first-order o.d.e.,

$$y' = f(x, y), \tag{3.4}$$

where $'$ denotes d/dx, subject to the general initial-value boundary condition

$$y(x_0) = y_0. \tag{3.5}$$

Clearly, if we can find a method for numerically solving this problem, then we should have little difficulty generalizing it to deal with a system of n simultaneous first-order o.d.e.s.

It is important to appreciate that the numerical solution to a differential equation is only an *approximation* to the actual solution. The actual solution, $y(x)$, to Eq. (3.4) is (presumably) a *continuous* function of a continuous variable, x. However, when we solve this equation numerically, the best that we can do is to evaluate approximations to the function $y(x)$ at a series of *discrete* grid-points, the x_n (say), where $n = 0, 1, 2, \cdots$ and $x_0 < x_1 < x_2 \cdots$. For the moment, we shall restrict our discussion to *equally spaced* grid-points, where

$$x_n = x_0 + n\,h. \tag{3.6}$$

Here, the quantity h is referred to as the *step-length*. Let y_n be our approximation to $y(x)$ at the grid-point x_n. A numerical integration scheme is essentially a method which somehow employs the information contained in the original o.d.e., Eq. (3.4), to construct a series of rules interrelating the various y_n.

The simplest possible integration scheme was invented by the celebrated 18th century Swiss mathematician Leonhard Euler, and is, therefore, called *Euler's method*. Incidentally, it is interesting to note that virtually all of the standard methods used in numerical analysis were invented *before* the advent of electronic computers. In olden days, people actually performed numerical calculations *by hand*—and a very long and tedious process it must have been! Suppose that we have evaluated an approximation, y_n, to the solution, $y(x)$, of Eq. (3.4) at the grid-point x_n. The approximate gradient of $y(x)$ at this point is, therefore, given
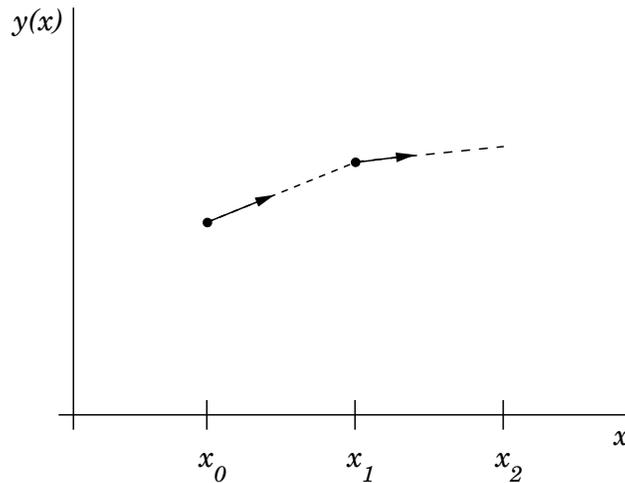
Figure 4: *Illustration of Euler's method.*

by

$$y'_n = f(x_n, y_n). \tag{3.7}$$

Let us approximate the curve $y(x)$ as a *straight-line* between the neighbouring grid-points x_n and x_{n+1}. It follows that

$$y_{n+1} = y_n + y'_n h, \tag{3.8}$$

or

$$y_{n+1} = y_n + f(x_n, y_n) h. \tag{3.9}$$

The above formula is the essence of Euler's method. It enables us to calculate all of the y_n, given the initial value, y_0, at the first grid-point, x_0. Euler's method is illustrated in Fig. 4.

3.3   Numerical Errors

There are two major sources of error associated with a numerical integration scheme for o.d.e.s: namely, *truncation error* and *round-off error*. Truncation error arises in Euler's method because the curve $y(x)$ is *not* generally a straight-line between the neighbouring grid-points x_n and x_{n+1}, as assumed above. The error associated with this approximation can easily be assessed by Taylor expanding

$y(x)$ about $x = x_n$:

$$
\begin{aligned}
y(x_n + h) &= y(x_n) + h\,y'(x_n) + \frac{h^2}{2}\,y''(x_n) + \cdots \\
&= y_n + h\,f(x_n, y_n) + \frac{h^2}{2}\,y''(x_n) + \cdots.
\end{aligned}
\tag{3.10}
$$

A comparison of Eqs. (3.9) and (3.10) yields

$$
y_{n+1} = y_n + h\,f(x_n, y_n) + O(h^2).
\tag{3.11}
$$

In other words, every time we take a step using Euler's method we incur a truncation error of $O(h^2)$, where h is the step-length. Suppose that we use Euler's method to integrate our o.d.e. over an x-interval of order unity. This requires $O(h^{-1})$ steps. If each step incurs an error of $O(h^2)$, and the errors are simply cumulative (a fairly conservative assumption), then the net truncation error is $O(h)$. In other words, the error associated with integrating an o.d.e. over a finite interval using Euler's method is directly proportional to the step-length. Thus, if we want to keep the relative error in the integration below about 10^{-6} then we would need to take about one million steps per unit interval in x. Incidentally, Euler's method is termed a *first-order* integration method because the truncation error associated with integrating over a finite interval scales like h^1. More generally, an integration method is conventionally called nth order if its truncation error *per step* is $O(h^{n+1})$.

Note that truncation error would be incurred even if computers performed floating-point arithmetic operations to infinite accuracy. Unfortunately, computers *do not* perform such operations to infinite accuracy. In fact, a computer is only capable of storing a floating-point number to a fixed number of decimal places. For every type of computer, there is a characteristic number, η, which is defined as the smallest number which when added to a number of order unity gives rise to a new number: *i.e.*, a number which when taken away from the original number yields a non-zero result. Every floating-point operation incurs a *round-off error* of $O(\eta)$ which arises from the finite accuracy to which floating-point numbers are stored by the computer. Suppose that we use Euler's method to integrate our o.d.e. over an x-interval of order unity. This entails $O(h^{-1})$ integration steps,

and, therefore, $O(h^{-1})$ floating-point operations. If each floating-point operation incurs an error of $O(\eta)$, and the errors are simply cumulative, then the net round-off error is $O(\eta/h)$.

The total error, ϵ, associated with integrating our o.d.e. over an x-interval of order unity is (approximately) the sum of the truncation and round-off errors. Thus, for Euler's method,

$$\epsilon \sim \frac{\eta}{h} + h. \tag{3.12}$$

Clearly, at large step-lengths the error is dominated by truncation error, whereas round-off error dominates at small step-lengths. The net error attains its minimum value, $\epsilon_0 \sim \eta^{1/2}$, when $h = h_0 \sim \eta^{1/2}$. There is clearly no point in making the step-length, h, any smaller than h_0, since this increases the number of floating-point operations but does not lead to an increase in the overall accuracy. It is also clear that the ultimate accuracy of Euler's method (or any other integration method) is determined by the accuracy, η, to which floating-point numbers are stored on the computer performing the calculation.

The value of η depends on how many bytes the computer hardware uses to store floating-point numbers. For IBM-PC clones, the appropriate value for *double precision* floating point numbers is $\eta = 2.22 \times 10^{-16}$ (this value is specified in the system header file `float.h`). It follows that the minimum practical step-length for Euler's method on such a computer is $h_0 \sim 10^{-8}$, yielding a minimum relative integration error of $\epsilon_0 \sim 10^{-8}$. This level of accuracy is perfectly adequate for most scientific calculations. Note, however, that the corresponding η value for *single precision* floating-point numbers is only $\eta = 1.19 \times 10^{-7}$, yielding a minimum practical step-length and a minimum relative error for Euler's method of $h_0 \sim 3 \times 10^{-4}$ and $\epsilon_0 \sim 3 \times 10^{-4}$, respectively. This level of accuracy is generally *not* adequate for scientific calculations, which explains why such calculations are invariably performed using double, rather than single, precision floating-point numbers on IBM-PC clones (and most other types of computer).

3.4   Numerical Instabilities

Consider the following example. Suppose that our o.d.e. is

$$y' = -\alpha\, y, \tag{3.13}$$

where $\alpha > 0$, subject to the boundary condition

$$y(0) = 1. \tag{3.14}$$

Of course, we can solve this problem analytically to give

$$y(x) = \exp(-\alpha\, x). \tag{3.15}$$

Note that the solution is a *monotonically decreasing* function of x. We can also solve this problem numerically using Euler's method. Appropriate grid-points are

$$x_n = n\, h, \tag{3.16}$$

where $n = 0, 1, 2, \cdots$. Euler's method yields

$$y_{n+1} = (1 - \alpha\, h)\, y_n. \tag{3.17}$$

Note one curious fact. If $h > 2/\alpha$ then $|y_{n+1}| > |y_n|$. In other words, if the step-length is made too large then the numerical solution becomes an oscillatory function of x of *monotonically increasing* amplitude: *i.e.,* the numerical solution *diverges* from the actual solution. This type of catastrophic failure of a numerical integration scheme is called a *numerical instability*. All simple integration schemes become unstable if the step-length is made sufficiently large.

3.5   Runge-Kutta Methods

There are two main reasons why Euler's method is *not* generally used in scientific computing. Firstly, the truncation error per step associated with this method is far larger than those associated with other, more advanced, methods (for a given value of h). Secondly, Euler's method is too prone to numerical instabilities.

The methods most commonly employed by scientists to integrate o.d.e.s were first developed by the German mathematicians C.D.T. Runge and M.W. Kutta in the latter half of the nineteenth century.[14] The basic reasoning behind so-called *Runge-Kutta* methods is outlined in the following.

The main reason that Euler's method has such a large truncation error per step is that in evolving the solution from x_n to x_{n+1} the method only evaluates derivatives at the beginning of the interval: *i.e.*, at x_n. The method is, therefore, very *asymmetric* with respect to the beginning and the end of the interval. We can construct a more symmetric integration method by making an Euler-like trial step to the midpoint of the interval, and then using the values of both x and y at the midpoint to make the real step across the interval. To be more exact,

$$k_1 = h\, f(x_n, y_n), \tag{3.18}$$

$$k_2 = h\, f(x_n + h/2, y_n + k_1/2), \tag{3.19}$$

$$y_{n+1} = y_n + k_2 + O(h^3). \tag{3.20}$$

As indicated in the error term, this symmetrization cancels out the first-order error, making the method *second-order*. In fact, the above method is generally known as a *second-order Runge-Kutta* method. Euler's method can be thought of as a first-order Runge-Kutta method.

Of course, there is no need to stop at a second-order method. By using two trial steps per interval, it is possible to cancel out both the first and second-order error terms, and, thereby, construct a third-order Runge-Kutta method. Likewise, three trial steps per interval yield a fourth-order method, and so on.[15]

The general expression for the total error, ϵ, associated with integrating our o.d.e. over an x-interval of order unity using an nth-order Runge-Kutta method is approximately

$$\epsilon \sim \frac{\eta}{h} + h^n. \tag{3.21}$$

Here, the first term corresponds to round-off error, whereas the second term

---

[14]*Numerical recipes in C: the art of scientific computing*, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press, Cambridge UK, 1992), p. 710.

[15]*Handbook of mathematical functions*, M. Abramowitz and I.A. Stegun (Dover, New York NY, 1965), p. 896.

n	h_0	ϵ_0
1	1.5×10^{-8}	1.5×10^{-8}
2	6.1×10^{-6}	3.7×10^{-11}
3	1.2×10^{-4}	1.8×10^{-12}
4	7.4×10^{-4}	3.0×10^{-13}
5	2.4×10^{-3}	9.0×10^{-14}

Table 1: *The minimum practical step-length, h_0, and minimum error, ϵ_0, for an nth-order Runge-Kutta method integrating over a finite interval using double precision arithmetic on an IBM-PC clone.*

represents truncation error. The minimum practical step-length, h_0, and the minimum error, ϵ_0, take the values

$$h_0 \sim \eta^{1/(n+1)}, \tag{3.22}$$

$$\epsilon_0 \sim \eta^{n/(n+1)}, \tag{3.23}$$

respectively. In Tab. 1, these values are tabulated against n using $\eta = 2.22 \times 10^{-16}$ (the value appropriate to double precision arithmetic on IBM-PC clones). It can be seen that h_0 increases and ϵ_0 decreases as n gets larger. However, the relative change in these quantities becomes progressively less dramatic as n increases.

In the majority of cases, the limiting factor when numerically integrating an o.d.e. is not round-off error, but rather the computational effort involved in calculating the function $f(x,y)$. Note that, in general, an nth-order Runge-Kutta method requires n evaluations of this function per step. It can easily be appreciated that as n is increased a point is quickly reached beyond which any benefits associated with the increased accuracy of a higher order method are more than offset by the computational "cost" involved in the necessary additional evaluation of $f(x,y)$ per step. Although there is no hard and fast general rule, in most problems encountered in computational physics this point corresponds to $n = 4$. In other words, in most situations of interest a *fourth-order Runge Kutta* integration method represents an appropriate compromise between the competing requirements of a low truncation error per step and a low computational cost per step.

The standard *fourth-order Runge-Kutta* method takes the form:

$$k_1 = h\,f(x_n, y_n), \tag{3.24}$$

$$k_2 = h\,f(x_n + h/2, y_n + k_1/2), \tag{3.25}$$

$$k_3 = h\,f(x_n + h/2, y_n + k_2/2), \tag{3.26}$$

$$k_4 = h\,f(x_n + h, y_n + k_3), \tag{3.27}$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5). \tag{3.28}$$

This is the method which we shall use, throughout this course, to integrate first-order o.d.e.s. The generalization of this method to deal with systems of coupled first-order o.d.e.s is (hopefully) fairly obvious.

3.6 An Example Fixed-Step RK4 routine

Listed below is an example fixed-step, fourth-order Runge-Kutta (RK4) integration routine which utilizes the `Blitz++` library (see Sect. 2.20).

```
// rk4_fixed.cpp

/*
  Function to advance set of coupled first-order o.d.e.s by single step
  using fixed step-length fourth-order Runge-Kutta scheme

     x        ... independent variable
     y        ... array of dependent variables
     h        ... fixed step-length

  Requires right-hand side routine

     void rhs_eval (double x, Array<double,1> y, Array<double,1>& dydx)

  which evaluates derivatives of y (w.r.t. x) in array dydx
*/

#include <blitz/array.h>

using namespace blitz;

void rk4_fixed (double& x, Array<double,1>& y,
                void (*rhs_eval)(double, Array<double,1>, Array<double,1>&),
                double h)
```

```
{
  // Array y assumed to be of extent n, where n is no. of coupled
  // equations
  int n = y.extent(0);

  // Declare local arrays
  Array<double,1> k1(n), k2(n), k3(n), k4(n), f(n), dydx(n);

  // Zeroth intermediate step
  (*rhs_eval) (x, y, dydx);
  for (int j = 0; j < n; j++)
    {
      k1(j) = h * dydx(j);
      f(j) = y(j) + k1(j) / 2.;
    }

  // First intermediate step
  (*rhs_eval) (x + h / 2., f, dydx);
  for (int j = 0; j < n; j++)
    {
      k2(j) = h * dydx(j);
      f(j) = y(j) + k2(j) / 2.;
    }

  // Second intermediate step
  (*rhs_eval) (x + h / 2., f, dydx);
  for (int j = 0; j < n; j++)
    {
      k3(j) = h * dydx(j);
      f(j) = y(j) + k3(j);
    }

  // Third intermediate step
  (*rhs_eval) (x + h, f, dydx);
  for (int j = 0; j < n; j++)
    {
      k4(j) = h * dydx(j);
    }

  // Actual step
  for (int j = 0; j < n; j++)
    {
      y(j) += k1(j) / 6. + k2(j) / 3. + k3(j) / 3. + k4(j) / 6.;
    }
  x += h;
```

```
  return;
}
```

3.7 An Example Calculation

Consider the following system of o.d.e.s:

$$\frac{dx}{dt} = v, \tag{3.29}$$

$$\frac{dv}{dt} = -k\,x, \tag{3.30}$$

subject to the initial conditions $x(0) = 0$ and $v(0) = \sqrt{k}$ at $t = 0$. In fact, this system can be solved analytically to give

$$x = \sin(\sqrt{k}\,t). \tag{3.31}$$

Let us compare the above solution with that obtained numerically using either Euler's method or a fourth-order Runge-Kutta method. Figure 5 shows the integration errors associated with these two methods (calculated by integrating the above system, with $k = 1$, from $t = 0$ to $t = 10$, and then taking the difference between the numerical and analytic solutions) plotted against the step-length, h, in a log-log graph. All calculations are performed to *single precision*: *i.e.*, by using `float`, rather than `double`, variables. It can be seen that at large values of h, the error associated with Euler's method becomes much greater than unity (*i.e.*, the magnitude of the numerical solution greatly exceeds that of the analytic solution), indicating the presence of a *numerical instability*. There are no similar signs of instability associated with the Runge-Kutta method. At intermediate h, the error associated with Euler's method decreases smoothly like h^{-1}: in this regime, the dominant error is *truncation error*, which is expected to scale like h^{-1} for a first-order method. The error associated with the Runge-Kutta method similarly scales like h^{-4}—as expected for a fourth-order scheme—in the truncation error dominated regime. Note that, as h is decreased, the error associated with both methods eventually starts to rise in a jagged curve that scales roughly like h^1. This is a manifestation of *round-off error*. The minimum error associated
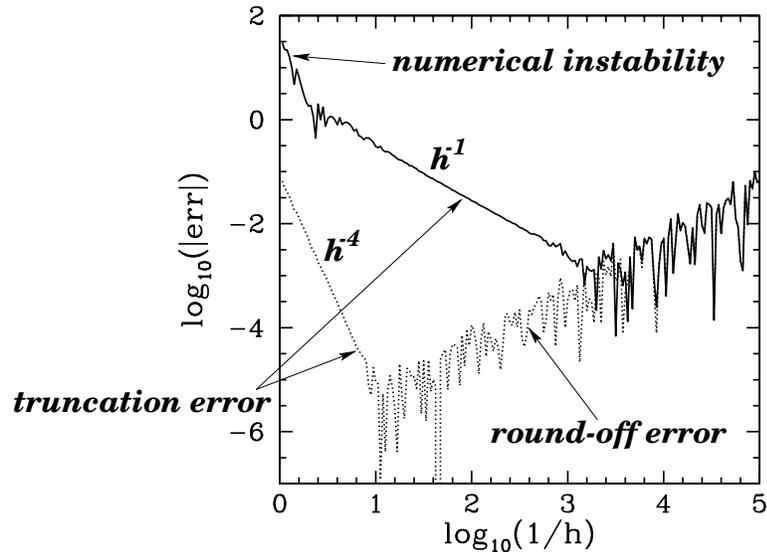
Figure 5: *Global integration errors associated with Euler's method (solid curve) and a fourth-order Runge-Kutta method (dotted curve) plotted against the step-length* h. *Single precision calculation.*

with both methods corresponds to the boundary between the truncation error and round-off error dominated regimes. Thus, for Euler's method the minimum error is about 10^{-3} at $h \sim 10^{-3}$, whereas for the Runge-Kutta method the minimum error is about 10^{-5} at $h \sim 10^{-1}$. Clearly, the performance of the Runge-Kutta method is vastly superior to that of Euler's method, since the former method is capable of attaining much greater accuracy than the latter using a far smaller number of steps (*i.e.,* a far larger h).

Figure 6 displays similar data to that shown in Fig. 5, except that now all of the calculations are performed to *double precision*. The figure exhibits the same broad features as those apparent in Fig. 5. The major difference is that the round-off error has been reduced by about nine orders of magnitude, allowing the Runge-Kutta method to attain a minimum error of about 10^{-12} (see Tab. 1)— a remarkably performance!

Figures 5 and 6 illustrate why scientists rarely use Euler's method, or single precision numerics, to integrate systems of o.d.e.s.
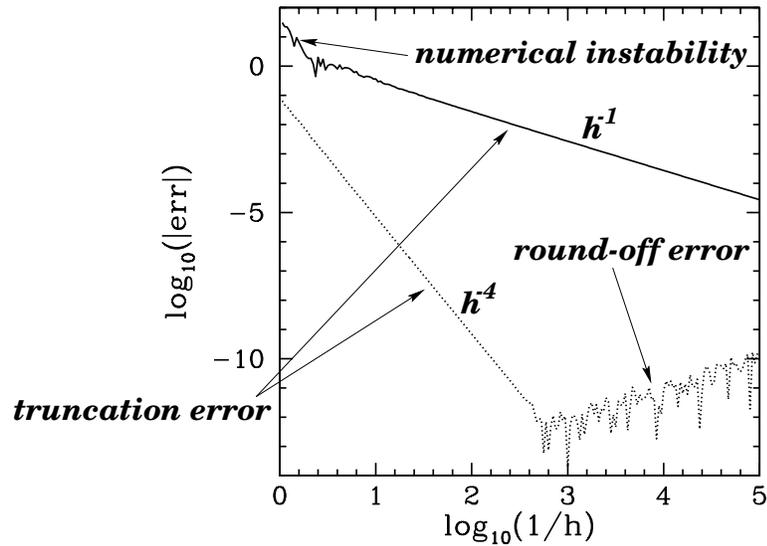
Figure 6: *Global integration errors associated with Euler's method (solid curve) and a fourth-order Runge-Kutta method (dotted curve) plotted against the step-length* h. *Double precision calculation.*

3.8 Adaptive Integration Methods

Consider the following system of o.d.e.s:

$$\frac{dx}{dt} = v, \tag{3.32}$$

$$\frac{dv}{dt} = \frac{v}{t} - 4\,k\,t^2\,x, \tag{3.33}$$

subject to the boundary conditions $x = \sqrt{k}\,v^2$ and $v = 2\sqrt{k}\,v$ at $x = v$, where $0 < v \ll 1$. This system can be solved analytically to give

$$x = \sin(\sqrt{k}\,t^2). \tag{3.34}$$

One peculiarity of the above solution is that its variation scale-length *decreases rapidly* as t increases.

Let us compare the above solution with that obtained numerically using a fourth-order Runge-Kutta method. Figure 7 shows the integration error associated with such a method (calculated by integrating the above system, with $k = 10$, from $t = 10^{-3}$ to $t = t$, and then taking the difference between the
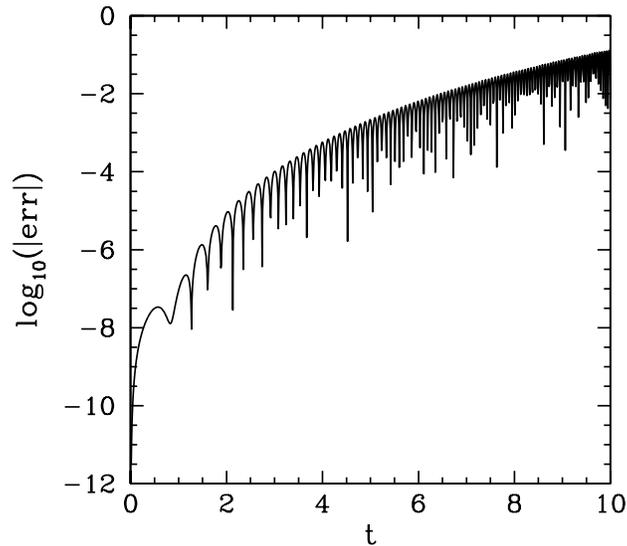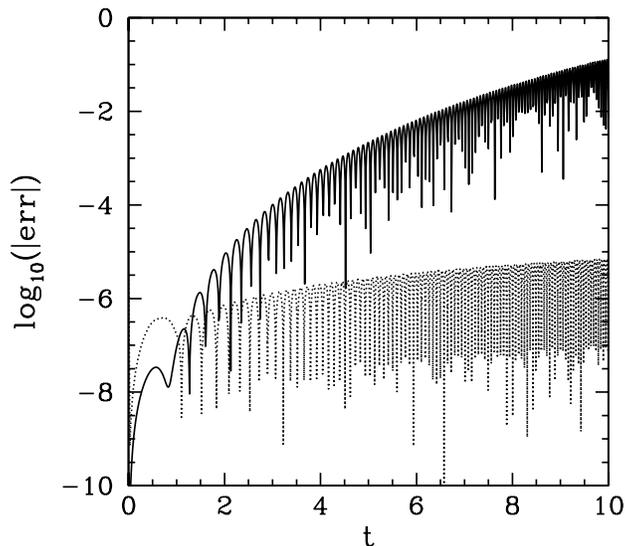
Figure 7: *Global integration error associated with a fixed step-length ($h = 0.01$), fourth-order Runge-Kutta method, plotted against the independent variable, t, for a system of o.d.e.s in which the variation scale-length decreases rapidly with increasing t. Double precision calculation.*

numerical and analytic solutions) with a fixed step-length of $h = 0.01$, plotted against the independent variable, t. It can be seen that, although the error starts off small, it rises rapidly as the variation scale-length of the solution decreases (*i.e.*, as t increases), and quickly becomes unacceptably large. Of course, we could reduce the error by simply reducing the step-length, h. However, this is a very inefficient solution. The step-length only needs to be reduced at large t. There is no need to reduce it, at all, at small t. Clearly, the ideal solution to this problem would be an integration method in which the step-length is *varied* so as to maintain a relatively constant truncation error per step. Such an *adaptive integration method* would take large steps when variation scale-length of the solution was large, and *vice versa*.

Let us investigate how we could convert our fixed step-length, fourth-order Runge-Kutta method into a corresponding adaptive method. First of all, we need an estimate of the truncation error at each step. Suppose that the current step-length is h. We can estimate the truncation error, ϵ, associated with the current step by taking the difference between the solutions obtained by stepping by $h/2$ twice and by h once (starting from the same point, in both cases). Let ϵ_0 be

114

the desired truncation error per step. How do we adjust h so as to ensure that the truncation error associated with the next step is closer to this value? Observe, from Eq. (3.28), that the truncation error per step in a fourth-order scheme scales like h^5. It follows, therefore, that our step-length adjustment formula should take the form[16]

$$h_{\text{new}} = h_{\text{old}} \left| \frac{\epsilon_0}{\epsilon} \right|^{1/5}. \tag{3.35}$$

According to this formula, the step-length should be increased if the truncation error per step is too small, and *vice versa*, in such a manner that the error per step remains relatively constant at ϵ_0.

There are a number of caveats to the above discussion. In a system of n coupled o.d.e.s, the overall truncation error per step, ϵ, should, of course, be some appropriately weighted average of the errors associated with each equation. There is also a question of whether ϵ should be an *absolute error* or a *relative error*. The relative error associated with the ith equation is simply the absolute error divided by $|y_i|$, where y_i is the current value of the ith dependent variable. An absolute error estimate is appropriate to a system of equations in which the amplitudes of the various dependent variables remain bounded. A relative error estimate is appropriate to a system in which the amplitudes of the dependent variables blow-up at some point, but the variables always remain the same sign. Finally, a *mixed error* estimate—usually the minimum of the absolute and relative errors—is appropriate to a system in which the amplitudes of the dependent variables blow-up at some point, but the signs of the variables oscillate. It is usually a good idea to place some limits on the allowed variation of the step-length from step to step: *e.g.*, by preventing the step-length from increasing or decreasing by more than some factor $S > 1$ per step. This prevents h from oscillating unduly about its optimum value. Obviously, if h becomes absurdly small then the integration method has failed, and should abort with an appropriate error message. Finally, a limit should be placed on how large h can become—unfortunately, adaptive methods have a tendency to become a little over optimistic when integration is easy.

---

[16]*Numerical recipes in C: the art of scientific computing*, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press, Cambridge, England, 1992), p. 714.

Figure 8: *Global integration errors associated with a fixed step-length ($h = 0.01$), fourth-order Runge-Kutta method (solid curve) and a corresponding adaptive method ($\epsilon_0 = 10^{-8}$)(dotted curve), plotted against the independent variable, t, for a system of o.d.e.s in which the variation scale-length decreases rapidly with increasing t. Double precision calculation.*

Figure 8 shows the integration errors associated with a fixed step-length, fourth-order Runge-Kutta method and a corresponding adaptive method—constructed along the lines discussed above—as functions of the independent variable, t. The errors are calculated by integrating the current system, with $k = 10$, from $t = 10^{-3}$ to $t = t$, and then taking the difference between the numerical and analytic solutions. The fixed step-length associated with the former method is $h = 0.01$. The desired truncation error per step associated with the latter is $\epsilon_0 = 10^{-8}$. It can be seen that the performance of the adaptive method is far superior to that of the fixed step-length method, since the former method maintains a relatively constant integration error as the variation scale-length of the solution decreases (*i.e.*, as t increases). Figure 9 illustrates how this is achieved. This figure shows the step-length, h, associated with the adaptive method as a function of t. It can be seen that the adaptive method maintains a relatively constant truncation error per step by decreasing h as t increases.

116
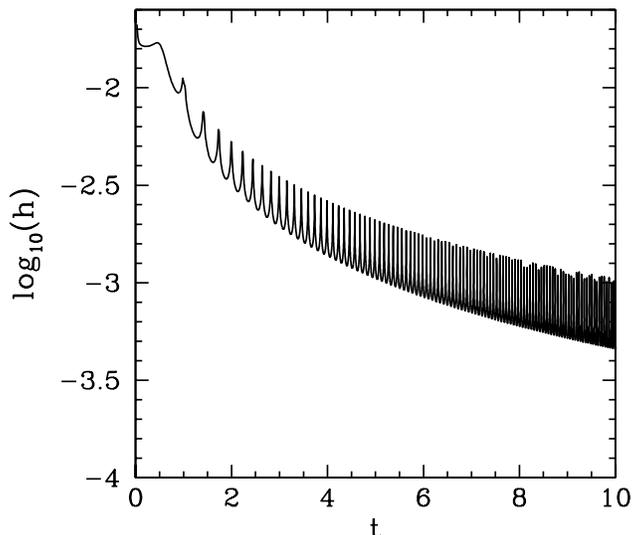
Figure 9: *The step-length,* h, *associated with the adaptive integration method shown in the previous figure, plotted as a function of the independent variable,* t. *Double precision calculation.*

3.9   An Example Adaptive-Step RK4 Routine

Listed below is an example adaptive-step RK4 routine which makes use of the previously listed fixed-step routine. Note that the routine recalculates all steps whose truncation error exceeds the desired value acc.

```
// rk4_adaptive.cpp

/*
  Function to advance set of coupled first-order o.d.e.s by single step
  using adaptive fourth-order Runge-Kutta scheme

     x        ... independent variable
     y        ... array of dependent variables
     h        ... step-length
     t_err    ... actual truncation error per step
     acc      ... desired truncation error per step
     S        ... step-length cannot change by more than this factor from
                   step to step
     rept     ... number of step recalculations
     maxrept  ... maximum allowable number of step recalculations
     h_min    ... minimum allowable step-length
     h_max    ... maximum allowable step-length
```

```
      flag    ... controls manner in which truncation error is calculated

  Requires right-hand side routine

        void rhs_eval (double x, Array<double,1> y, Array<double,1>& dydx)

  which evaluates derivatives of y (w.r.t. x) in array dydx.

  Function advances equations by single step whilst attempting to maintain
  constant truncation error per step of acc:

    flag = 0 ... error is absolute
    flag = 1 ... error is relative
    flag = 2 ... error is mixed

  If step-length falls below h_min then routine aborts
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <blitz/array.h>

using namespace blitz;

void rk4_fixed (double&, Array<double,1>&,
                void (*)(double, Array<double,1>, Array<double,1>&),
                double);

void rk4_adaptive (double& x, Array<double,1>& y,
                   void (*rhs_eval)(double, Array<double,1>, Array<double,1>&),
                   double& h, double& t_err, double acc,
                   double S, int& rept, int maxrept,
                   double h_min, double h_max, int flag)
{
  // Array y assumed to be of extent n,  where n is no. of coupled
  // equations
  int n = y.extent(0);

  // Declare local arrays
  Array<double,1> y0(n), y1(n);

  // Declare repetition counter
  static int count = 0;
```

```
// Save initial data
double x0 = x;
y0 = y;

// Take full step
rk4_fixed (x, y, rhs_eval, h);

// Save data
y1 = y;

// Restore initial data
x = x0;
y = y0;

// Take two half-steps
rk4_fixed (x, y, rhs_eval, h/2.);
rk4_fixed (x, y, rhs_eval, h/2.);

// Calculate truncation error
t_err = 0.;
double err, err1, err2;
if (flag == 0)
  {
    // Use absolute truncation error
    for (int i = 0; i < n; i++)
      {
        err = fabs (y(i) - y1(i));
        t_err = (err > t_err) ? err : t_err;
      }
  }
else if (flag == 1)
  {
    // Use relative truncation error
    for (int i = 0; i < n; i++)
      {
        err = fabs ((y(i) - y1(i)) / y(i));
        t_err = (err > t_err) ? err : t_err;
      }
  }
else
  {
    // Use mixed truncation error
    for (int i = 0; i < n; i++)
      {
        err1 = fabs ((y(i) - y1(i)) / y(i));
```

```
        err2 = fabs (y(i) - y1(i));
        err = (err1 < err2) ? err1 : err2;
        t_err = (err > t_err) ? err : t_err;
      }
  }

// Prevent small truncation error from rounding to zero
if (t_err == 0.) t_err = 1.e-15;

// Calculate new step-length
double h_est = h * pow (fabs (acc / t_err), 0.2);

// Prevent step-length from changing by more than factor S
if (h_est / h > S)
  h *= S;
else if (h_est / h < 1. / S)
  h /= S;
else
  h = h_est;

// Prevent step-length from exceeding h_max
h = (fabs(h) > h_max) ? h_max * h / fabs(h) : h;

// Abort if step-length falls below h_min
if (fabs(h) < h_min)
  {
    printf ("Error - |h| < hmin\n");
    exit (1);
  }

// If truncation error acceptable take step
if ((t_err <= acc) || (rept >= maxrept))
  {
    rept = count;
    count = 0;
  }
// If truncation error unacceptable repeat step
else
  {
    count++;
    x = x0;
    y = y0;
    rk4_adaptive (x, y, rhs_eval, h, t_err, acc,
                  S, rept, maxrept, h_min, h_max, flag);
  }
```

```
   return;
}
```

3.10   Advanced Integration Methods

Of course, Runge-Kutta methods are not the last word in integrating o.d.e.s. Far from it! Runge-Kutta methods are sometimes referred to as *single-step* methods, since they evolve the solution from x_n to x_{n+1} without needing to know the solutions at x_{n-1}, x_{n-2}, *etc.* There is a broad class of more sophisticated integration methods, known as *multi-step* methods, which utilize the previously calculated solutions at x_{n-1}, x_{n-2}, *etc.* in order to evolve the solution from x_n to x_{n+1}. Examples of these methods are the various Adams methods[17] and the various Predictor-Corrector methods.[18] The main advantages of Runge-Kutta methods are that they are easy to implement, they are very stable, and they are "self-starting" (*i.e.*, unlike muti-step methods, we do not have to treat the first few steps taken by a single-step integration method as special cases). The primary disadvantages of Runge-Kutta methods are that they require significantly more computer time than multi-step methods of comparable accuracy, and they do not easily yield good *global* estimates of the truncation error. However, for the straightforward dynamical systems under investigation in this course, the advantage of the relative simplicity and ease of use of Runge-Kutta methods far outweighs the disadvantage of their relatively high computational cost.

3.11   The Physics of Baseball Pitching

Baseball is the oldest professional sport in the U.S. It is a game of great subtlety (like cricket!) which has fascinated fans for over a hundred years. It has also fascinated physicists—partly, because many physicists are avid baseball fans, but, partly, also, because there are clearly delineated physics principles at work in

---

[17]*Numerical methods*, R.W. Hornbeck (Prentice-Hall, Englewood Cliffs NJ, 1975), p. 196.
[18]*ibid*, p. 199.

this game. Indeed, more books and papers have been written on the physics of baseball than on any other sport.

A baseball is formed by winding yarn around a small sphere of cork. The ball is then covered with two interlocking pieces of white cowhide, which are tightly stitched together. The mass and circumference of a regulation baseball are 5 oz and 9 in (*i.e.*, about 150 g and 23 cm), respectively. In the major leagues, the ball is pitched a distance of 60 feet 6 inches (*i.e.*, 18.44 m), towards the hitter, at speeds which typically lie in the range 60 to 100 mph (*i.e.*, about 30 to 45 m/s). As is well-known to baseball fans, there are a surprising variety of different pitches. "Sliders" deviate sideways through the air. "Curveballs" deviate sideways, but also dip unusually rapidly. Conversely, "fastballs" dip unusually slowly. Finally, the mysterious "knuckleball" can weave from side to side as it moves towards the hitter. How is all this bizarre behaviour possible? Let us investigate.

3.12 Air Drag

A baseball in flight is subject to three distinct forces. The first is *gravity*, which causes the ball to accelerate vertically downwards at $g = 9.8\,\mathrm{m/s}^{-2}$. The second is *air drag*, which impedes the ball's motion through the air. The third is the *Magnus force*, which permits the ball to curve laterally. Let us discuss the latter two forces in more detail.

As is well-known, the drag force acting on an object which moves *very slowly* through a viscous fluid is directly proportional to the speed of that object with respect to the fluid. For example, a sphere of radius r, moving with speed v through a fluid whose coefficient of viscosity is η, experiences a drag force given by Stokes' law:[19]

$$f_D = 6\,\pi\,\eta\,r\,v. \tag{3.36}$$

As students who have attempted to reproduce Millikan's oil drop experiment will recall, this force is the dominant drag force acting on a microscopic oil drop falling through air. However, for most *macroscopic* projectiles moving through

---

[19]*Methods of theoretical physics*, Vol. II, P.M. Morse, and H. Feshbach, (McGraw-Hill, New York NY, 1953).

air, the above force is dwarfed by a second drag force which is proportional to v^2.

The origin of this second force is fairly easy to understand. At velocities sufficiently low for Stokes' law to be valid, air is able to flow smoothly around a passing projectile. However, at higher velocities, the projectile's motion is too rapid for this to occur. Instead, the projectile effectively knocks the air out of its way. The total mass of air which the projectile comes into contact with per second is $\rho\,v\,A$, where ρ is the air density, v the projectile speed, and A the projectile's cross-sectional area. Suppose, as seems reasonable, that the projectile imparts to this air mass a speed v' which is directly proportional to v. The rate of momentum gain of the air, which is equal to the drag force acting on the projectile, is approximately

$$f_D = \frac{1}{2}\,C_D(v)\,\rho\,A\,v^2, \tag{3.37}$$

where the *drag coefficient*, $C_D(v)$, is a dimensionless quantity.

The drag force acting on a projectile, passing through air, always points in the *opposite* direction to the projectile's instantaneous direction of motion. Thus, the vector drag force takes the form

$$\mathbf{f}_D = -\frac{1}{2}\,C_D(v)\,\rho\,A\,v\,\mathbf{v}. \tag{3.38}$$

When a projectile moves through air it leaves a *turbulent* wake. The value of the drag coefficient, C_D, is closely related to the properties of this wake. Turbulence in fluids is conventionally characterized in terms of a dimensionless quantity known as a *Reynolds number*:[20]

$$R_e = \frac{\rho\,v\,d}{\eta}. \tag{3.39}$$

Here, d is the typical length-scale (*e.g.*, the diameter) of the projectile. For sufficiently small Reynolds numbers, the air flow immediately above the surface of the projectile remains smooth, despite the presence of the turbulent wake, and C_D takes an approximately constant value which depends on the projectile's shape. However, above a certain critical value of R_e—which corresponds to 2×10^5 for

---

[20]R.E. Reynolds, Phil. Trans. Roy. Soc. **174**, 935 (1883).
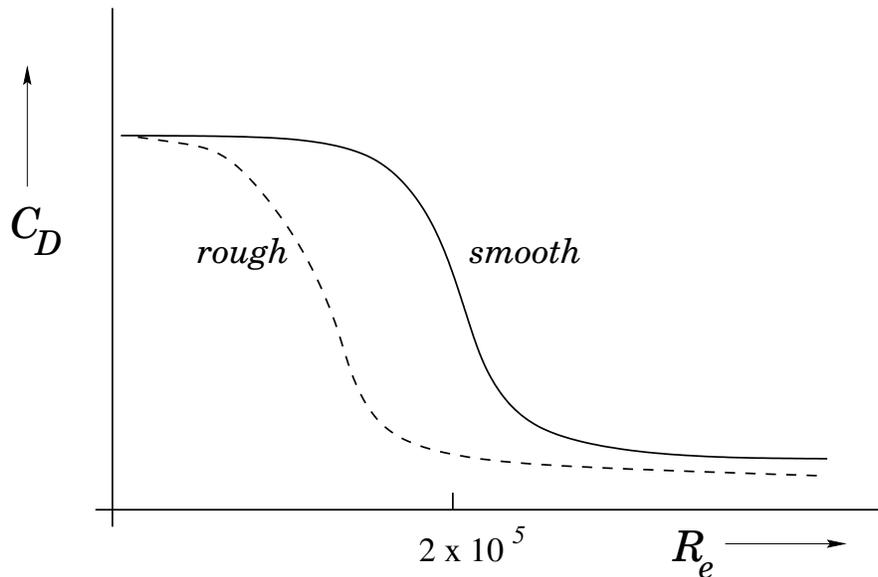
Figure 10: *Typical dependence of the drag coefficient,* C_D*, on the Reynolds number,* R_e*.*

a *smooth* projectile—the air flow immediately above the surface of the projectile becomes turbulent, and the drag coefficient drops: *i.e.*, a projectile slips more easily through the air when the surrounding flow is completely turbulent. In this high Reynolds number regime, the drag coefficient generally falls rapidly by a factor of between 3 and 10, as R_e is increased, and then settles down to a new, roughly constant value. Note that the critical Reynolds number is significantly less than 2×10^5 for projectiles with rough surfaces. Paradoxically, a rough projectile generally experiences less air drag, at high velocities, than a smooth one. The typical dependence of the drag coefficient on the Reynolds number is illustrated schematically in Fig. 10.

Wind tunnel measurements reveal that the drag coefficient is a strong function of speed for baseballs, as illustrated in Fig. 11. At low speeds, the drag coefficient is approximately constant. However, as the speed increases, C_D drops substantially, and is more than a factor of 2 smaller at high speeds. This behaviour is similar to that described above. The sudden drop in the drag coefficient is triggered by a transition from laminar to turbulent flow in the air layer immediately above the ball's surface. The critical speed (to be more exact, the critical Reynolds number) at which this transition occurs depends on the properties of the surface. Thus, for a completely smooth baseball the transition occurs at speeds well be-
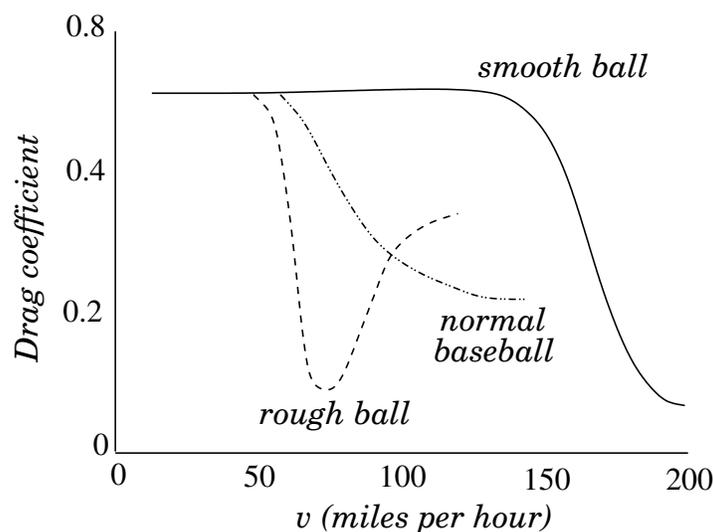
Figure 11: *Variation of the drag coefficient, C_D, with speed, v, for normal, rough, and smooth baseballs. From* The physics of baseball, *R.K. Adair (Harper & Row, New York NY, 1990).*

yond the capabilities of the fastest pitchers. Conversely, the transition takes place at comparatively low speeds if the surface of the ball is rough. In fact, the raised stitches on the otherwise smooth regulation baseball cause the transition to occur at an intermediate speed which is easily within the range of major league pitchers. Note that the magnitude of the drag force is substantial—it actually exceeds the force due to gravity for ball speeds above about 95 mph.

The above discussion leads to a number of interesting observations. First, the raised stitches on a baseball have an important influence on its aerodynamic properties. Without them, the air drag acting on the ball at high speeds would increase substantially. Indeed, it seems unlikely that major league pitchers could throw 95 mph fastballs if baseballs were completely smooth. On the other hand, a scuffed-up baseball experiences even less air drag than a regulation ball. Presumably, such a ball can be thrown faster—which explains why balls are so regularly renewed in major league games.

Giordano[21] has developed the following useful formula which quantifies the

---

[21]*Computational physics*, N.J. Giordano, (Prentice-Hall, Upper Saddle River NJ, 1997).

drag force acting on a baseball:

$$\frac{\mathbf{f}_D}{m} = -F(v)\, v\, \mathbf{v}, \tag{3.40}$$

where

$$F(v) = 0.0039 + \frac{0.0058}{1 + \exp[(v - v_d)/\Delta]}. \tag{3.41}$$

Here, $v_d = 35\,\text{m/s}$ and $\Delta = 5\,\text{m/s}$.

3.13   The Magnus Force

We have not yet explained how a baseball is able to *curve* through the air. Physicists in the last century could not account for this effect, and actually tried to dismiss it as an "optical illusion." It turns out that this strange phenomenon is associated with the fact that the balls thrown in major league games tend to *spin* fairly rapidly—typically, at 1500 rpm.

The origin of the force which makes a spinning baseball curve can readily be appreciated once we recall that the drag force acting on a baseball increases with increasing speed.[22] For a ball spinning about an axis perpendicular to its direction of travel, the speed of the ball, relative to the air, is *different* on opposite sides of the ball, as illustrated in Fig. 12. It can be seen, from the diagram, that the lower side of the ball has a larger speed relative to the air than the upper side. This results in a larger drag force acting on the lower surface of the ball than on the upper surface. If we think of drag forces as exerting a sort of pressure on the ball then we can readily appreciate that when the unequal drag forces acting on the ball's upper and lower surfaces are added together there is a component of the resultant force acting *upwards*. This force—which is known as the *Magnus force,* after the German physicist Heinrich Magnus, who first described it in 1853—is the dominant spin-dependent force acting on baseballs. The Magnus force can be written

$$\mathbf{f}_M = S(v)\, \boldsymbol{\omega} \times \mathbf{v}, \tag{3.42}$$

---

[22]Although the drag coefficient, C_D, decreases with increasing speed, the drag force, which is proportional to $C_D\, v^2$, always increases.
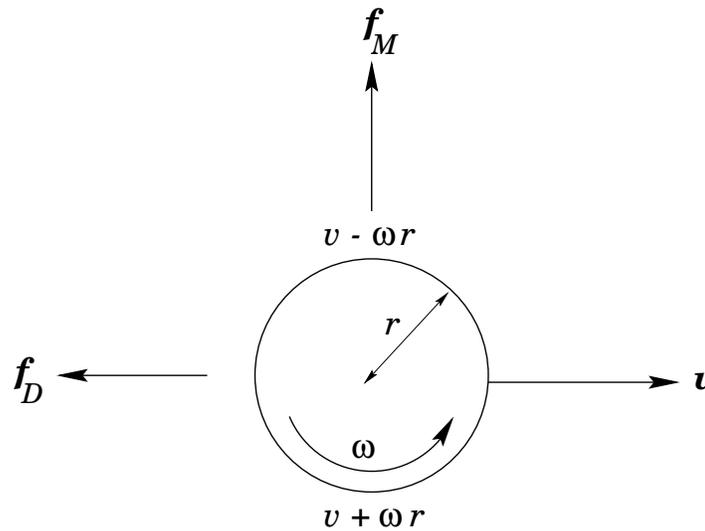
Figure 12: *Origin of the Magnus force for a ball of radius r, moving with speed v, and spinning with angular velocity ω about an axis perpendicular to its direction of motion.*

where ω is the angular velocity vector of the ball. According to Adair and Giordano, it is a fairly good approximation to take

$$B = \frac{S}{m} = 4.1 \times 10^{-4} \tag{3.43}$$

for baseballs. Note that B is a dimensionless quantity. The magnitude of the Magnus force is about one third of the force due to gravity for typical curveballs.

3.14   Simulations of Baseball Pitches

Let us adopt a set of coordinates such that x measures displacement from the pitcher to the hitter, y measures horizontal displacement (a displacement in the $+y$ direction corresponds to a displacement to the hitter's right-hand side), and z measures vertical displacement (a displacement in the $+z$ direction corresponds to an upward displacement). Using these coordinates, the equations of motion of a baseball can be written as the following set of coupled first-order o.d.e.s:

$$\frac{dx}{dt} = v_x, \tag{3.44}$$

$$\frac{dy}{dt} = v_y, \tag{3.45}$$

$$\frac{dz}{dt} = v_z, \tag{3.46}$$

$$\frac{dv_x}{dt} = -F(v)\,v\,v_x + B\,\omega\,(v_z\,\sin\phi - v_y\,\cos\phi), \tag{3.47}$$

$$\frac{dv_y}{dt} = -F(v)\,v\,v_y + B\,\omega\,v_x\,\cos\phi, \tag{3.48}$$

$$\frac{dv_z}{dt} = -g - F(v)\,v\,v_z - B\,\omega\,v_x\,\sin\phi. \tag{3.49}$$

Here, the ball's angular velocity vector has been written $\omega = \omega\,(0, \sin\phi, \cos\phi)$. Appropriate boundary conditions at $t = 0$ are:

$$x(t=0) = 0, \tag{3.50}$$

$$y(t=0) = 0, \tag{3.51}$$

$$z(t=0) = 0, \tag{3.52}$$

$$v_x(t=0) = v_0\,\cos\theta \tag{3.53}$$

$$v_y(t=0) = 0, \tag{3.54}$$

$$v_z(t=0) = v_0\,\sin\theta, \tag{3.55}$$

where v_0 is the initial speed of the pitch, and θ is its initial angle of elevation. Note that, in writing the above equations, we are neglecting any decrease in the ball's rate of spin as it moves towards the hitter.

The above set of equations have been solved numerically using a fixed step-length, fourth-order Runge-Kutta method. The step-length, h, is conveniently expressed as a fraction of the pitch's estimated time-of-flight, $T = l/v_0$, where $l = 18.44\,\text{m}$ is the horizontal distance between the pitcher and the hitter. Thus, 1/h is approximately the number of steps used to integrate the trajectory.

It is helpful, at this stage, to describe the conventional classification of baseball pitches in terms of the direction of the ball's axis of spin. Figure 13 shows the direction of rotation of various different pitches thrown by a right-handed pitcher, as seen by the hitter. Obviously, the directions are reversed for the corresponding pitches thrown by a left-handed pitcher. Note, from Eq. (3.42), that the arrows in Fig. 13 show both the direction of the ball's spin and the direction of the Magnus
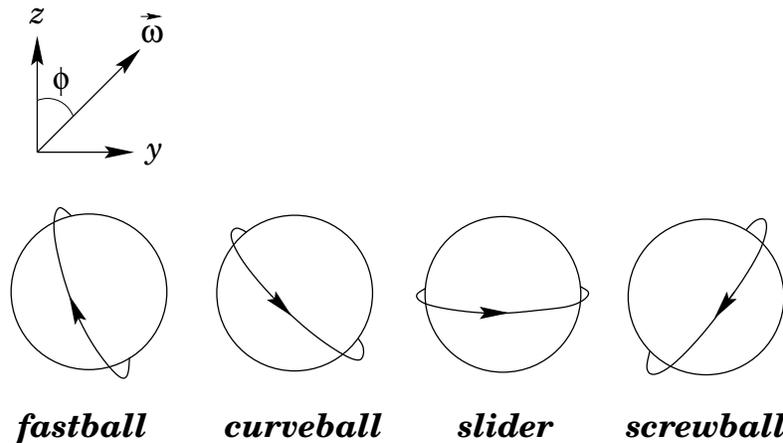
Figure 13: *Rotation direction, as seen by hitter, for various pitches thrown by a right-handed pitcher. The arrow shows the direction of rotation, which is also the direction of the Magnus force. From* The physics of baseball, *R.K. Adair (Harper & Row, New York NY, 1990).*

force.

Figure 14 shows the numerical trajectory of a "slider" delivered by a right-handed pitcher. The ball is thrown such that its axis of rotation points vertically upwards, which is the direction of spin generated by the natural clockwise (seen from below) rotation of a right-handed pitcher's wrist. The associated Magnus force causes the ball to curve sideways, *away* from a right-handed hitter (*i.e.*, in the $+y$-direction). As can be seen, from the figure, the sideways displacement for an 85 mph pitch spinning at 1800 rpm is over 1 foot. Of course, a slider delivered by a left-handed pitcher would curve *towards* a right-handed hitter. It turns out that pitches which curve towards a hitter are far harder to hit than pitches which curve away. Thus, a right-handed hitter is at a distinct disadvantage when facing a left-handed pitcher, and *vice versa*. A lot of strategy in baseball games is associated with teams trying to match the handedness of hitters and pitchers so as to gain an advantage over their opponents.

Figure 14 shows the numerical trajectory of a "curveball" delivered by a right-handed pitcher. The ball is thrown such that its axis of rotation, as seen by the hitter, points upwards, but also *tilts* to the right. The hand action associated with throwing a curveball is actually somewhat more natural than that associated with a slider. The Magnus force acting on a curveball causes the ball to deviate both
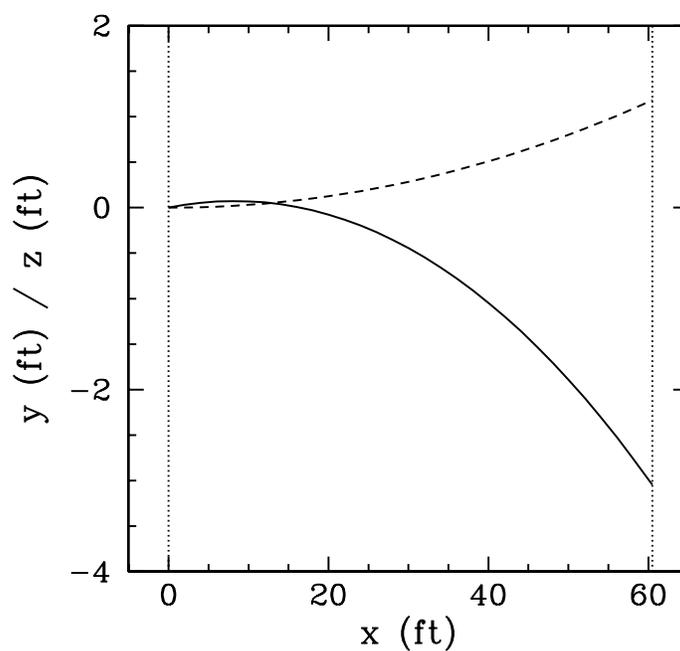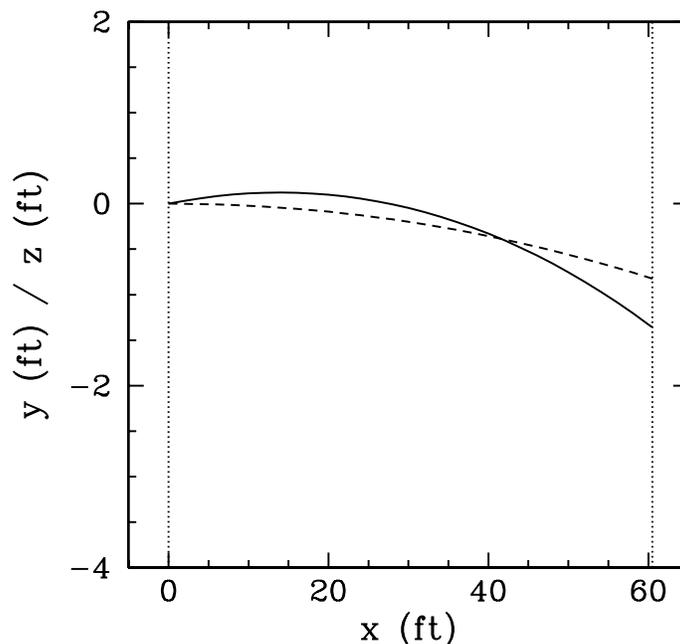
Figure 14: *Numerical trajectory of a slider delivered by a right-handed pitcher. The solid and dashed curves show the vertical and horizontal displacements of the ball, respectively. The parameters for this pitch are $v_0 = 85\,mph$, $\theta = 1°$, $\omega = 1800\,rpm$, $\phi = 0°$, and $\mathtt{h} = 1 \times 10^{-4}$. The ball passes over the plate at 76 $\mathtt{mph}$ about 0.52 seconds after it is released by the pitcher.*
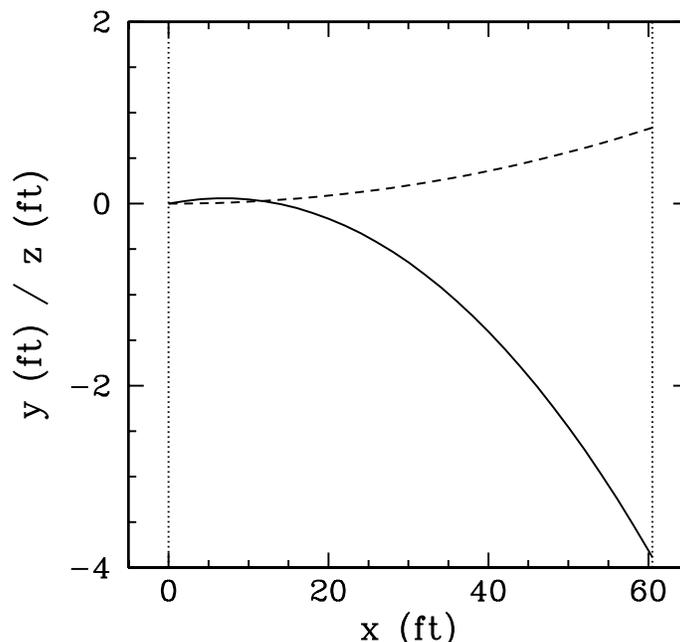
Figure 15: *Numerical trajectory of a curveball delivered by a right-handed pitcher. The solid and dashed curves show the vertical and horizontal displacements of the ball, respectively. The parameters for this pitch are $v_0 = 85\,mph$, $\theta = 1°$, $\omega = 1800\,rpm$, $\phi = 45°$, and $h = 1 \times 10^{-4}$. The ball passes over the plate at* 76 mph *about* 0.52 *seconds after it is released by the pitcher.*

sideways and *downwards*. Thus, although a curveball generally does not move laterally as far as a slider, it dips unusually rapidly—which makes it difficult to hit. The anomalously large dip of a typical curveball is apparent from a comparison of Figs. 14 and 15.

As we have already mentioned, a pitch which curves towards a hitter is harder to hit than one which curves away. It is actually possible for a right-handed pitcher to throw an inward curving pitch to a right-handed hitter. Such a pitch is known as a "screwball." Unfortunately, throwing screwballs involves a completely unnatural wrist rotation which is extremely difficult to master. Figure 16 shows the numerical trajectory of a screwball delivered by a right-handed pitcher. The ball is thrown such that its axis of rotation, as seen by the hitter, points upwards and tilts to the *left*. Note that the pitch dips rapidly—like a curveball—but has a sideways displacement in the *opposite* direction to normal.

Probably the most effective pitch in baseball is the so-called "fastball." As the
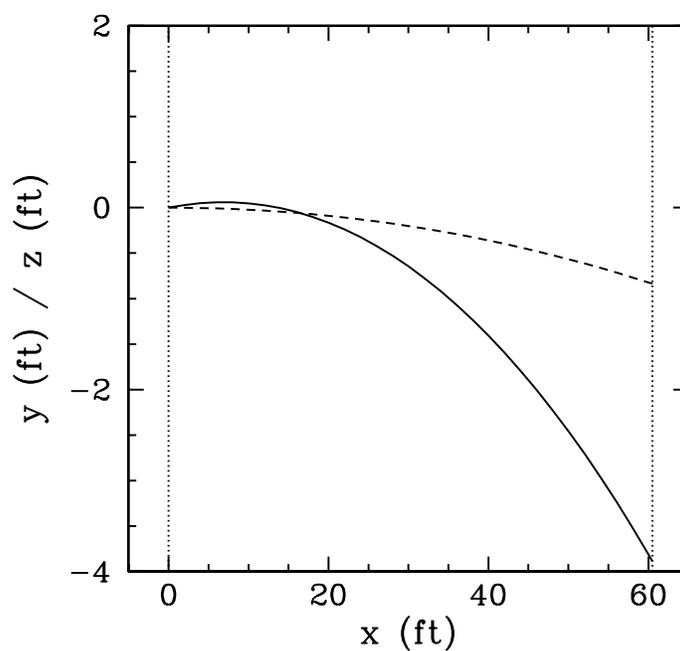
Figure 16: *Numerical trajectory of a screwball delivered by a right-handed pitcher. The solid and dashed curves show the vertical and horizontal displacements of the ball, respectively. The parameters for this pitch are $v_0 = 85\,mph$, $\theta = 1°$, $\omega = 1800\,rpm$, $\phi = 135°$, and $h = 1 \times 10^{-4}$. The ball passes over the plate at $76\,mph$ about 0.52 seconds after it is released by the pitcher.*

Figure 17: *Numerical trajectory of a fastball delivered by a right-handed pitcher. The solid and dashed curves show the vertical and horizontal displacements of the ball, respectively. The parameters for this pitch are $v_0 = 95\,mph$, $\theta = 1°$, $\omega = 1800\,rpm$, $\phi = 225°$, and $h = 1 \times 10^{-4}$. The ball passes over the plate at 86 mph about 0.46 seconds after it is released by the pitcher.*

name suggests, a fastball is thrown extremely rapidly—typically, at 95 mph. The natural hand action associated with throwing this type of pitch tends to impart a significant amount of *backspin* to the ball. Thus, the Magnus force acting on a fastball has a large upwards component, slowing the ball's rate of fall. Figure 17 shows the numerical trajectory of a fastball delivered by a right-handed pitcher. The ball is thrown such that its axis of rotation, as seen by the hitter, points *downwards* and tilts to the left. Note that the pitch falls an unusually small distance. This is partly because the ball takes less time than normal to reach the hitter, but partly also because the Magnus force has a substantial upward component. This type of pitch is often called a "rising fastball," because hitters often claim that the ball rises as it moves towards them. Actually, this is an optical illusion created by the ball's smaller than expected rate of fall.

3.15   The Knuckleball

Probably the most entertaining pitch in baseball is the so-called "knuckleball."
Unlike the other types of pitch we have encountered, knuckleballs are low speed
pitches (typically, 65 mph) in which the ball is purposely thrown with as little spin
as possible. It turns out that knuckleballs are hard to hit because they have un-
stable trajectories which shift from side to side in a highly unpredictable manner.
How is this possible?

Suppose that a moving ball is not spinning at all, and is orientated such that
one of its stitches is exposed on one side, whereas the other side is smooth. It
follows, from Fig. 11, that the drag force on the smooth side of the ball is greater
than that on the stitch side. Hence, the ball experiences a lateral force in the
direction of the exposed stitch. Suppose, now, that the ball is rotating slowly as it
moves towards the hitter. As the ball moves forward its orientation changes, and
the exposed stitch shifts from side to side, giving rise to a lateral force which also
shifts from side to side. Of course, if the ball is rotating sufficiently rapidly then
the oscillations in the lateral force average out. However, for a slowly rotating
ball these oscillations can profoundly affect the ball's trajectory.

Watts and Sawyer[23] have performed wind tunnel measurements of the lateral
force acting on a baseball as a function of its angular orientation. Note that the
stitches on a baseball pass any given point *four* times for each complete revolution
of the ball about an axis passing through its centre. Hence, we expect the lateral
force to exhibit four maxima and four minima as the ball is rotated once. This is
exactly what Watts and Sawyer observed. For the case of a 65 mph knuckleball,
Giordano has extracted the following useful expression for the lateral force, as a
function of angular orientation, φ, from Watts and Sawyer's data:

$$\frac{f_y}{m\,g} = G(\varphi) = 0.5\left[\sin(4\varphi) - 0.25\,\sin(8\varphi) + 0.08\,\sin(12\varphi) - 0.025\,\sin(16\varphi)\right].$$

$$(3.56)$$

The function $G(\varphi)$ is plotted in Fig. 18. Note the very rapid changes in this
function in certain narrow ranges of φ.

---

[23]*Aerodynamics of a knuckleball*, R.G. Watts, and E. Sawyer, Am. J. of Phys. **43**, 960 (1975).
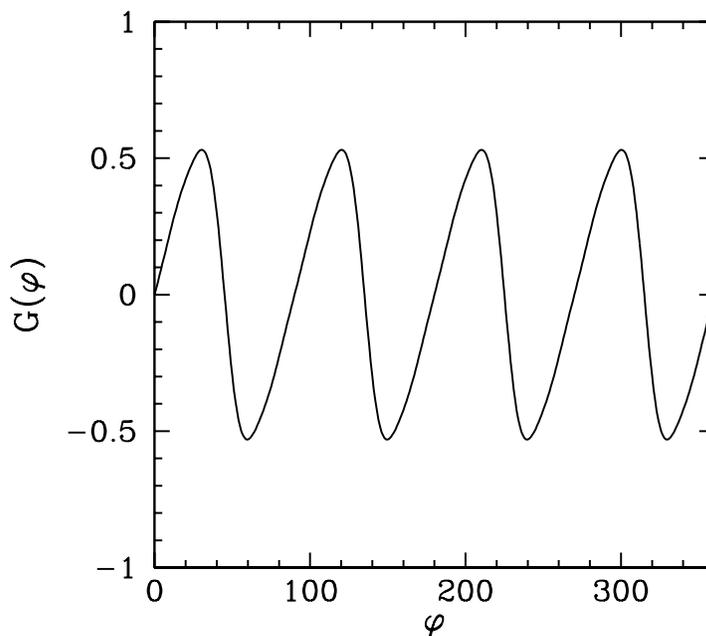
Figure 18: *The lateral force function,* $G(\varphi)$, *for a 65 mph knuckleball.*

Using the above expression, the equations of motion of a knuckleball can be written as the following set of coupled first-order o.d.e.s:

$$\frac{dx}{dt} = v_x, \tag{3.57}$$

$$\frac{dy}{dt} = v_y, \tag{3.58}$$

$$\frac{dz}{dt} = v_z, \tag{3.59}$$

$$\frac{dv_x}{dt} = -F(v)\,v\,v_x, \tag{3.60}$$

$$\frac{dv_y}{dt} = -F(v)\,v\,v_y + g\,G(\varphi), \tag{3.61}$$

$$\frac{dv_z}{dt} = -g - F(v)\,v\,v_z. \tag{3.62}$$

$$\frac{d\varphi}{dt} = \omega. \tag{3.63}$$

Note that we have added an equation of motion for the angular orientation, φ,

of the ball, which is assumed to rotate at a fixed angular velocity, ω, about a vertical axis. Here, we are neglecting the Magnus force, which is expected to be negligibly small for a slowly spinning pitch. Appropriate boundary conditions at $t = 0$ are:

$$x(t = 0) = 0, \tag{3.64}$$
$$y(t = 0) = 0, \tag{3.65}$$
$$z(t = 0) = 0, \tag{3.66}$$
$$v_x(t = 0) = v_0 \cos\theta \tag{3.67}$$
$$v_y(t = 0) = 0, \tag{3.68}$$
$$v_z(t = 0) = v_0 \sin\theta, \tag{3.69}$$
$$\varphi(t = 0) = \varphi_0, \tag{3.70}$$

where v_0 is the initial speed of the pitch, θ is the pitch's initial angle of elevation, and φ_0 is the ball's initial angular orientation.

The above set of equations have been solved numerically using a fixed step-length, fourth-order Runge-Kutta method. The step-length, `h`, is conveniently expressed as a fraction of the pitch's estimated time-of-flight, $T = l/v_0$, where $l = 18.44\,\text{m}$ is the horizontal distance between the pitcher and the hitter. Thus, `1/h` is approximately the number of steps used to integrate the trajectory.

Figure 19 shows the lateral displacement of a set of four knuckleballs thrown with the same rate of spin, $\omega = 20\,\text{rpm}$, but starting with different angular orientations. Note the striking difference between the various trajectories shown in this figure. Clearly, even a fairly small change in the initial orientation of the ball translates into a large change in its subsequent trajectory through the air. For this reason, knuckleballs are extremely unpredictable—neither the pitcher, the hitter, nor the catcher can be really sure where one of these pitches is going to end up. Needless to say, baseball teams always put their best catcher behind the plate when a knuckleball pitcher is on the mound!

Figure 20 shows the lateral displacement of a knuckleball thrown with a somewhat higher rate of spin than those shown previously: *i.e.,* $\omega = 40\,\text{rpm}$. Note the
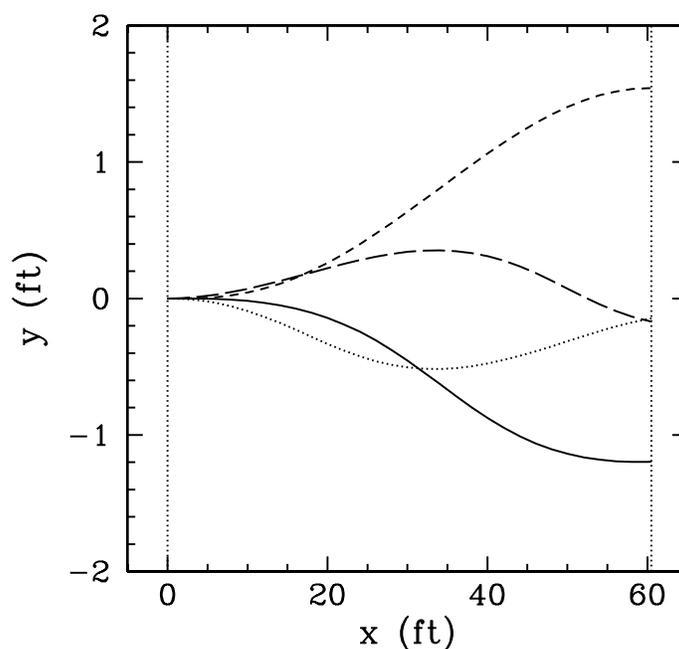
Figure 19: *Numerical trajectories of knuckleballs with the same angular velocity but different initial orientations. The solid, dotted, long-dashed, and short-dashed curves show the horizontal displacement of the ball for $\varphi_0 = 0°$, $22.5°$, $45°$, and $67.5°$, respectively. The other parameters for this pitch are $v_0 = 65\,mph$, $\theta = 4°$, $\omega = 20\,rpm$, and $\mathtt{h} = 1 \times 10^{-4}$. The ball passes over the plate at $56\,mph$ about 0.69 seconds after it is released by the pitcher. The ball rotates about $83°$ whilst it is in the air.*
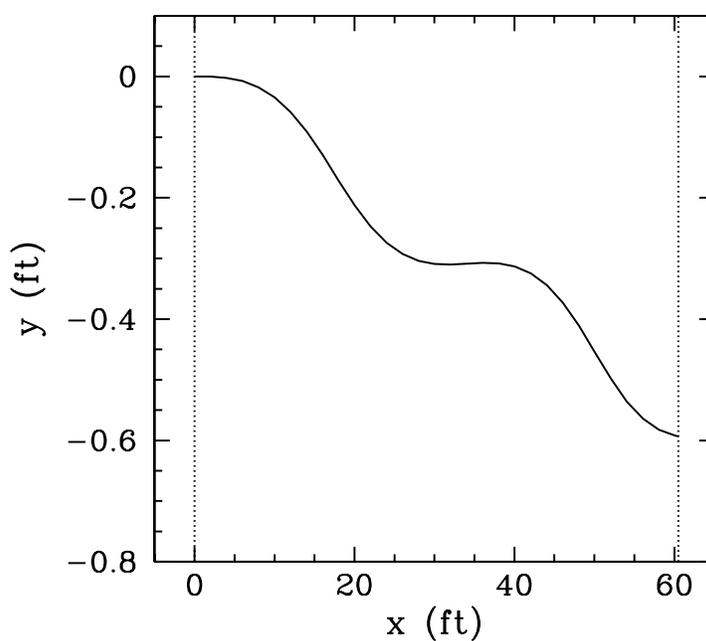
Figure 20: *Numerical trajectory of a knuckleball. The parameters for this pitch are* $v_0 = 65\,mph$, $\theta = 4°$, $\varphi_0 = 0°$, $\omega = 40\,rpm$, *and* $\mathtt{h} = 1 \times 10^{-4}$. *The ball passes over the plate at* $56\,\mathtt{mph}$ *about* 0.69 *seconds after it is released by the pitcher. The ball rotates about* $166°$ *whilst it is in the air.*

curious way in which the ball "dances" through the air. Given that the difference between a good hit and a bad hit can correspond to a shift in the strike point on the bat by as little as 1/4 of an inch, it must be quite a challenge to hit such a pitch well!

4   The Chaotic Pendulum

4.1   Introduction

Up to now, we have mostly dealt with problems which are capable of analytic solution (so that we can easily validate our numerical solutions). Let us now investigate a problem which is quite intractable analytically, and in which meaningful progress can only be made via numerical means.

Consider a simple pendulum consisting of a point mass m, at the end of a light rigid rod of length l, attached to a fixed frictionless pivot which allows the rod (and the mass) to move freely under gravity in the vertical plane. Such a pendulum is sketched in Fig. 21. Let us parameterize the instantaneous position of the pendulum via the angle θ the rod makes with the downward vertical. It is assumed that the pendulum is free to swing through 360 degrees. Hence, θ and $\theta + 2\pi$ both correspond to the same pendulum position.

The angular equation of motion of the pendulum is simply

$$m\,l\,\frac{d^2\theta}{dt^2} + m\,g\,\sin\theta = 0, \qquad (4.1)$$

where g is the downward acceleration due to gravity. Suppose that the pendulum is embedded in a viscous medium (*e.g.*, air). Let us assume that the viscous drag torque acting on the pendulum is governed by Stokes' law (see Sect. 3.12) and is, thus, directly proportional to the pendulum's instantaneous velocity. It follows that, in the presence of viscous drag, the above equation generalizes to

$$m\,l\,\frac{d^2\theta}{dt^2} + \nu\,\frac{d\theta}{dt} + m\,g\,\sin\theta = 0, \qquad (4.2)$$

where ν is a positive constant parameterizing the viscosity of the medium in question. Of course, viscous damping will eventually drain all energy from the pendulum, leaving it in a stationary state. In order to maintain the motion against viscosity, it is necessary to add some external driving. For the sake of simplicity, we choose a fixed amplitude, periodic drive (which could arise, for instance, via
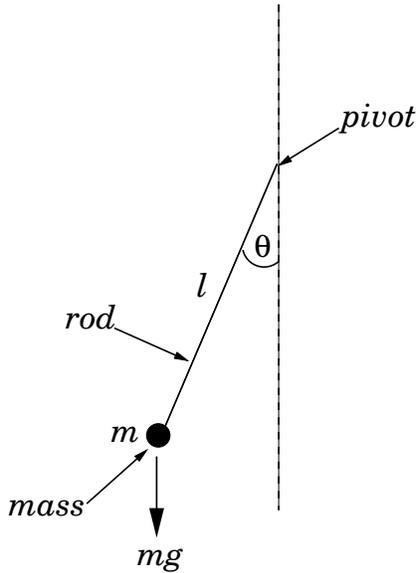
Figure 21: *A simple pendulum.*

periodic oscillations of the pendulum's pivot point). Thus, the final equation of motion of the pendulum is written

$$ m\,l\,\frac{d^2\theta}{dt^2} + \nu\,\frac{d\theta}{dt} + m\,g\,\sin\theta = A\,\cos\omega t, \tag{4.3} $$

where A and ω are constants parameterizing the amplitude and angular frequency of the external driving torque, respectively.

Let

$$ \omega_0 = \sqrt{\frac{g}{l}}. \tag{4.4} $$

Of course, we recognize ω_0 as the natural (angular) frequency of small amplitude oscillations of the pendulum. We can conveniently normalize the pendulum's equation of motion by writing,

$$ \hat{t} = \omega_0\,t, \tag{4.5} $$

$$ \hat{\omega} = \frac{\omega}{\omega_0}, \tag{4.6} $$

$$ Q = \frac{m\,g}{\omega_0\,\nu}, \tag{4.7} $$

$$ \hat{A} = \frac{A}{m\,g}, \tag{4.8} $$

141

in which case Eq. (4.3) becomes

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + \sin\theta = \hat{A}\,\cos\hat{\omega}\hat{t}. \tag{4.9}$$

From now on, the hats on normalized quantities will be omitted, for ease of notation. Note that, in normalized units, the natural frequency of small amplitude oscillations is *unity*. Moreover, Q is the familiar *quality-factor*—roughly, the number of oscillations of the undriven system which must elapse before its energy is significantly reduced via the action of viscosity. The quantity A is the amplitude of the external torque measured in units of the maximum possible gravitational torque. Finally, ω is the angular frequency of the external torque measured in units of the pendulum's natural frequency.

Equation (4.9) is clearly a second-order o.d.e. It can, therefore, also be written as two coupled first-order o.d.e.s:

$$\frac{d\theta}{dt} = v, \tag{4.10}$$

$$\frac{dv}{dt} = -\frac{v}{Q} - \sin\theta + A\,\cos\omega t. \tag{4.11}$$

4.2   Analytic Solution

Before attempting to solve the equations of motion of any dynamical system using a computer, we should, first, investigate them as thoroughly as possible via standard analytic techniques. Unfortunately, Eqs. (4.10) and (4.11) constitute a *non-linear* dynamical system—because of the presence of the $\sin\theta$ term on the right-hand side of Eq. (4.11). This system, like most non-linear systems, does not possess a simple analytic solution. Fortunately, however, if we restrict our attention to *small amplitude* oscillations, such that the approximation

$$\sin\theta \simeq \theta \tag{4.12}$$

is valid, then the system becomes *linear*, and can easily be solved analytically.

The linearized equations of motion of the pendulum take the form:

$$\frac{d\theta}{dt} = v, \tag{4.13}$$

$$\frac{dv}{dt} = -\frac{v}{Q} - \theta + A \cos \omega t. \tag{4.14}$$

Suppose that the pendulum's position, $\theta(0)$, and velocity, $v(0)$, are specified at time $t = 0$. As is well-known, in this case, the above equations of motion can be solved analytically to give:

$$\theta(t) = \left\{ \theta(0) - \frac{A(1 - \omega^2)}{[(1 - \omega^2)^2 + \omega^2/Q^2]} \right\} e^{-t/2Q} \cos \omega_* t$$

$$+ \frac{1}{\omega_*} \left\{ v(0) + \frac{\theta(0)}{2Q} - \frac{A(1 - 3\omega^2)/2Q}{[(1 - \omega^2)^2 + \omega^2/Q^2]} \right\} e^{-t/2Q} \sin \omega_* t$$

$$+ \frac{A\left[(1 - \omega^2) \cos \omega t + (\omega/Q) \sin \omega t\right]}{[(1 - \omega^2)^2 + \omega^2/Q^2]}, \tag{4.15}$$

$$v(t) = \left\{ v(0) - \frac{A\,\omega^2/Q}{[(1 - \omega^2)^2 + \omega^2/Q^2]} \right\} e^{-t/2Q} \cos \omega_* t$$

$$- \frac{1}{\omega_*} \left\{ \theta(0) + \frac{v(0)}{2Q} - \frac{A\left[(1 - \omega^2) - \omega^2/2Q^2\right]}{[(1 - \omega^2)^2 + \omega^2/Q^2]} \right\} e^{-t/2Q} \sin \omega_* t$$

$$+ \frac{\omega A\left[-(1 - \omega^2) \sin \omega t + (\omega/Q) \cos \omega t\right]}{[(1 - \omega^2)^2 + \omega^2/Q^2]}. \tag{4.16}$$

Here,

$$\omega_* = \sqrt{1 - \frac{1}{4Q^2}}, \tag{4.17}$$

and it is assumed that $Q > 1/2$. It can be seen that the above expressions for θ and v both consist of *three* terms. The first two terms clearly represent *transients*—they depend on the initial conditions, and *decay* exponentially in time. In fact, the e-folding time for the decay of these terms is $2\,Q$ (in normalized time units). The final term represents the *time-asymptotic motion* of the pendulum, and is manifestly *independent* of the initial conditions.
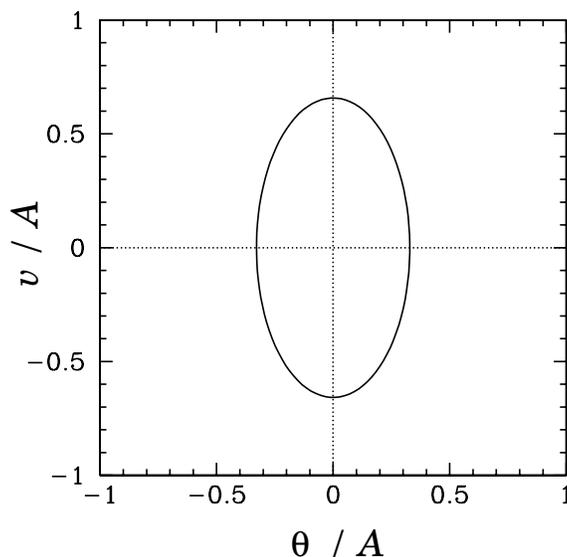
Figure 22: *A phase-space plot of the periodic attractor for a linear, damped, periodically driven, pendulum. Data calculated analytically for Q = 4 and ω = 2.*

It is often convenient to *visualize* the motion of a dynamical system as an orbit, or trajectory, in *phase-space*, which is defined as the space of all of the dynamical variables required to specify the instantaneous state of the system. For the case in hand, there are two dynamical variables, v and θ, and so phase-space corresponds to the θ-v plane. Note that each different point in this plane corresponds to a unique instantaneous state of the pendulum. [Strictly speaking, we should also consider t to be a dynamical variable, since it appears explicitly on the right-hand side of Eq. (4.11).]

It is clear, from Eqs. (4.15) and (4.16), that if we wait long enough for all of the transients to decay away then the motion of the pendulum settles down to the following simple orbit in phase-space:

$$\theta(t) \;=\; \frac{A\left[(1-\omega^2)\,\cos\omega t + (\omega/Q)\,\sin\omega t\right]}{[(1-\omega^2)^2 + \omega^2/Q^2]}, \tag{4.18}$$

$$v(t) \;=\; \frac{\omega A\left[-(1-\omega^2)\,\sin\omega t + (\omega/Q)\,\cos\omega t\right]}{[(1-\omega^2)^2 + \omega^2/Q^2]}. \tag{4.19}$$

144

This orbit traces out the closed curve

$$\left(\frac{\theta}{\tilde{A}}\right)^2 + \left(\frac{v}{\omega\,\tilde{A}}\right)^2 = 1, \tag{4.20}$$

in phase-space, where

$$\tilde{A} = \frac{A}{\sqrt{(1-\omega^2)^2 + \omega^2/Q^2}}. \tag{4.21}$$

As illustrated in Fig. 22, this curve is an *ellipse* whose principal axes are aligned with the v and θ axes. Observe that the curve is *closed*, which suggests that the associated motion is *periodic* in time. In fact, the motion repeats itself exactly every

$$\tau = \frac{2\pi}{\omega} \tag{4.22}$$

normalized time units. The maximum angular displacement of the pendulum from its undriven rest position ($\theta = 0$) is $\tilde{A}$. As illustrated in Fig. 23, the variation of $\tilde{A}$ with driving frequency ω [see Eq. (4.21)] displays all of the features of a classic resonance curve. The maximum amplitude of the driven oscillation is proportional to the quality-factor, Q, and is achieved when the driving frequency matches the natural frequency of the pendulum (*i.e.*, when $|\omega| = 1$). Moreover, the width of the resonance in ω-space is proportional to $1/Q$.

The phase-space curve shown in Fig. 22 is called a *periodic attractor*. It is termed an "attractor" because, irrespective of the initial conditions, the trajectory of the system in phase-space tends asymptotically to—in other words, is attracted to—this curve as $t \to \infty$. This gravitation of phase-space trajectories towards the attractor is illustrated in Figs. 24 and 25. Of course, the attractor is termed "periodic" because it corresponds to motion which is periodic in time.

Let us summarize our findings, so far. We have discovered that if a damped pendulum is subject to a low amplitude, periodic, drive then its *time-asymptotic* response (*i.e.*, its response after any transients have died away) is *periodic*, with the same period as the driving torque. Moreover, the response exhibits *resonant* behaviour as the driving frequency approaches the natural frequency of oscillation of the pendulum. The amplitude of the resonant response, as well as the width of the resonant window, is governed by the amount of damping in the system. After a little reflection, we can easily appreciate that all of these results are a
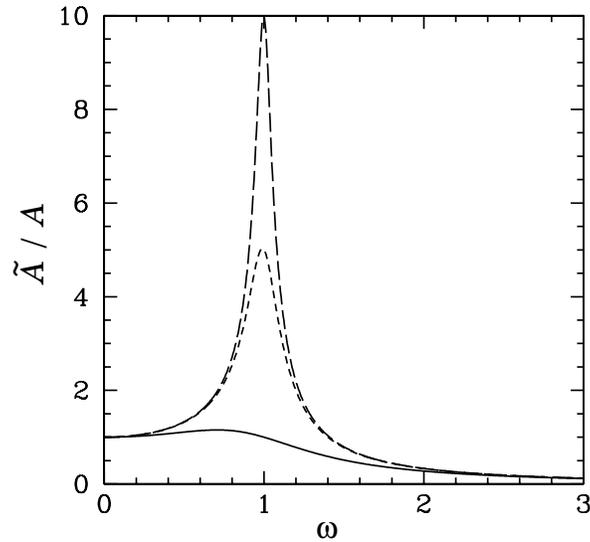
Figure 23: *The maximum angular displacement of a linear, damped, periodically driven, pendulum as a function of driving frequency. The solid curve corresponds to* Q = 1. *The short-dashed curve corresponds to* Q = 5. *The long-dashed curve corresponds to* Q = 10. *Analytic data.*
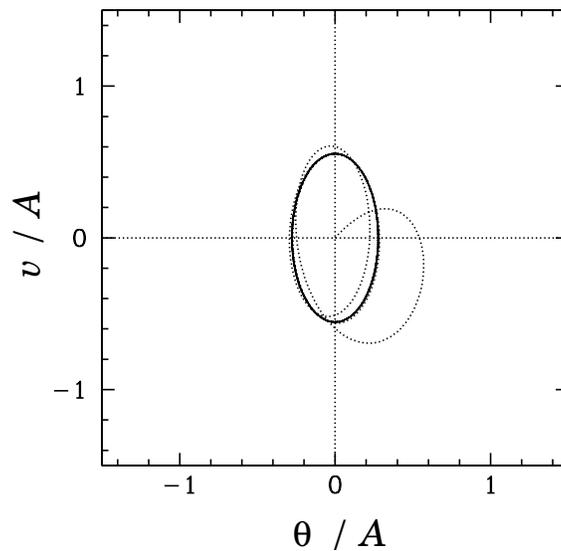
Figure 24: *The phase-space trajectory of a linear, damped, periodically driven, pendulum. Data calculated analytically for* Q = 1 *and* ω = 2. *Here,* $v(0)/A = 0$ *and* $\theta(0)/A = 0$.

Figure 25: *The phase-space trajectory of a linear, damped, periodically driven, pendulum. Data calculated analytically for* $Q = 1$ *and* $\omega = 2$*. Here,* $v(0)/A = 0.5$ *and* $\theta(0)/A = 0.5$*.*

direct consequence of the *linearity* of the pendulum's equations of motion in the low amplitude limit. In fact, it is easily demonstrated that the time-asymptotic response of *any* intrinsically stable linear system (with a discrete spectrum of normal modes) to a periodic drive is periodic, with the same period as the drive. Moreover, if the driving frequency approaches one of the natural frequencies of oscillation of the system then the response exhibits resonant behaviour. But, is this the only allowable time-asymptotic response of a dynamical system to a periodic drive? Most undergraduate students might be forgiven for answering this question in the affirmative. After all, the majority of undergraduate classical dynamics courses focus almost exclusively on linear systems. The correct answer, as we shall see, is no. The response of a *non-linear* system to a periodic drive is generally far more rich and diverse than simple periodic motion. Since the majority of naturally occurring dynamical systems are non-linear, it is clearly important that we gain a basic understanding of this phenomenon. Unfortunately, we cannot achieve this goal via a standard analytic approach—non-linear equations of motion generally do not possess simple analytic solutions. Instead, we must use *computers*. As an example, let us investigate the dynamics of a damped pendulum, subject to a periodic drive, with *no restrictions* on the amplitude of the

pendulum's motion.

4.3 Numerical Solution

In the following, we present numerical solutions of Eqs. (4.10) and (4.11) obtained using a fixed step-length, fourth-order, Runge-Kutta integration scheme. The step-length is conveniently parameterized by `Nacc`, which is defined as the number of time-steps taken by the integration scheme per period of the external drive.

4.4 Validation of Numerical Solutions

Before proceeding with our investigation, we must first convince ourselves that our numerical solutions are valid. Now, the usual method of validating a numerical solution is to look for some special limits of the input parameters for which analytic solutions are available, and then to test the numerical solution in one of these limits against the associated analytic solution.

One special limit of Eqs. (4.10) and (4.11) occurs when there is no viscous damping (*i.e.*, $Q \to \infty$) and no external driving (*i.e.*, $A \to 0$). In this case, we expect the normalized energy of the pendulum

$$\mathcal{E} = 1 + \frac{v^2}{2} - \cos\theta \tag{4.23}$$

to be a *constant* of the motion. Note that $\mathcal{E}$ is defined such that the energy is zero when the pendulum is in its stable equilibrium state (*i.e.*, at rest, pointing vertically downwards). Figure 26 shows $\mathcal{E}$ versus time, calculated numerically for an undamped, undriven, pendulum. Curves are plotted for various values of the parameter `Nacc`, which, in this special case, measures the number of time-steps taken by the integrator per (low amplitude) natural period of oscillation of the pendulum. It can be seen that for `Nacc` $= 12$ there is a strong spurious loss of energy, due to truncation error in the numerical integration scheme, which eventually drains all energy from the pendulum after about 2000 oscillations.
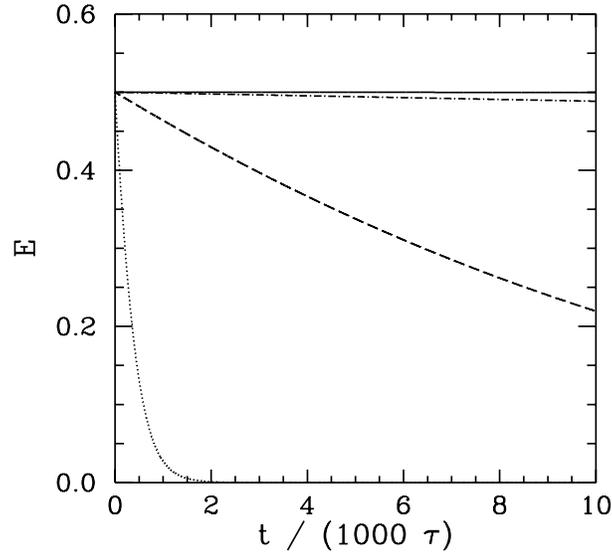
Figure 26: *The normalized energy $\mathcal{E}$ of an undamped, undriven, pendulum versus time (measured in natural periods of oscillation τ). Data calculated numerically for $Q = 10^{16}$, $A = 10^{-16}$, $\omega = 1$, $\theta(0) = 0$, and $v(0) = 1$. The dotted curve shows data for* Nacc $= 12$. *The dashed curve shows data for* Nacc $= 24$. *The dot-dashed curve shows data for* Nacc $= 48$. *Finally, the solid curve shows data for* Nacc $= 96$.

For Nacc $= 24$, the spurious energy loss is less severe, but, nevertheless, still causes a more than 50% reduction in pendulum energy after 10,000 oscillations. For Nacc $= 48$, the reduction in energy after 10,000 oscillations is only about 1%. Finally, for Nacc $= 96$, the reduction in energy after 10,000 oscillation is completely negligible. This test seems to indicate that when Nacc ≥ 100 our numerical solution describes the pendulum's motion to a high degree of precision for at least 10,000 oscillations.

Another special limit of Eqs. (4.10) and (4.11) occurs when these equations are *linearized* to give Eqs. (4.13) and (4.14). In this case, we expect

$$R = \sqrt{\theta^2 + (v/\omega)^2} \tag{4.24}$$

to be a constant of the motion, *after* all transients have died away (see Sect. 4.2). Figure 27 shows R versus time, calculated numerically, for a linearized, damped, periodically driven, pendulum. Curves are plotted for various values of the parameter Nacc, which measures the number of time-steps taken by the integrator per period of oscillation of the external drive. As Nacc increases, it can be seen
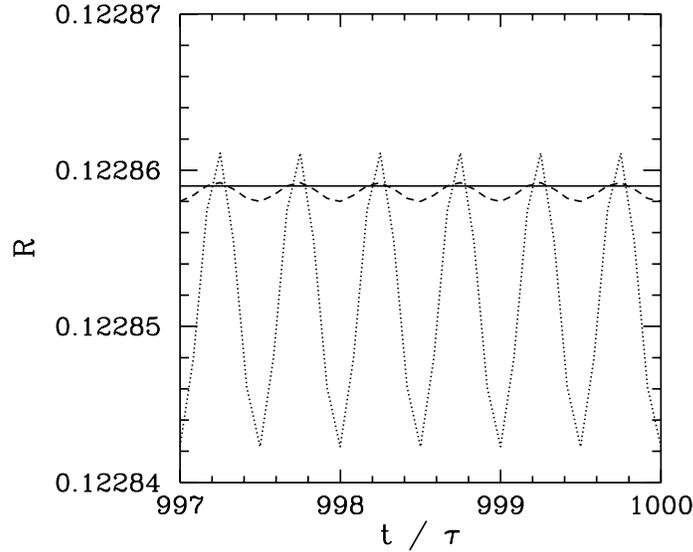
Figure 27: *The parameter* R *associated with a linearized, damped, periodically driven, pendulum versus time (measured in units of the period of oscillation τ of the external drive). Data calculated numerically for* $Q = 2$, $A = 1$, $\omega = 3$, $\theta(0) = 0$, *and* $v(0) = 0$. *The dotted curve shows data for* `Nacc` = 12. *The dashed curve shows data for* `Nacc` = 24. *The solid curve shows data for* `Nacc` = 48.

that the amplitude of the spurious oscillations in R, which are due to truncation error in the numerical integration scheme, decreases rapidly. Indeed, for `Nacc` ≥ 48 these oscillations become effectively undetectable. According to the analysis in Sect. 4.2, the parameter R should take the value

$$R = \frac{A}{\sqrt{(1 - \omega^2)^2 + \omega^2/Q^2}}. \tag{4.25}$$

Thus, for the case in hand (*i.e.*, $Q = 2, A = 1, \omega = 3$), we expect $R = 0.122859$. It can be seen that this prediction is borne out very accurately in Fig. 27. The above test essentially confirms our previous conclusion that when `Nacc` ≥ 100 our numerical solution matches pendulum's actual motion to a high degree of accuracy for many thousands of oscillation periods.
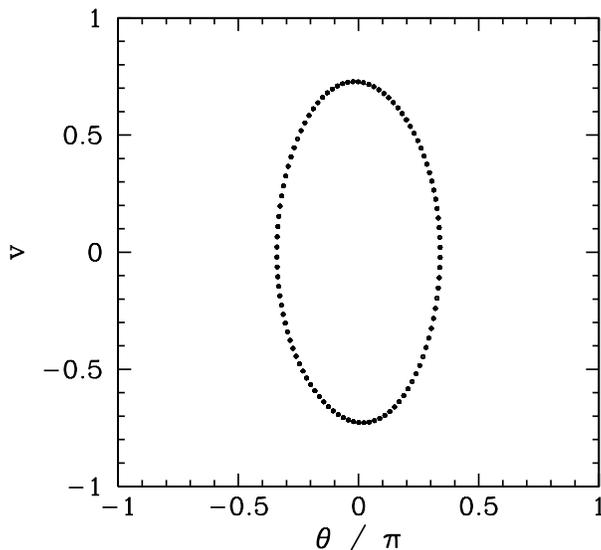
Figure 28: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* $Q = 0.5$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, *and* `Nacc` $= 100$.

4.5   The Poincaré Section

For the sake of definiteness, let us fix the normalized amplitude and frequency of the external drive to be $A = 1.5$ and $\omega = 2/3$, respectively.[24] Furthermore, let us investigate any changes which may develop in the nature of the pendulum's time-asymptotic motion as the quality-factor Q is varied. Of course, if Q is made sufficiently small (*i.e.*, if the pendulum is embedded in a sufficiently viscous medium) then we expect the amplitude of the pendulum's time-asymptotic motion to become low enough that the linear analysis outlined in Sect. 4.2 remains valid. Indeed, we expect non-linear effects to manifest themselves as Q is gradually made larger, and the amplitude of the pendulum's motion consequently increases to such an extent that the small angle approximation breaks down.

Figure 28 shows a time-asymptotic orbit in phase-space calculated numerically for a case where Q is sufficiently small (*i.e.*, $Q = 1/2$) that the small angle approximation holds reasonably well. Not surprisingly, the orbit is very similar to the analytic orbits described in Sect. 4.2. The fact that the orbit consists of a *single* loop, and forms a *closed* curve in phase-space, strongly suggests that the

[24]G.L. Baker, *Control of the chaotic driven pendulum*, Am. J. Phys. **63**, 832 (1995).

corresponding motion is periodic with the same period as the external drive—we term this type of motion *period-1* motion. More generally, period-n motion consists of motion which repeats itself exactly every n periods of the external drive (and, obviously, does not repeat itself on any time-scale less than n periods). Of course, period-1 motion is the only allowed time-asymptotic motion in the small angle limit.

It would certainly be helpful to possess a graphical test for period-n motion. In fact, such a test was developed more than a hundred years ago by the French mathematician Henry Poincaré—nowadays, it is called a *Poincaré section* in his honour. The idea of a Poincaré section, as applied to a periodically driven pendulum, is very simple. As before, we calculate the time-asymptotic motion of the pendulum, and visualize it as a series of points in θ-v phase-space. However, we only plot *one point per period* of the external drive. To be more exact, we only plot a point when

$$\omega\, t = \phi + k\, 2\pi \tag{4.26}$$

where k is any integer, and ϕ is referred to as the *Poincaré phase*. For period-1 motion, in which the motion repeats itself exactly every period of the external drive, we expect the Poincaré section to consist of only *one* point in phase-space (*i.e.*, we expect all of the points to plot on top of one another). Likewise, for period-2 motion, in which the motion repeats itself exactly every two periods of the external drive, we expect the Poincaré section to consist of *two* points in phase-space (*i.e.*, we expect alternating points to plot on top of one another). Finally, for period-n motion we expect the Poincaré section to consist of n points in phase-space.

Figure 29 displays the Poincaré section of the orbit shown in Fig. 28. The fact that the section consists of a single point confirms that the motion displayed in Fig. 28 is indeed period-1 motion.

4.6 Spatial Symmetry Breaking

Suppose that we now gradually increase the quality-factor Q. What happens to the simple orbit shown in Fig. 28? It turns out that, at first, nothing particularly
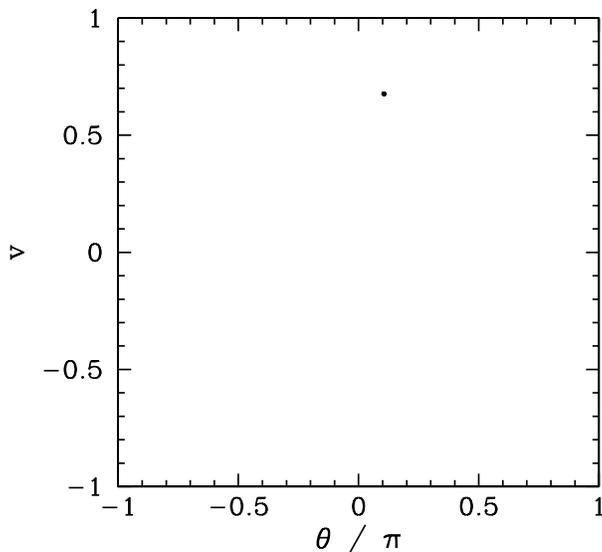
Figure 29: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 0.5$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$.
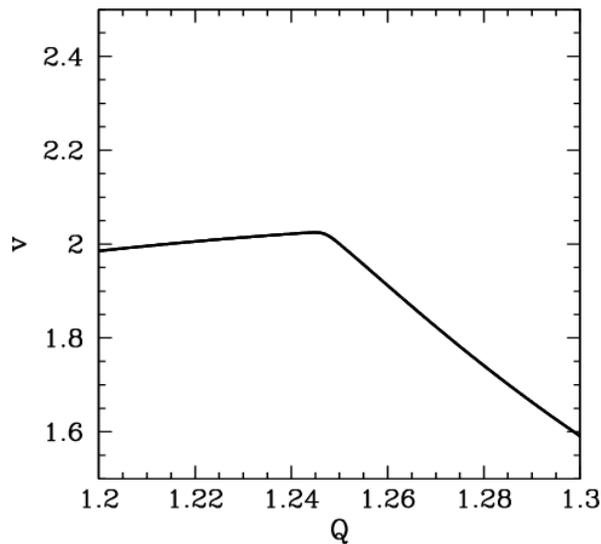
Figure 30: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor* Q. *Data calculated numerically for* $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$.

exciting happens. The size of the orbit gradually increases, indicating a corresponding increase in the amplitude of the pendulum's motion, but the general nature of the motion remains unchanged. However, something interesting does occur when Q is increased beyond about 1.2. Figure 30 shows the v-coordinate of the orbit's Poincaré section plotted against Q in the range 1.2 and 1.3. Note the sharp downturn in the curve at $Q \simeq 1.245$. What does this signify? Well, Fig. 31 shows the time-asymptotic phase-space orbit just before the downturn (*i.e.*, at $Q = 1.24$), and Fig. 32 shows the orbit somewhat after the downturn (*i.e.*, at $Q = 1.30$). It is clear that the downturn is associated with a sudden change in the nature of the pendulum's time-asymptotic phase-space orbit. Prior to the downturn, the orbit spends as much time in the region $\theta < 0$ as in the region $\theta > 0$. However, after the downturn the orbit spends the majority of its time in the region $\theta < 0$. In other words, after the downturn the pendulum bob favours the region to the *left* of the pendulum's vertical. This is somewhat surprising, since there is nothing in the pendulum's equations of motion which differentiates between the regions to the left and to the right of the vertical. We refer to a solution of this type—which fails to realize the full symmetry of the dynamical system in question—as a *symmetry breaking* solution. In this case, because the particular symmetry which is broken is a *spatial* symmetry, we refer to the process by which the symmetry breaking solution suddenly appears, as the control parameter Q is adjusted, as *spatial symmetry breaking*. Needless to say, spatial symmetry breaking is an intrinsically *non-linear* process—it cannot take place in dynamical systems possessing linear equations of motion.

It stands to reason that since the pendulum's equations of motion favour neither the left nor the right then the left-favouring orbit pictured in Fig. 32 must be accompanied by a mirror image right-favouring orbit. How do we obtain this mirror image orbit? It turns out that all we have to do is keep the physics parameters Q, A, and ω fixed, but *change the initial conditions* $\theta(0)$ and $v(0)$. Figure 33 shows a time-asymptotic phase-space orbit calculated with the same physics parameters used in Fig. 32, but with the initial conditions $\theta(0) = 0$ and $v(0) = -3$, instead of $\theta(0) = 0$ and $v(0) = 0$. It can be seen that the orbit is indeed the mirror image of that pictured in Fig. 32.

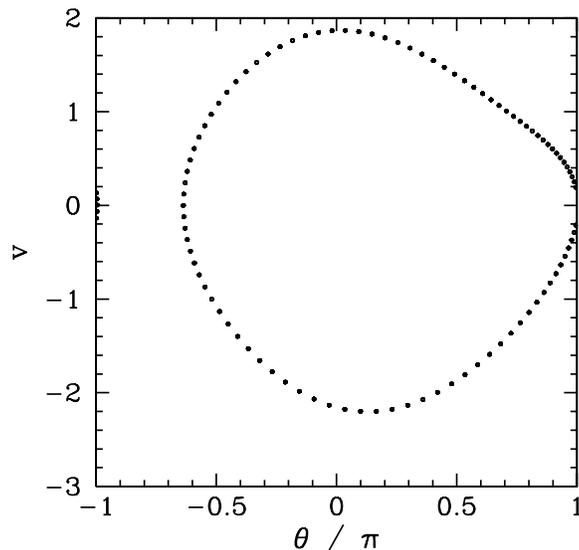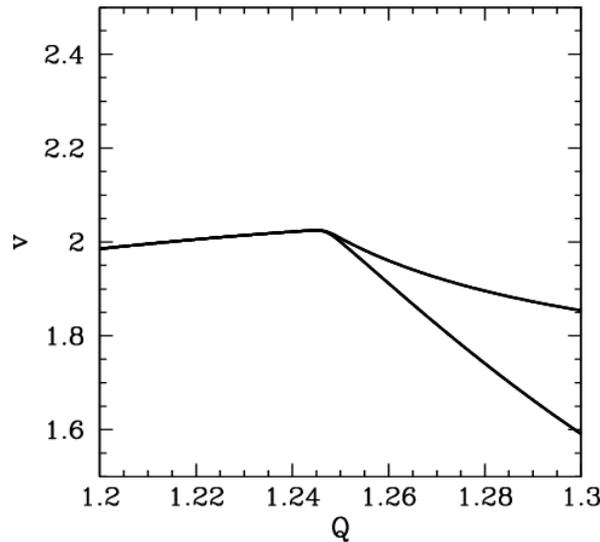Figure 34 shows the v-coordinate of the Poincaré section of a time-asymptotic
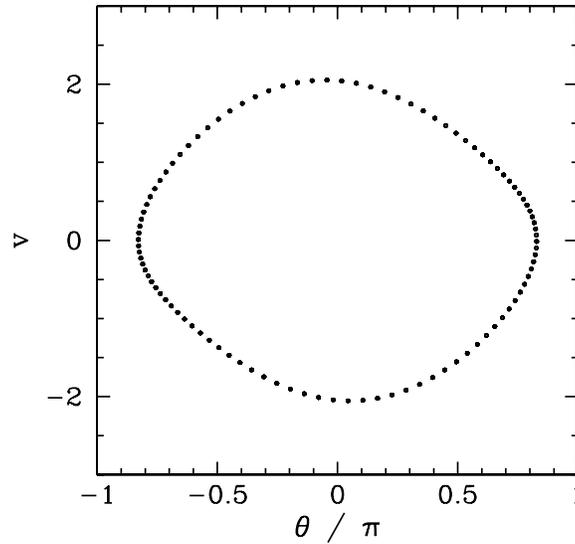
Figure 31: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* $Q = 1.24$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, *and* `Nacc` $= 100$.
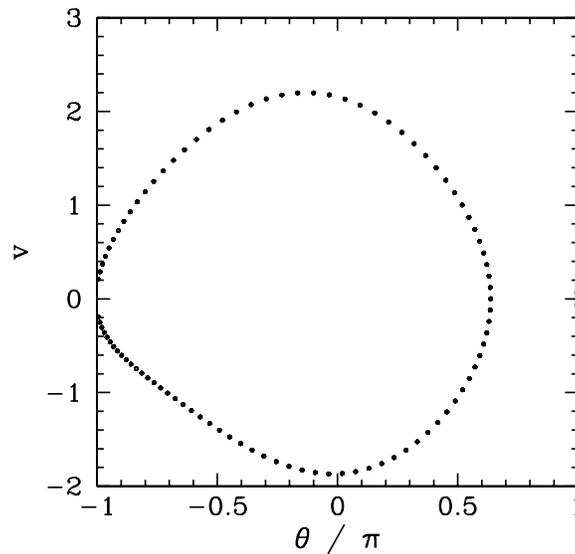
Figure 32: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* $Q = 1.30$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, *and* `Nacc` $= 100$.

Figure 33: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* $Q = 1.30$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = -3$, *and* `Nacc` $= 100$.

orbit, calculated with the same physics parameters used in Fig. 30, versus Q in the range 1.2 and 1.3. Data is shown for the two sets of initial conditions discussed above. The figure is interpreted as follows. When Q is less than a critical value, which is about 1.245, then the two sets of initial conditions lead to motions which converge on the *same*, left-right symmetric, period-1 attractor. However, once Q exceeds the critical value then the attractor *bifurcates* into two asymmetric, mirror image, period-1 attractors. Obviously, the bifurcation is indicated by the forking of the curve shown in Fig. 34. The lower and upper branches correspond to the left- and right-favouring attractors, respectively.

Spontaneous symmetry breaking, which is the fundamental non-linear process illustrated in the above discussion, plays an important role in many areas of physics. For instance, symmetry breaking gives mass to elementary particles in the unified theory of electromagnetic and weak interactions.[25] Symmetry breaking also plays a pivotal role in the so-called "inflation" theory of the expansion of the early universe.[26]

---

[25]E.S. Albers and B.W. Lee, Phys. Rep. 9**C**, 1 (1973).

[26]P. Coles, and F. Lucchin, *Cosmology: The origin and evolution of cosmic structure*, (J. Wiley & Sons, Chichester UK, 1995).

Figure 34: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor Q. Data calculated numerically for A = 1.5, ω = 2/3, and* Nacc *= 100. Data is shown for two sets of initial conditions: θ(0) = 0 and v(0) = 0 (lower branch); and θ(0) = 0 and v(0) = −3 (upper branch).*

4.7   Basins of Attraction

We have seen that when $Q = 1.3$, $A = 1.5$, and $\omega = 2/3$ there are two co-existing period-1 attractors in θ–v phase-space. The time-asymptotic trajectory of the pendulum through phase-space converges on one or other of these attractors depending on the initial conditions: *i.e.*, depending on the values of $\theta(0)$ and $v(0)$. Let us define the *basin of attraction* of a given attractor as the locus of all points in the $\theta(0)$–$v(0)$ plane which lead to motion which ultimately converges on that attractor. We have seen that in the low-amplitude (*i.e.*, linear) limit (see Sect. 4.2) there is only a single period-1 attractor in phase-space, and all possible initial conditions lead to motion which converges on this attractor. In other words, the basin of attraction for the low-amplitude attractor constitutes the entire $\theta(0)$–$v(0)$ plane. The present case, in which there are *two* co-existing attractors in phase-space, is somewhat more complicated.

Figure 35 shows the basins of attraction, in $\theta(0)$–$v(0)$ space, of the asymmetric, mirror image, attractors pictured in Figs. 32 and 33. The basin of attraction of
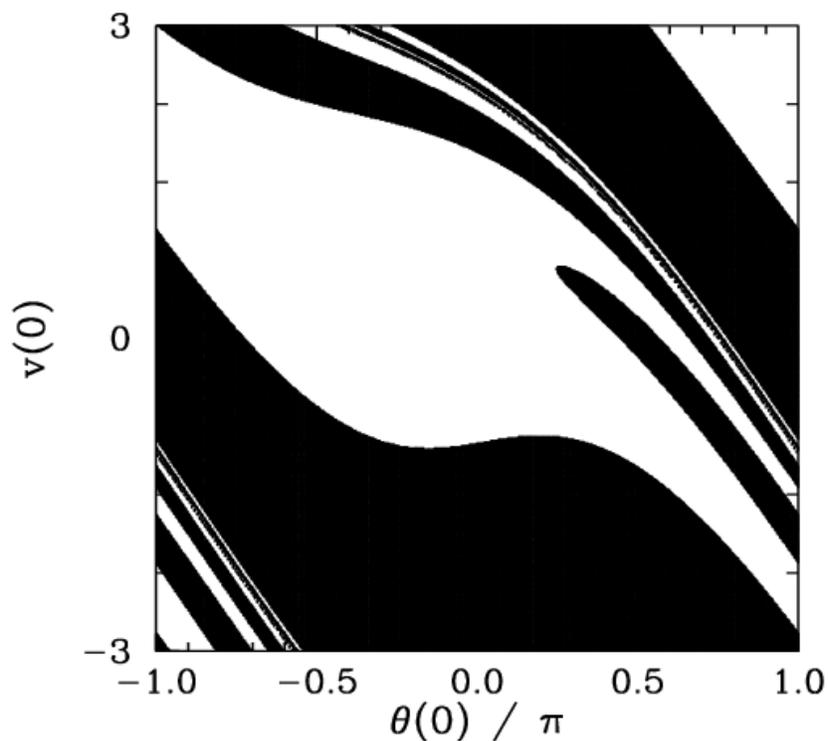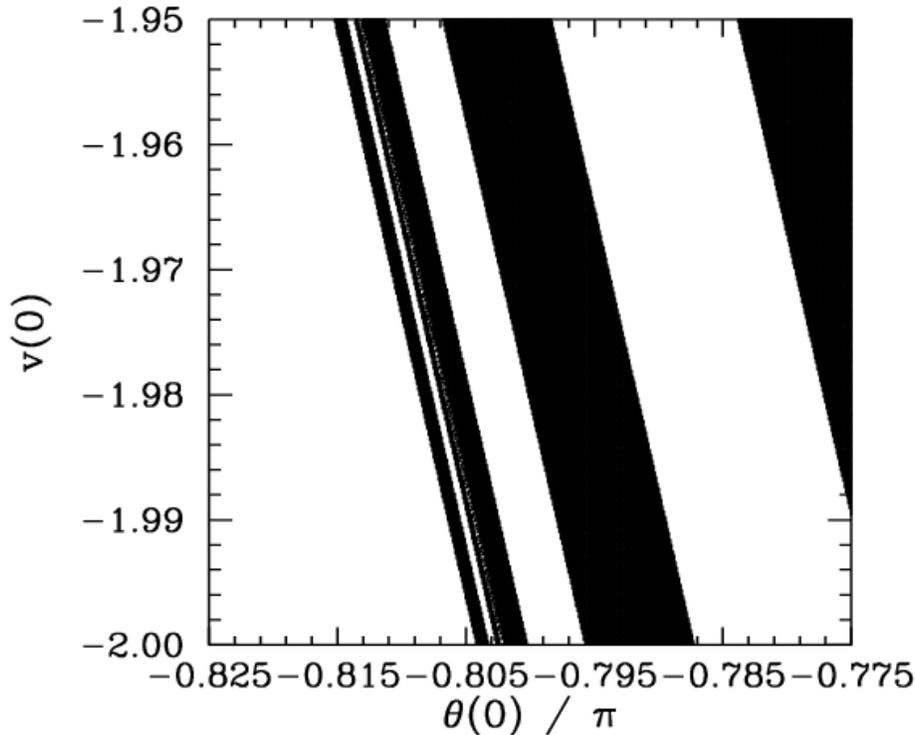
Figure 35: *The basins of attraction for the asymmetric, mirror image, attractors pictured in Figs. 32 and 33. Regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the left-favouring attractor shown in Fig. 32 are coloured white: regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the right-favouring attractor shown in Fig. 33 are coloured black. Data calculated numerically for $Q = 1.3$, $A = 1.5$, $\omega = 2/3$, Nacc $= 100$, and $\phi = 0$.*
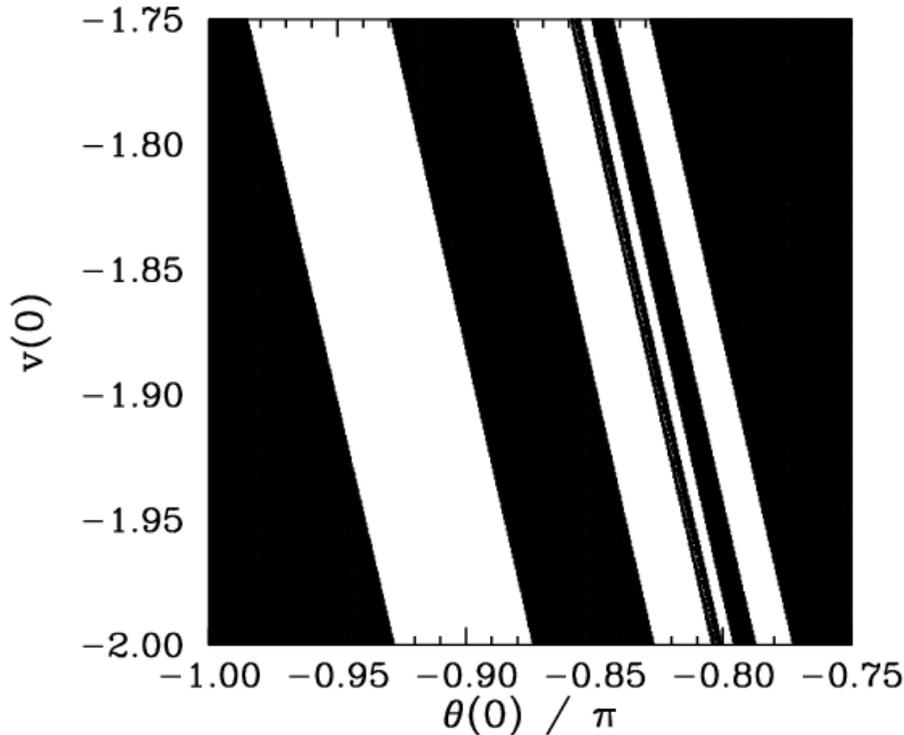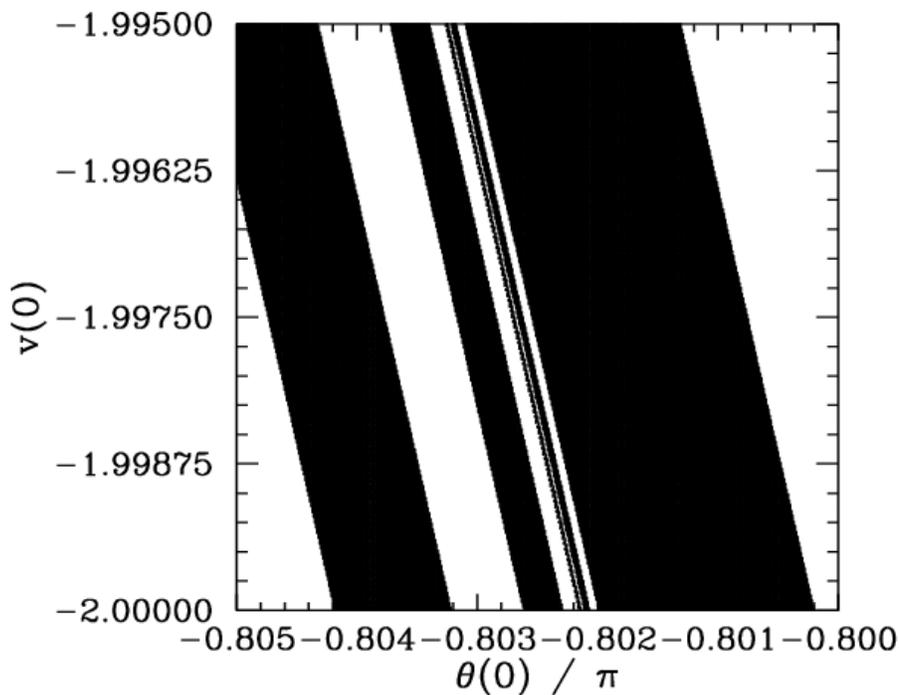
Figure 36: *Detail of the basins of attraction for the asymmetric, mirror image, attractors pictured in Figs. 32 and 33. Regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the left-favouring attractor shown in Fig. 32 are coloured white: regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the right-favouring attractor shown in Fig. 33 are coloured black. Data calculated numerically for* $Q = 1.3$, $A = 1.5$, $\omega = 2/3$, `Nacc` $= 100$, *and* $\phi = 0$.

the left-favoring attractor shown in Fig. 32 is coloured black, whereas the basin of attraction of the right-favoring attractor shown in Fig. 33 is coloured white. It can be seen that the two basins form a complicated interlocking pattern. Since we can identify the angles π and $-\pi$, the right-hand edge of the pattern connects smoothly with its left-hand edge. In fact, we can think of the pattern as existing on the surface of a *cylinder*.

Suppose that we take a diagonal from the bottom left-hand corner of Fig. 35 to its top right-hand corner. This diagonal is intersected by a number of black bands of varying thickness. Observe that the two narrowest bands (*i.e.*, the fourth band from the bottom left-hand corner and the second band from the upper right-hand corner) both exhibit structure which is not very well resolved in the present

Figure 37: *Detail of the basins of attraction for the asymmetric, mirror image, attractors pictured in Figs. 32 and 33. Regions of θ(0)–v(0) space which lead to motion converging on the left-favouring attractor shown in Fig. 32 are coloured white: regions of θ(0)–v(0) space which lead to motion converging on the right-favouring attractor shown in Fig. 33 are coloured black. Data calculated numerically for Q = 1.3, A = 1.5, ω = 2/3, Nacc = 100, and φ = 0.*

picture.

Figure 36 is a blow-up of a region close to the lower left-hand corner of Fig. 35. It can be seen that the unresolved band in the latter figure (*i.e.*, the second and third bands from the right-hand side in the former figure) actually consists of a closely spaced *pair* of bands. Note, however, that the narrower of these two bands exhibits structure which is not very well resolved in the present picture.

Figure 37 is a blow-up of a region of Fig. 36. It can be seen that the unresolved band in the latter figure (*i.e.*, the first and second bands from the left-hand side in the former figure) actually consists of a closely spaced *pair* of bands. Note, however, that the broader of these two bands exhibits structure which is not very well resolved in the present picture.

Figure 38: *Detail of the basins of attraction for the asymmetric, mirror image, attractors pictured in Figs. 32 and 33. Regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the left-favouring attractor shown in Fig. 32 are coloured white: regions of $\theta(0)$–$v(0)$ space which lead to motion converging on the right-favouring attractor shown in Fig. 33 are coloured black. Data calculated numerically for $Q = 1.3$, $A = 1.5$, $\omega = 2/3$, Nacc $= 100$, and $\phi = 0$.*

Figure 38 is a blow-up of a region of Fig. 37. It can be seen that the unresolved band in the latter figure (*i.e.*, the first, second, and third bands from the right-hand side in the former figure) actually consists of a closely spaced *triplet* of bands. Note, however, that the narrowest of these bands exhibits structure which is not very well resolved in the present picture.

It should be clear, by this stage, that no matter how closely we look at Fig. 35 we are going to find structure which we cannot resolve. In other words, the separatrix between the two basins of attraction shown in this figure is a curve which exhibits structure *at all scales*. Mathematicians have a special term for such a curve—they call it a *fractal*.[27]

Many people think of fractals as mathematical toys whose principal use is the generation of pretty pictures. However, it turns out that there is a close connection between fractals and the dynamics of non-linear systems—particularly systems which exhibit chaotic dynamics. We have just seen an example in which the boundary between the basins of attraction of two co-existing attractors in phase-space is a fractal curve. This turns out to be a fairly general result: *i.e.*, when multiple attractors exist in phase-space the separatrix between their various basins of attraction is invariably fractal. What is this telling us about the nature of non-linear dynamics? Well, returning to Fig. 35, we can see that in the region of phase-space in which the fractal behaviour of the separatrix manifests itself most strongly (*i.e.*, the region where the light and dark bands fragment) the system exhibits abnormal sensitivity to initial conditions. In other words, we only have to change the initial conditions slightly (*i.e.*, so as to move from a dark to a light band, or *vice versa*) in order to significantly alter the time-asymptotic motion of the pendulum (*i.e.*, to cause the system to converge to a left-favouring instead of a right-favouring attractor, or *vice versa*). Fractals and extreme sensitivity to initial conditions are themes which will reoccur in our investigation of non-linear dynamics.
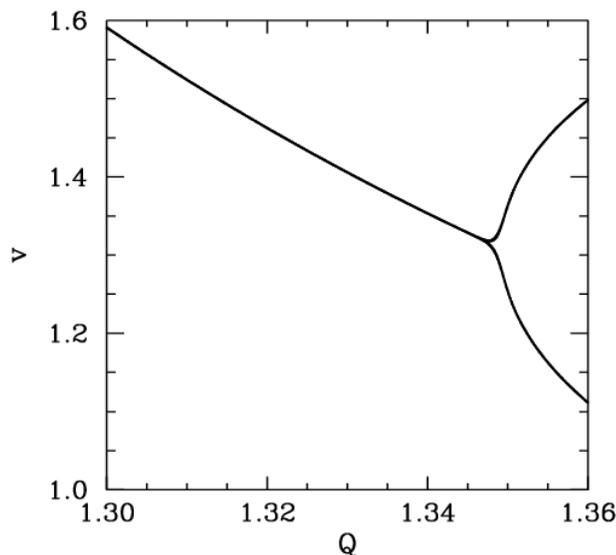
Figure 39: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor* Q. *Data calculated numerically for* A = 1.5, ω = 2/3, θ(0) = 0, v(0) = 0, `Nacc` = 100, *and* φ = 0.

4.8 Period-Doubling Bifurcations

Let us now return to Fig. 30. Recall, that as the quality-factor Q is gradually increased, the time-asymptotic orbit of the pendulum through phase-space undergoes a sudden transition, at Q ≃ 1.245, from a left-right symmetric, period-1 orbit to a left-favouring, period-1 orbit. What happens if we continue to increase Q? Figure 39 is basically a continuation of Fig. 30. It can be seen that as Q is increased the left-favouring, period-1 orbit gradually evolves until a critical value of Q, which is about 1.348, is reached. When Q exceeds this critical value the nature of the orbit undergoes another sudden change: this time from a left-favouring, period-1 orbit to a left-favouring, *period-2* orbit. Obviously, the change is indicated by the forking of the curve in Fig. 39. This type of transition is termed a *period-doubling bifurcation*, since it involves a sudden doubling of the repetition period of the pendulum's time-asymptotic motion.

We can represent period-1 motion schematically as $AAAAAA\cdots$, where A represents a pattern of motion which is repeated every period of the external

---

[27]B.B. Mandelbrot, *The fractal geometry of nature*, (W.H. Freeman, New York NY, 1982).
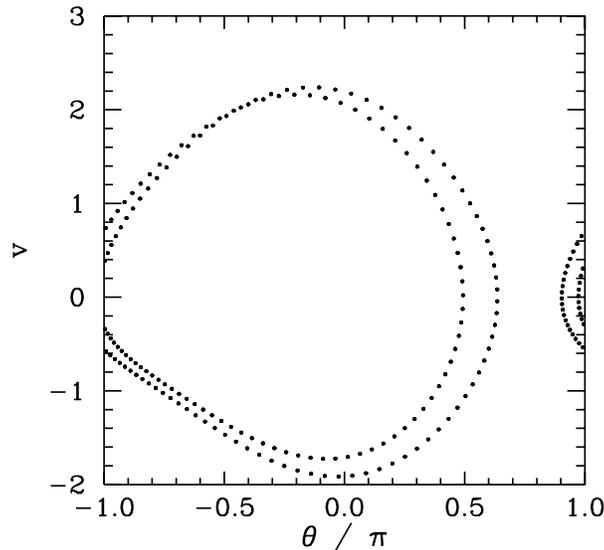
Figure 40: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* Q = 1.36, A = 1.5, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = -3$, *and* `Nacc` = 100.

drive. Likewise, we can represent period-2 motion as ABABAB $\cdots$, where A and B represent *distinguishable* patterns of motion which are repeated every alternate period of the external drive. A period-doubling bifurcation is represented: AAAAAA $\cdots \rightarrow$ ABABAB $\cdots$. Clearly, all that happens in such a bifurcation is that the pendulum suddenly decides to do something slightly different in alternate periods of the external drive.

Figure 40 shows the time-asymptotic phase-space orbit of the pendulum calculated for a value of Q somewhat higher than that required to trigger the above mentioned period-doubling bifurcation. It can be seen that the orbit is left-favouring (*i.e.*, it spends the majority of its time on the left-hand side of the plot), and takes the form of a closed curve consisting of *two* interlocked loops in phase-space. Recall that for period-1 orbits there was only a single closed loop in phase-space. Figure 41 shows the Poincaré section of the orbit shown in Fig. 40. The fact that the section consists of *two* points confirms that the orbit does indeed correspond to period-2 motion.

A period-doubling bifurcation is an example of *temporal symmetry breaking*. The equations of motion of the pendulum are invariant under the transformation
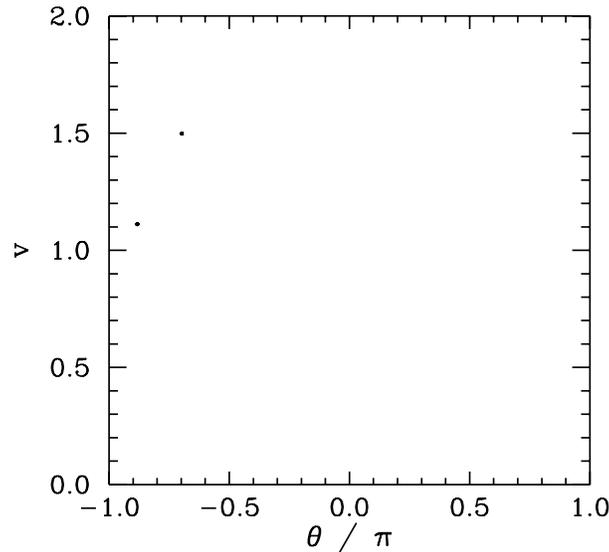
Figure 41: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 1.36$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$.

$t \to t + \tau$, where τ is the period of the external drive. In the low amplitude (*i.e.*, linear) limit the time-asymptotic motion of the pendulum always respects this symmetry. However, as we have just seen, in the non-linear regime it is possible to obtain solutions which spontaneously break this symmetry. Obviously, motion which repeats itself every two periods of the external drive is not as temporally symmetric as motion which repeats every period of the drive.

Figure 42 is essentially a continuation of Fig 34. Data is shown for two sets of initial conditions which lead to motions converging on left-favouring (lower branch) and right-favouring (upper branch) periodic attractors. We have already seen that the left-favouring periodic attractor undergoes a period-doubling bifurcation at $Q = 1.348$. It is clear from Fig. 42 that the right-favouring attractor undergoes a similar bifurcation at almost exactly the same Q-value. This is hardly surprising since, as has already been mentioned, for fixed physics parameters (*i.e.*, Q, A, ω), the left- and right-favouring attractors are mirror-images of one another.
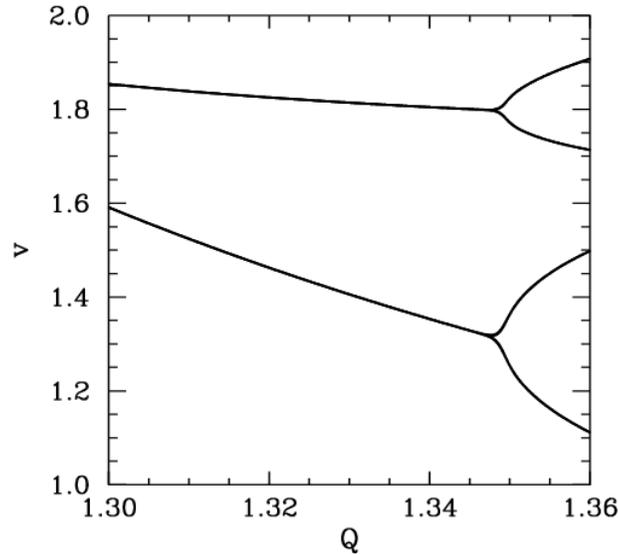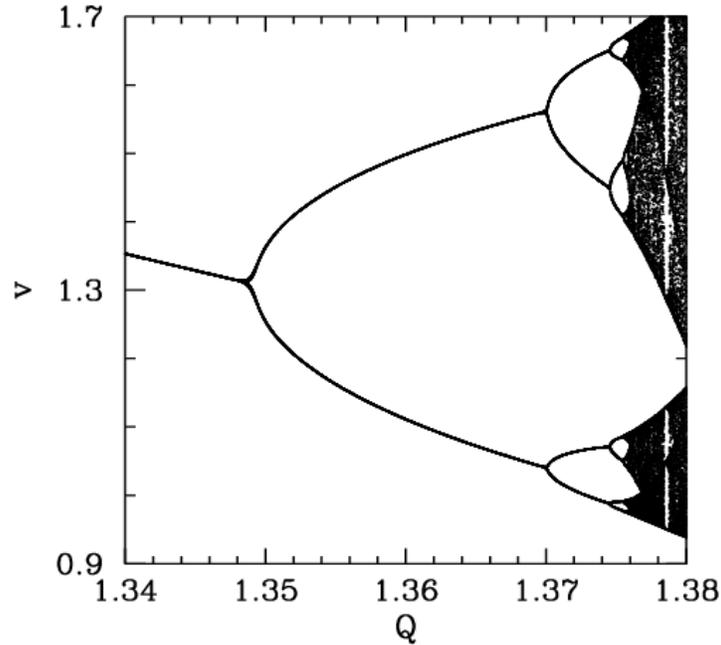
Figure 42: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor Q. Data calculated numerically for A = 1.5, ω = 2/3,* Nacc *= 100, and φ = 0. Data is shown for two sets of initial conditions: θ(0) = 0 and v(0) = 0 (lower branch); and θ(0) = 0 and v(0) = −2 (upper branch).*

4.9   The Route to Chaos

Let us return to Fig. 39, which tracks the evolution of a left-favouring periodic attractor as the quality-factor Q is gradually increased. Recall that when Q exceeds a critical value, which is about 1.348, then the attractor undergoes a period-doubling bifurcation which converts it from a period-1 to a period-2 attractor. This bifurcation is indicated by the forking of the curve in Fig. 39. Let us now investigate what happens as we continue to increase Q. Fig. 43 is basically a continuation of Fig. 39. It can be seen that, as Q is gradually increased, the attractor undergoes a period-doubling bifurcation at Q = 1.348, as before, but then undergoes a *second* period-doubling bifurcation (indicated by the second forking of the curves) at Q ≃ 1.370, and a *third* bifurcation at Q ≃ 1.375. Obviously, the second bifurcation converts a period-2 attractor into a period-4 attractor (hence, two curves split apart to give four curves). Likewise, the third bifurcation converts a period-4 attractor into a period-8 attractor (hence, four curves split into eight curves). Shortly after the third bifurcation, the various curves in the figure seem to expand explosively and merge together to produce an area of almost
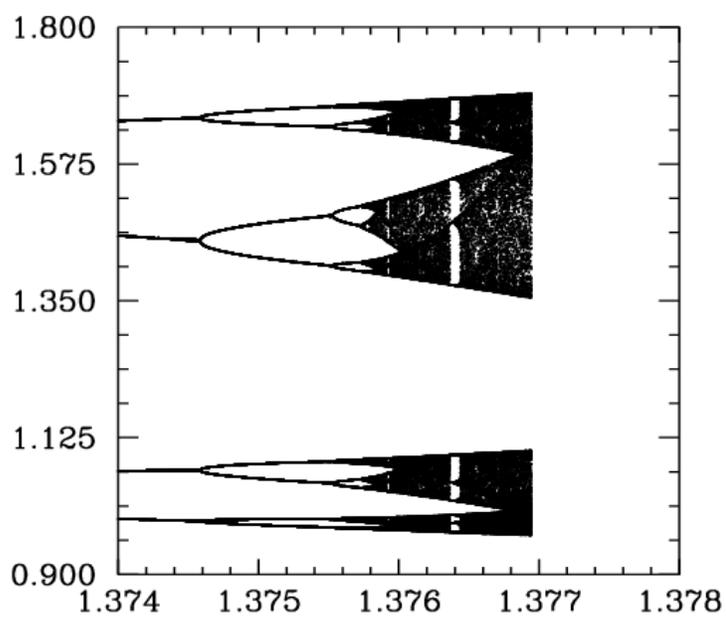
Figure 43: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor Q. Data calculated numerically for A = 1.5, ω = 2/3, θ(0) = 0, v(0) = 0, Nacc = 100, and φ = 0.*

solid black. As we shall see, this behaviour is indicative of the onset of *chaos*.

Figure 44 is a blow-up of Fig. 43, showing more details of the onset of chaos. The period-4 to period-8 bifurcation can be seen quite clearly. However, we can also see a period-8 to period-16 bifurcation, at $Q \simeq 1.3755$. Finally, if we look carefully, we can see a hint of a period-16 to period-32 bifurcation, just before the start of the solid black region. Figures 43 and 44 seem to suggest that the onset of chaos is triggered by an *infinite series* of period-doubling bifurcations.

Table 2 gives some details of the sequence of period-doubling bifurcations shown in Figs. 43 and 44. Let us introduce a bifurcation index n: the period-1 to period-2 bifurcation corresponds to $n = 1$; the period-2 to period-4 bifurcation corresponds to $n = 2$; and so on. Let Q_n be the critical value of the quality-factor Q above which the nth bifurcation is triggered. Table 2 shows the Q_n, determined from Figs. 43 and 44, for $n = 1$ to 5. Also shown is the ratio

$$F_n = \frac{Q_{n-1} - Q_{n-2}}{Q_n - Q_{n-1}} \tag{4.27}$$

167

Figure 44: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor* Q. *Data calculated numerically for* A = 1.5, ω = 2/3, θ(0) = 0, v(0) = 0, Nacc = 100, *and* φ = 0.

Bifurcation	n	Q_n	$Q_n - Q_{n-1}$	F_n
period-1→period-2	1	1.34870	-	-
period-2→period-4	2	1.37003	0.02133	-
period-4→period-8	3	1.37458	0.00455	4.69 ± 0.01
period-8→period-16	4	1.37555	0.00097	4.69 ± 0.04
period-16→period-32	5	1.37575	0.00020	4.9 ± 0.20

Table 2: *The period-doubling cascade.*

for $n = 3$ to 5. It can be seen that Tab. 2 offers reasonably convincing evidence that this ratio takes the *constant* value $F = 4.69$. It follows that we can estimate the critical Q-value required to trigger the nth bifurcation via the following formula:

$$Q_n = Q_1 + (Q_2 - Q_1) \sum_{j=0}^{n-2} \frac{1}{F^j}, \tag{4.28}$$

for $n > 1$. Note that the distance (in Q) between bifurcations decreases rapidly as n increases. In fact, the above formula predicts an *accumulation* of period-doubling bifurcations at $Q = Q_\infty$, where

$$Q_\infty = Q_1 + (Q_2 - Q_1) \sum_{j=0}^{\infty} \frac{1}{F^j} \equiv Q_1 + (Q_2 - Q_1) \frac{F}{F-1} = 1.3758. \tag{4.29}$$

Note that our calculated accumulation point corresponds almost exactly to the onset of the solid black region in Fig. 44. By the time that Q exceeds Q_∞, we expect the attractor to have been converted into a *period-infinity* attractor via an infinite series of period-doubling bifurcations. A period-infinity attractor is one whose corresponding motion *never* repeats itself, no matter how long we wait. In dynamics, such bounded aperiodic motion is generally referred to as *chaos*. Hence, a period-infinity attractor is sometimes called a *chaotic attractor*. Now, period-n motion is represented by n separate curves in Fig. 44. It is, therefore, not surprising that chaos (*i.e.*, period-infinity motion) is represented by an infinite number of curves which merge together to form a region of solid black.

Let us examine the onset of chaos in a little more detail. Figures 45–48 show details of the pendulum's time-asymptotic motion at various stages on the period-doubling cascade discussed above. Figure 45 shows period-4 motion: note that the Poincaré section consists of four points, and the associated sequence of net
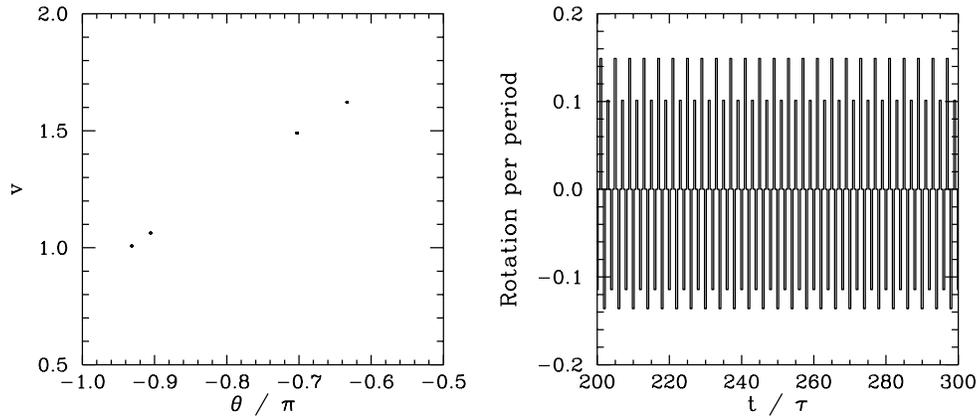
Figure 45: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 1.372$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$. *Also, shown is the net rotation per period,* $\Delta\theta/2\pi$, *calculated at the Poincaré phase* $\phi = 0$.

rotations per period of the pendulum repeats itself every four periods. Figure 46 shows period-8 motion: now the Poincaré section consists of eight points, and the rotation sequence repeats itself every eight periods. Figure 47 shows period-16 motion: as expected, the Poincaré section consists of sixteen points, and the rotation sequence repeats itself every sixteen periods. Finally, Fig. 48 shows chaotic motion. Note that the Poincaré section now consists of a set of four *continuous* line segments, which are, presumably, made up of an infinite number of points (corresponding to the infinite period of chaotic motion). Note, also, that the associated sequence of net rotations per period shows no obvious sign of ever repeating itself. In fact, this sequence looks rather like one of the previously shown periodic sequences with the addition of a small random component. The generation of *apparently random* motion from equations of motion, such as Eqs. (4.10) and (4.11), which contain no overtly random elements is one of the most surprising features of non-linear dynamics.

Many non-linear dynamical systems, found in nature, exhibit a transition from periodic to chaotic motion as some control parameter is varied. Now, there are various known mechanisms by which chaotic motion can arise from periodic motion. However, a transition to chaos via an infinite series of period-doubling bifurcations, as illustrated above, is certainly amongst the most commonly occurring of these mechanisms. Around 1975, the physicist Mitchell Feigenbaum
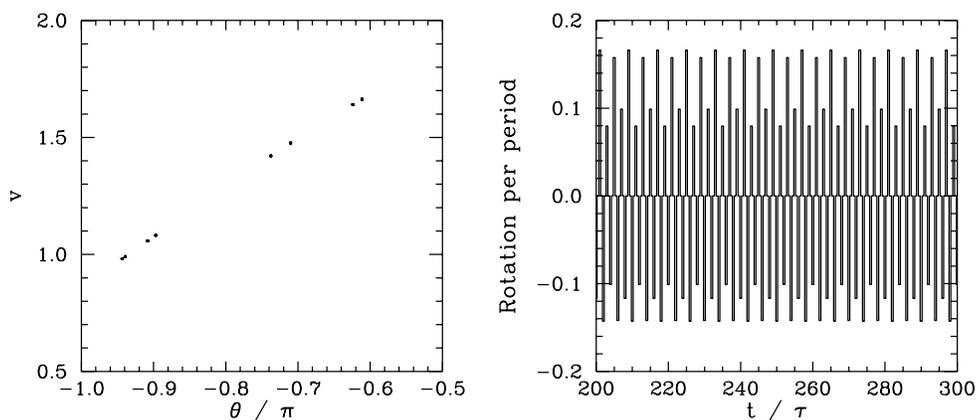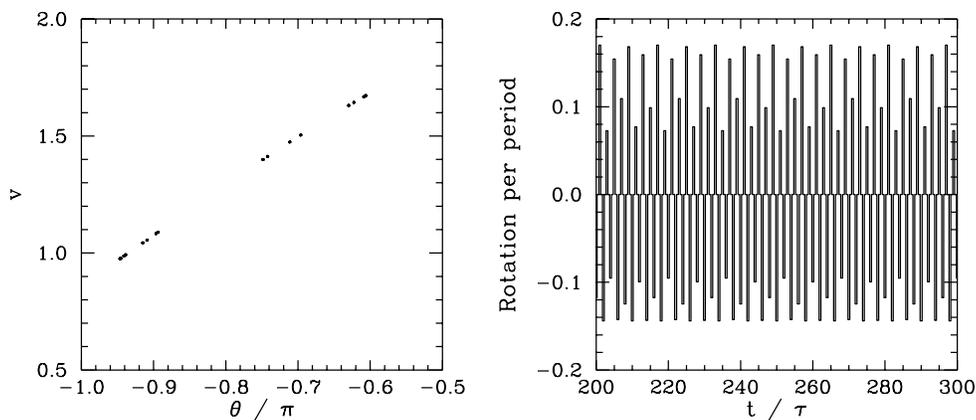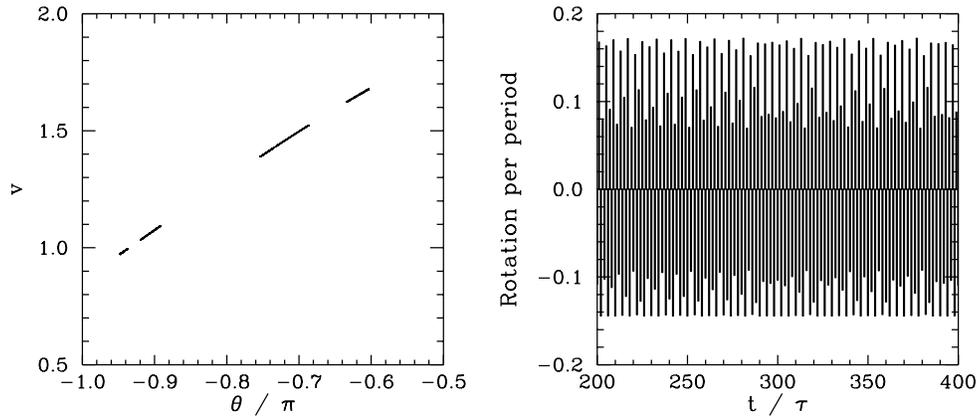
170

Figure 46: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 1.375$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, Nacc $= 100$, *and* $\phi = 0$. *Also, shown is the net rotation per period,* $\Delta\theta/2\pi$, *calculated at the Poincaré phase* $\phi = 0$.

Figure 47: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 1.3757$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, Nacc $= 100$, *and* $\phi = 0$. *Also, shown is the net rotation per period,* $\Delta\theta/2\pi$, *calculated at the Poincaré phase* $\phi = 0$.

Figure 48: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 1.376$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$. *Also, shown is the net rotation per period*, $\Delta\theta/2\pi$, *calculated at the Poincaré phase* $\phi = 0$.

was investigating a simple mathematical model, known as the *logistic map*, which exhibits a transition to chaos, via a sequence of period-doubling bifurcations, as a control parameter r is increased. Let r_n be the value of r at which the first 2^n-period cycle appears. Feigenbaum noticed that the ratio

$$F_n = \frac{r_{n-1} - r_{n-2}}{r_n - r_{n-1}} \tag{4.30}$$

converges rapidly to a constant value, $F = 4.669\ldots$, as n increases. Feigenbaum was able to demonstrate that this value of F is common to a wide range of different mathematic models which exhibit transitions to chaos via period-doubling bifurcations.[28] Feigenbaum went on to argue that the *Feigenbaum ratio*, F_n, should converge to the value $4.669\ldots$ in *any* dynamical system exhibiting a transition to chaos via period-doubling bifurcations.[29] This amazing prediction has been verified experimentally in a number of quite different physical systems.[30] Note that our best estimate of the Feigenbaum ratio (see Tab. 2) is 4.69 ± 0.01, in good agreement with Feigenbaum's prediction.

The existence of a universal ratio characterizing the transition to chaos via period-doubling bifurcations is one of many pieces of evidence indicating that

[28]M.J. Feigenbaum, *Quantitative universality for a class of nonlinear transformations*, J. Stat. Phys. **19**, 25 (1978).

[29]M.J. Feigenbaum, *The universal metric properties of nonlinear transformations*, J. Stat. Phys. **21**, 69 (1979).

[30]P. Citanovic, *Universality in chaos*, (Adam Hilger, Bristol UK, 1989).

172

chaos is a *universal* phenomenon (*i.e.*, the onset and nature of chaotic motion in different dynamical systems has many common features). This observation encourages us to believe that in studying the chaotic motion of a damped, periodically driven, pendulum we are learning lessons which can be applied to a wide range of non-linear dynamical systems.

4.10 Sensitivity to Initial Conditions

Suppose that we launch our pendulum and then wait until its motion has converged onto a particular attractor. The subsequent motion can be visualized as a trajectory $\theta_0(t)$, $v_0(t)$ through phase-space. Suppose that we somehow perturb the pendulum, at time $t = t_0$, such that its position in phase-space is instantaneously changed from $\theta_0(t_0)$, $v_0(t_0)$ to $\theta_0(t_0) + \delta\theta_0$, $v_0(t_0) + \delta v_0$. The subsequent motion can be visualized as a second trajectory $\theta_1(t)$, $v_1(t)$ through phase-space. What is the relationship between the original trajectory $\theta_0(t)$, $v_0(t)$ and the perturbed trajectory $\theta_1(t)$, $v_1(t)$? In other words, does the phase-space separation between the two trajectories, whose components are

$$\delta\theta(\Delta t) \;=\; \theta_1(t_0 + \Delta t) - \theta_0(t_0 + \Delta t), \tag{4.31}$$

$$\delta v(\Delta t) \;=\; v_1(t_0 + \Delta t) - v_0(t_0 + \Delta t), \tag{4.32}$$

grow in time, decay in time, or stay more or less the same? What we are really investigating is how sensitive the time-asymptotic motion of the pendulum is to initial conditions.

According to the linear analysis of Sect. 4.2,

$$\delta\theta(\Delta t) \;=\; \delta\theta_0 \cos(\omega_* \Delta t)\, e^{-\Delta t/2Q}$$

$$+ \frac{1}{\omega_*} \left\{ \delta v_0 + \frac{\delta\theta_0}{2Q} \right\} \sin(\omega_* \Delta t)\, e^{-\Delta t/2Q}, \tag{4.33}$$

$$\delta v(\Delta t) \;=\; \delta v_0 \cos(\omega_* \Delta t)\, e^{-\Delta t/2Q}$$

$$- \frac{1}{\omega_*} \left\{ \delta\theta_0 + \frac{\delta v_0}{2Q} \right\} \sin(\omega_* \Delta t)\, e^{-\Delta t/2Q}, \tag{4.34}$$
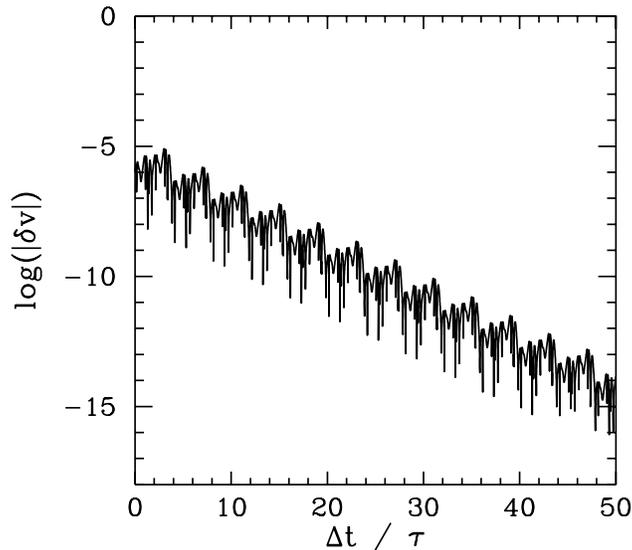
Figure 49: *The v-component of the separation between two neighbouring phase-space trajectories (one of which lies on an attractor) plotted against normalized time. Data calculated numerically for $Q = 1.372$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, and* `Nacc` $= 100$. *The separation between the two trajectories is initialized to $\delta\theta_0 = \delta v_0 = 10^{-6}$ at $\Delta t = 0$.*

assuming $\sin(\omega_* t_0) = 0$. It is clear that in the linear regime, at least, the pendulum's time-asymptotic motion is not particularly sensitive to initial conditions. In fact, if we move the pendulum's phase-space trajectory slightly off the linear attractor, as described above, then the perturbed trajectory decays back to the attractor *exponentially* in time. In other words, if we wait long enough then the perturbed and unperturbed motions of the pendulum become effectively indistinguishable. Let us now investigate whether this insensitivity to initial conditions carries over into the non-linear regime.

Figures 49–52 show the results of the experiment described above, in which the pendulum's phase-space trajectory is moved slightly off an attractor and the phase-space separation between the perturbed and unperturbed trajectories is then monitored as a function of time, at various stages on the period-doubling cascade discussed in the previous section. To be more exact, the figures show the logarithm of the absolute magnitude of the v-component of the phase-space separation between the perturbed and unperturbed trajectories as a function of normalized time.
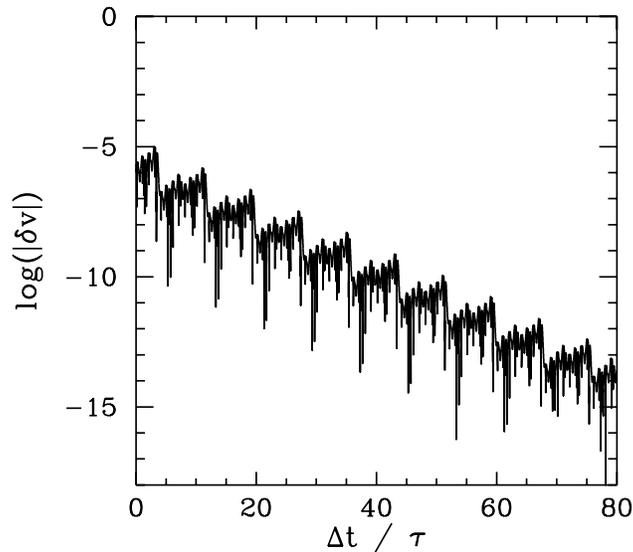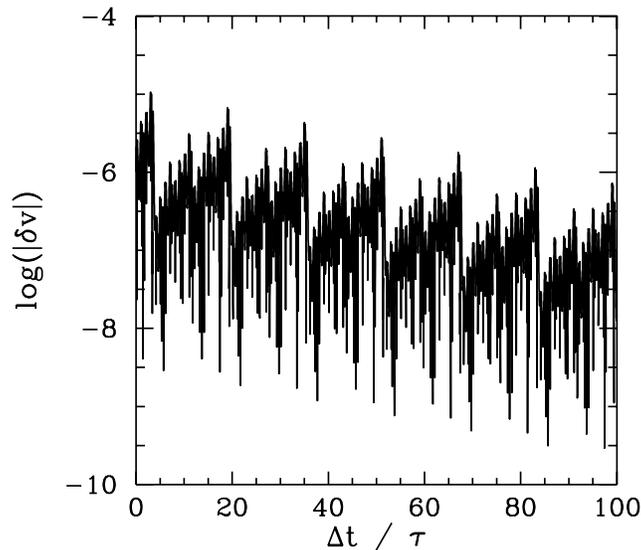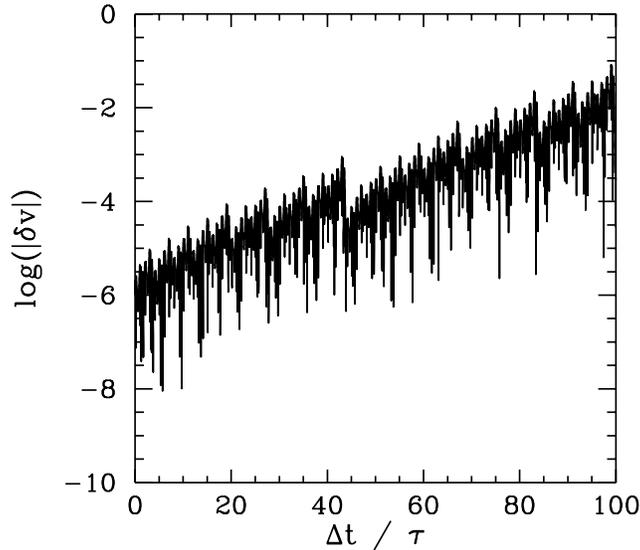
Figure 50: *The v-component of the separation between two neighbouring phase-space trajectories (one of which lies on an attractor) plotted against normalized time. Data calculated numerically for $Q = 1.375$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, and* `Nacc` $= 100$. *The separation between the two trajectories is initialized to $\delta\theta_0 = \delta v_0 = 10^{-6}$ at $\Delta t = 0$.*

Figure 51: *The v-component of the separation between two neighbouring phase-space trajectories (one of which lies on an attractor) plotted against normalized time. Data calculated numerically for $Q = 1.3757$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, and* `Nacc` $= 100$. *The separation between the two trajectories is initialized to $\delta\theta_0 = \delta v_0 = 10^{-6}$ at $\Delta t = 0$.*

Figure 52: *The v-component of the separation between two neighbouring phase-space trajectories (one of which lies on an attractor) plotted against normalized time. Data calculated numerically for $Q = 1.376$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, and* `Nacc` $= 100$. *The separation between the two trajectories is initialized to $\delta\theta_0 = \delta v_0 = 10^{-6}$ at $\Delta t = 0$.*

Figure 49 shows the time evolution of the v-component of the phase-space separation, δv, between two neighbouring trajectories, one of which is the period-4 attractor illustrated in Fig. 45. It can be seen that δv decays rapidly in time. In fact, the graph of $\log(|\delta v|)$ versus Δt can be plausibly represented as a *straight-line* of negative gradient λ. In other words,

$$|\delta v(\Delta t)| \simeq \delta v_0 \, e^{\lambda \Delta t}, \tag{4.35}$$

where the quantity λ is known as the *Liapunov exponent*. Clearly, in this case, λ measures the strength of the exponential convergence of the two trajectories in phase-space. Of course, the graph of $\log(|\delta v|)$ versus Δt is not exactly a straight-line. There are deviations due to the fact that δv oscillates, as well as decays, in time. There are also deviations because the strength of the exponential convergence between the two trajectories varies along the attractor.

The above definition of the Liapunov exponent is rather inexact, for two main reasons. In the first place, the strength of the exponential convergence/divergence between two neighbouring trajectories in phase-space, one of which is an attractor, generally *varies* along the attractor. Hence, we should really take formula

176

(4.35) and somehow *average* it over the attractor, in order to obtain a more unambiguous definition of λ. In the second place, since the dynamical system under investigation is a second-order system, it actually possesses *two* different Liapunov exponents. Consider the evolution of an infinitesimal circle of perturbed initial conditions, centred on a point in phase-space lying on an attractor. During its evolution, the circle will become distorted into an infinitesimal ellipse. Let δ_k, where $k = 1, 2$, denote the phase-space length of the kth principal axis of the ellipse. The two Liapunov exponents, λ_1 and λ_2, are defined via $\delta_k(\Delta t) \simeq \delta_k(0) \exp(\lambda_k \Delta t)$. However, for large Δt the diameter of the ellipse is effectively controlled by the Liapunov exponent with the most positive real part. Hence, when we refer to *the* Liapunov exponent, λ, what we generally mean is the Liapunov exponent with the most positive real part.

Figure 50 shows the time evolution of the v-component of the phase-space separation, δv, between two neighbouring trajectories, one of which is the period-8 attractor illustrated in Fig. 46. It can be seen that δv decays in time, though not as rapidly as in Fig. 49. Another way of saying this is that the Liapunov exponent of the periodic attractor shown in Fig. 46 is negative (*i.e.*, it has a negative real part), though not as negative as that of the periodic attractor shown in Fig. 45.

Figure 51 shows the time evolution of the v-component of the phase-space separation, δv, between two neighbouring trajectories, one of which is the period-16 attractor illustrated in Fig. 47. It can be seen that δv decays weakly in time. In other words, the Liapunov exponent of the periodic attractor shown in Fig. 47 is small and negative.

Finally, Fig. 52 shows the time evolution of the v-component of the phase-space separation, δv, between two neighbouring trajectories, one of which is the chaotic attractor illustrated in Fig. 48. It can be seen that δv *increases* in time. In other words, the Liapunov exponent of the chaotic attractor shown in Fig. 48 is *positive*. Further investigation reveals that, as the control parameter Q is gradually increased, the Liapunov exponent changes sign and becomes positive at exactly the same point that chaos ensues in Fig 44.

The above discussion strongly suggests that periodic attractors are characterized by negative Liapunov exponents, whereas chaotic attractors are character-

ized by positive exponents. But, how can an attractor have a positive Liapunov exponent? Surely, a positive exponent necessarily implies that neighbouring phase-space trajectories *diverge* from the attractor (and, hence, that the attractor is not a true attractor)? It turns out that this is not the case. The chaotic attractor shown in Fig. 48 is a true attractor, in the sense that neighbouring trajectories rapidly converge onto it—*i.e.*, after a few periods of the external drive their Poincaré sections plot out the same four-line segment shown in Fig. 48. Thus, the exponential divergence of neighbouring trajectories, characteristic of chaotic attractors, takes place *within the attractor* itself. Obviously, this exponential divergence must come to an end when the phase-space separation of the trajectories becomes comparable to the extent of the attractor.

A dynamical system characterized by a positive Liapunov exponent, λ, has a *time horizon* beyond which regular deterministic prediction breaks down. Suppose that we measure the initial conditions of an experimental system very accurately. Obviously, no measurement is perfect: there is always some error δ_0 between our estimate and the true initial state. After a time t, the discrepancy grows to $\delta(t) \sim \delta_0 \exp(\lambda t)$. Let a be a measure of our tolerance: *i.e.*, a prediction within a of the true state is considered acceptable. It follows that our prediction becomes unacceptable when $\delta \gg a$, which occurs when

$$t > t_h \sim \frac{1}{\lambda} \ln\left(\frac{a}{\delta_0}\right). \tag{4.36}$$

Note the *logarithmic* dependence on δ_0. This ensures that, in practice, no matter how hard we work to reduce our initial measurement error, we cannot predict the behaviour of the system for longer than a few multiples of $1/\lambda$.

It follows, from the above discussion, that chaotic attractors are associated with motion which is essentially *unpredictable*. In other words, if we attempt to integrate the equations of motion of a chaotic system then even the slightest error made in the initial conditions will be amplified exponentially over time and will rapidly destroy the accuracy of our prediction. Eventually, all that we will be able to say is that the motion lies somewhere on the chaotic attractor in phase-space, but exactly where it lies on the attractor at any given time will be unknown to us.

The hyper-sensitivity of chaotic systems to initial conditions is sometimes called

the *butterfly effect*. The idea is that a butterfly flapping its wings in a South American rain-forest could, in principle, affect the weather in Texas (since the atmosphere exhibits chaotic dynamics). This idea was first publicized by the meteorologist Edward Lorenz, who constructed a very crude model of the convection of the atmosphere when it is heated from below by the ground.[31] Lorenz discovered, much to his surprise, that his model atmosphere exhibited chaotic motion—which, at that time, was virtually unknown to physics. In fact, Lorenz was essentially the first scientist to fully understand the nature and ramifications of chaotic motion in physical systems. In particular, Lorenz realized that the chaotic dynamics of the atmosphere spells the doom of long-term weather forecasting: the best one can hope to achieve is to predict the weather a few days in advance ($1/\lambda$ for the atmosphere is of order a few days).

4.11   The Definition of Chaos

There is no universally agreed definition of chaos. However, most people would accept the following working definition:

> Chaos is aperiodic time-asymptotic behaviour in a deterministic system which exhibits sensitive dependence on initial conditions.

This definition contains three main elements:

1. Aperiodic time-asymptotic behaviour—this implies the existence of phase-space trajectories which do not settle down to fixed points or periodic orbits. For practical reasons, we insist that these trajectories are not too rare. We also require the trajectories to be *bounded*: *i.e.*, they should not go off to infinity.

2. Deterministic—this implies that the equations of motion of the system possess no random inputs. In other words, the irregular behaviour of the system arises from non-linear dynamics and not from noisy driving forces.

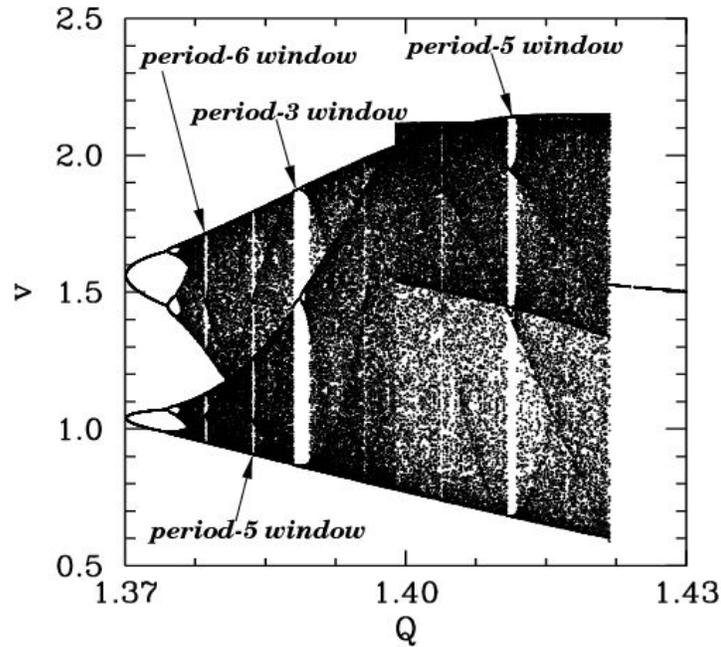[31]E. Lorenz, *Deterministic nonperiodic flow*, J. Atmospheric Science **20**, 130 (1963).

Figure 53: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor Q. Data calculated numerically for* A = 1.5, ω = 2/3, θ(0) = 0, v(0) = −0.75, Nacc = 100, *and* φ = 0.

3. Sensitive dependence on initial conditions—this implies that nearby trajectories in phase-space separate exponentially fast in time: *i.e.*, the system has a positive Liapunov exponent.

4.12   Periodic Windows

Let us return to Fig. 44. Recall, that this figure shows the onset of chaos, via a cascade of period-doubling bifurcations, as the quality-factor Q is gradually increased. Figure 53 is essentially a continuation of Fig. 44 which shows the full extent of the chaotic region (in Q-v space). It can be seen that the chaotic region ends abruptly when Q exceeds a critical value, which is about 1.4215. Beyond this critical value, the time-asymptotic motion appears to revert to period-1 motion (*i.e.*, the solid black region collapses to a single curve). It can also be seen that the chaotic region contains many narrow windows in which chaos reverts to periodic motion (*i.e.*, the solid black region collapses to n curves, where n is the period of
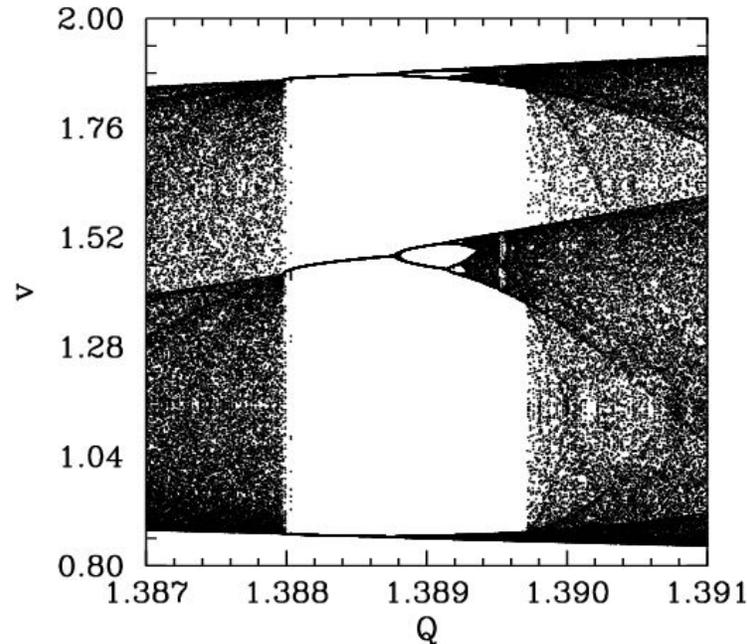
Figure 54: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor Q. Data calculated numerically for* A = 1.5, ω = 2/3, θ(0) = 0, v(0) = −0.75, `Nacc` = 100, *and* φ = 0.

the motion) for a short interval in Q. The four widest windows are indicated on the figure.

Figure 54 is a blow-up of the period-3 window shown in Fig. 53. It can be seen that the window appears "out of the blue" as Q is gradually increased. However, it can also be seen that, as Q is further increased, the window breaks down, and eventually disappears, due to the action of a cascade of period-doubling bifurcations. The same basic mechanism operates here as in the original period-doubling cascade, discussed in Sect. 4.9, except that now the orbits are of period $3 \cdot 2^n$, instead of $2 \cdot 2^n$. Note that all of the other periodic windows seen in Fig. 53 break down in an analogous manner, as Q is increased.

We now understand how periodic windows break down. But, how do they appear in the first place? Figures 55–57 show details of the pendulum's time-asymptotic motion calculated *just before* the appearance of the period-3 window (shown in Fig. 54), *just at* the appearance of the window, and *just after* the ap-
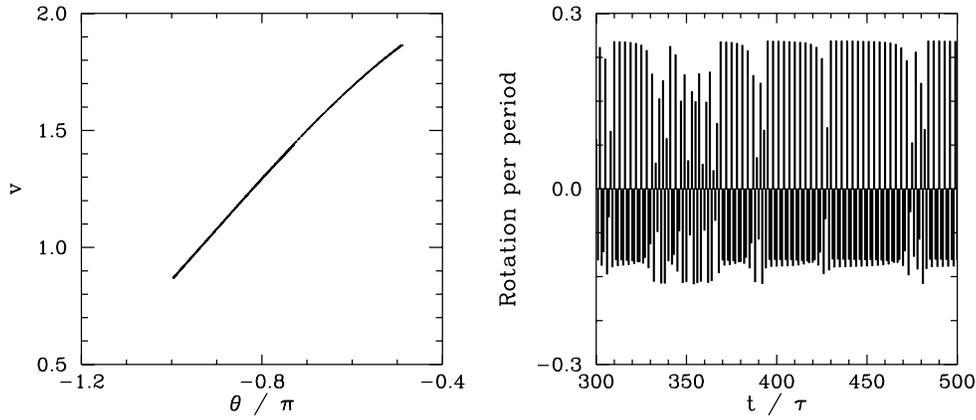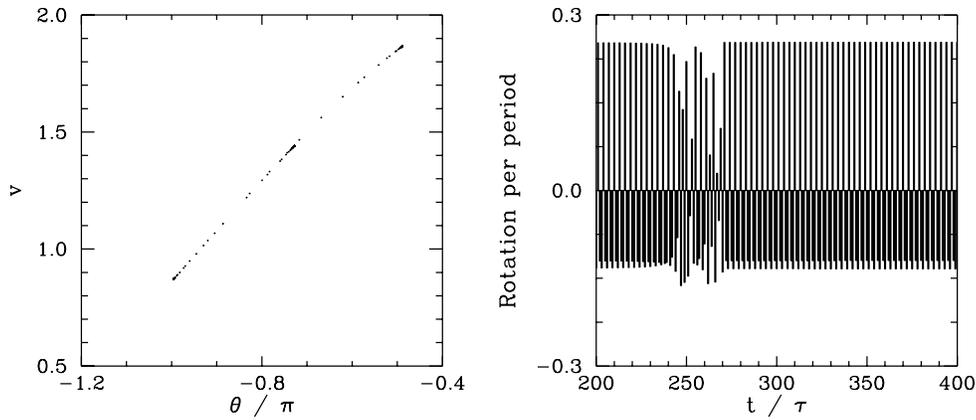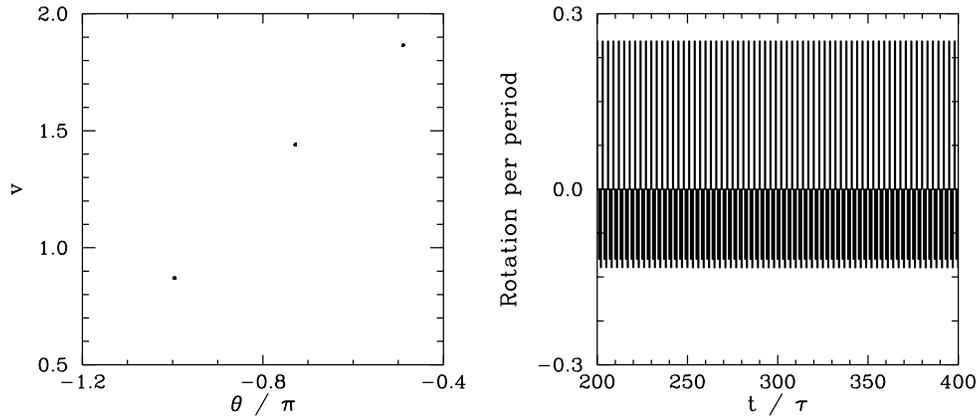
Figure 55: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* Q = 1.387976, A = 1.5, ω = 2/3, θ(0) = 0, ν(0) = −0.75, Nacc = 100, *and* φ = 0. *Also, shown is the net rotation per period,* Δθ/2π, *calculated at the Poincaré phase* φ = 0.

Figure 56: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* Q = 1.387977, A = 1.5, ω = 2/3, θ(0) = 0, ν(0) = −0.75, Nacc = 100, *and* φ = 0. *Also, shown is the net rotation per period,* Δθ/2π, *calculated at the Poincaré phase* φ = 0.

Figure 57: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for Q =
1.387978, A = 1.5, ω = 2/3, θ(0) = 0, v(0) = −0.75, Nacc = 100, and ϕ = 0. Also, shown is the
net rotation per period, Δθ/2π, calculated at the Poincaré phase ϕ = 0.*

pearance of the window, respectively. It can be seen, from Fig. 55, that just before
the appearance of the window the attractor is chaotic (*i.e.*, its Poincaré section
consists of a line, rather than a discrete set of points), and the time-asymptotic
motion of the pendulum consists of intervals of period-3 motion interspersed with
intervals of chaotic motion. Figure 56 shows that just at the appearance of the
window the attractor loses much of its chaotic nature (*i.e.*, its Poincaré section
breaks up into a series of points), and the chaotic intervals become shorter and
much less frequent. Finally, Fig. 57 shows that just after the appearance of the
window the attractor collapses to a period-3 attractor, and the chaotic intervals
cease altogether. All of the other periodic windows seen in Fig. 53 appear in an
analogous manner to that just described.

According to the above discussion, the typical time-asymptotic motion seen
just prior to the appearance of a period-n window consists of intervals of period-
n motion interspersed with intervals of chaotic motion. This type of behaviour
is called *intermittency*, and is observed in a wide variety of non-linear systems.
As we move away from the window, in parameter space, the intervals of periodic
motion become gradually shorter and more infrequent. Eventually, they cease
altogether. Likewise, as we move towards the window, the intervals of periodic
motion become gradually longer and more frequent. Eventually, the whole mo-
tion becomes periodic.

In 1973, Metropolis and co-workers investigated a class of simple mathematical models which all exhibit a transition to chaos, via a cascade of period-doubling bifurcations, as some control parameter r is increased.[32] They were able to demonstrate that, for these maps, the order in which stable periodic orbits occur as r is increased is fixed. That is, *stable periodic attractors always occur in the same sequence* as r is varied. This sequence is called the universal or *U-sequence*. It is possible to make a fairly convincing argument that any physical system which exhibits a transition to chaos via a sequence of period-doubling bifurcations should also exhibit the U-sequence of stable periodic attractors. Up to period-6, the U-sequence is

$$1,\ 2,\ 2 \times 2,\ 6,\ 5,\ 3,\ 2 \times 3,\ 5,\ 6,\ 4,\ 6,\ 5,\ 6.$$

The beginning of this sequence is familiar: periods $1, 2, 2 \times 2$ are the first stages of the period-doubling cascade. (The later period-doublings give rise to periods greater than 6, and so are omitted here). The next periods, $6, 5, 3$ correspond to the first three periodic windows shown in Fig. 53. Period 2×3 is the first component of the period-doubling cascade which breaks up the period-3 window. The next period, 5, corresponds to the last periodic window shown in Fig. 53. The remaining periods, 6, 4, 6, 5, 6, correspond to tiny periodic windows, which, in practice, are virtually impossible to observe. It follows that our driven pendulum system exhibits the U-sequence of stable periodic orbits fairly convincingly. This sequence has also been observed experimentally in other, quite different, dynamical systems.[33] The existence of a universal sequence of stable periodic orbits in dynamical systems which exhibit a transition to chaos via a cascade of period-doubling bifurcations is another indication that chaos is a universal phenomenon.

4.13   Further Investigation

Figure 58 shows the complete *bifurcation diagram* for the damped, periodically driven, pendulum (with $A = 1.5$ and $\omega = 2/3$). It can be seen that the chaotic

---

[32]N. Metropolis, M.L. Stein, and P.R. Stein, *On finite limit sets for transformations on the unit interval*, J. Combin. Theor. **15**, 25 (1973).

[33]R.H. Simoyi, A. Wolf, and H.L. Swinney, *One-dimensional dynamics in a multi-component chemical reaction*, Phys. Rev. Lett. **49**, 245 (1982).
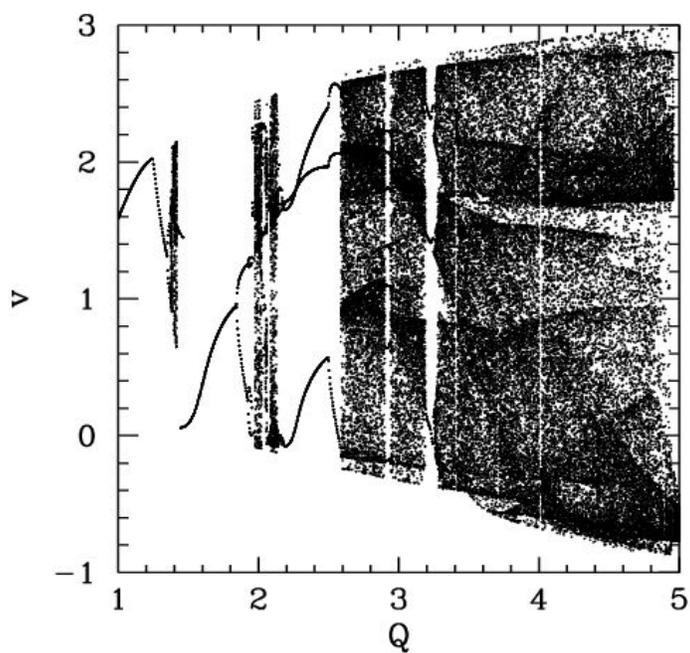
Figure 58: *The v-coordinate of the Poincaré section of a time-asymptotic orbit plotted against the quality-factor* Q. *Data calculated numerically for* A = 1.5, ω = 2/3, θ(0) = 0, v(0) = 0, `Nacc` = 100, *and* φ = 0.

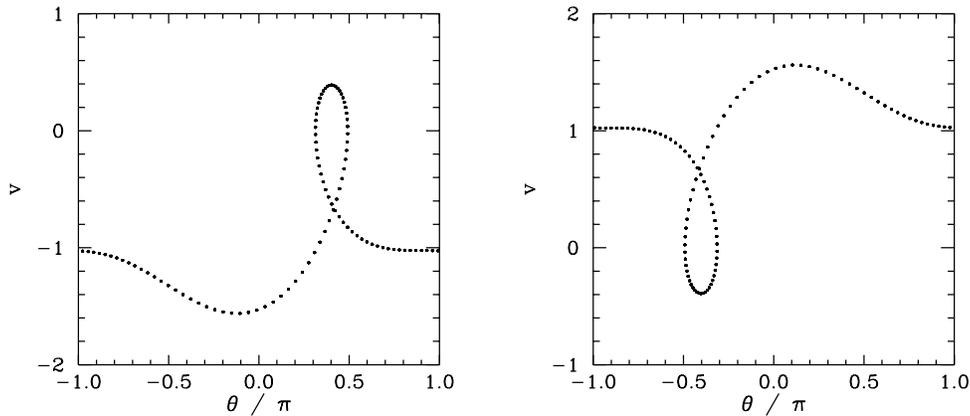Figure 59: *Equally spaced (in time) points on a time-asymptotic orbit in phase-space. Data calculated numerically for* Q = 1.5, A = 1.5, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, *and* `Nacc` = 100. *Also shown is the time-asymptotic orbit calculated for the modified initial conditions* $\theta(0) = 0$, *and* $v(0) = -1$.

region investigated in the previous section is, in fact, the first, and least extensive, of *three* different chaotic regions.

The interval between the first and second chaotic regions is occupied by the period-1 orbits shown in Fig. 59. Note that these orbits differ somewhat from previously encountered period-1 orbits, because the pendulum executes a complete rotation (either to the left or to the right) every period of the external drive. Now, an n, l periodic orbit is defined such that

$$\theta(t + n\,\tau) = \theta(t) + 2\pi\,l$$

for all t (after the transients have died away). It follows that all of the periodic orbits which we encountered in previous sections were $l = 0$ orbits: *i.e.*, their associated motions did not involve a net rotation of the pendulum. The orbits show in Fig. 59 are $n = 1, l = -1$ and $n = 1, l = +1$ orbits, respectively. The existence of periodic orbits in which the pendulum undergoes a net rotation, either to the left or to the right, is another example of spatial symmetry breaking—there is nothing in the pendulum's equations of motion which distinguishes between the two possible directions of rotation.

Figure 60 shows the Poincaré section of a typical attractor in the second chaotic region shown in Fig. 58. It can be seen that this attractor is far more convoluted and extensive than the simple 4-line chaotic attractor pictured in Fig. 48. In fact,
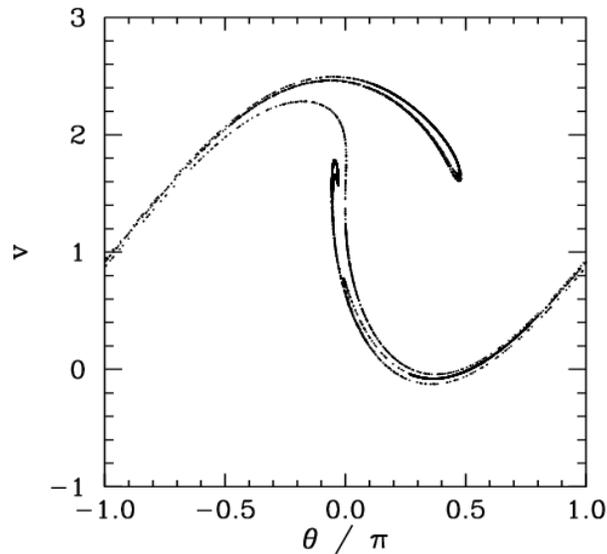
Figure 60: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* Q = 2.13, A = 1.5, ω = 2/3, θ(0) = 0, v(0) = 0, `Nacc` = 100, *and* φ = 0.

the attractor shown in Fig. 60 is clearly a *fractal* curve. It turns out that virtually all chaotic attractors exhibit fractal structure.

The interval between the second and third chaotic regions shown in Fig. 58 is occupied by n = 3, l = 0 periodic orbits. Figure 61 shows the Poincaré section of a typical attractor in the third chaotic region. It can be seen that this attractor is even more overtly fractal in nature than that pictured in the previous figure. Note that the fractal nature of chaotic attractors is closely associated with some of their unusual properties. Trajectories on a chaotic attractor remain confined to a bounded region of phase-space, and yet they separate from their neighbours exponentially fast (at least, initially). How can trajectories diverge endlessly and still stay bounded? The basic mechanism is described below. If we imagine a blob of initial conditions in phase-space, then these undergo a series of repeated *stretching and folding* episodes, as the chaotic motion unfolds. The stretching is what gives rise to the divergence of neighbouring trajectories. The folding is what ensures that the trajectories remain bounded. The net result is a phase-space structure which looks a bit like filo dough—in other words, a fractal structure.
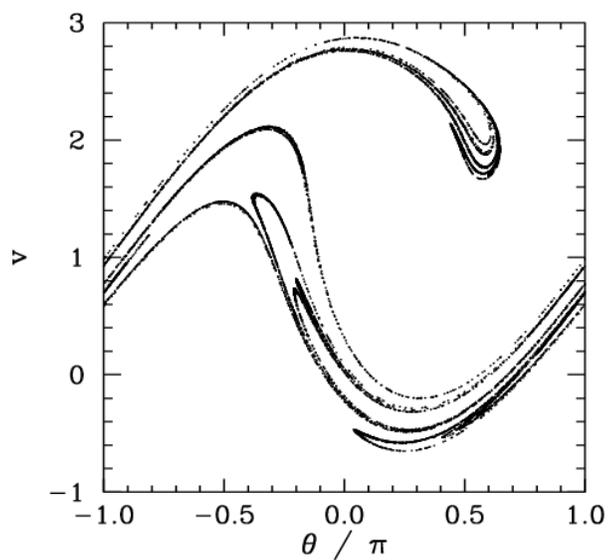
Figure 61: *The Poincaré section of a time-asymptotic orbit. Data calculated numerically for* $Q = 3.9$, $A = 1.5$, $\omega = 2/3$, $\theta(0) = 0$, $v(0) = 0$, `Nacc` $= 100$, *and* $\phi = 0$.

5 Poisson's Equation

5.1 Introduction

In this section, we shall discuss some simple numerical techniques for solving Poisson's equation:

$$\nabla^2 u(\mathbf{r}) = v(\mathbf{r}). \tag{5.1}$$

Here, $u(\mathbf{r})$ is usually some sort of potential, whereas $v(\mathbf{r})$ is a source term. The solution to the above equation is generally required in some simply-connected volume V bounded by a closed surface S. There are two main types of boundary conditions to Poisson's equation. In so-called *Dirichlet* boundary conditions, the potential u is specified on the bounding surface S. In so-called *Neumann* boundary conditions, the normal gradient of the potential $\nabla u \cdot d\mathbf{S}$ is specified on the bounding surface.

Poisson's equation is of particular importance in electrostatics and Newtonian gravity. In electrostatics, we can write the electric field $\mathbf{E}$ in terms of an electric potential ϕ:

$$\mathbf{E} = -\nabla \phi. \tag{5.2}$$

The potential itself satisfies Poisson's equation:

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}, \tag{5.3}$$

where $\rho(\mathbf{r})$ is the charge density, and ϵ_0 the permittivity of free-space. In Newtonian gravity, we can write the force $\mathbf{f}$ exerted on a unit test mass in terms of a gravitational potential ϕ:

$$\mathbf{f} = -\nabla \phi. \tag{5.4}$$

The potential satisfies Poisson's equation:

$$\nabla^2 \phi = 4\pi^2 \, G \, \rho, \tag{5.5}$$

where $\rho(\mathbf{r})$ is the mass density, and G the universal gravitational constant.

5.2   1-D Problem with Dirichlet Boundary Conditions

As a simple test case, let us consider the solution of Poisson's equation in one dimension. Suppose that

$$\frac{d^2 u(x)}{dx^2} = v(x), \tag{5.6}$$

for $x_l \leq x \leq x_h$, subject to the Dirichlet boundary conditions $u(x_l) = u_l$ and $u(x_h) = u_h$.

As a first step, we divide the domain $x_l \leq x \leq x_h$ into equal segments whose vertices are located at the grid-points

$$x_i = x_l + \frac{i\,(x_h - x_l)}{N + 1}, \tag{5.7}$$

for $i = 1, N$. The boundaries, x_l and x_h, correspond to $i = 0$ and $i = N + 1$, respectively.

Next, we discretize the differential term $d^2 u/dx^2$ on the grid-points. The most straightforward discretization is

$$\frac{d^2 u(x_i)}{dx^2} = \frac{u_{i-1} - 2\,u_i + u_{i+1}}{(\delta x)^2} + O(\delta x)^2. \tag{5.8}$$

Here, $\delta x = (x_h - x_l)/(N + 1)$, and $u_i \equiv u(x_i)$. This type of discretization is termed a *second-order*, *central difference* scheme. It is "second-order" because the truncation error is $O(\delta x)^2$, as can easily be demonstrated via Taylor expansion. Of course, an nth order scheme would have a truncation error which is $O(\delta x)^n$. It is a "central difference" scheme because it is symmetric about the central grid-point, x_i. Our discretized version of Poisson's equation takes the form

$$u_{i-1} - 2\,u_i + u_{i+1} = v_i\,(\delta x)^2, \tag{5.9}$$

for $i = 1, N$, where $v_i \equiv v(x_i)$. Furthermore, $u_0 = u_l$ and $u_{N+1} = u_h$.

It is helpful to regard the above set of discretized equations as a matrix equation. Let $\mathbf{u} = (u_1, u_2, \cdots, u_N)$ be the vector of the u-values, and let

$$\mathbf{w} = [v_1\,(\delta x)^2 - u_l, v_2\,(\delta x)^2, v_3\,(\delta x)^2, \cdots, v_{N-1}\,(\delta x)^2, v_N\,(\delta x)^2 - u_h] \tag{5.10}$$

be the vector of the source terms. The discretized equations can be written as:

$$\mathbf{M}\,\mathbf{u} = \mathbf{w}. \tag{5.11}$$

The matrix $\mathbf{M}$ takes the form

$$\mathbf{M} = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix} \tag{5.12}$$

for $N = 6$. The generalization to other N values is fairly obvious. Matrix $\mathbf{M}$ is termed a *tridiagonal* matrix, since only those elements which lie on the three leading diagonals are non-zero.

The formal solution to Eq. (5.11) is

$$\mathbf{u} = \mathbf{M}^{-1}\,\mathbf{w}, \tag{5.13}$$

where $\mathbf{M}^{-1}$ is the inverse matrix to $\mathbf{M}$. Unfortunately, the most efficient general purpose algorithm for inverting an $N \times N$ matrix—namely, Gauss-Jordan elimination with partial pivoting—requires $O(N^3)$ arithmetic operations. It is fairly clear that this is a disastrous scaling for finite-difference solutions of Poisson's equation. Every time we doubled the resolution (*i.e.*, doubled the number of grid-points) the required cpu time would increase by a factor of about eight. Consequently, adding a second dimension (which effectively requires the number of grid-points to be squared) would be prohibitively expensive in terms of cpu time. Fortunately, there is a well-known trick for inverting an $N \times N$ *tridiagonal* matrix which only requires $O(N)$ arithmetic operations.

Consider a general $N \times N$ tridiagonal matrix equation $\mathbf{M}\,\mathbf{u} = \mathbf{w}$. Let $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ be the vectors of the left, center and right diagonal elements of the matrix, respectively Note that a_1 and c_N are undefined, and can be conveniently set to zero. Our matrix equation can now be written

$$a_i\,u_{i-1} + b_i\,u_i + c_i\,u_{i+1} = w_i, \tag{5.14}$$

for $i = 1, N$. Let us search for a solution of the form

$$u_{i+1} = x_i\, u_i + y_i. \tag{5.15}$$

Substitution into Eq. (5.14) yields

$$a_i\, u_{i-1} + b_i\, u_i + c_i\, (x_i\, u_i + y_i) = w_i, \tag{5.16}$$

which can be rearranged to give

$$u_i = -\frac{a_i\, u_{i-1}}{b_i + c_i\, x_i} + \frac{w_i - c_i\, y_i}{b_i + c_i\, x_i}. \tag{5.17}$$

However, if Eq. (5.15) is general then we can write $u_i = x_{i-1}\, u_{i-1} + y_{i-1}$. Comparison with the previous equation yields

$$x_{i-1} = -\frac{a_i}{b_i + c_i\, x_i}, \tag{5.18}$$

and

$$y_{i-1} = \frac{w_i - c_i\, y_i}{b_i + c_i\, x_i}. \tag{5.19}$$

We can now solve our tridiagonal matrix equation in two stages. In the first stage, we scan *up* the leading diagonal from $i = N$ to 1 using Eqs. (5.18) and (5.19). Thus,

$$x_{N-1} = -\frac{a_N}{b_N}, \qquad y_{N-1} = \frac{w_N}{b_N}, \tag{5.20}$$

since $c_N = 0$. Furthermore,

$$x_i = -\frac{a_{i+1}}{b_{i+1} + c_{i+1}\, x_{i+1}}, \qquad y_i = \frac{w_{i+1} - c_{i+1}\, y_{i+1}}{b_{i+1} + c_{i+1}\, x_{i+1}} \tag{5.21}$$

for $i = N - 2, 1$. Finally,

$$x_0 = 0, \qquad y_0 = \frac{w_1 - c_1\, y_1}{b_1 + c_1\, x_1}, \tag{5.22}$$

since $a_1 = 0$. We have now defined all of the x_i and y_i. In the second stage, we scan *down* the leading diagonal from $i = 0$ to $N - 1$ using Eq. (5.15). Thus,

$$u_1 = y_0, \tag{5.23}$$

since $x_0 = 0$, and

$$u_{i+1} = x_i \, u_i + y_i \tag{5.24}$$

for $i = 1, N - 1$. We have now inverted our tridiagonal matrix equation using $O(N)$ arithmetic operations.

Clearly, we can use the above algorithm to invert Eq. (5.11), with the source terms specified in Eq. (5.10), and the diagonals of matrix $\mathbf{M}$ given by $a_i = 1$ for $i = 2, N$, plus $b_i = -2$ for $i = 1, N$, and $c_i = 1$ for $i = 1, N - 1$.

5.3   An Example Tridiagonal Matrix Solving Routine

Listed below is an example tridiagonal matrix solving routine which utilizes the `Blitz++` library (see Sect. 2.20).

```
// Tridiagonal.cpp

// Function to invert tridiagonal matrix equation.
// Left, centre, and right diagonal elements of matrix
// stored in arrays a, b, c, respectively.
// Right-hand side stored in array w.
// Solution written to array u.

// Matrix is NxN. Arrays a, b, c, w, u assumed to be of extent N+2,
// with redundant 0 and N+1 elements.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u)
{
  // Find N. Declare local arrays.
  int N = a.extent(0) - 2;
  Array<double,1> x(N), y(N);

  // Scan up diagonal from i = N to 1
  x(N-1) = - a(N) / b(N);
  y(N-1) = w(N) / b(N);
  for (int i = N-2; i > 0; i--)
```

```
    {
      x(i) = - a(i+1) / (b(i+1) + c(i+1) * x(i+1));
      y(i) = (w(i+1) - c(i+1) * y(i+1)) / (b(i+1) + c(i+1) * x(i+1));
    }
  x(0) = 0.;
  y(0) = (w(1) - c(1) * y(1)) / (b(1) + c(1) * x(1));

  // Scan down diagonal from i = 0 to N-1
  u(1) = y(0);
  for (int i = 1; i < N; i++)
    u(i+1) = x(i) * u(i) + y(i);
}
```

5.4   1-D problem with Mixed Boundary Conditions

Previously, we solved Poisson's equation in one dimension subject to Dirichlet boundary conditions, which are the simplest conceivable boundary conditions. Let us now consider the following much more general set of boundary conditions:

$$\alpha_l \, u(x) + \beta_l \frac{du(x)}{dx} = \gamma_l, \tag{5.25}$$

at $x = x_l$, and

$$\alpha_h \, u(x) + \beta_h \frac{du(x)}{dx} = \gamma_h, \tag{5.26}$$

at $x = x_h$. Here, α_l, β_l, *etc.*, are known constants. The above boundary conditions are termed *mixed*, since they are a mixture of Dirichlet and Neumann boundary conditions.

Using the previous notation, the discretized versions of Eq. (5.25) and (5.26) are:

$$\alpha_l \, u_0 + \beta_l \frac{u_1 - u_0}{\delta x} = \gamma_l, \tag{5.27}$$

$$\alpha_h \, u_{N+1} + \beta_h \frac{u_{N+1} - u_N}{\delta x} = \gamma_h, \tag{5.28}$$

respectively. The above expressions can be rearranged to give

$$u_0 = \frac{\gamma_l \, \delta x - \beta_l \, u_1}{\alpha_l \, \delta x - \beta_l}, \tag{5.29}$$

$$u_{N+1} = \frac{\gamma_h \, \delta x + \beta_h \, u_N}{\alpha_h \, \delta x + \beta_h}. \tag{5.30}$$

Using Eqs. (5.8), (5.29), and (5.30), the problem can be reduced to a tridiagonal matrix equation $\mathbf{M}\,\mathbf{u} = \mathbf{w}$, where the left, center, and right diagonals of $\mathbf{M}$ possess the elements $a_i = 1$ for $i = 2, N$, with

$$b_1 = -2 - \frac{\beta_l}{\alpha_l \, \delta x - \beta_l}, \tag{5.31}$$

and $b_i = -2$ for $i = 2, N - 1$, plus

$$b_N = -2 + \frac{\beta_h}{\alpha_h \, \delta x + \beta_h}, \tag{5.32}$$

and $c_i = 1$ for $i = 1, N - 1$, respectively. The elements of the right-hand side are

$$w_1 = v_1 \, (\delta x)^2 - \frac{\gamma_l \, \delta x}{\alpha_l \, \delta x - \beta_l}, \tag{5.33}$$

with $w_i = v_i \, (\delta x)^2$ for $i = 2, N - 1$, and

$$w_N = v_N \, (\delta x)^2 - \frac{\gamma_h \, \delta x}{\alpha_h \, \delta x + \beta_h}. \tag{5.34}$$

Our tridiagonal matrix equation can be inverted using the algorithm discussed previously.

5.5 An Example 1-D Poisson Solving Routine

Listed below is an example 1-d Poisson solving routine which utilizes the previously listed tridiagonal matrix solver and the `Blitz++` library (see Sect. 2.20).

```
// Poisson1D.cpp

// Function to solve Poisson's equation in 1-d:

//   d^2 u / dx^2 = v   for   xl <= x <= xh
```

```
//  alpha_l u + beta_l du/dx = gamma_l  at x=xl

//  alpha_h u + beta_h du/dx = gamma_h  at x=xh

// Arrays u and v assumed to be of extent N+2.

// Now, ith elements of arrays correspond to

//  x_i = xl + i * dx    i=0,N+1

// Here, dx = (xh - xl) / (N+1) is grid spacing.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void Poisson1D (Array<double,1>& u, Array<double,1> v,
                double alpha_l, double beta_l, double gamma_l,
                double alpha_h, double beta_h, double gamma_h,
                double dx)
{
  // Find N. Declare local arrays.
  int N = u.extent(0) - 2;
  Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2);

  // Initialize tridiagonal matrix
  for (int i = 2; i <= N; i++) a(i) = 1.;
  for (int i = 1; i <= N; i++) b(i) = -2.;
  b(1) -= beta_l / (alpha_l * dx - beta_l);
  b(N) += beta_h / (alpha_h * dx + beta_h);
  for (int i = 1; i <= N-1; i++) c(i) = 1.;

  // Initialize right-hand side vector
  for (int i = 1; i <= N; i++)
      w(i) = v(i) * dx * dx;
  w(1) -= gamma_l * dx / (alpha_l * dx - beta_l);
  w(N) -= gamma_h * dx / (alpha_h * dx + beta_h);

  // Invert tridiagonal matrix equation
  Tridiagonal (a, b, c, w, u);

  // Calculate i=0 and i=N+1 values
```

```
  u(0) = (gamma_l * dx - beta_l * u(1)) /
    (alpha_l * dx - beta_l);
  u(N+1) = (gamma_h * dx + beta_h * u(N)) /
    (alpha_h * dx + beta_h);
}
```

5.6   An Example Solution of Poisson's Equation in 1-D

Let us now solve Poisson's equation in one dimension, with mixed boundary conditions, using the finite difference technique discussed above. We seek the solution of

$$\frac{d^2 u(x)}{dx^2} = v(x), \tag{5.35}$$

in the region $0 \leq x \leq 1$, with $v(x) = 1 - 2 x^2$. The boundary conditions at $x_l = 0$ and $x_h = 1$ take the mixed form specified in Eqs. (5.25) and (5.26). Of course, we can solve this problem analytically. In fact,

$$u(x) = g + h x + \frac{x^2}{2} - \frac{x^4}{6}, \tag{5.36}$$

where

$$g = \frac{\gamma_l \left(\alpha_h + \beta_h \right) - \beta_l \left[\gamma_h - (\alpha_h + \beta_h)/3 \right]}{\alpha_l \, \alpha_h + \alpha_l \, \beta_h - \beta_l \, \alpha_h}, \tag{5.37}$$

$$h = \frac{\alpha_l \left[\gamma_h - (\alpha_h + \beta_h)/3 \right] - \gamma_l \, \alpha_h}{\alpha_l \, \alpha_h + \alpha_l \, \beta_h - \beta_l \, \alpha_h}. \tag{5.38}$$

Figure 62 shows a comparison between the analytic and finite difference solutions for $N = 100$. It can be seen that the finite difference solution mirrors the analytic solution almost exactly.

5.7   2-D problem with Dirichlet Boundary Conditions

Let us consider the solution of Poisson's equation in two dimensions. Suppose that

$$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = v(x, y), \tag{5.39}$$
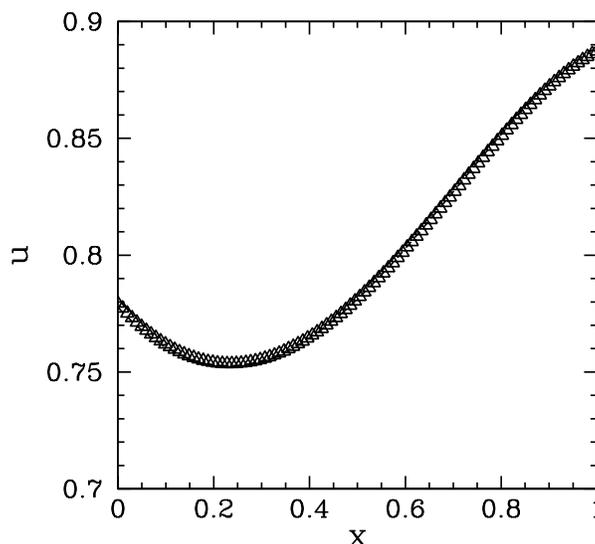
Figure 62: *Solution of Poisson's equation in one dimension with* $v = 1 - 2\,x^2$, $\alpha_l = 1$, $\beta_l = -1$, $\gamma_l = 1$, $\alpha_h = 1$, $\beta_h = 1$, *and* $\gamma_h = 1$. *The dotted curve (obscured) shows the analytic solution, whereas the open triangles show the finite difference solution for* $N = 100$.

for $x_l \leq x \leq x_h$, and $0 \leq y \leq L$. By direct analogy with our previous method of solution in the 1-d case, we could discretize the above 2-d problem using a second-order, central difference scheme in both the x- and y-directions. Unfortunately, such a discretization scheme yields a set of equations which *cannot* be reduced to a simple tridiagonal matrix equation. In fact, all of the efficient numerical algorithms for solving this type of problem are *iterative* in nature. For instance, the *Jacobi* method, the *Gauss-Seidel* method, the *successive over-relaxation* method, and the *multi-grid* method.[34] Regrettably, unless such iteration methods are extremely sophisticated (*e.g.*, the multi-grid method), and, hence, beyond the scope of this course, they tend to converge very poorly. In the following, rather than discuss iterative methods which do not work very well, we shall instead discuss a non-iterative method which works effectively for a restricted set of problems. The method in question is termed a *spectral method,* since it involves expanding u and v as truncated Fourier series in the y-direction.

Suppose that $u(x, y)$ satisfies mixed boundary conditions in the x-direction:

---

[34]See *Numerical recipes in C: the art of scientific computing*, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press, Cambridge, England, 1992), Sect. 19.5.

i.e.,

$$\alpha_l\, u(x,y) + \beta_l\, \frac{\partial u(x,y)}{\partial x} = \gamma_l(y), \tag{5.40}$$

at $x = x_l$, and

$$\alpha_h\, u(x,y) + \beta_h\, \frac{\partial u(x,y)}{\partial x} = \gamma_h(y), \tag{5.41}$$

at $x = x_h$. Here, α_l, β_l, *etc.*, are known constants, whereas γ_l, γ_h are known functions of y. Furthermore, suppose that $u(x,y)$ satisfies the following simple Dirichlet boundary conditions in the y-direction:

$$u(x,0) = u(x,L) = 0. \tag{5.42}$$

Note that, since $u(x,y)$ is a potential, and, hence, probably undetermined to an arbitrary additive constant, the above boundary conditions are equivalent to demanding that u take the *same* constant value on both the upper and lower boundaries in the y-direction.

Let us write $u(x,y)$ as a Fourier series in the y-direction:

$$u(x,y) = \sum_{j=0}^{\infty} U_j(x)\, \sin(j\,\pi y/L). \tag{5.43}$$

Note that the above expression for u automatically satisfies the boundary conditions in the y-direction. The $\sin(j\,\pi y/L)$ functions are *orthogonal*, and form a complete set, in the interval $y = 0, L$. In fact,

$$\frac{2}{L} \int_0^L \sin(j\,\pi y/L)\, \sin(k\,\pi y/L)\, dy = \delta_{jk}. \tag{5.44}$$

Thus, we can write the source term as

$$v(x,y) = \sum_{j=0}^{\infty} V_j(x)\, \sin(j\,\pi y/L), \tag{5.45}$$

where

$$V_j(x) = \frac{2}{L} \int_0^L v(x,y)\, \sin(j\,\pi y/L)\, dy. \tag{5.46}$$

Furthermore, the boundary conditions in the x-direction become

$$\alpha_l \, U_j(x) + \beta_l \, \frac{dU_j(x)}{dx} = \Gamma_{lj}, \tag{5.47}$$

at $x = x_l$, and

$$\alpha_h \, U_j(x) + \beta_h \, \frac{dU_j(x)}{dx} = \Gamma_{hj}, \tag{5.48}$$

at $x = x_h$, where

$$\Gamma_{lj} = \frac{2}{L} \int_0^L \gamma_l(y) \, \sin(j \, \pi \, y/L) \, dy, \tag{5.49}$$

etc.

   Substituting Eqs. (5.43) and (5.45) into Eq. (5.39), and equating the coefficients of the $\sin(j \, \pi \, y/L)$ (since these functions are orthogonal), we obtain

$$\frac{d^2 U_j(x)}{dx^2} - \frac{j^2 \, \pi^2}{L^2} \, U_j(x) = V_j(x), \tag{5.50}$$

for $j = 0, \infty$. Now, we can discretize the problem in the y-direction by truncating our Fourier expansion: *i.e.*, by only solving the above equations for $j = 0, J$, rather than $j = 0, \infty$. This is essentially equivalent to discretization in the y-direction on the equally-spaced grid-points $y_j = j \, L/J$. The problem is discretized in the x-direction by dividing the domain into equal segments, according to Eq. (5.7), and approximating d^2/dx^2 via the second-order, central difference scheme specified in Eq. (5.8). Thus, we obtain

$$U_{i-1,j} - (2 + j^2 \, \kappa^2) \, U_{i,j} + U_{i+1,j} = V_{i,j} \, (\delta x)^2, \tag{5.51}$$

for $i = 1, N$ and $j = 0, J$. Here, $U_{i,j} \equiv U_j(x_i)$, $V_{i,j} \equiv V_j(x_i)$, and $\kappa = \pi \, \delta x/L$. The boundary conditions (5.47) and (5.48) discretize to give:

$$U_{0,j} = \frac{\Gamma_{lj} \, \delta x - \beta_l \, U_{1,j}}{\alpha_l \, \delta x - \beta_l}, \tag{5.52}$$

$$U_{N+1,j} = \frac{\Gamma_{hj} \, \delta x + \beta_h \, U_{N,j}}{\alpha_h \, \delta x + \beta_h}, \tag{5.53}$$

for $j = 0, J$. Eqs. (5.51), (5.52), and (5.53) constitute a set of $J + 1$ *uncoupled* tridiagonal matrix equations (with one equation for each separate j value). These

equations can be inverted, using the algorithm discussed in Sect. 5.4, to give the $U_{i,j}$. Finally, the $u(x_i, y_j)$ values can be reconstructed from Eq. (5.43). Hence, we have solved the problem.

5.8   2-d Problem with Neumann Boundary Conditions

Let us redo the above calculation, replacing the Dirichlet boundary conditions (5.42) with the following simple Neumann boundary conditions:

$$\frac{\partial u(x, y = 0)}{\partial y} = \frac{\partial u(x, y = L)}{\partial y} = 0. \tag{5.54}$$

In this case, we can express $u(x, y)$ in the form

$$u(x, y) = \sum_{j=0}^{\infty} U_j(x) \, \cos(j\,\pi y/L), \tag{5.55}$$

which automatically satisfies the boundary conditions in the y-direction. Likewise, we can write the source term $v(x, y)$ as

$$v(x, y) = \sum_{j=0}^{\infty} V_j(x) \, \cos(j\,\pi y/L), \tag{5.56}$$

where

$$V_j(x) = \frac{2}{L} \int_0^L v(x, y) \, \cos(j\,\pi y/L) \, dy, \tag{5.57}$$

since

$$\frac{2}{L} \int_0^L \cos(j\,\pi y/L) \, \cos(k\,\pi y/L) \, dy = \delta_{jk}. \tag{5.58}$$

Finally, the boundary conditions in the x-direction become

$$\alpha_l \, U_j(x) + \beta_l \, \frac{dU_j(x)}{dx} = \Gamma_{lj}, \tag{5.59}$$

at $x = x_l$, and

$$\alpha_h \, U_j(x) + \beta_h \, \frac{dU_j(x)}{dx} = \Gamma_{hj}, \tag{5.60}$$

at $x = x_h$, where

$$\Gamma_{lj} = \frac{2}{L} \int_0^L \gamma_l(y) \, \cos(j\pi y/L) \, dy, \tag{5.61}$$

etc. Note, however, that the factor in front of the integrals in Eqs. (5.57) and (5.61) takes the special value $1/L$ for the $j = 0$ harmonic.

As before, we truncate the Fourier expansion in the y-direction, and discretize in the x-direction, to obtain the set of tridiagonal matrix equations specified in Eqs. (5.51), (5.52), and (5.53). We can solve these equations to obtain the $U_{i,j}$, and then reconstruct the $u(x_i, y_j)$ from Eq. (5.55). Hence, we have solved the problem.

5.9   The Fast Fourier Transform

The method outlined in Sect. 5.7 for solving Poisson's equation in 2-d with simple Dirichlet boundary conditions in the y-direction requires us to perform very many *Fourier-sine transforms*:

$$F_j^S = \frac{2}{J} \sum_{k=1}^{J-1} f_k \, \sin(j\,k\,\pi/J) \tag{5.62}$$

for $j = 0, J$, and inverse Fourier-sine transforms:

$$f_j = \sum_{k=1}^{J-1} F_k^S \, \sin(j\,k\,\pi/J). \tag{5.63}$$

Here, f_j is the value of $f(y)$ at $y_j = j\,L/J$. Thus, Eq. (5.62) is analogous to Eqs. (5.46) and (5.49), whereas Eq. (5.63) can be used to reconstruct the $u(x_i, y_j)$ from the $U_{i,j}$. Likewise, the method outlined in Sect. 5.8 for solving Poisson's equation in 2-d with simple Neumann boundary conditions in the y-direction requires us to perform very many *Fourier-cosine transforms*:

$$F_j^C = \frac{f_0}{J} + \frac{2}{J} \sum_{k=1}^{J-1} f_k \, \cos(j\,k\,\pi/J) + \frac{(-1)^j \, f_J}{J} \tag{5.64}$$

for $j = 0, J$, and inverse Fourier-cosine transforms:

$$f_j = \sum_{k=0}^{J} F_k^C \cos(j\,k\,\pi/J). \tag{5.65}$$

Unfortunately, performing such transforms directly requires $O(J^2)$ arithmetic operations, which means that they are *extremely expensive* in terms of cpu resources. There is, however, an ingenious algorithm for performing Fourier transforms which only takes $O(J \ln J)$ arithmetic operations [which is much less than $O(J^2)$ operations when J is large]. This algorithm is known as the *fast Fourier transform* or FFT.[35]

   The details of the FFT algorithm lie beyond the scope of this course. Roughly speaking, the algorithm works by building up the transform in stages using 2, 4, 8, 16, *etc.* grid-points. In this course, we shall employ the best-known publicly available FFT library, called the `fftw` library,[36] to perform all of our Fourier-sine and -cosine transforms. Unfortunately, the `fftw` library does not directly calculate Fourier-sine and -cosine transforms.[37] Instead, it calculates *complex* Fourier transforms:

$$F_j = \frac{1}{2J} \sum_{k=0}^{2J-1} f_k \exp(-i\,j\,k\,\pi/J) \tag{5.66}$$

for $j = 0, 2J - 1$, and complex inverse Fourier transforms:

$$f_j = \sum_{k=0}^{2J-1} F_k \exp(i\,j\,k\,\pi/J). \tag{5.67}$$

Note that f_j and F_j are *periodic* in j with period $2\,J$. Note, further, that the data-sets associated with complex Fourier transforms contain *twice* as many elements as the data-sets associated with sine and cosine transforms. However, we can easily convert a sine or cosine transform into a complex transform by *extending* its data-set. Thus, for a sine transform we write:

$$f_{2J-j} = -f_j, \tag{5.68}$$

$$F_{2J-j} = -F_j, \tag{5.69}$$

---

[35]See *Numerical recipes in C: the art of scientific computing*, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press, Cambridge, England, 1992), Sect. 12.2.

[36]See `http://www.fftw.org`

[37]This is the case for version 2 of the library (which is the version used in this course), but not version 3.

for $j = 1, J - 1$, in which case

$$F_j^S = 2\,i\,F_j. \qquad (5.70)$$

Likewise, for a cosine transform we write:

$$f_{2J-j} \; = \; f_j, \qquad (5.71)$$

$$F_{2J-j} \; = \; F_j, \qquad (5.72)$$

for $j = 1, J - 1$, in which case

$$F_j^C = 2\,F_j. \qquad (5.73)$$

Listed below are a set of wrapper routines which employ the `fftw` library to perform Fourier-sine and -cosine transforms.

```
// FFT.cpp

// Set of functions to calculate Fourier-cosine and -sine transforms
// of real data using fftw Fast-Fourier-Transform library.
// Input/ouput arrays are assumed to be of extent J+1.
// Uses version 2 of fftw library (incompatible with vs 3).

#include <fftw.h>
#include <blitz/array.h>

using namespace blitz;

// Calculates Fourier-cosine transform of array f in array F
void fft_forward_cos (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (ff[0]) = f(0); c_im (ff[0]) = 0.;
  c_re (ff[J]) = f(J); c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = f(j); c_im (ff[2*J-j]) = 0.;
    }
```

```cpp
  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = c_re (FF[0]); F(J) = c_re (FF[J]);
  for (int j = 1; j < J; j++)
    {
      F(j) = 2. * c_re (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}

// Calculates inverse Fourier-cosine transform of array F in array f
void fft_backward_cos (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = F(0); c_im (FF[0]) = 0.;
  c_re (FF[J]) = F(J); c_im (FF[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (FF[j]) = F(j) / 2.; c_im (FF[j]) = 0.;
      FF[2*J-j] = FF[j];
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  f(0) = c_re (ff[0]); f(J) = c_re (ff[J]);
  for (int j = 1; j < J; j++)
    {
      f(j) = c_re (ff[j]);
    }
}
```

```
// Calculates Fourier-sine transform of array f in array F
void fft_forward_sin (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (ff[0]) = 0.; c_im (ff[0]) = 0.;
  c_re (ff[J]) = 0.; c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = - f(j); c_im (ff[2*J-j]) = 0.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = 0.; F(J) = 0.;
  for (int j = 1; j < J; j++)
    {
      F(j) = - 2. * c_im (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}

// Calculates inverse Fourier-sine transform of array F in array f
void fft_backward_sin (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = 0.; c_im (FF[0]) = 0.;
  c_re (FF[J]) = 0.; c_im (FF[J]) = 0.;
```

```
  for (int j = 1; j < J; j++)
    {
      c_re (FF[j]) = 0.; c_im (FF[j]) = - F(j) / 2.;
      c_re (FF[2*J-j]) = 0.; c_im (FF[2*J-j]) = F(j) / 2.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  f(0) = 0.; f(J) = 0.;
  for (int j = 1; j < J; j++)
    {
      f(j) = c_re (ff[j]);
    }
}
```

5.10   An Example 2-D Poisson Solving Routine

Listed below is an example 2-d Poisson solving routine which employs the previously listed tridiagonal matrix inversion and FFT wrapper routines, as well as the `Blitz++` library.

```
// Poisson2d.cpp

// Function to solve Poisson's equation in 2-d:

//  d^2 u / dx^2 + d^2 u / dy^2 = v  for  xl <= x <= xh  and  0 <= y <= L

//  alphaL u + betaL du/dx = gammaL(y)  at x=xl

//  alphaH u + betaH du/dx = gammaH(y)  at x=xh

// In y-direction, either simple Dirichlet boundary conditions:

//  u(x,0) = u(x,L) = 0

// or simple Neumann boundary conditions:

//  du/dy(x,0) = du/dy(x,L) = 0
```

```cpp
// Matrices u and v assumed to be of extent N+2, J+1.
// Arrays gammaL, gammaH assumed to be of extent J+1.

// Now, (i,j)th elements of matrices correspond to

//   x_i = xl + i * dx     i=0,N+1

//   y_j = j * L / J       j=0,J

// Here, dx = (xh - xl) / (N+1) is grid spacing in x-direction.

// Now, kappa = pi * dx / L

// Finally, Neumann=0/1 selects Dirichlet/Neumann bcs in y-direction.

#include <blitz/array.h>

using namespace blitz;

void fft_forward_cos (Array<double,1> f, Array<double,1>& F);
void fft_backward_cos (Array<double,1> F, Array<double,1>& f);
void fft_forward_sin (Array<double,1> f, Array<double,1>& F);
void fft_backward_sin (Array<double,1> F, Array<double,1>& f);
void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void Poisson2D (Array<double,2>& u, Array<double,2> v,
                double alphaL, double betaL, Array<double,1> gammaL,
                double alphaH, double betaH, Array<double,1> gammaH,
                double dx, double kappa, int Neumann)
{
  // Find N and J. Declare local arrays.
  int N = u.extent(0) - 2;
  int J = u.extent(1) - 1;
  Array<double,2> V(N+2, J+1), U(N+2, J+1);
  Array<double,1> GammaL(J+1), GammaH(J+1);

  // Fourier transform boundary conditions
  if (Neumann)
    {
      fft_forward_cos (gammaL, GammaL);
      fft_forward_cos (gammaH, GammaH);
    }
  else
```

```
    {
      fft_forward_sin (gammaL, GammaL);
      fft_forward_sin (gammaH, GammaH);
    }

  // Fourier transform source term
  for (int i = 1; i <= N; i++)
    {
      Array<double,1> In(J+1), Out(J+1);

      for (int j = 0; j <= J; j++) In(j) = v(i, j);

      if (Neumann)
        fft_forward_cos (In, Out);
      else
        fft_forward_sin (In, Out);

      for (int j = 0; j <= J; j++) V(i, j) = Out(j);
    }

  // Solve tridiagonal matrix equations
  if (Neumann)
    {
      for (int j = 0; j <= J; j++)
        {
          Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2), uu(N+2);

          // Initialize tridiagonal matrix
          for (int i = 2; i <= N; i++) a(i) = 1.;
          for (int i = 1; i <= N; i++)
            b(i) = -2. - double (j * j) * kappa * kappa;
          b(1) -= betaL / (alphaL * dx - betaL);
          b(N) += betaH / (alphaH * dx + betaH);
          for (int i = 1; i <= N-1; i++) c(i) = 1.;

          // Initialize right-hand side vector
          for (int i = 1; i <= N; i++)
            w(i) = V(i, j) * dx * dx;
          w(1) -= GammaL(j) * dx / (alphaL * dx - betaL);
          w(N) -= GammaH(j) * dx / (alphaH * dx + betaH);

          // Invert tridiagonal matrix equation
          Tridiagonal (a, b, c, w, uu);
          for (int i = 1; i <= N; i++) U(i, j) = uu(i);
        }
```

```
    }
  else
    {
      for (int j = 1; j < J; j++)
        {
          Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2), uu(N+2);

          // Initialize tridiagonal matrix
          for (int i = 2; i <= N; i++) a(i) = 1.;
          for (int i = 1; i <= N; i++)
            b(i) = -2. - double (j * j) * kappa * kappa;
          b(1) -= betaL / (alphaL * dx - betaL);
          b(N) += betaH / (alphaH * dx + betaH);
          for (int i = 1; i <= N-1; i++) c(i) = 1.;

          // Initialize right-hand side vector
          for (int i = 1; i <= N; i++)
            w(i) = V(i, j) * dx * dx;
          w(1) -= GammaL(j) * dx / (alphaL * dx - betaL);
          w(N) -= GammaH(j) * dx / (alphaH * dx + betaH);

          // Invert tridiagonal matrix equation
          Tridiagonal (a, b, c, w, uu);
          for (int i = 1; i <= N; i++) U(i, j) = uu(i);
        }

      for (int i = 1; i <= N  ; i++)
        {
          U(i, 0) = 0.; U(i, J) = 0.;
        }
    }

  // Reconstruct solution via inverse Fourier transform
  for (int i = 1; i <= N; i++)
    {
      Array<double,1> In(J+1), Out(J+1);

      for (int j = 0; j <= J; j++) In(j) = U(i, j);

      if (Neumann)
        fft_backward_cos (In, Out);
      else
        fft_backward_sin (In, Out);

      for (int j = 0; j <= J; j++) u(i, j) = Out(j);
```

```
    }

  // Calculate i=0 and i=N+1 values
  for (int j = 0; j <= J; j++)
    {
      u(0, j) = (gammaL(j) * dx - betaL * u(1, j)) /
        (alphaL * dx - betaL);
      u(N+1, j) = (gammaH(j) * dx + betaH * u(N, j)) /
        (alphaH * dx + betaH);
    }
}
```

5.11   An Example Solution of Poisson's Equation in 2-D

Let us now use the techniques discussed above to solve Poisson's equation in two dimensions. Suppose that the source term is

$$v(x, y) = 6\,x\,y\,(1 - y) - 2\,x^3 \tag{5.74}$$

for $0 \le x \le 1$ and $0 \le y \le 1$. The boundary conditions at $x = 0$ are $\alpha_l = 1$, $\beta_l = 0$, and $\gamma_l = 0$ [see Eq. (5.40)], whereas the boundary conditions at $x = 1$ are $\alpha_h = 1$, $\beta_h = 0$, and $\gamma_h = y\,(1 - y)$ [see Eq. (5.41)]. The simple Dirichlet boundary conditions $u(x, 0) = u(x, 1) = 0$ are applied at $y = 0$ and $y = 1$. Of course, this problem can be solved analytically to give

$$u(x, y) = y\,(1 - y)\,x^3. \tag{5.75}$$

Figures 63 and 64 show comparisons between the analytic and finite difference solutions for $N = J = 64$. It can be seen that the finite difference solution mirrors the analytic solution almost exactly.

As a second example, suppose that the source term is

$$v(x, y) = -2\,(2\,y^3 - 3\,y^2 + 1) + 6\,(1 - x^2)\,(2\,y - 1) \tag{5.76}$$

for $0 \le x \le 1$ and $0 \le y \le 1$. The boundary conditions at $x = 0$ are $\alpha_l = 1$, $\beta_l = 0$, and $\gamma_l = 2\,y^3 - 3\,y^2 + 1$ [see Eq. (5.40)], whereas the boundary conditions at $x = 1$ are $\alpha_h = 1$, $\beta_h = 0$, and $\gamma_h = 0$ [see Eq. (5.41)]. The simple Neumann boundary
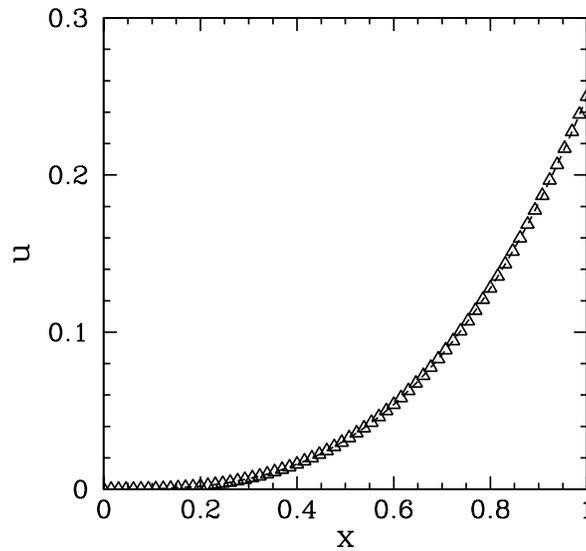
Figure 63: *Solution of Poisson's equation in two dimensions with simple Dirichlet boundary conditions in the y-direction. The solution is plotted versus x at y = 0.5. The dotted curve (obscured) shows the analytic solution, whereas the open triangles show the finite difference solution for N = J = 64.*
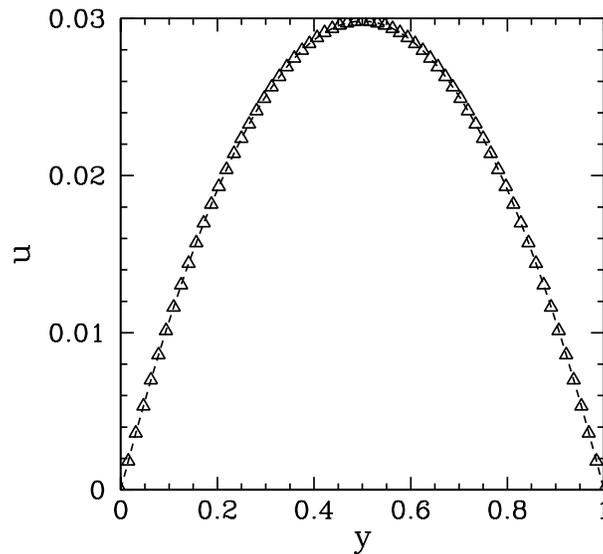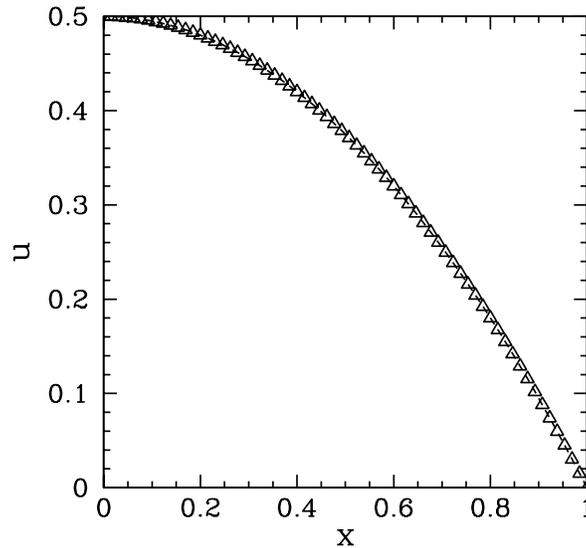

Figure 64: *Solution of Poisson's equation in two dimensions with simple Dirichlet boundary conditions in the y-direction. The solution is plotted versus y at x = 0.5. The dotted curve (obscured) shows the analytic solution, whereas the open triangles show the finite difference solution for N = J = 64.*
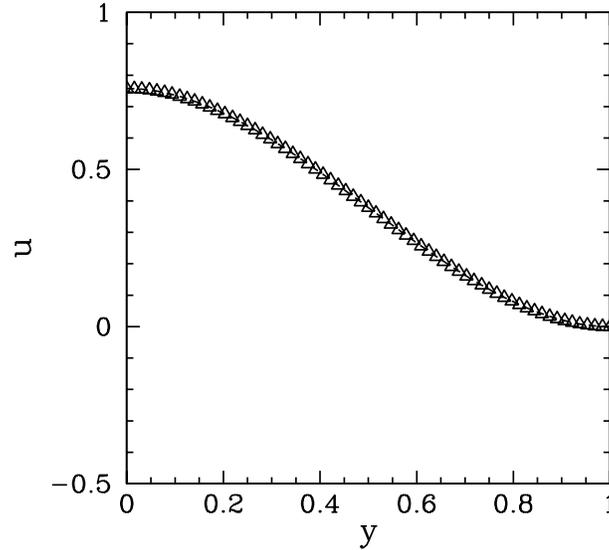
Figure 65: *Solution of Poisson's equation in two dimensions with simple Neumann boundary conditions in the y-direction. The solution is plotted versus x at y = 0.5. The dotted curve (obscured) shows the analytic solution, whereas the open triangles show the finite difference solution for N = J = 64.*

conditions $\partial u(x,0)/\partial y = \partial u(x,1)/\partial y = 0$ are applied at $y = 0$ and $y = 1$. Of course, this problem can be solved analytically to give

$$u(x, y) = (1 - x^2)(2y^3 - 3y^2 + 1). \tag{5.77}$$

Figures 65 and 66 show comparisons between the analytic and finite difference solutions for $N = J = 64$. It can be seen that the finite difference solution mirrors the analytic solution almost exactly.

5.12   Example 2-D Electrostatic Calculation

Let us perform an example 2-d electrostatic calculation. Consider a charged wire running parallel to the axis of a uniform, hollow, rectangular, conducting channel. Suppose that the vertices of the channel lie at $(x, y) = (0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. Suppose, further, that the wire carries a uniform charge per unit length of magnitude unity. The electric potential $\phi(x, y)$ inside the channel satisfies [see

213

Figure 66: *Solution of Poisson's equation in two dimensions with simple Neumann boundary conditions in the y-direction. The solution is plotted versus y at x = 0.5. The dotted curve (obscured) shows the analytic solution, whereas the open triangles show the finite difference solution for N = J = 64.*

Eq. (5.3)]

$$\frac{\partial^2 \phi(x,y)}{\partial x^2} + \frac{\partial^2 \phi(x,y)}{\partial y^2} = v(x,y) = -\delta(x - x_0)\,\delta(y - y_0), \qquad (5.78)$$

where (x_0, y_0) are the coordinates of the wire. Here, we have conveniently normalized our units such that the factor ϵ_0 is absorbed into the normalization. Assuming that the box is grounded, the potential is subject to the Dirichlet boundary conditions $\phi = 0$ at $x = 0$, $x = 1$, $y = 0$, and $y = 1$. We require the solution in the region $0 \le x \le 1$ and $0 \le y \le 1$.

Note that when discretizing Eq. (5.78) the right-hand side becomes

$$v(x_i, y_j) = -\frac{1}{\delta x \, \delta y} \qquad (5.79)$$

on the grid-point closest to the wire, with $v(x_i, y_j) = 0$ on the remaining grid-points. Here, δx and δy are the grid spacings in the x- and y- directions, respectively.

Figures 67 and 68 show the electric potential $\phi(x,y)$ and electric field $\mathbf{E} = -\nabla\phi$ generated by a wire placed at the center of the channel: *i.e.*, $(x_0, y_0) =$
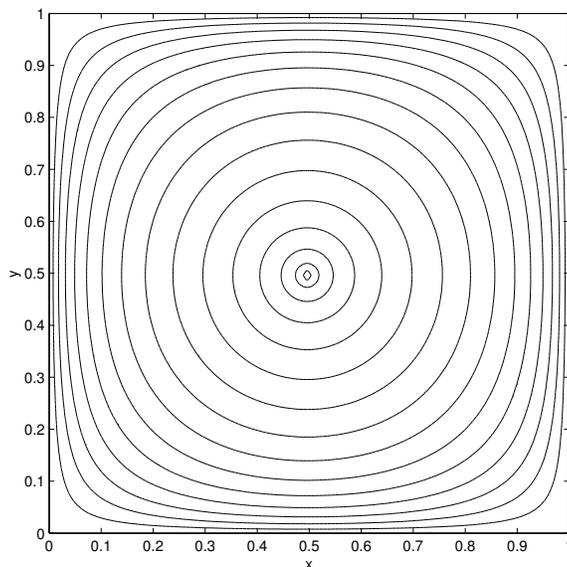
Figure 67: *Contour plot of the electric potential generated by a charged wire placed at the center of a grounded rectangular channel. The wire is located at* $(x, y) = (0.5, 0.5)$, *whereas the channel walls are at* $x = 0$, $x = 1$, $y = 0$, *and* $y = 1$. *Calculation performed with* $N = J = 128$.
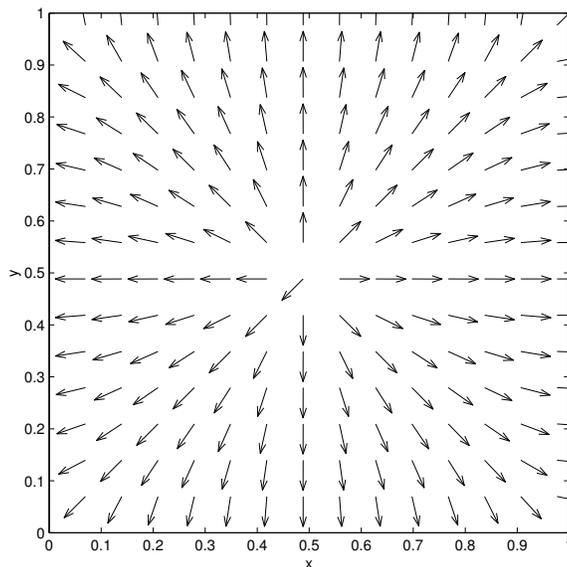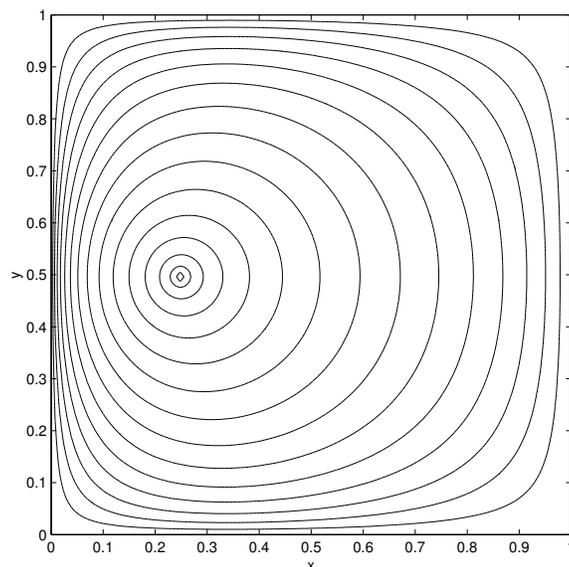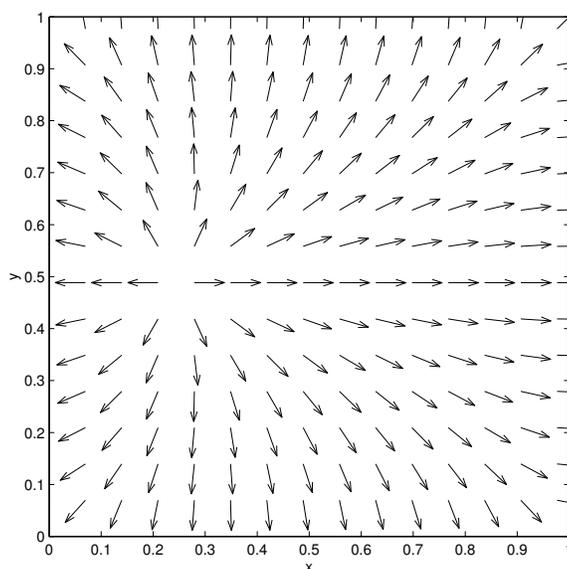

Figure 68: *Vector plot showing the direction of the electric field generated by a charged wire placed at the center of a grounded rectangular channel. The wire is located at* $(x, y) = (0.5, 0.5)$, *whereas the channel walls are at* $x = 0$, $x = 1$, $y = 0$, *and* $y = 1$. *Calculation performed with* $N = J = 128$.

Figure 69: *Contour plot of the electric potential generated by a charged wire offset from the center of a grounded rectangular channel. The wire is located at* $(x, y) = (0.25, 0.5)$*, whereas the channel walls are at* $x = 0$*,* $x = 1$*,* $y = 0$*, and* $y = 1$*. Calculation performed with* $N = J = 128$*.*

$(0.5, 0.5)$. The calculation was performed with the previously listed 2-d Poisson solver using $N = J = 128$.

Figures 69 and 70 show the electric potential $\phi(x, y)$ and electric field $\mathbf{E} = -\nabla\phi$ generated by a wire offset from the center of the channel: *i.e.,* $(x_0, y_0) = (0.25, 0.5)$. The calculation was performed with the previously listed 2-d Poisson solver using $N = J = 128$.

5.13 3-D Problems

The techniques discussed in Sects. 5.7 and 5.8 for solving Poisson's equation in two dimensions with a restricted class of boundary conditions can easily be generalized to three dimensions. In the 3-d case, it is necessary to Fourier transform in *two* directions (the y and z directions, say) in order to reduce the problem to a system of uncoupled tridiagonal matrix equations. These equations can be inverted in the usual manner, and the solution can then be reconstructed via a double inverse Fourier transform.

Figure 70: *Vector plot showing the direction of the electric field generated by a charged wire offset from the center of a grounded rectangular channel. The wire is located at $(x, y) = (0.25, 0.5)$, whereas the channel walls are at $x = 0$, $x = 1$, $y = 0$, and $y = 1$. Calculation performed with $N = J = 128$.*

6   The Diffusion Equation

6.1   Introduction

The diffusion equation

$$\frac{\partial T(\mathbf{r}, t)}{\partial t} = D \, \nabla^2 T(\mathbf{r}, t), \tag{6.1}$$

where $D > 0$ is the (uniform) coefficient of diffusion, describes many interesting physical phenomena. For instance, in heat conduction we can write

$$\mathbf{q} = -\kappa \, \nabla T, \tag{6.2}$$

where $\mathbf{q}$ is the heat flux, T the temperature, and κ the coefficient of thermal conductivity. The above equation merely states that heat flows down a temperature gradient. In the absence of sinks or sources of heat, energy conservation requires that

$$-\frac{\partial Q}{\partial t} = \int \mathbf{q} \cdot d\mathbf{S}, \tag{6.3}$$

where Q is the thermal energy contained in some volume V bounded by a closed surface S. The above equation states that the rate of decrease of the thermal energy content of volume V equals the instantaneous heat flux flowing across its boundary. However, $Q = \int c \, T \, dV$, where c is the heat capacity per unit volume. Making use of the previous equations, as well as the divergence theorem, we obtain the following diffusion equation for the temperature:

$$\frac{\partial T}{\partial t} = D \, \nabla^2 T, \tag{6.4}$$

where $D = \kappa/c$. In a typical heat conduction problem, we are given the temperature $T(\mathbf{r}, t_0)$ at some initial time t_0, and then asked to evaluate $T(\mathbf{r}, t)$ at all subsequent times. Such a problem is known as an *initial value problem*. The spatial boundary conditions can be either of type Dirichlet (*i.e.*, T specified on the boundary), type Neumann (*i.e.*, ∇T specified on the boundary), or some combination.

6.2   1-D Problem with Mixed Boundary Conditions

Consider the solution of the diffusion equation in one dimension. Suppose that

$$\frac{\partial T(x,t)}{\partial t} = D \frac{\partial^2 T(x,t)}{\partial x^2}, \tag{6.5}$$

for $x_l \leq x \leq x_h$, subject to the mixed spatial boundary conditions

$$\alpha_l(t)\, T(x,t) + \beta_l(t)\, \frac{\partial T(x,t)}{\partial x} = \gamma_l(t), \tag{6.6}$$

at $x = x_l$, and

$$\alpha_h(t)\, T(x,t) + \beta_h(t)\, \frac{\partial T(x,t)}{\partial x} = \gamma_h(t), \tag{6.7}$$

at $x = x_h$. Here, α_l, β_l, *etc.*, are known functions of time. Of course, $T(x,t_0)$ must be specified at some initial time t_0.

Equation (6.5) needs to be discretized in both time and space. In time, we discretize on the equally spaced grid

$$t_n = t_0 + n\,\delta t, \tag{6.8}$$

where δt is the *time-step*. Adopting a simple first-order differencing scheme, Eq. (6.5) becomes

$$\frac{T(x,t_{n+1}) - T(x,t_n)}{\delta t} = D\,\frac{\partial^2 T(x,t_n)}{\partial x^2} + O(\delta t). \tag{6.9}$$

In space, we discretize on the usual equally spaced grid-points specified in Eq. (5.7), and approximate d^2/dx^2 via the second-order, central difference scheme introduced in Eq. (5.8). The spatial boundary conditions are discretized in a similar manner to Eqs. (5.27) and (5.28). Thus, Eq. (6.9) yields

$$\frac{T_i^{n+1} - T_i^n}{\delta t} = D\,\frac{T_{i-1}^n - 2\,T_i^n + T_{i+1}^n}{(\delta x)^2}, \tag{6.10}$$

or

$$T_i^{n+1} = T_i^n + C\left(T_{i-1}^n - 2\,T_i^n + T_{i+1}^n\right) \tag{6.11}$$

for $i = 1, N$, where $T_i^n \equiv T(x_i, t_n)$, and $C = D\, \delta t/(\delta x)^2$. The discretized boundary conditions take the form

$$T_0^n = \frac{\gamma_l^n\, \delta x - \beta_l^n\, T_1^n}{\alpha_l^n\, \delta x - \beta_l^n}, \tag{6.12}$$

$$T_{N+1}^n = \frac{\gamma_h^n\, \delta x + \beta_h^n\, T_N^n}{\alpha_h^n\, \delta x + \beta_h^n}, \tag{6.13}$$

where $\gamma_l^n \equiv \gamma_l(t_n)$, *etc.* The discretization scheme outlined above is termed first-order in time and second-order in space.

Equations (6.11)–(6.13) constitute a fairly straightforward iterative scheme which can be used to evolve the $T(x, t)$ in time.

6.3   An Example 1-D Diffusion Equation Solver

Listed below is an example 1-d diffusion equation solving routine which makes use of the `Blitz++` library.

```
// Diffusion1D.cpp

// Function to evolve diffusion equation in 1-d:

//   dT / dt = D d^2 T / dx^2   for   xl <= x <= xh

//   alpha_l T + beta_l dT/dx = gamma_l   at x=xl

//   alpha_h T + beta_h dT/dx = gamma_h   at x=xh

// Array T assumed to be of extent N+2.

// Now, ith element of array corresponds to

//   x_i = xl + i * dx    i=0,N+1

// Here, dx = (xh - xl) / (N+1) is grid spacing.

// Function evolves T by single time-step.

// C = D dt / dx^2, where dt is time-step.
```

```
// Uses explicit scheme.

#include <blitz/array.h>

using namespace blitz;

void Diffusion1D (Array<double,1>& T,
                  double alpha_l, double beta_l, double gamma_l,
                  double alpha_h, double beta_h, double gamma_h,
                  double dx, double C)
{
  // Set N. Declare local array.
  int N = T.extent(0) - 2;
  Array<double,1> T0(N+2);

  // Evolve T
  T0 = T;
  for (int i = 1; i <= N; i++)
    T(i) += C * (T0(i-1) - 2. * T0(i) + T0(i+1));

  // Set boundary conditions
  T(0) = (gamma_l * dx - beta_l * T(1)) / (alpha_l * dx - beta_l);
  T(N+1) = (gamma_h * dx + beta_h * T(N)) / (alpha_h * dx + beta_h);
}
```

6.4   An Example 1-D Solution of the Diffusion Equation

Let us now solve the diffusion equation in 1-d using the finite difference technique discussed above. We seek the solution of Eq. (6.5) in the region $-x_0 \leq x \leq x_0$, subject to the initial condition

$$T(x, t_0) = \exp\left(\frac{-x^2}{4\,D\,t_0}\right), \tag{6.14}$$

where $t_0 > 0$. The spatial boundary conditions are

$$T(\pm x_0, t) = \sqrt{\frac{t_0}{t}} \, \exp\left(\frac{-x_0^2}{4\,D\,t}\right). \tag{6.15}$$
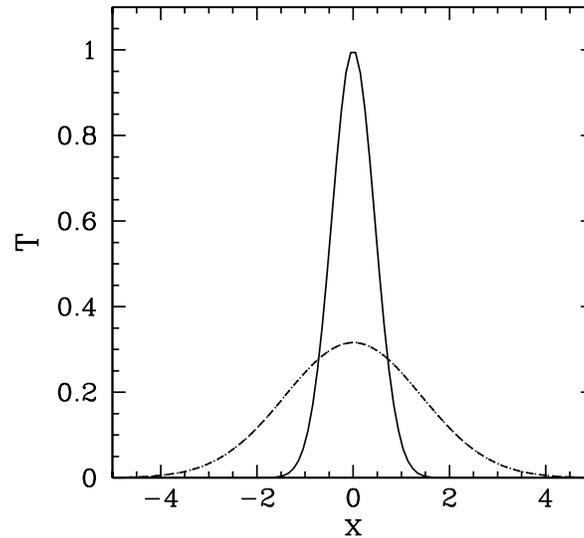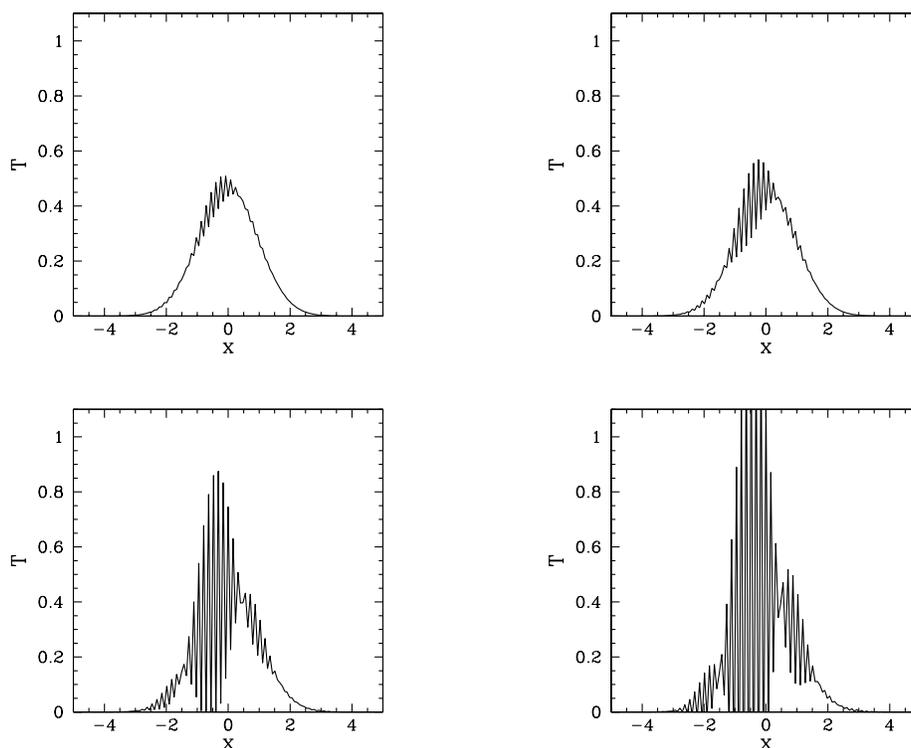
Figure 71: *Diffusive evolution of a 1-d Gaussian pulse. Numerical calculation performed using* $D = 1$, $x_0 = 5$, $\delta t = 4 \times 10^{-3}$, *and* $N = 100$. *The pulse is evolved from* $t = 0.1$ *to* $t = 1$. *The solid curve shows the initial condition at* $t = 0.1$, *the dashed curve the numerical solution at* $t = 1$, *and the dotted curve (obscured by the dashed curve) the analytic solution at* $t = 1$.

Of course, we can solve this problem analytically to give

$$T(x, t) = \sqrt{\frac{t_0}{t}} \, \exp\left(\frac{-x^2}{4 \, D \, t}\right). \tag{6.16}$$

Note that the above equation describes a Gaussian pulse which gradually decreases in height and broadens in width in such a manner that its area is conserved. The width of the pulse varies approximately as

$$\Delta x \sim \sqrt{D \, t}. \tag{6.17}$$

Moreover, the pulse approaches a δ-function as $t \to 0$.

Figure 71 shows a comparison between the analytic and numerical solutions for a calculation performed using $D = 1$, $x_0 = 5$, $t_0 = 0.1$, $\delta t = 4 \times 10^{-3}$, and $N = 100$. It can be seen that the analytic and numerical solutions are in excellent agreement.

It is reasonable to expect that as N increases at fixed δt (*i.e.*, the spatial resolution increases at fixed temporal resolution) the numerical solution should become

Figure 72: *Diffusive evolution of a 1-d Gaussian pulse. Numerical calculation performed using* $D = 1$, $x_0 = 5$, $\delta t = 4 \times 10^{-3}$, *and* $N = 125$. *The simulation is started at* $t = 0.1$. *The top-left, top-right, bottom-left, and bottom-right panels show the solution at* $t = 0.45$, $t = 0.46$, $t = 0.47$, *and* $t = 0.48$, *respectively.*

more and more accurate. This is indeed the case—at least, until N exceeds a critical value. Beyond this value, there is a catastrophic breakdown in the numerical solution. This breakdown is illustrated in Fig. 72. It can be seen that the solution develops rapidly growing short-wavelength oscillations. Indeed, the solution eventually becomes effectively infinite. Let us investigate this unusual and rather disturbing phenomenon.

6.5   von Neumann Stability Analysis

Clearly, our simple finite difference algorithm for solving the 1-d diffusion equation is subject to a numerical instability under certain circumstances. Let us try to establish when this instability occurs. Consider the time evolution of a single Fourier mode of wave-number k:

$$T(x, t) = \hat{T}(t)\, e^{i\,k\,x}. \tag{6.18}$$

Substitution of the above expression into our finite difference scheme (6.11) yields

$$\hat{T}^{n+1}\, e^{i\,k\,x_n} = \hat{T}^n\, e^{i\,k\,x_n}\left[1 + C\,(e^{-i\,k\,\delta x} - 2 + e^{+i\,k\,\delta x})\right], \tag{6.19}$$

or

$$\hat{T}^{n+1} = A\,\hat{T}^n, \tag{6.20}$$

where

$$A = 1 - 2\,C\,(1 - \cos k\,\delta x) = 1 - 4\,C\,\sin^2(k\,\delta x/2). \tag{6.21}$$

Thus, the amplitude of the Fourier mode is amplified by a factor A at each time-step. In order for the differencing scheme to be stable, the modulus of this amplification factor must be *less than unity* for *all* possible values of k. Now, the largest possible value of $\sin^2(k\,\delta x/2)$ is unity: hence, the wave-length corresponding to this value is that of the most unstable Fourier mode. In fact, the most unstable mode possesses a wave-length which is *half* the grid-spacing: *i.e.*, $\lambda = \delta x/2$. It follows from Eq. (6.21) that this mode is stable provided

$$C < \frac{1}{2}. \tag{6.22}$$

Finally, from the definition of C, our stability condition can be written

$$\delta t < \frac{(\delta x)^2}{2\,D}. \tag{6.23}$$

Note that $C = 0.408$ for the stable calculation shown in Fig. 71, whereas $C = 0.635$ for the unstable calculation shown in Fig. 72. Incidentally, the type of stability analysis outlined above is called *von Neumann stability analysis*. Note that the neglect of the spatial boundary conditions in the above calculation is justified because the unstable modes vary on very small length-scales which are typically of order the grid spacing.

According to Eq. (6.23), our finite difference scheme for solving the 1-d diffusion equation is only stable provided that the time-step remains below some critical value. Note that this critical value scales like the *square* of the grid-spacing. This is a very unfavorable scaling, since it implies that a doubling of the spatial resolution requires a simultaneous reduction in the time-step by a factor of four in order to maintain numerical stability. Certainly, the above constraint limits us to absurdly small time-steps in high resolution calculations. Is there any way of overcoming this constraint?

6.6   The Crank-Nicholson Scheme

Let us revisit our temporal differencing scheme:

$$\frac{T(x, t_{n+1}) - T(x, t_n)}{\delta t} = D\,\frac{\partial^2 T(x, t_n)}{\partial x^2} + O(\delta t). \tag{6.24}$$

Note that the right-hand side is evaluated entirely at the *start* of the time-step: *i.e.*, at t_n. This type of temporal differencing scheme is termed an *explicit* scheme. Now, explicit schemes are very straightforward to implement, but are also notoriously prone to numerical instabilities. Fortunately, we can often overcome these instabilities by making our differencing scheme *implicit* in nature. An implicit scheme is one in which the right-hand side is evaluated partly (or wholly) at the *end* of the time-step: *i.e.*, at t_{n+1}. Unfortunately, implicit schemes are generally a great deal more complicated to implement than explicit schemes.

The well-known Crank-Nicholson implicit method for solving the diffusion equation involves taking the average of the right-hand side between the beginning and end of the time-step. In other words,

$$\frac{T(x, t_{n+1}) - T(x, t_n)}{\delta t} = \frac{D}{2} \frac{\partial^2 T(x, t_n)}{\partial x^2} + \frac{D}{2} \frac{\partial^2 T(x, t_{n+1})}{\partial x^2} + O(\delta t)^2. \qquad (6.25)$$

As indicated by the error term, this method is actually *second-order* in time.

Adopting our usual spatial differencing scheme, the above expression yields

$$T_i^{n+1} - \frac{C}{2} \left(T_{i-1}^{n+1} - 2\, T_i^{n+1} + T_{i+1}^{n+1} \right) = T_i^n + \frac{C}{2} \left(T_{i-1}^n - 2\, T_i^n + T_{i+1}^n \right). \qquad (6.26)$$

When we perform a von Neumann stability analysis of the above scheme, we obtain the following expression for the amplification factor:

$$A = \frac{1 - 2\, C\, \sin^2(k\, \delta x/2)}{1 + 2\, C\, \sin^2(k\, \delta x/2)}. \qquad (6.27)$$

Note that $|A| < 1$ for all values of k. It follows that the Crank-Nicholson scheme is *unconditionally stable*. Unfortunately, Eq. (6.26) constitutes a tridiagonal matrix equation linking the T_i^{n+1} and the T_i^n. Thus, the price we pay for the high accuracy and unconditional stability of the Crank-Nicholson scheme is having to invert a tridiagonal matrix equation at each time-step. Usually, this price is well worth paying.

6.7   An Improved 1-D Diffusion Equation Solver

Listed below is an improved 1-d diffusion equation solver which uses the Crank-Nicholson scheme, as well as the previous listed tridiagonal matrix solver and the `Blitz++` library. Note the great structural similarity between this solver and the previously listed 1-d Poisson solver (see Sect. 5.5).

```
// CrankNicholson1D.cpp
```

```
// Function to evolve diffusion equation in 1-d:
```

```
//  dT / dt = D d^2 T / dx^2  for  xl <= x <= xh

//  alpha_l T + beta_l dT/dx = gamma_l  at x=xl

//  alpha_h T + beta_h dT/dx = gamma_h  at x=xh

// Array T assumed to be of extent N+2.

// Now, ith element of array corresponds to

//  x_i = xl + i * dx    i=0,N+1

// Here, dx = (xh - xl) / (N+1) is grid spacing.

// Function evolves T by single time-step.

// C = D dt / dx^2, where dt is time-step.

// Uses Crank-Nicholson implicit scheme.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void CrankNicholson1D (Array<double,1>& T,
                  double alpha_l, double beta_l, double gamma_l,
                  double alpha_h, double beta_h, double gamma_h,
                  double dx, double C)
{
  // Find N. Declare local arrays.
  int N = T.extent(0) - 2;
  Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2);

  // Initialize tridiagonal matrix
  for (int i = 2; i <= N; i++) a(i) = - 0.5 * C;
  for (int i = 1; i <= N; i++) b(i) = 1. + C;
  b(1) += 0.5 * C * beta_l / (alpha_l * dx - beta_l);
  b(N) -= 0.5 * C * beta_h / (alpha_h * dx + beta_h);
  for (int i = 1; i <= N-1; i++) c(i) = - 0.5 * C;

  // Initialize right-hand side vector
  for (int i = 1; i <= N; i++)
```

```
    w(i) = T(i) + 0.5 * C * (T(i-1) - 2. * T(i) + T(i+1));
  w(1) += 0.5 * C * gamma_l * dx / (alpha_l * dx - beta_l);
  w(N) += 0.5 * C * gamma_h * dx / (alpha_h * dx + beta_h);

  // Invert tridiagonal matrix equation
  Tridiagonal (a, b, c, w, T);

  // Calculate i=0 and i=N+1 values
  T(0) = (gamma_l * dx - beta_l * T(1)) /
    (alpha_l * dx - beta_l);
  T(N+1) = (gamma_h * dx + beta_h * T(N)) /
    (alpha_h * dx + beta_h);
}
```

6.8   An Improved 1-D Solution of the Diffusion Equation

Let us now solve the simple diffusion problem introduced in Sect. 6.4 with the above listed Crank-Nicholson routine. Figure 73 shows a comparison between the analytic and numerical solutions for a calculation performed using $D = 1$, $x_0 = 5$, $t_0 = 0.1$, $\delta t = 0.1$, and $N = 100$. It can be seen that the analytic and numerical solutions are in excellent agreement. Note, however, that the time-step used in this calculation (*i.e.*, $\delta t = 0.1$) is much larger than that used in our previous calculation (*i.e.*, $\delta t = 4 \times 10^{-3}$), which employed an explicit differencing scheme—see Fig. 71. According to Eq. (6.23), an explicit scheme is limited to time-steps less than about 5×10^{-3} for the problem under investigation. Thus, we have been able to exceed this limit by a factor of 20 with our implicit scheme, yet still maintain numerical stability. Note that our Crank-Nicholson scheme is able to obtain accurate results with a time-step as large as 0.1 because it is *second-order* in time.
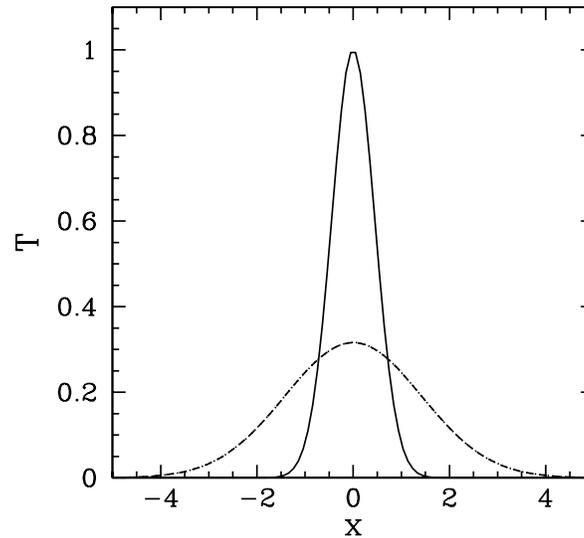
Figure 73: *Diffusive evolution of a 1-d Gaussian pulse. Numerical calculation performed using* $D = 1$, $x_0 = 5$, $\delta t = 0.1$, *and* $N = 100$. *The pulse is evolved from* $t = 0.1$ *to* $t = 1$. *The solid curve shows the initial condition at* $t = 0.1$, *the dashed curve the numerical solution at* $t = 1$, *and the dotted curve (obscured by the dashed curve) the analytic solution at* $t = 1$.

6.9   2-D Problem with Dirichlet Boundary Conditions

Let us consider the solution of the diffusion equation in two dimensions. Suppose that

$$\frac{\partial T(x,y,t)}{\partial t} = D\,\frac{\partial^2 T(x,y,t)}{\partial x^2} + D\,\frac{\partial^2 T(x,y,t)}{\partial y^2}, \qquad (6.28)$$

for $x_l \leq x \leq x_h$, and $0 \leq y \leq L$. Suppose that $T(x,y,t)$ satisfies mixed boundary conditions in the x-direction:

$$\alpha_l(t)\,T(x,y,t) + \beta_l(t)\,\frac{\partial T(x,y,t)}{\partial x} = \gamma_l(y,t), \qquad (6.29)$$

at $x = x_l$, and

$$\alpha_h(t)\,T(x,y,t) + \beta_h(t)\,\frac{\partial T(x,y,t)}{\partial x} = \gamma_h(y,t), \qquad (6.30)$$

at $x = x_h$. Here, α_l, β_l, *etc.*, are known functions of t, whereas γ_l, γ_h are known functions of y and t. Furthermore, suppose that $T(x,y,t)$ satisfies the following

simple Dirichlet boundary conditions in the y-direction:

$$T(x, 0, t) = T(x, L, t) = 0. \tag{6.31}$$

As before, we discretize in time on the uniform grid $t_n = t_0 + n\,\delta t$, for $n = 0, 1, 2, \cdots$. Furthermore, in the x-direction, we discretize on the uniform grid $x_i = x_l + i\,\delta x$, for $i = 0, N+1$, where $\delta x = (x_h - x_l)/(N+1)$. Finally, in the y-direction, we discretize on the uniform grid $y_j = j\,\delta y$, for $j = 0, J$, where $\delta y = L/J$. Adopting the Crank-Nicholson temporal differencing scheme discussed in Sect. 6.6, and the second-order spatial differencing scheme outlined in Sect. 5.2, Eq. (6.28) yields

$$\frac{T_{i,j}^{n+1} - T_{i,j}^n}{\delta t} - \frac{D}{2} \frac{T_{i-1,j}^{n+1} - 2\,T_{i,j}^{n+1} + T_{i+1,j}^{n+1}}{(\delta x)^2} - \frac{D}{2}\left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j}^{n+1} =$$

$$\frac{D}{2} \frac{T_{i-1,j}^n - 2\,T_{i,j}^n + T_{i+1,j}^n}{(\delta x)^2} + \frac{D}{2}\left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j}^n, \tag{6.32}$$

where $T_{i,j}^n \equiv T(x_i, y_j, t_n)$. The discretized boundary conditions take the form

$$T_{0,j}^n = \frac{\gamma_{lj}^n\,\delta x - \beta_l^n\,T_{1,j}^n}{\alpha_l^n\,\delta x - \beta_l^n}, \tag{6.33}$$

$$T_{N+1,j}^n = \frac{\gamma_{hj}^n\,\delta x + \beta_h^n\,T_{N,j}^n}{\alpha_h^n\,\delta x + \beta_h^n}, \tag{6.34}$$

plus

$$T_{i,0}^n = T_{i,J}^n = 0. \tag{6.35}$$

Here, $\alpha_l^n \equiv \alpha_l(t_n)$, *etc.*, and $\gamma_{lj}^n \equiv \gamma_l(y_j, t_n)$, *etc.*

Adopting the Fourier method introduced in Sect. 5.7, we write the $T_{i,j}^n$ in terms of their Fourier-sine harmonics:

$$T_{i,j}^n = \sum_{k=0}^{J} \hat{T}_{i,k}^n\,\sin(j\,k\,\pi/J), \tag{6.36}$$

which automatically satisfies the boundary conditions (6.35). The above expression can be inverted to give (see Sect. 5.9)

$$\hat{T}_{i,j}^n = \frac{2}{J} \sum_{k=0}^{J} T_{i,k}^n\,\sin(j\,k\,\pi/J). \tag{6.37}$$

When Eq. (6.32) is written in terms of the $\hat{T}_{i,j}^n$, it reduces to

$$-\frac{C}{2}\,\hat{T}_{i-1,j}^{n+1} + \left\{1 + C\left(1 + j^2\,\kappa^2/2\right)\right\}\hat{T}_{i,j}^{n+1} - \frac{C}{2}\,\hat{T}_{i+1,j}^{n+1} =$$

$$\frac{C}{2}\,\hat{T}_{i-1,j}^n + \left\{1 - C\left(1 + j^2\,\kappa^2/2\right)\right\}\hat{T}_{i,j}^n + \frac{C}{2}\,\hat{T}_{i+1,j}^n, \tag{6.38}$$

for $i = 1, N$, and $j = 0, J$. Here, $C = D\,\delta t/(\delta x)^2$, and $\kappa = \pi\,\delta x/L$. Moreover, the boundary conditions (6.33) and (6.34) yield

$$\hat{T}_{0,j}^n = \frac{\Gamma_{lj}^n\,\delta x - \beta_l^n\,\hat{T}_{1,j}^n}{\alpha_l^n\,\delta x - \beta_l^n}, \tag{6.39}$$

$$\hat{T}_{N+1,j}^n = \frac{\Gamma_{hj}^n\,\delta x + \beta_h^n\,\hat{T}_{N,j}^n}{\alpha_h^n\,\delta x + \beta_h^n}, \tag{6.40}$$

where

$$\hat{\Gamma}_{lj}^n = \frac{2}{J}\sum_{k=0}^{J}\gamma_{l,k}^n\,\sin(j\,k\,\pi/J), \tag{6.41}$$

etc. Equations (6.38)—(6.40) constitute a set of $J + 1$ uncoupled tridiagonal matrix equations for the $\hat{T}_{i,j}^{n+1}$, with one equation for each separate value of j.

In order to advance our solution by one time-step, we first Fourier transform the $T_{i,j}^n$ and the boundary conditions, according to Eqs. (6.37) and (6.41). Next, we invert the $J + 1$ tridiagonal equations (6.38)—(6.40) to obtain the $\hat{T}_{i,j}^{n+1}$. Finally, we reconstruct the $T_{i,j}^n$ via Eq. (6.36).

6.10   2-D Problem with Neumann Boundary Conditions

Let us replace the Dirichlet boundary conditions (6.35) by the following simple Neumann boundary conditions:

$$\frac{\partial T(x,0,t)}{\partial y} = \frac{\partial T(x,L,t)}{\partial y} = 0. \tag{6.42}$$

The method of solution outlined in the previous section is unaffected, except that the Fourier-sine transforms are replaced by Fourier-cosine transforms—see Sects. 5.8 and 5.9.

6.11   An Example 2-D Diffusion Equation Solver

Listed below is an example 2-d diffusion equation solver which uses the Crank-Nicholson scheme, as well as the previous listed tridiagonal matrix solver and the `Blitz++` library. Note the great structural similarity between this solver and the previously listed 2-d Poisson solver (see Sect. 5.10).

```
// CrankNicholson2D.cpp

// Function to evolve diffusion equation in 2-d:

//   dT / dt = D d^2 T / dx^2 +  D d^2 T / dy^2  for  xl <= x <= xh
//                                               and  0 <= y <= L

//   alphaL T + betaL dT/dx = gammaL(y)  at x=xl

//   alphaH T + betaH dT/dx = gammaH(y)  at x=xh

// In y-direction, either simple Dirichlet boundary conditions:

//   T(x,0) = T(x,L) = 0

// or simple Neumann boundary conditions:

//   dT/dy(x,0) = dT/dy(x,L) = 0

// Matrix T assumed to be of extent N+2, J+1.
// Arrays gammaL, gammaH assumed to be of extent J+1.

// Now, (i,j)th elements of matrices correspond to

//   x_i = xl + i * dx    i=0,N+1

//   y_j = j * L / J      j=0,J

// Here, dx = (xh - xl) / (N+1) is grid spacing in x-direction.

// Now, C =  D dt / dx^2, and kappa = pi * dx / L

// Finally, Neumann=0/1 selects Dirichlet/Neumann bcs in y-direction.

// Uses Crank-Nicholson scheme.

#include <blitz/array.h>
```

```
using namespace blitz;

void fft_forward_cos (Array<double,1> f, Array<double,1>& F);
void fft_backward_cos (Array<double,1> F, Array<double,1>& f);
void fft_forward_sin (Array<double,1> f, Array<double,1>& F);
void fft_backward_sin (Array<double,1> F, Array<double,1>& f);
void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void CrankNicholson2D (Array<double,2>& T,
                       double alphaL, double betaL, Array<double,1> gammaL,
                       double alphaH, double betaH, Array<double,1> gammaH,
                       double dx, double C, double kappa, int Neumann)
{
  // Find N and J. Declare local arrays.
  int N = T.extent(0) - 2;
  int J = T.extent(1) - 1;
  Array<double,2> TT(N+2, J+1), V(N+2, J+1);
  Array<double,1> GammaL(J+1), GammaH(J+1);

  // Fourier transform T
  for (int i = 0; i <= N+1; i++)
    {
      Array<double,1> In(J+1), Out(J+1);

      for (int j = 0; j <= J; j++) In(j) = T(i, j);

      if (Neumann)
        fft_forward_cos (In, Out);
      else
        fft_forward_sin (In, Out);

      for (int j = 0; j <= J; j++) TT(i, j) = Out(j);
    }

  // Fourier transform boundary conditions
  if (Neumann)
    {
      fft_forward_cos (gammaL, GammaL);
      fft_forward_cos (gammaH, GammaH);
    }
  else
    {
      fft_forward_sin (gammaL, GammaL);
```

```
      fft_forward_sin (gammaH, GammaH);
   }


 // Construct source term
 for (int i = 1; i <= N; i++)
   for (int j = 0; j <= J; j++)
     V(i, j) =
       0.5 * C * TT(i-1, j) +
       (1. - C * (1. + 0.5 * double (j * j) * kappa * kappa)) * TT(i, j) +
       0.5 * C * TT(i+1, j);


 // Solve tridiagonal matrix equations
 if (Neumann)
   {
     for (int j = 0; j <= J; j++)
      {
         Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2), u(N+2);

         // Initialize tridiagonal matrix
         for (int i = 2; i <= N; i++) a(i) = - 0.5 * C;
         for (int i = 1; i <= N; i++)
          b(i) = 1. + C * (1. + 0.5 * double (j * j) * kappa * kappa);
         b(1) += 0.5 * C * betaL / (alphaL * dx - betaL);
         b(N) -= 0.5 * C * betaH / (alphaH * dx + betaH);
         for (int i = 1; i <= N-1; i++) c(i) = - 0.5 * C;

         // Initialize right-hand side vector
         for (int i = 1; i <= N; i++)
          w(i) = V(i, j);
         w(1) += 0.5 * C * GammaL(j) * dx / (alphaL * dx - betaL);
         w(N) += 0.5 * C * GammaH(j) * dx / (alphaH * dx + betaH);

         // Invert tridiagonal matrix equation
         Tridiagonal (a, b, c, w, u);
         for (int i = 1; i <= N; i++) TT(i, j) = u(i);
      }
   }
 else
   {
     for (int j = 1; j < J; j++)
      {
         Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2), u(N+2);

         // Initialize tridiagonal matrix
         for (int i = 2; i <= N; i++) a(i) = - 0.5 * C;
```

```
      for (int i = 1; i <= N; i++)
       b(i) = 1. + C * (1. + 0.5 * double (j * j) * kappa * kappa);
      b(1) -= betaL / (alphaL * dx - betaL);
      b(N) += betaH / (alphaH * dx + betaH);
      for (int i = 1; i <= N-1; i++) c(i) = - 0.5 * C;

      // Initialize right-hand side vector
      for (int i = 1; i <= N; i++)
       w(i) = V(i, j);
      w(1) += 0.5 * C * GammaL(j) * dx / (alphaL * dx - betaL);
      w(N) += 0.5 * C * GammaH(j) * dx / (alphaH * dx + betaH);

      // Invert tridiagonal matrix equation
      Tridiagonal (a, b, c, w, u);
      for (int i = 1; i <= N; i++) TT(i, j) = u(i);
     }

   for (int i = 1; i <= N  ; i++)
     {
       TT(i, 0) = 0.; TT(i, J) = 0.;
     }
  }

// Reconstruct solution
for (int i = 1; i <= N; i++)
  {
    Array<double,1> In(J+1), Out(J+1);

    for (int j = 0; j <= J; j++) In(j) = TT(i, j);

    if (Neumann)
      fft_backward_cos (In, Out);
    else
      fft_backward_sin (In, Out);

    for (int j = 0; j <= J; j++) T(i, j) = Out(j);
  }

// Calculate i=0 and i=N+1 values
for (int j = 0; j <= J; j++)
  {
    T(0, j) = (gammaL(j) * dx - betaL * T(1, j)) /
      (alphaL * dx - betaL);
    T(N+1, j) = (gammaH(j) * dx + betaH * T(N, j)) /
      (alphaH * dx + betaH);
```

```
    }
}
```

6.12 An Example 2-D Solution of the Diffusion Equation

Let us now solve the diffusion equation in 2-d using the finite difference technique discussed above. We seek the solution of Eq. (6.28) in the region $0 \leq x \leq 1$ and $0 \leq y \leq 1$, subject to the following initial condition at $t = 0$:

$$
\begin{aligned}
T(x, y, 0) &= 1 &&\text{for } |x - 0.5| < 0.1 \text{ and } |y - 0.5| < 0.1, \\
T(x, y, 0) &= 0 &&\text{otherwise.}
\end{aligned}
\tag{6.43}
$$

The boundary conditions are simply $T(0, y) = T(1, y) = T(x, 0) = T(x, 1) = 0$.

Figure 74 shows the evolution of $T(x, y, t)$ for a calculation performed with the previously listed 2-d diffusion equation solver using $D = 1$, $\delta t = 10^{-4}$, and $N = J = 128$.

6.13 3-D Problems

The techniques discussed above for solving the diffusion equation in two dimensions with a restricted class of boundary conditions can easily be generalized to three dimensions. In the 3-d case, it is necessary to Fourier transform in *two* directions (the y and z directions, say) in order to reduce the problem to a system of uncoupled tridiagonal matrix equations. These equations can be inverted in the usual manner, and the solution can then be reconstructed via a double inverse Fourier transform.
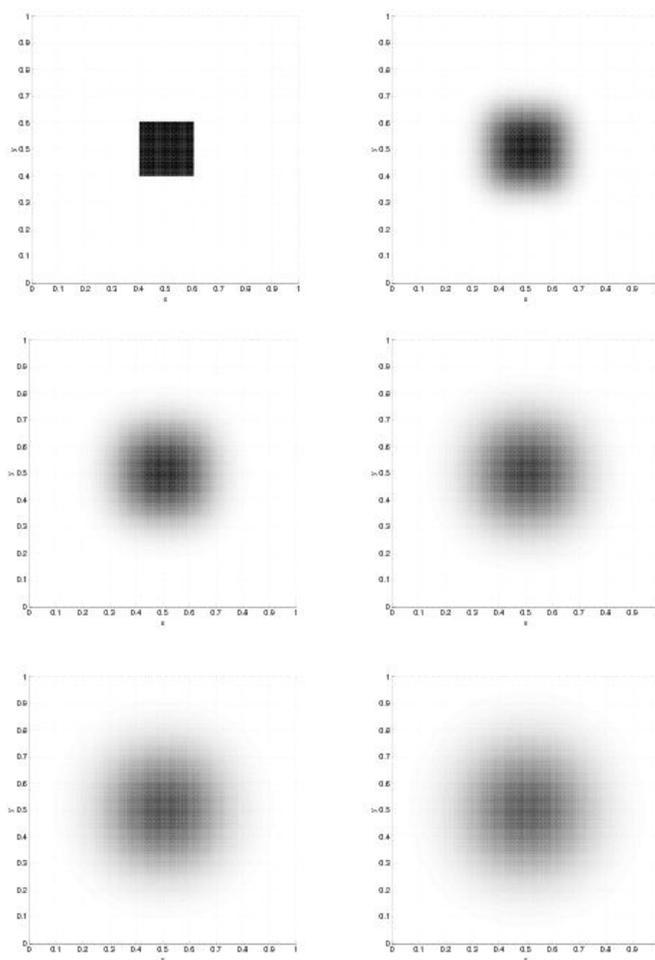
Figure 74: *Diffusion in two dimensions. Numerical calculation performed using* $D = 1$, $\delta t = 10^{-4}$, *and* $N = J = 128$. *Density plots of* $T(x, y, t)$ *are shown at* $t = 0.000$ *(top-left),* $t = 0.001$ *(top-right),* $t = 0.002$ *(middle-left),* $t = 0.003$ *(middle-right),* $t = 0.004$ *(bottom-left), and* $t = 0.005$ *(bottom-right).*

7   The Wave Equation

7.1   Introduction

The wave equation, which in one dimension takes the form

$$\frac{\partial^2 \xi}{\partial t^2} = c^2 \frac{\partial^2 \xi}{\partial x^2}, \tag{7.1}$$

occurs so frequently in physics that it is not necessary to enumerate examples. Here, ξ is usually some sort of displacement or perturbation, whereas c is the (constant) wave speed. The wave equation possesses the formal solution

$$\xi(x, t) = F(x - c\,t) + G(x + c\,t), \tag{7.2}$$

where F and G are arbitrary functions. The above solution represents arbitrarily shaped wave pulses propagating with speed c in the $+x$ and $-x$ directions, respectively, without changing shape.

The wave equation, which is second-order in space and time, can be written as two coupled first-order equations by defining the new variables $v = \partial \xi / \partial t$ and $\theta = -c\,\partial \xi / \partial x$. Expressing Eq. (7.1) in terms of these new variables, we obtain

$$\frac{\partial v}{\partial t} + c\,\frac{\partial \theta}{\partial x} = 0, \tag{7.3}$$

$$\frac{\partial \theta}{\partial t} + c\,\frac{\partial v}{\partial x} = 0. \tag{7.4}$$

Note that when solving the wave equation numerically it is generally preferable to write it as a set of coupled first-order equations, as shown above.

7.2   The 1-D Advection Equation

The wave equation is closely related to the so-called *advection equation*, which in one dimension takes the form

$$\frac{\partial u}{\partial t} = -v\,\frac{\partial u}{\partial x}. \tag{7.5}$$

This equation describes the passive advection of some scalar field $u(x, t)$ carried along by a flow of constant speed v. Since the advection equation is somewhat simpler than the wave equation, we shall discuss it first. The advection equation possesses the formal solution

$$u(x, t) = F(x - v\,t), \tag{7.6}$$

where F is an arbitrary function. This solution describes an arbitrarily shaped pulse which is swept along by the flow, at constant speed v, without changing shape.

We seek the solution of Eq. (7.5) in the region $x_l \leq x \leq x_h$, subject to the simple Dirichlet boundary conditions $u(x_l) = u(x_h) = 0$. As usual, we discretize in time on the uniform grid $t_n = t_0 + n\,\delta t$, for $n = 0, 1, 2, \cdots$. Furthermore, in the x-direction, we discretize on the uniform grid $x_i = x_l + i\,\delta x$, for $i = 0, N + 1$, where $\delta x = (x_h - x_l)/(N+1)$. Adopting an explicit temporal differencing scheme, and a centered spatial differencing scheme, Eq. (7.5) yields

$$\frac{u_i^{n+1} - u_i^n}{\delta t} = -v\,\frac{u_{i+1}^n - u_{i-1}^n}{2\,\delta x}, \tag{7.7}$$

where $u_i^n \equiv u(x_i, t_n)$. The above equation can be rewritten

$$u_i^{n+1} = u_i^n - \frac{C}{2}\,(u_{i+1}^n - u_{i-1}^n), \tag{7.8}$$

where $C = v\,\delta t/\delta x$.

Let us perform a von Neumann stability analysis of the above differencing scheme. Writing $u(x, t) = \hat{u}(t)\,\exp(i\,k\,x)$, we obtain $\hat{u}_i^{n+1} = A\,\hat{u}_i^n$, where

$$A = 1 - i\,C\,\sin(k\,\delta x). \tag{7.9}$$

Note that

$$|A|^2 = 1 + C^2\,\sin^2(k\,\delta x) > 1. \tag{7.10}$$

Thus, the magnitude of the amplification factor is greater than unity for all k. This implies, unfortunately, that the simple differencing scheme (7.8) is *unconditionally unstable*.

7.3   The Lax Scheme

The instability in the differencing scheme (7.8) can be fixed by replacing u_i^n on the right-hand side by the *spatial average* of u taken over the neighbouring grid points. Thus, we obtain

$$u_i^{n+1} = \frac{1}{2}\left(u_{i+1}^n + u_{i-1}^n\right) - \frac{C}{2}\left(u_{i+1}^n - u_{i-1}^n\right), \tag{7.11}$$

which is known as the *Lax* scheme. A von Neumann stability analysis of the Lax scheme yields the following expression for the amplification factor:

$$A = \cos(k\,\delta x) - i\,C\,\sin(k\,\delta x). \tag{7.12}$$

Now

$$|A|^2 = 1 - (1 - C^2)\,\sin^2(k\,\delta x). \tag{7.13}$$

It follows that the Lax scheme is unconditionally stable (*i.e.*, $|A| < 1$ for all k), provided that $C < 1$. From the definition of C, the inequality $C < 1$ can also be written

$$\delta t < \frac{\delta x}{v}. \tag{7.14}$$

This is the famous Courant-Friedrichs-Lewy (or CFL) stability criterion. In fact, *all* stable *explicit* differencing schemes for solving the advection equation are subject to the CFL constraint, which determines the maximum allowable time-step.

Listed below is a routine which solves the 1-d advection equation via the Lax method.

```
// Lax1D.cpp

// Function to evolve advection equation in 1-d:

//   du / dt + v du / dx = 0   for   xl <= x <= xh

//   u = 0   at x=xl and x=xh

// Array u assumed to be of extent N+2.

// Now, ith element of array corresponds to
```

```
//  x_i = xl + i * dx     i=0,N+1

// Here, dx = (xh - xl) / (N+1) is grid spacing.

// Function evolves u by single time-step.

// C = v dt / dx, where dt is time-step.

// Uses Lax scheme.

#include <blitz/array.h>

using namespace blitz;

void Lax1D (Array<double,1>& u, double C)
{
  // Set N. Declare local array.
  int N = u.extent(0) - 2;
  Array<double,1> u0(N+2);

  // Evolve u
  u0 = u;
  for (int i = 1; i <= N; i++)
    u(i) = 0.5 * (u0(i+1) + u0(i-1)) - 0.5 * C * (u0(i+1) - u0(i-1));

  // Set boundary conditions
  u(0) = 0.;
  u(N+1) = 0.;
}
```

Figure 75 shows an example calculation which uses the above routine to advect a Gaussian pulse. The initial condition is

$$u(x, 0) = \exp[-100\,(x - 0.5)^2], \tag{7.15}$$

and the calculation is performed with $v = 1$, $\delta t = 2.49 \times 10^{-3}$, and $N = 200$. Furthermore, $x_l = v\,t$ and $x_h = 1 + v\,t$. Note that $C = 0.5$ for these parameters. It can be seen that the pulse is advected at the correct speed: *i.e.*, the pulse appears approximately stationary when plotted versus $x - v\,t$. Unfortunately, the pulse does not remain the same shape (as it should). Instead, the pulse becomes gradually lower and wider as it propagates, and eventually diffuses away entirely.

It is clear, from the above calculation, that the Lax scheme introduces a spu-
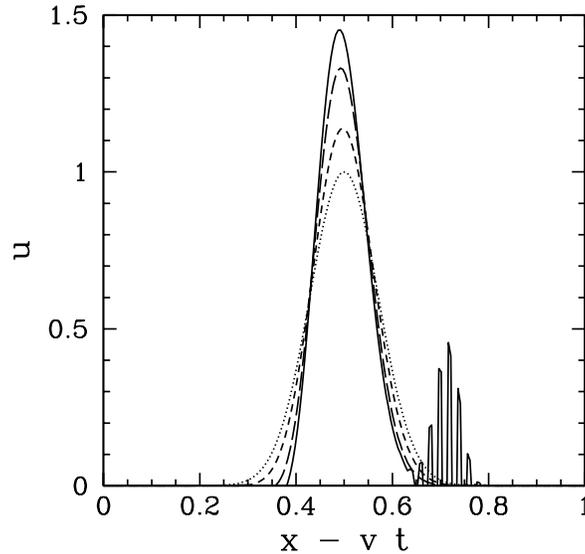
241
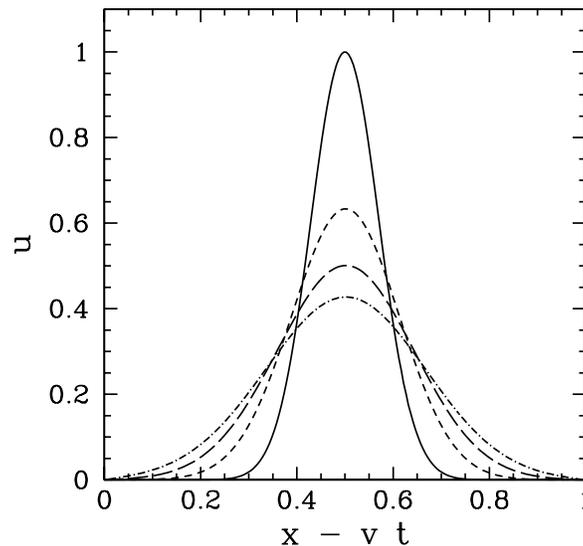
Figure 75: *Advection of a 1-d Gaussian pulse. Numerical calculation performed using $v = 1$, $\delta t =$ 2.49 $\times$ 10^{-3}, and $N = 200$. The solid curve shows the initial condition at $t = 0$, the short-dashed curve the numerical solution at $t = 1$, the long-dashed curve the numerical solution at $t = 2$, and the dot-dashed curve the numerical solution at $t = 3$.*

rious dispersion effect into the advection problem. We can understand the origin of this effect by attempting a Fourier solution, $u(x, t) = \hat{u}_k(t) \exp(i k x)$, of Eq. (7.5). We easily obtain

$$\hat{u}_k(t) = \hat{u}_k(0) \exp(-i k v t). \tag{7.16}$$

Note that $|\hat{u}_k|$ is *constant* in time for all values of k. In other words, the *amplitudes* of the Fourier harmonics of a true solution of the advection equation remain constant in time—it is the *phases* of the harmonics which evolve. Let us now examine Eq. (7.13). It can be seen that, provided the CFL condition $C < 1$ is satisfied, the magnitude of the amplification factor, $|A|$, is *less than unity* for all Fourier harmonics. In other words, the Lax differencing scheme causes the Fourier harmonics to *decay* in time. It is this unphysical attenuation of the Fourier harmonics which gives rise to the strong dispersion effect illustrated in Fig. 75.

Figure 76 shows a calculation made using the Lax scheme in which the CFL condition is violated. This calculation is identical to the one discussed previously, except that the time-step has been increased to $\delta t = 9.95 \times 10^{-3}$, yielding a CFL parameter, $C = 2.0$, which exceeds unity. It can be seen that the pulse grows in

Figure 76: *Advection of a 1-d Gaussian pulse. Numerical calculation performed using $v = 1$, $\delta t = 9.95 \times 10^{-3}$, and $N = 200$. The dotted curve shows the initial condition at $t = 0.00$, the short-dashed curve the numerical solution at $t = 0.15$, the long-dashed curve the numerical solution at $t = 0.30$, and the solid curve the numerical solution at $t = 0.37$.*

amplitude, and eventually starts to break up due to a short-wavelength instability.

7.4   The Crank-Nicholson Scheme

The Crank-Nicholson implicit scheme for solving the diffusion equation (see Sect. 6.6) can be adapted to solve the advection equation. Thus, taking the average of the right-hand side of Eq. (7.5) between the beginning and end of the time-step, we obtain the differencing scheme written below:

$$u_i^{n+1} + \frac{C}{4}\left(u_{i+1}^{n+1} - u_{i-1}^{n+1}\right) = u_i^n - \frac{C}{4}\left(u_{i+1}^n - u_{i-1}^n\right). \tag{7.17}$$

A von Neumann stability analysis of the above scheme yields the following expression for the amplification factor:

$$A = \frac{1 - i\,(C/2)\,\sin(k\,\delta x)}{1 + i\,(C/2)\,\sin(k\,\delta x)}. \tag{7.18}$$

Note that $|A| = 1$ for all values of k, irrespective of the value of C. This implies that the Crank-Nicholson implicit scheme is *not* subject to the CFL constraint, $C <$

1 (since there is no value of k for which $|A| > 1$). Moreover, there is no spurious decay in the Fourier harmonics of the solution (since $|A| = 1$). Hence, unlike the Lax scheme, we would not expect the Crank-Nicholson scheme to introduce strong numerical dispersion into the advection problem.

Listed below is a routine which solves the 1-d advection equation via the Crank-Nicholson method.

```cpp
// Advect1D.cpp

// Function to evolve advection equation in 1-d:

//   du / dt + v du / dx = 0   for   xl <= x <= xh

//   u = 0   at x=xl and x=xh

// Array u assumed to be of extent N+2.

// Now, ith element of array corresponds to

//   x_i = xl + i * dx     i=0,N+1

// Here, dx = (xh - xl) / (N+1) is grid spacing.

// Function evolves u by single time-step.

// C = v dt / dx, where dt is time-step.

// Uses Crank-Nicholson scheme.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void Advect1D (Array<double,1>& u, double C)
{
  // Find N. Declare local arrays.
  int N = u.extent(0) - 2;
  Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2);

  // Initialize tridiagonal matrix
```

```
  for (int i = 2; i <= N;    i++) a(i) = - 0.25 * C;
  for (int i = 1; i <= N; i++) b(i) = 1.;
  for (int i = 1; i <= N-1; i++) c(i) = + 0.25 * C;

  // Initialize right-hand side vector
  for (int i = 1; i <= N; i++)
    w(i) = u(i) - 0.25 * C * (u(i+1) - u(i-1));

  // Invert tridiagonal matrix equation
  Tridiagonal (a, b, c, w, u);

  // Calculate i=0 and i=N+1 values
  u(0) = 0.;
  u(N+1) = 0.;
}
```

Figure 75 shows an example calculation which uses the above routine to advect a Gaussian pulse. The initial condition is as specified in Eq. (7.15), and the calculation is performed with $v = 1$, $\delta t = 9.95 \times 10^{-3}$, and $N = 200$. Note that $C = 2.0$ for these parameters. It can be seen that the pulse propagates at (almost) the correct speed, and maintains approximately the same shape. Clearly, the performance of the Crank-Nicholson scheme is vastly superior to that of the Lax scheme.

7.5   Upwind Differencing

We might be forgiven for concluding that the Crank-Nicholson scheme represents an efficient and accurate general purpose numerical method for solving the advection equation. This is indeed the case, provided we restrict ourselves to fairly *smooth* wave-forms. Unfortunately, the Crank-Nicholson scheme does a very poor job at advecting wave-forms with sharp leading or trailing edges. This is illustrated in Fig. 78, which shows a calculation in which the Crank-Nicholson scheme is used to advect a square wave-pulse. It can be seen that spurious oscillations are generated at both the leading and trailing edges of the wave-form. It turns out that all *central difference* schemes for solving the advection equation suffer from a similar problem.
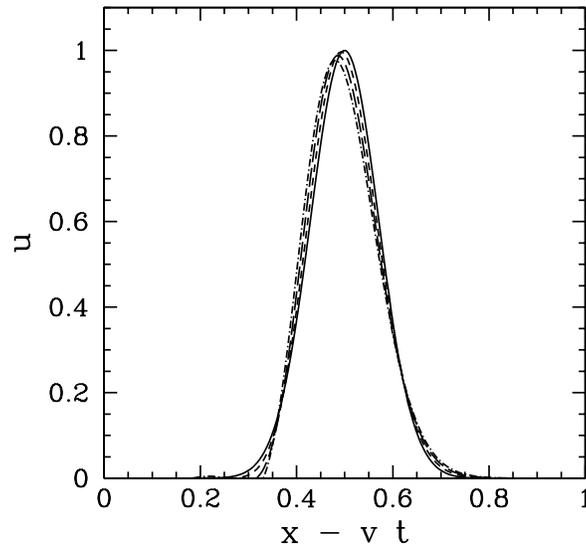
Figure 77: *Advection of a 1-d Gaussian pulse. Numerical calculation performed using $\nu = 1$, $\delta t = 9.95 \times 10^{-3}$, and $N = 200$. The solid curve shows the initial condition at $t = 0$, the short-dashed curve the numerical solution at $t = 1$, the long-dashed curve the numerical solution at $t = 2$, and the dot-dashed curve the numerical solution at $t = 3$.*
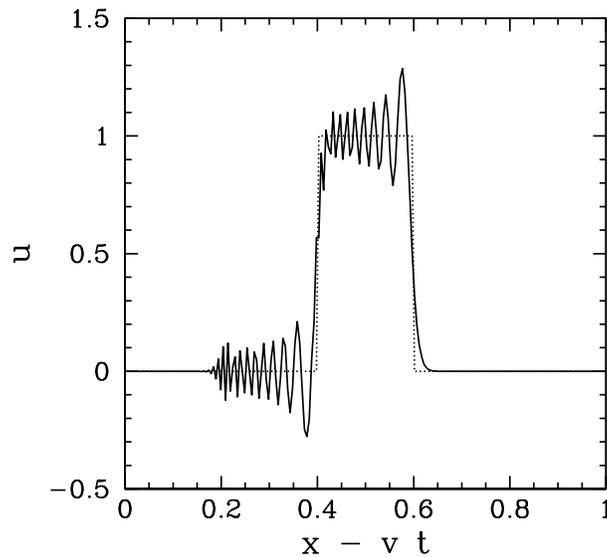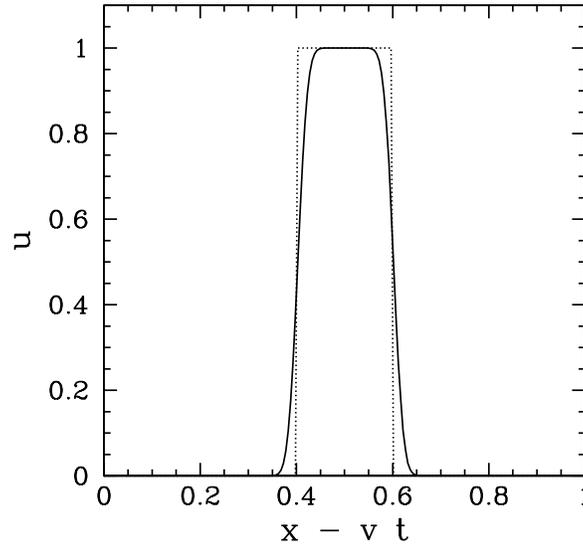

Figure 78: *Advection of a 1-d square wave-pulse. Numerical calculation performed using $\nu = 1$, $\delta t = 2.49 \times 10^{-3}$, and $N = 200$. The dotted curve shows the initial condition at $t = 0.0$, whereas the solid curve shows the numerical solution at $t = 0.1$.*

The only known way to suppress spurious oscillations at the leading and trailing edges of a sharp wave-form is to adopt a so-called *upwind* differencing scheme. In such a scheme, the spatial differences are skewed in the "upwind" direction: *i.e.*, the direction from which the advecting flow emanates. Thus, the upwind version of the simple explicit differencing scheme (7.7) is written

$$\frac{u_i^{n+1} - u_i^n}{\delta t} = -v \, \frac{u_i^n - u_{i-1}^n}{\delta x}, \tag{7.19}$$

or

$$u_i^{n+1} = u_i^n - C \, (u_i^n - u_{i-1}^n), \tag{7.20}$$

Note that this scheme is only *first-order* in space, whereas every other scheme we have discussed has been *second-order*. A von Neumann stability analysis of the above scheme yields

$$A = 1 - C \, [1 - \cos(k \, \delta x)] - i \, C \, \sin(k \, \delta x). \tag{7.21}$$

Note that

$$|A|^2 = 1 - 2 \, C \, (1 - C) \, [1 - \cos(k \, \delta x)]. \tag{7.22}$$

It follows that $|A| < 1$ for all k provided that $C < 1$. Thus, the upwind differencing scheme is stable provided that the CFL condition is satisfied. Fig. 79 shows a calculation in which the above scheme is used to advect a square wave-pulse. There are now no spurious oscillations generated at the sharp edges of the wave-form. On the other hand, the wave-form shows evidence of dispersion. Indeed, the upwind differencing scheme suffers from the same type of spurious dispersion problem as the Lax scheme. Unfortunately, there is no known differencing scheme which is both non-dispersive and capable of dealing well with sharp wave-fronts. In fact, sophisticated codes which solve the advection (or wave) equation generally employ an upwind scheme in regions close to sharp wave-fronts, or shocks, and a more accurate non-dispersive scheme elsewhere.

Incidentally, it is easily demonstrated that the downwind differencing scheme,

$$\frac{u_i^{n+1} - u_i^n}{\delta t} = -v \, \frac{u_{i+1}^n - u_i^n}{\delta x}, \tag{7.23}$$

is unconditionally unstable.

Figure 79: *Advection of a 1-d square wave-pulse. Numerical calculation performed using $v = 1$, $\delta t = 2.49 \times 10^{-3}$, and $N = 200$. The dotted curve shows the initial condition at $t = 0.0$, whereas the solid curve shows the numerical solution at $t = 0.1$.*

7.6   The 1-D Wave Equation

Consider a plane polarized electromagnetic wave propagating *in vacuo* along the z-axis. Suppose that the electric and magnetic fields take the form $\mathbf{E} = [E_x(z, t), 0, 0]$, and $\mathbf{B} = [0, B_y(z, t), 0]$. Now, according to Maxwell's equations,

$$\frac{\partial E_x}{\partial t} + c \frac{\partial H_y}{\partial z} = 0, \tag{7.24}$$

$$\frac{\partial H_y}{\partial t} + c \frac{\partial E_x}{\partial z} = 0, \tag{7.25}$$

where $H_y = c\,B_y$, and c is the velocity of light. Note that the above equations take the form of two coupled advection equations. Let us find the numerical solution of these equations in some region $0 \leq z \leq L$ which is bounded by *perfectly conducting walls* at $z = 0$ and $z = L$. Now, both the *tangential* electric field and the *normal* magnetic field must be zero at a perfect conductor. It follows that

$$E_x(0, t) = E_x(L, t) = 0. \tag{7.26}$$

Moreover, Eq. (7.24) yields

$$\frac{\partial H_y(0, t)}{\partial z} = \frac{\partial H_y(L, t)}{\partial z} = 0. \tag{7.27}$$

We expect Eqs. (7.24) and (7.25) to support wave-like solutions which bounce backwards and forwards between the conducting walls.

As before, we discretize in time on the uniform grid $t_n = t_0 + n \,\delta t$, for $n = 0, 1, 2, \cdots$. Furthermore, in the z-direction, we discretize on the uniform grid $z_i = i \,\delta z$, for $i = 0, I$, where $\delta z = L/I$. Adopting a Crank-Nicholson temporal differencing scheme similar to that discussed in Sect. 7.4, Eqs. (7.24)–(7.25) yield

$$\frac{(E_x)_i^{n+1} - (E_x)_i^n}{\delta t} + \frac{c}{2} \left(\frac{\partial H_y}{\partial z} \right)_i^{n+1} + \frac{c}{2} \left(\frac{\partial H_y}{\partial z} \right)_i^n = 0, \tag{7.28}$$

$$\frac{(H_y)_i^{n+1} - (H_y)_i^n}{\delta t} + \frac{c}{2} \left(\frac{\partial E_x}{\partial z} \right)_i^{n+1} + \frac{c}{2} \left(\frac{\partial E_x}{\partial z} \right)_i^n = 0, \tag{7.29}$$

where $(E_x)_i^n \equiv E_x(z_i, t_n)$, *etc.*

Adopting a Fourier approach, we write

$$(E_x)_i^n = \sum_{j=0,I} \hat{E}_j^n \sin(i j \pi/I), \tag{7.30}$$

$$(H_y)_i^n = \sum_{j=0,I} \hat{H}_j^n \cos(i j \pi/I), \tag{7.31}$$

which automatically satisfies the boundary conditions (7.26) and (7.27). Equations (7.28) and (7.29) yield

$$\hat{E}_i^{n+1} - \hat{E}_i^n - i \, D \, (\hat{H}_i^{n+1} + \hat{H}_i^n) = 0, \tag{7.32}$$

$$\hat{H}_i^{n+1} - \hat{H}_i^n + i \, D \, (\hat{E}_i^{n+1} + \hat{E}_i^n) = 0, \tag{7.33}$$

where $D = \pi c \,\delta t/(2 \, L)$. It follows that

$$\hat{E}_i^{n+1} = +\frac{2 \, i \, D}{1 + i^2 \, D^2} \, \hat{H}_i^n + \frac{1 - i^2 \, D^2}{1 + i^2 \, D^2} \, \hat{E}_i^n, \tag{7.34}$$

$$\hat{H}_i^{n+1} = -\frac{2 \, i \, D}{1 + i^2 \, D^2} \, \hat{E}_i^n + \frac{1 - i^2 \, D^2}{1 + i^2 \, D^2} \, \hat{H}_i^n, \tag{7.35}$$

for $i = 0, I$.

Let us determine the eigenvalues λ of the above linear system, assuming that $(\hat{E}_i^{n+1}, \hat{H}_i^{n+1}) = \lambda\,(\hat{E}_i^n, \hat{H}_i^n)$. After a little analysis, we obtain

$$\lambda^2 - 2\,\frac{1 - i^2\,D^2}{1 + i^2\,D^2}\,\lambda + 1 = 0. \tag{7.36}$$

For all values of i, the two roots of the above quadratic are complex, but have modulus unity: *i.e.*, $|\lambda| = 1$. This implies that our differencing scheme is both numerically stable (if $|\lambda| > 1$ then the scheme would be unstable, since the associated eigenfunction would be amplified at each time-step) and non-dispersive (if $|\lambda| < 1$ then all Fourier harmonics would eventually decay away, and, hence, so would the solution). Note that the above calculation is equivalent to von Neumann stability analysis.

The routine listed below solves the 1-d wave equation using the Crank-Nicholson scheme discussed above. The routine first Fourier transforms E_x and H_y, takes a time-step using Eqs. (7.34) and (7.35), and then reconstructs E_x and H_y via an inverse Fourier transform.

```
// Wave1D.cpp

// Function to evolve 1-d wave equation:

//   d E_x / dt + c d H_y / dz = 0

//   d H_y / dt + c d E_x / dz = 0

// in region 0 < z < L.

// Boundary conditions:

//   E_x(0,t) = E_x(L,t) = 0

//   d H_y(0,t) / dz =  d H_y(L,t) / dz = 0

// Arrays Ex, Hy assumed to be of extent I+1.

// Now, ith elements of arrays correspond to

//   z_i = i * L / J      i=0,I
```

```
// Also, D = pi c dt / (2 L), where dt is time-step.

// Uses Crank-Nicholson scheme.

#include <blitz/array.h>

using namespace blitz;

void fft_forward_cos (Array<double,1> f, Array<double,1>& F);
void fft_backward_cos (Array<double,1> F, Array<double,1>& f);
void fft_forward_sin (Array<double,1> f, Array<double,1>& F);
void fft_backward_sin (Array<double,1> F, Array<double,1>& f);

void Wave1D (Array<double,1>& Ex, Array<double,1>& Hy, double D)
{
  // Find I. Declare local arrays
  int I = Ex.extent(0) - 1;
  Array<double,1> EE(I+1), HH(I+1), EE0(I+1), HH0(I+1);

  // Fourier transform Ex and Hy
  fft_forward_sin (Ex, EE0);
  fft_forward_cos (Hy, HH0);

  // Evolve EE and HH
  for (int i = 0; i <= I; i++)
    {
      double x = double (i) * D;
      double fp = 1. + x*x;
      double fm = 1. - x*x;

      EE(i) = 2. * x * HH0(i) + fm * EE0(i);
      EE(i) /= fp;
      HH(i) = - 2. * x * EE0(i) + fm * HH0(i);
      HH(i) /= fp;
    }

  // Reconstruct Ex and Hy via inverse Fourier transform
  fft_backward_sin (EE, Ex);
  fft_backward_cos (HH, Hy);
}
```

Figure 7.1 shows an example calculation which uses the above routine to prop-

agate a Gaussian wave-pulse. The initial condition is

$$E_x(z, 0) = H_y(z, 0) = \exp[-100\,(z - 0.5)^2], \qquad (7.37)$$

and the calculation is performed with $L = 1$, $c = 1$, $\delta t = 5 \times 10^{-3}$, and $N = 100$. It can be seen that the pulse moves to the right, reflects off the conducting wall at $z = 1$, and then moves to the left. Note, that $E_x = +H_y$ when the wave is far from the conducting walls and moving to the right, whereas $E_x = -H_y$ when the wave is far from the conducting walls and moving to the left.

Figure 7.24 shows a second example calculation performed with the same parameters as the first. In this calculation, the pulse is allowed to reflect off the conducting walls *ten* times before returning to its initial position. Note that the pulse amplitude and shape remain constant to a very good approximation during this process. The fact that the pulse returns almost exactly to its initial position after ten time periods have elapsed (*i.e.*, at $t = 10$) demonstrates that it is propagating at the correct speed (*i.e.*, $c = 1$).

Figure 82 shows a third example calculation which uses the above listed routine to propagate a square wave-pulse. Note that the routine does a very poor job, since spurious oscillations are generated at the sharp leading and trailing edges of the wave-form. As discussed in Sect. 7.5, such oscillations can only be suppressed by adopting an *upwind* differencing scheme, which in this case means that the spatial differences must be skewed in the direction from which the wave is propagating. Unfortunately, simple explicit upwind schemes are subject to the CFL constraint,

$$\delta t < \frac{\delta z}{c}, \qquad (7.38)$$

and also tend to be highly dispersive.

7.7 The 2-D Resonant Cavity

Figure 83 shows a 2-d resonant cavity consisting of a hollow, rectangular, perfectly conducting channel of dimensions $L_x \times L_y$. Suppose that the walls of the channel are aligned along the x- and y-axes. We shall excite this cavity in a
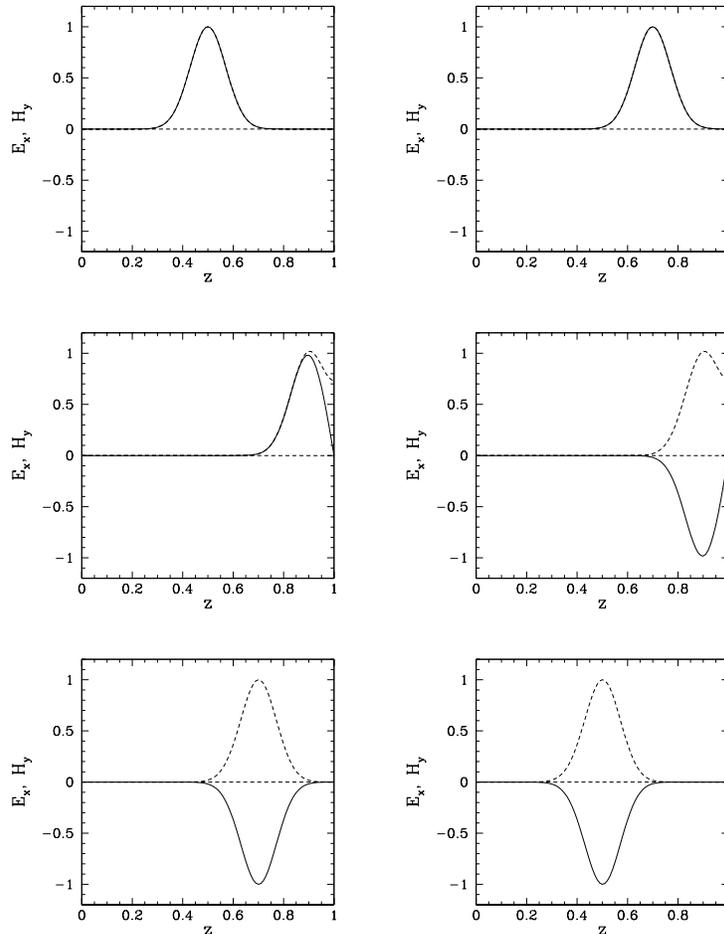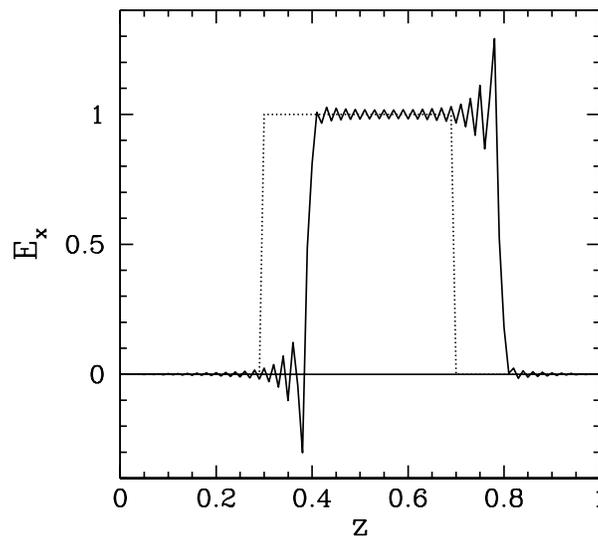
Figure 80: *Propagation of a 1-d Gaussian wave-pulse. Numerical calculation performed using* $c = 1$, $\delta t = 5 \times 10^{-3}$, *and* $N = 100$. *The solid curves show* E_x, *whereas the dashed curves show* H_y. *The top-left, top-right, middle-left, middle-right, bottom-left, and bottom-right panels show the solution at* $t = 0.0$, 0.2, 0.4, 0.6, 0.8, *and* 1.0, *respectively.*
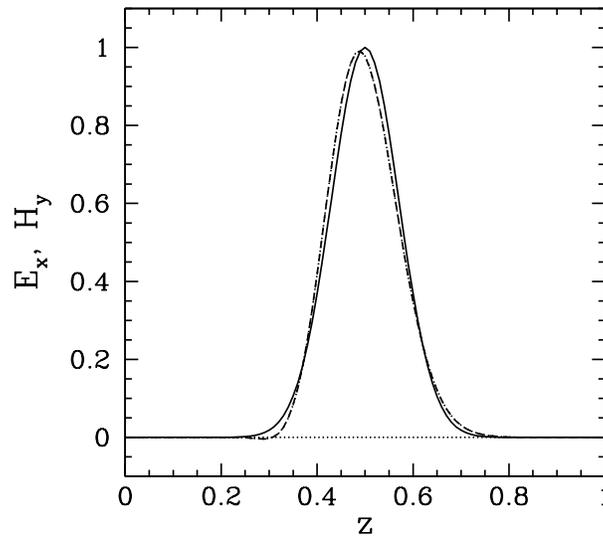
Figure 81: *Propagation of a 1-d Gaussian wave-pulse. Numerical calculation performed using* $c = 1$, $\delta t = 5 \times 10^{-3}$, *and* $N = 100$. *The solid curve shows* E_x *at* $t = 0.$, *the dashed curve shows* E_x *at* $t = 10.$, *and the dotted curve (obscured by the dashed curve) shows* B_y *at* $t = 10$.

Figure 82: *Propagation of a 1-d square wave-pulse. Numerical calculation performed using* $c = 1$, $\delta t = 5 \times 10^{-3}$, *and* $N = 100$. *The dotted curve shows* E_x *at* $t = 0.0$, *and the solid curve shows* E_x *at* $t = 0.1$.
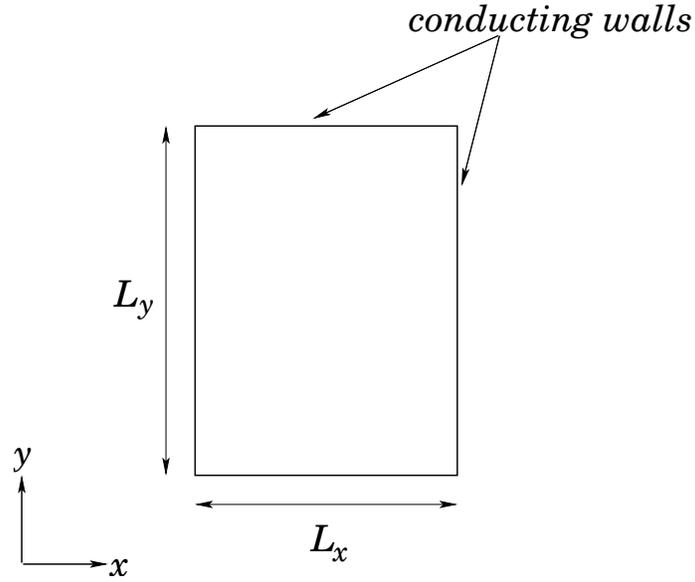
Figure 83: *A 2-d resonant cavity.*

rather artificial manner by imposing a *z*-directed alternating current pattern of frequency f, which has the same spatial structure as the mode in which we are interested. Let us calculate the electric and magnetic field patterns excited within the cavity by such a current pattern.

The electric and magnetic fields within the cavity can be written $\mathbf{E} = [0, 0, E_z(x, y, t)]$, and $\mathbf{B} = [B_x(x, y, t), B_y(x, y, t), 0]$, respectively. It follows from Maxwell's equations that

$$\frac{\partial H_x}{\partial t} + c\,\frac{\partial E_z}{\partial y} = 0, \tag{7.39}$$

$$\frac{\partial H_y}{\partial t} + c\,\frac{\partial E_z}{\partial x} = 0, \tag{7.40}$$

$$\frac{\partial E_z}{\partial t} + c\,\frac{\partial H_y}{\partial x} + c\,\frac{\partial H_x}{\partial y} = J_z, \tag{7.41}$$

where c is the velocity of light, $H_x = c\,B_x$, $H_y = -c\,B_y$, and $J_z = -\mu_0\,c^2\,j_z$. Note that the above system of equations takes the form of three coupled advection equations with a source term. The boundary conditions are that the *tangential* electric field and the *normal* magnetic field must be zero at the conducting walls.

255

It follows that

$$E_z = 0 \qquad (7.42)$$

at all the walls (which are located at $x = 0, L_x$ and $y = 0, L_y$),

$$H_x = \frac{\partial H_y}{\partial x} = 0 \qquad (7.43)$$

at $x = 0, L_x$, and

$$H_y = \frac{\partial H_x}{\partial y} = 0 \qquad (7.44)$$

at $y = 0, L_y$. Finally, the normalized current pattern associated with the (m, n) mode takes the form

$$J_z(x, y, t) = J_0 \, \sin(m \, \pi \, x / L_x) \, \sin(n \, \pi \, y / L_y) \, \sin(2 \, \pi \, f \, t). \qquad (7.45)$$

As usual, we discretize in time on the uniform grid $t_n = t_0 + n \, \delta t$, for $n = 0, 1, 2, \cdots$. Furthermore, in the x-direction, we discretize on the uniform grid $x_i = i \, \delta x$, for $i = 0, I$, where $\delta x = L_x / I$. Finally, in the y-direction, we discretize on the uniform grid $y_j = j \, \delta y$, for $j = 0, J$, where $\delta y = L_y / J$. Adopting a Crank-Nicholson temporal differencing scheme similar to that discussed in Sects. 7.4 and 7.6, Eqs. (7.39)–(7.41) yield

$$\frac{(H_x)_{i,j}^{n+1} - (H_x)_{i,j}^n}{\delta t} + \frac{c}{2} \left(\frac{\partial E_z}{\partial y} \right)_{i,j}^{n+1} + \frac{c}{2} \left(\frac{\partial E_z}{\partial y} \right)_{i,j}^n = 0, \qquad (7.46)$$

$$\frac{(H_y)_{i,j}^{n+1} - (H_y)_{i,j}^n}{\delta t} + \frac{c}{2} \left(\frac{\partial E_z}{\partial x} \right)_{i,j}^{n+1} + \frac{c}{2} \left(\frac{\partial E_z}{\partial x} \right)_{i,j}^n = 0, \qquad (7.47)$$

$$\frac{(E_z)_{i,j}^{n+1} - (E_z)_{i,j}^n}{\delta t} + \frac{c}{2} \left(\frac{\partial H_y}{\partial x} \right)_{i,j}^{n+1} + \frac{c}{2} \left(\frac{\partial H_y}{\partial x} \right)_{i,j}^n$$

$$+ \frac{c}{2} \left(\frac{\partial H_x}{\partial y} \right)_{i,j}^{n+1} + \frac{c}{2} \left(\frac{\partial H_x}{\partial y} \right)_{i,j}^n = (J_z)_{i,j}^n, \qquad (7.48)$$

where $(H_x)_{i,j}^n \equiv H_x(x_i, y_j, t_n)$, *etc.*

Adopting a Fourier approach, we write

$$(E_z)_{i,j}^n = \sum_{i'=0,I}^{j'=0,J} \hat{E}_{i',j'}^n \, \sin(i \, i' \, \pi / I) \, \sin(j \, j' \, \pi / J), \qquad (7.49)$$

$$(H_x)_{i,j}^n = \sum_{i'=0,I}^{j'=0,J} \hat{X}_{i',j'}^n \sin(i\,i'\,\pi/I)\,\cos(j\,j'\,\pi/J), \qquad (7.50)$$

$$(H_y)_{i,j}^n = \sum_{i'=0,I}^{j'=0,J} \hat{Y}_{i',j'}^n \cos(i\,i'\,\pi/I)\,\sin(j\,j'\,\pi/J), \qquad (7.51)$$

$$(J_z)_{i,j}^n = \sum_{i'=0,I}^{j'=0,J} \hat{J}_{i',j'}^n \sin(i\,i'\,\pi/I)\,\sin(j\,j'\,\pi/J) \qquad (7.52)$$

which automatically satisfies the boundary conditions (7.42)–(7.44). Equations (7.46)–(7.48) yield

$$\hat{X}_{i,j}^{n+1} - \hat{X}_{i,j}^n + j\,D_y\,(\hat{E}_{i,j}^{n+1} + \hat{E}_{i,j}^n) = 0, \qquad (7.53)$$

$$\hat{Y}_{i,j}^{n+1} - \hat{Y}_{i,j}^n + i\,D_x\,(\hat{E}_{i,j}^{n+1} + \hat{E}_{i,j}^n) = 0, \qquad (7.54)$$

$$\hat{E}_{i,j}^{n+1} - \hat{E}_{i,j}^n - i\,D_x\,(\hat{Y}_{i,j}^{n+1} + \hat{Y}_{i,j}^n) - j\,D_j\,(\hat{X}_{i,j}^{n+1} + \hat{X}_{i,j}^n) = \delta t\,\hat{J}_{i,j}^n, \qquad (7.55)$$

for $i = 0, I$ and $j = 0, J$, where $D_x = \pi\,c\,\delta t/(2\,L_x)$ and $D_y = \pi\,c\,\delta t/(2\,L_y)$. It follows that

$$\hat{E}_{i,j}^{n+1} = \frac{(1 - i^2\,D_x^2 - j^2\,D_y^2)\,\hat{E}_{i,j}^n + 2j\,D_y\,\hat{X}_{i,j}^n + 2i\,D_x\,\hat{Y}_{i,j}^n + \delta t\,\hat{J}_{i,j}^n}{1 + i^2\,D_x^2 + j^2\,D_y^2}$$

$$(7.56)$$

$$\hat{X}_{i,j}^{n+1} = \hat{X}_{i,j}^n - j\,D_y\,(\hat{E}_{i,j}^{n+1} + \hat{E}_{i,j}^n), \qquad (7.57)$$

$$\hat{Y}_{i,j}^{n+1} = \hat{Y}_{i,j}^n - i\,D_x\,(\hat{E}_{i,j}^{n+1} + \hat{E}_{i,j}^n). \qquad (7.58)$$

The routine listed below solves the 2-d wave equation in a resonant cavity using the Crank-Nicholson scheme discussed above. The routine first Fourier transforms H_x, H_y, E_z, and J_z in both the x- and y-directions, takes a time-step using Eqs. (7.56)–(7.58), and then reconstructs H_x, H_y, and E_z via an double inverse Fourier transform.

```
// Wave2D.cpp
```

```
// Function to evolve 2-d wave equation:
```

```
//   d H_x / dt + c d E_z / dy = 0

//   d H_y / dt + c d E_z / dx = 0

//   d E_z / dt + c d H_y / dx + c d H_x / dy = J_z

// in region 0 < x < L_x and 0 < y < L_y

// Boundary conditions:

//   E_z(0, y) = E_z(L_x, y) = E_z(x, 0) = E_z(x, L_y) = 0

//   H_x(0, y) = H_x(L_x, y) = d H_y(0, y) / dx =  d H_y(L_x, y) / dx = 0

//   H_y(x, 0) = H_y(x, L_y) = d H_x(x, 0) / dy =  d H_x(x, L_y) / dy = 0

// Matrices Hx, Hy, Ez, Jz assumed to be of extent I+1, J+1.
// Now, (i,j)th elements of matrices correspond to

//   x_i = i * dx     i=0,I

//   y_j = j * dy     j=0,J

// Here, dx = L_x / I is grid spacing in x-direction,

// and dy = L_y / J is grid spacing in x-direction.

// Now, Dx = pi c dt / (2 L_x) and Dy = pi c dt / (2 L_y),

// where dt is time-step.

// Uses Crank-Nicholson scheme.

#include <blitz/array.h>

using namespace blitz;

void fft_forward_cos (Array<double,1> f, Array<double,1>& F);
void fft_backward_cos (Array<double,1> F, Array<double,1>& f);
void fft_forward_sin (Array<double,1> f, Array<double,1>& F);
void fft_backward_sin (Array<double,1> F, Array<double,1>& f);

void Wave2D (Array<double,2>& Hx, Array<double,2>& Hy, Array<double,2>& Ez,
             Array<double,2> Jz, double Dx, double Dy, double dt)
{
```

```cpp
// Find I and J. Declare local arrays
int I = Hx.extent(0) - 1;
int J = Hx.extent(1) - 1;
Array<double,2> X(I+1, J+1), XX(I+1, J+1), XXX(I+1, J+1);
Array<double,2> Y(I+1, J+1), YY(I+1, J+1), YYY(I+1, J+1);
Array<double,2> E(I+1, J+1), EE(I+1, J+1), EEE(I+1, J+1);
Array<double,2> K(I+1, J+1), KK(I+1, J+1);

// Fourier transform solution in x-direction
for (int j = 0; j <= J; j++)
  {
    Array<double,1> In(I+1), Out(I+1);

    // Fourier transform Hx
    for (int i = 0; i <= I; i++) In(i) = Hx(i, j);
    fft_forward_sin (In, Out);
    for (int i = 0; i <= I; i++) X(i, j) = Out(i);

    // Fourier transform Hy
    for (int i = 0; i <= I; i++) In(i) = Hy(i, j);
    fft_forward_cos (In, Out);
    for (int i = 0; i <= I; i++) Y(i, j) = Out(i);

    // Fourier transform Ez
    for (int i = 0; i <= I; i++) In(i) = Ez(i, j);
    fft_forward_sin (In, Out);
    for (int i = 0; i <= I; i++) E(i, j) = Out(i);

    // Fourier transform Jz
    for (int i = 0; i <= I; i++) In(i) = dt * Jz(i, j);
    fft_forward_sin (In, Out);
    for (int i = 0; i <= I; i++) K(i, j) = Out(i);
  }

// Fourier transform solution in y-direction
for (int i = 0; i <= I; i++)
  {
    Array<double,1> In(J+1), Out(J+1);

    // Fourier transform Hx
    for (int j = 0; j <= J; j++) In(j) = X(i, j);
    fft_forward_cos (In, Out);
    for (int j = 0; j <= J; j++) XX(i, j) = Out(j);

    // Fourier transform Hy
```

```
    for (int j = 0; j <= J; j++) In(j) = Y(i, j);
    fft_forward_sin (In, Out);
    for (int j = 0; j <= J; j++) YY(i, j) = Out(j);

    // Fourier transform Ez
    for (int j = 0; j <= J; j++) In(j) = E(i, j);
    fft_forward_sin (In, Out);
    for (int j = 0; j <= J; j++) EE(i, j) = Out(j);

    // Fourier transform Jz
    for (int j = 0; j <= J; j++) In(j) = K(i, j);
    fft_forward_sin (In, Out);
    for (int j = 0; j <= J; j++) KK(i, j) = Out(j);
  }

// Evolve XX, YY, and EE
for (int i = 0; i <= I; i++)
  for (int j = 0; j <= J; j++)
    {
      double x = double (i) * Dx;
      double y = double (j) * Dy;
      double fp = 1. + x*x + y*y;
      double fm = 1. - x*x - y*y;

      EEE(i, j) = fm * EE(i, j) + 2. * y * XX(i, j) +
        2. * x * YY(i,j) + KK(i, j);
      EEE(i, j) /= fp;
      XXX(i, j) = XX(i, j) - y * (EEE(i, j) + EE(i, j));
      YYY(i, j) = YY(i, j) - x * (EEE(i, j) + EE(i, j));
    }

// Reconstruct solution via inverse Fourier transform in y-direction
for (int i = 0; i <= I; i++)
  {
    Array<double,1> In(J+1), Out(J+1);

    // Reconstruct Hx
    for (int j = 0; j <= J; j++) In(j) = XXX(i, j);
    fft_backward_cos (In, Out);
    for (int j = 0; j <= J; j++) X(i, j) = Out(j);

    // Reconstruct Hy
    for (int j = 0; j <= J; j++) In(j) = YYY(i, j);
    fft_backward_sin (In, Out);
    for (int j = 0; j <= J; j++) Y(i, j) = Out(j);
```

```
      // Reconstruct Ez
      for (int j = 0; j <= J; j++) In(j) = EEE(i, j);
      fft_backward_sin (In, Out);
      for (int j = 0; j <= J; j++) E(i, j) = Out(j);
    }

  // Reconstruct solution via inverse Fourier transform in x-direction
  for (int j = 0; j <= J; j++)
    {
      Array<double,1> In(I+1), Out(I+1);

      // Reconstruct Hx
      for (int i = 0; i <= I; i++) In(i) = X(i, j);
      fft_backward_sin (In, Out);
      for (int i = 0; i <= I; i++) Hx(i, j) = Out(i);

      // Reconstruct Hy
      for (int i = 0; i <= I; i++) In(i) = Y(i, j);
      fft_backward_cos (In, Out);
      for (int i = 0; i <= I; i++) Hy(i, j) = Out(i);

      // Reconstruct Ez
      for (int i = 0; i <= I; i++) In(i) = E(i, j);
      fft_backward_sin (In, Out);
      for (int i = 0; i <= I; i++) Ez(i, j) = Out(i);
    }
}
```

The numerical calculations discussed below were performed using the above routine. The electromagnetic fields H_x, H_y, and E_z were all initialized to zero everywhere at $t = 0$. Figure 84 shows the maximum amplitude of E_z versus the frequency, f, for an $m = 1/n = 1$ driving current distribution. It can be seen that there is a clear resonance at $f \simeq 0.7$.

Figures 85 and 86 illustrate the typical time variation of E_z, H_x, and H_y for a non-resonant and a resonant case, respectively. For the non-resonant case, the traces take the form of interference patterns between the directly driven response, which oscillates at the driving frequency f, and the transient response, which oscillates at the natural frequency f_0 of the cavity. Note that the transients never decay, since there is no dissipation in the present problem. Incidentally, it is easily
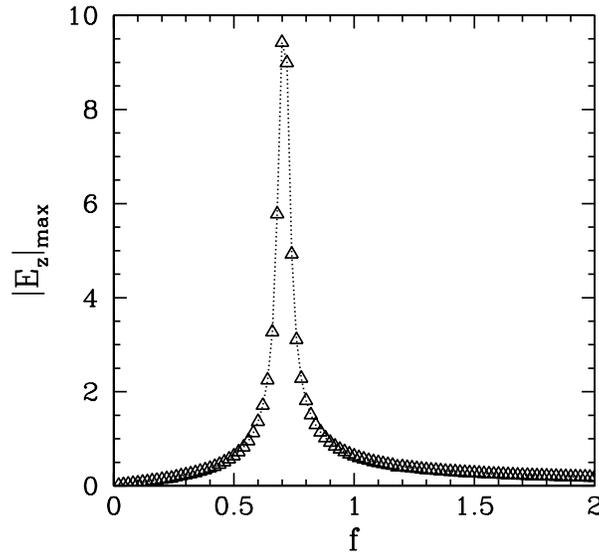
Figure 84: *Electromagnetic waves in a 2-d resonant cavity. The maximum amplitude of* $E_z(x = 0.5, y = 0.5)$ *between* $t = 0$ *and* $t = 20$ *versus the driving frequency,* f. *Numerical calculation performed using* $m = 1$, $n = 1$, $L_x = 1$, $L_y = 1$, $c = 1$, $I = J = 32$, *and* $\delta t = 10^{-2}$.

demonstrated that

$$f_0 = \frac{c}{2}\sqrt{\frac{1}{(n\,L_x)^2} + \frac{1}{(m\,L_y)^2}}. \tag{7.59}$$

Hence, it follows that $f_0 = 1/\sqrt{2} = 0.7071$ for $n = m = L_x = L_y = c = 1$, which corresponds very well to the resonant frequency found in Fig. 84. For the resonant case, the traces take the form of waves of ever increasing amplitude which oscillate at the natural frequency f_0.

Finally, Figs. 86 and 87 illustrate the spatial variation of the electromagnetic fields driven within the cavity when $m = 1$ and $n = 1$.

262
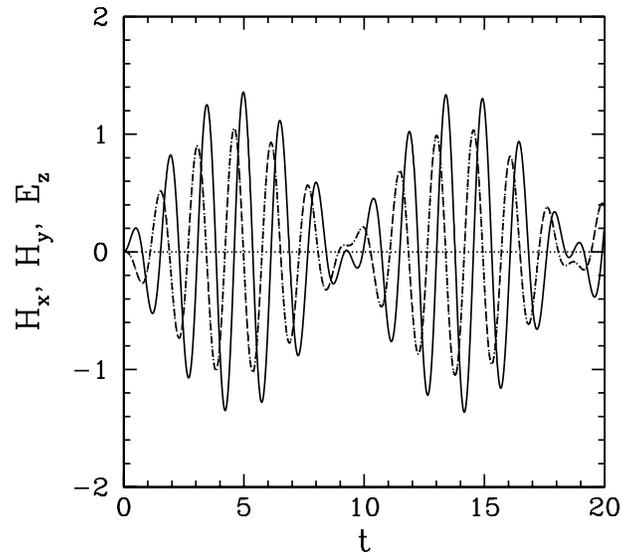
Figure 85: *Electromagnetic waves in a 2-d resonant cavity. Time traces of* $E_z(x = 0.5, y = 0.5)$ *(solid curve),* $H_x(x = 0.5, y = 0.0)$ *(dashed curve), and* $H_y(x = 0.0, y = 0.5)$ *(dotted curve—obscured by dashed curve). Numerical calculation performed using* $m = 1$, $n = 1$, $L_x = 1$, $L_y = 1$, $c = 1$, $I = J = 32$, $\delta t = 10^{-2}$, *and* $f = 0.6$.
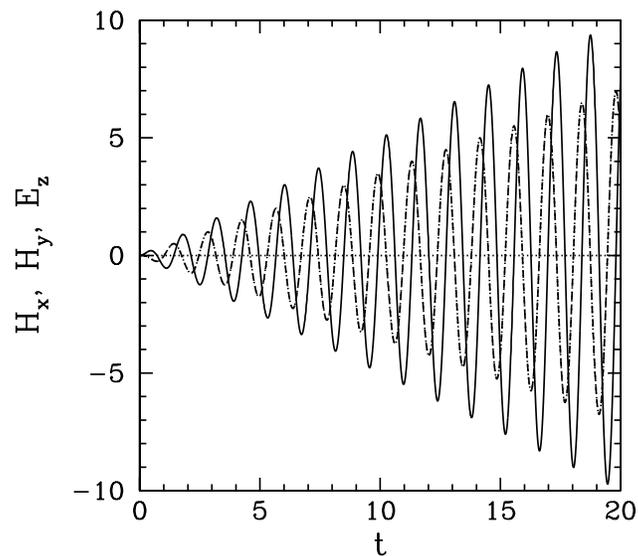
Figure 86: *Electromagnetic waves in a 2-d resonant cavity. Time traces of* $E_z(x = 0.5, y = 0.5)$ *(solid curve),* $H_x(x = 0.5, y = 0.0)$ *(dashed curve), and* $H_y(x = 0.0, y = 0.5)$ *(dotted curve—obscured by dashed curve). Numerical calculation performed using* $m = 1$, $n = 1$, $L_x = 1$, $L_y = 1$, $c = 1$, $I = J = 32$, $\delta t = 10^{-2}$, *and* $f = 0.7071$.
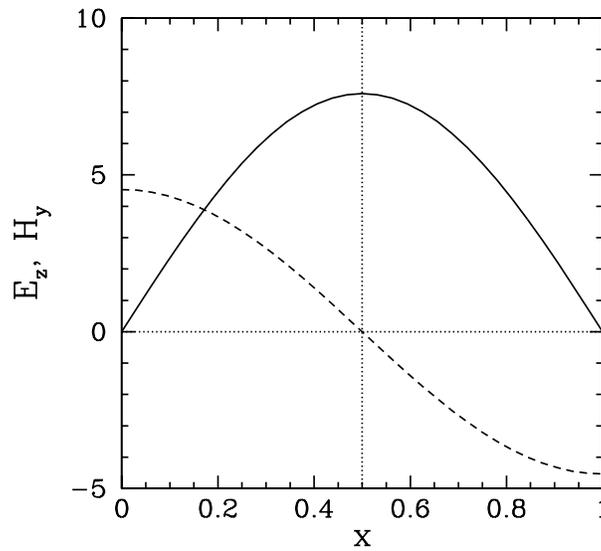
Figure 87: *Electromagnetic waves in a 2-d resonant cavity. Spatial variation of* E_z *(solid curve) and* H_y *(dashed curve) in the x-direction at* $t = 20$ *and* $y = 0.5$. *Numerical calculation performed using* $m = 1$, $n = 1$, $L_x = 1$, $L_y = 1$, $c = 1$, $I = J = 32$, $\delta t = 10^{-2}$, *and* $f = 0.7071$.
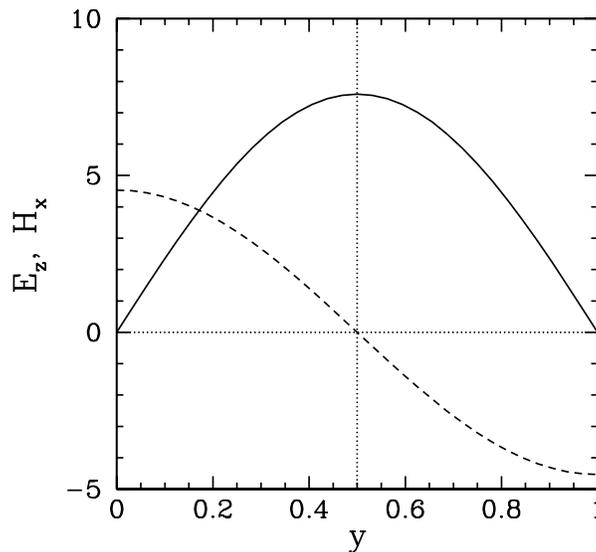
Figure 88: *Electromagnetic waves in a 2-d resonant cavity. Spatial variation of* E_z *(solid curve) and* H_x *(dashed curve) in the y-direction at* $t = 20$ *and* $x = 0.5$. *Numerical calculation performed using* $m = 1$, $n = 1$, $L_x = 1$, $L_y = 1$, $c = 1$, $I = J = 32$, $\delta t = 10^{-2}$, *and* $f = 0.7071$.

8   Particle-in-Cell Codes

8.1   Introduction

Consider an unmagnetized, uniform, 1-dimensional plasma consisting of N electrons and N unit-charged ions. Now, ions are much more massive than electrons. Hence, on short time-scales, we can treat the ions as a static neutralizing background, and only consider the motion of the electrons. Let r_i be the x-coordinate of the ith electron. The equations of motion of the ith electron are written:

$$\frac{dr_i}{dt} = v_i, \tag{8.1}$$

$$\frac{dv_i}{dt} = -\frac{e\,E(r_i)}{m_e}, \tag{8.2}$$

where $e > 0$ is the magnitude of the electron charge, m_e the electron mass, and $E(x)$ the x-component of the electric field-strength at position x. Now, the electric field-strength can be expressed in terms of an electric potential:

$$E(x) = -\frac{d\phi(x)}{dx}. \tag{8.3}$$

Furthermore, from the Poisson-Maxwell equation, we have

$$\frac{d^2\phi(x)}{dx^2} = -\frac{e}{\epsilon_0}\{n_0 - n(x)\}, \tag{8.4}$$

where ϵ_0 is the permittivity of free-space, $n(x)$ the electron number density (*i.e.*, $n(x)\,dx$ is the number of electrons in the interval x to $x + dx$), and n_0 the uniform ion number density. Of course, the average value of $n(x)$ is equal to n_0, since there are equal numbers of ions and electrons.

Let us consider an initial electron distribution function consisting of two counter-propagating Maxwellian beams of mean speed v_b and thermal spread v_{th}: *i.e.*,

$$f(x,v) = \frac{n_0}{2}\left\{\frac{1}{\sqrt{2\pi}\,v_{th}}\,e^{-(v-v_b)^2/2\,v_{th}^2} + \frac{1}{\sqrt{2\pi}\,v_{th}}\,e^{-(v+v_b)^2/2\,v_{th}^2}\right\}. \tag{8.5}$$

Here, $f(x, v) \, dx \, dv$ is the number of electrons between x and $x + dx$ with velocities in the range v to $v + dv$. Of course, $n(x) = \int_{-\infty}^{\infty} f(x, v) \, dv$. The beam temperature T is related to the thermal velocity via $v_{th} = \sqrt{k_B \, T / m_e}$, where k_B is the Boltzmann constant. It is well-known that if v_b is significantly larger than v_{th} then the above distribution is unstable to a plasma instability called the *two-stream instability*.[38] Let us investigate this instability numerically.

8.2   Normalization Scheme

It is convenient to normalize time with respect to ω_p^{-1}, where

$$\omega_p^2 = \frac{n_0 \, e^2}{\epsilon_0 \, m_e} \tag{8.6}$$

is the so-called *plasma frequency*: *i.e.*, the typical frequency of electrostatic electron oscillations. Likewise it is convenient to normalize length with respect to the so-called *Debye length*:

$$\lambda_D = \frac{v_{th}}{\omega_p},$$

which is the length-scale above which the electrons exhibit collective (*i.e.*, plasma-like) effects, instead of acting like individual particles.

Our normalized equations take the form:

$$\frac{dx_i}{dt} = v_i, \tag{8.7}$$

$$\frac{dv_i}{dt} = -E(x_i), \tag{8.8}$$

$$E(x) = -\frac{d\phi(x)}{dx}, \tag{8.9}$$

$$\frac{d^2\phi(x)}{dx^2} = \frac{n(x)}{n_0} - 1. \tag{8.10}$$

---

[38]T.H. Stix, *The theory of plasma waves*, 1st Ed. (McGraw-Hill, New York NY, 1962).

whereas our initial distribution function becomes

$$f(x, v) = \frac{n_0}{2} \left\{ \frac{1}{\sqrt{2\pi}} e^{-(v-v_b)^2/2} + \frac{1}{\sqrt{2\pi}} e^{-(v+v_b)^2/2} \right\}. \tag{8.11}$$

Note that $v_{\text{th}} = 1$ in normalized units.

Let us solve the above system of equations in the domain $0 \le x \le L$. Furthermore, for the sake of simplicity, let us adopt *periodic* boundary conditions: *i.e.*, let us identify the left and right boundaries of our solution domain. It follows that $n(0) = n(L)$, $\phi(0) = \phi(L)$, and $E(0) = E(L)$. Moreover, any electron which crosses the right boundary of the solution domain must reappear at the left boundary with the same velocity, and *vice versa*.

8.3   Solution of Electron Equations of Motion

We can solve the electron equations of motion, (8.7) and (8.8), as a set of 2N coupled first-order ODEs using the RK4 methods discussed in Sect. 3. However, in order to evaluate the right-hand sides of these equations we need to know the electric field $E(x)$ at each time-step. We can achieve this by solving Poisson's equation, (8.10), every time-step. However, in order to determine the source term for this equation we need to calculate the electron number density $n(x)$, which is, of course, a function of the instantaneous electron locations.

8.4   Evaluation of Electron Number Density

In order to obtain the electron number density $n(x)$ from the electron coordinates r_i we adopt a so-called *particle-in-cell* (PIC) approach. Let us define a set of J equally spaced spatial grid-points located at coordinates

$$x_j = j \, \delta x, \tag{8.12}$$

for $j = 0, J - 1$, where $\delta x = L/J$. Let $n_j \equiv n(x_j)$. Suppose that the ith electron lies between the jth and $(j+1)$th grid-points: *i.e.*, $x_j < r_i < x_{j+1}$. We let

$$n_j \rightarrow n_j + \left(\frac{x_{j+1} - r_i}{x_{j+1} - x_j}\right) \bigg/ \delta x, \tag{8.13}$$

and

$$n_{j+1} \rightarrow n_{j+1} + \left(\frac{r_i - x_j}{x_{j+1} - x_j}\right) \bigg/ \delta x. \tag{8.14}$$

Thus, $n_j \, \delta x$ increases by 1 if the electron is at the jth grid-point, $n_{j+1} \, \delta x$ increases by 1 if the electron is at the $(j+1)$th grid-point, and $n_j \, \delta x$ and $n_{j+1} \, \delta x$ both increase by $1/2$ if the electron is halfway between the two grid-points, *etc.* Performing a similar assignment for each electron in turn allows us to build up the n_j from the electron coordinates (assuming that all the n_j are initialized to zero at the start of this process).

8.5 Solution of Poisson's Equation

Consider the solution of Poisson's equation:

$$\frac{d^2\phi(x)}{dx^2} = \rho(x), \tag{8.15}$$

where $\rho(x) = n(x)/n_0 - 1$. Note that $n_0 = N/L$ in normalized units. Let $\phi_j \equiv \phi(x_j)$ and $\rho_j \equiv \rho(x_j)$. We can write

$$\phi_j = \sum_{j'=0,J-1} \hat{\phi}_{j'} \, e^{i j j' 2\pi/J}, \tag{8.16}$$

$$\rho_j = \sum_{j'=0,J-1} \hat{\rho}_{j'} \, e^{i j j' 2\pi/J}, \tag{8.17}$$

which automatically satisfies the periodic boundary conditions $\phi_J = \phi_0$ and $\rho_J = \rho_0$. Note that $\hat{\rho}_0 = 0$, since $\int_0^L n(x) \, dx = n_0$. The other $\hat{\rho}_j$ are obtained from

$$\hat{\rho}_j = \frac{1}{J} \sum_{j'=0,J-1} \rho_{j'} \, e^{-i j j' 2\pi/J}, \tag{8.18}$$

for $j = 1, J - 1$. The Fourier transformed version of Poisson's equation yields

$$\hat{\phi}_0 = 0 \tag{8.19}$$

and

$$\hat{\phi}_j = -\frac{\hat{\rho}_j}{j^2 \, \kappa^2} \tag{8.20}$$

for $j = 1, J/2$, where $\kappa = 2\pi/L$. Finally,

$$\hat{\phi}_j = \hat{\phi}_{J-j}^* \tag{8.21}$$

for $j = J/2 + 1$ to $J - 1$, which ensures that the ϕ_j remain real. The discretized version of Eq. (8.9) is

$$E_j = \frac{\phi_{j-1} - \phi_{j+1}}{2 \, \delta x}. \tag{8.22}$$

Of course, $j = 0$ and $j = J - 1$ are special cases which can be resolved using the periodic boundary conditions. Finally, suppose that the coordinate of the ith electron lies between the jth and $(j+1)$th grid-points: *i.e.*, $x_j < r_i < x_{j+1}$. We can then use linear interpolation to evaluate the electric field seen by the ith electron:

$$E(r_i) = \left(\frac{x_{j+1} - r_i}{x_{j+1} - x_j} \right) E_j + \left(\frac{r_i - x_j}{x_{j+1} - x_j} \right) E_{j+1}. \tag{8.23}$$

8.6 An example 1-D PIC Code

The following code is an implementation of the ideas developed above.

The main function reads in the calculation parameters, checks that they are sensible, initializes the electron coordinates, and then evolves the electron equations of motion from $t = 0$ to some specified t_{max}, using a fixed step RK4 routine with some specified time-step δt. Information on the electron phase-space coordinates and the electric field is periodically written to various data-files.

```
// 1-d PIC code to solve plasma two-stream instability problem.

#include <stdlib.h>
```

```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <blitz/array.h>
#include <fftw.h>

using namespace blitz;

void Output (char* fn1, char* fn2, double t,
     Array<double,1> r, Array<double,1> v);
void Density (Array<double,1> r, Array<double,1>& n);
void Electric (Array<double,1> phi, Array<double,1>& E);
void Poisson1D (Array<double,1>& u, Array<double,1> v, double kappa);
void rk4_fixed (double& x, Array<double,1>& y,
                void (*rhs_eval)(double, Array<double,1>, Array<double,1>&),
                double h);
void rhs_eval (double t, Array<double,1> y, Array<double,1>& dydt);
void Load (Array<double,1> r, Array<double,1> v, Array<double,1>& y);
void UnLoad (Array<double,1> y, Array<double,1>& r, Array<double,1>& v);
double distribution (double vb);

double L; int N, J;

int main()
{
  // Parameters
  L;            // Domain of solution 0 <= x <= L (in Debye lengths)
  N;            // Number of electrons
  J;            // Number of grid points
  double vb;    // Beam velocity
  double dt;    // Time-step (in inverse plasma frequencies)
  double tmax;  // Simulation run from t = 0. to t = tmax

  // Get parameters
```

```
printf ("Please input N:  "); scanf ("%d", &N);
printf ("Please input vb:  "); scanf ("%lf", &vb);
printf ("Please input L:  "); scanf ("%lf", &L);
printf ("Please input J:  "); scanf ("%d", &J);
printf ("Please input dt:  "); scanf ("%lf", &dt);
printf ("Please input tmax:  "); scanf ("%lf", &tmax);
int skip = int (tmax / dt) / 10;
if ((N < 1) || (J < 2) || (L <= 0.) || (vb <= 0.)
    || (dt <= 0.) || (tmax <= 0.) || (skip < 1))
  {
    printf ("Error - invalid input parameters\n");
    exit (1);
  }

// Set names of output files
char* phase[11]; char* data[11];
phase[0] = "phase0.out";phase[1] = "phase1.out";phase[2] = "phase2.out";
phase[3] = "phase3.out";phase[4] = "phase4.out";phase[5] = "phase5.out";
phase[6] = "phase6.out";phase[7] = "phase7.out";phase[8] = "phase8.out";
phase[9] = "phase9.out";phase[10] = "phase10.out";data[0] = "data0.out";
data[1] = "data1.out"; data[2] = "data2.out"; data[3] = "data3.out";
data[4] = "data4.out"; data[5] = "data5.out"; data[6] = "data6.out";
data[7] = "data7.out"; data[8] = "data8.out"; data[9] = "data9.out";
data[10] = "data10.out";

// Initialize solution
double t = 0.;
int seed = time (NULL); srand (seed);
Array<double,1> r(N), v(N);
for (int i = 0; i < N; i++)
  {
    r(i) = L * double (rand ()) / double (RAND_MAX);
    v(i) = distribution (vb);
  }
```

```
  Output (phase[0], data[0], t, r, v);

  // Evolve solution
  Array<double,1> y(2*N);
  Load (r, v, y);
  for (int k = 1; k <= 10; k++)
    {
      for (int kk = 0; kk < skip; kk++)
        {
          // Take time-step
          rk4_fixed (t, y, rhs_eval, dt);

          // Make sure all coordinates in range 0 to L.
          for (int i = 0; i < N; i++)
            {
              if (y(i) < 0.) y(i) += L;
              if (y(i) > L) y(i) -= L;
            }

          printf ("t = %11.4e\n", t);
        }
      printf ("Plot %3d\n", k);

      // Output data
      UnLoad (y, r, v);
      Output(phase[k], data[k], t, r, v);
    }

  return 0;
}
```

The following routine outputs the simulation data to various data-files.

```
// Write data to output files
```

```
void Output (char* fn1, char* fn2, double t,
     Array<double,1> r, Array<double,1> v)
{
  // Write phase-space data
  FILE* file = fopen (fn1, "w");
  for (int i = 0; i < N; i++)
    fprintf (file, "%e %e\n", r(i), v(i));
  fclose (file);

  // Write electric field data
  Array<double,1> ne(J), n(J), phi(J), E(J);
  Density (r, ne);
  for (int j = 0; j < J; j++)
    n(j) = double (J) * ne(j) / double (N) - 1.;
  double kappa = 2. * M_PI / L;
  Poisson1D (phi, n, kappa);
  Electric (phi, E);

  file = fopen (fn2, "w");
  for (int j = 0; j < J; j++)
    {
      double x = double (j) * L / double (J);
      fprintf (file, "%e %e %e %e\n", x, ne(j), n(j), E(j));
    }
  double x = L;
  fprintf (file, "%e %e %e %e\n", x, ne(0), n(0), E(0));
  fclose (file);
}
```

The following routine returns a random velocity distributed on a double Maxwellian distribution function corresponding to two counter-streaming beams. The algorithm used to achieve this is called the *rejection method,* and will be discussed later in this course.

```
// Function to distribute electron velocities randomly so as
// to generate two counter propagating warm beams of thermal
// velocities unity and mean velocities +/- vb.
// Uses rejection method.

double distribution (double vb)
{
  // Initialize random number generator
  static int flag = 0;
  if (flag == 0)
    {
      int seed = time (NULL);
      srand (seed);
      flag = 1;
    }

  // Generate random v value
  double fmax = 0.5 * (1. + exp (-2. * vb * vb));
  double vmin = - 5. * vb;
  double vmax = + 5. * vb;
  double v = vmin + (vmax - vmin) * double (rand ()) / double (RAND_MAX);

  // Accept/reject value
  double f = 0.5 * (exp (-(v - vb) * (v - vb) / 2.) +
    exp (-(v + vb) * (v + vb) / 2.));
  double x = fmax * double (rand ()) / double (RAND_MAX);
  if (x > f) return distribution (vb);
  else return v;
}
```

The routine below evaluates the electron number density on an evenly spaced mesh given the instantaneous electron coordinates.

```
// Evaluates electron number density n(0:J-1) from
```

```
// array r(0:N-1) of electron coordinates.

void Density (Array<double,1> r, Array<double,1>& n)
{
  // Initialize
  double dx = L / double (J);
  n = 0.;

  // Evaluate number density.
  for (int i = 0; i < N; i++)
    {
      int j = int (r(i) / dx);
      double y = r(i) / dx - double (j);
      n(j) += (1. - y) / dx;
      if (j+1 == J) n(0) += y / dx;
      else n(j+1) += y / dx;
    }
}
```

The following functions are wrapper routines for using the fftw library with periodic functions.

```
// Functions to calculate Fourier transforms of real data
// using fftw Fast-Fourier-Transform routine.
// Input/ouput arrays are assumed to be of extent J.

// Calculates Fourier transform of array f in arrays Fr and Fi
void fft_forward (Array<double,1>f, Array<double,1>&Fr,
      Array<double,1>& Fi)
{
  fftw_complex ff[J], FF[J];

  // Load data
  for (int j = 0; j < J; j++)
```

```
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (J, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  for (int j = 0; j < J; j++)
    {
      Fr(j) = c_re (FF[j]); Fi(j) = c_im (FF[j]);
    }

  // Normalize data
  Fr /= double (J);
  Fi /= double (J);
}

// Calculates inverse Fourier transform of arrays Fr and Fi in array f
void fft_backward (Array<double,1> Fr, Array<double,1> Fi,
      Array<double,1>& f)
{
  fftw_complex ff[J], FF[J];

  // Load data
  for (int j = 0; j < J; j++)
    {
      c_re (FF[j]) = Fr(j); c_im (FF[j]) = Fi(j);
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (J, FFTW_BACKWARD, FFTW_ESTIMATE);
```

```
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  for (int j = 0; j < J; j++)
      f(j) = c_re (ff[j]);
}
```

The following routine solves Poisson's equation in 1-D to find the instanta-
neous electric potential on a uniform grid.

```
// Solves 1-d Poisson equation:
//    d^u / dx^2 = v    for  0 <= x <= L
// Periodic boundary conditions:
//    u(x + L) = u(x),  v(x + L) = v(x)
// Arrays u and v assumed to be of length J.
// Now, jth grid point corresponds to
//    x_j = j dx  for j = 0,J-1
// where dx = L / J.
// Also,
//    kappa = 2 pi / L

void Poisson1D (Array<double,1>& u, Array<double,1> v, double kappa)
{
  // Declare local arrays.
  Array<double,1> Vr(J), Vi(J), Ur(J), Ui(J);

  // Fourier transform source term
  fft_forward (v, Vr, Vi);

  // Calculate Fourier transform of u
  Ur(0) = Ui(0) = 0.;
  for (int j = 1; j <= J/2; j++)
    {
```

```
     Ur(j) = - Vr(j) / double (j * j) / kappa / kappa;
     Ui(j) = - Vi(j) / double (j * j) / kappa / kappa;
   }
 for (int j = J/2; j < J; j++)
   {
     Ur(j) = Ur(J-j);
     Ui(j) = - Ui(J-j);
   }


 // Inverse Fourier transform to obtain u
 fft_backward (Ur, Ui, u);
}
```

The following function evaluates the electric field on a uniform grid from the electric potential.

```
// Calculate electric field from potential

void Electric (Array<double,1> phi, Array<double,1>& E)
{
  double dx = L / double (J);

  for (int j = 1; j < J-1; j++)
    E(j) = (phi(j-1) - phi(j+1)) / 2. / dx;
  E(0) = (phi(J-1) - phi(1)) / 2. / dx;
  E(J-1) = (phi(J-2) - phi(0)) / 2. / dx;
}
```

The following routine is the right-hand side routine for the electron equations of motion. Is is designed to be used with the fixed-step RK4 solver described earlier in this course.

```
// Electron equations of motion:
//     y(0:N-1)   = r_i
```

```
//     y(N:2N-1) = dr_i/dt

void rhs_eval (double t, Array<double,1> y, Array<double,1>& dydt)
{
  // Declare local arrays
  Array<double,1> r(N), v(N), rdot(N), vdot(N), r0(N);
  Array<double,1> ne(J), rho(J), phi(J), E(J);

  // Unload data from y
  UnLoad (y, r, v);

  // Make sure all coordinates in range 0 to L
  r0 = r;
  for (int i = 0; i < N; i++)
    {
      if (r0(i) < 0.) r0(i) += L;
      if (r0(i) > L) r0(i) -= L;
    }

  // Calculate electron number density
  Density (r0, ne);

  // Solve Poisson's equation
  double n0 = double (N) / L;
  for (int j = 0; j < J; j++)
    rho(j) = ne(j) / n0 - 1.;
  double kappa = 2. * M_PI / L;
  Poisson1D (phi, rho, kappa);

  // Calculate electric field
  Electric (phi, E);

  // Equations of motion
  for (int i = 0; i < N; i++)
```

```
    {
      double dx = L / double (J);
      int j = int (r0(i) / dx);
      double y = r0(i) / dx - double (j);

      double Efield;
      if (j+1 == J)
         Efield = E(j) * (1. - y) + E(0) * y;
      else
         Efield = E(j) * (1. - y) + E(j+1) * y;

      rdot(i) = v(i);
      vdot(i) = - Efield;
    }

  // Load data into dydt
  Load (rdot, vdot, dydt);
}
```

The following functions load and unload the electron phase-space coordinates into the solution vector y used by the RK4 routine.

```
// Load particle coordinates into solution vector

void Load (Array<double,1> r, Array<double,1> v, Array<double,1>& y)
{
  for (int i = 0; i < N; i++)
    {
      y(i) = r(i);
      y(N+i) = v(i);
    }
}


// Unload particle coordinates from solution vector
```
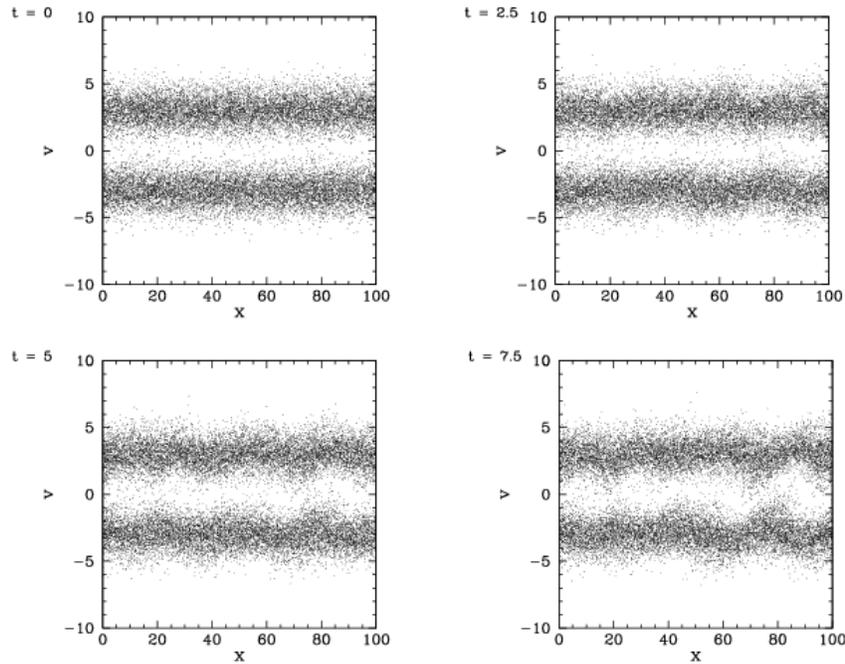
Figure 89: *The electron phase-space distribution evaluated at various times for a 1-dimensional two-stream instability calculation performed with* $N = 20000$, $J = 1000$, $L = 100$, $v_b = 3$, *and* $\delta t = 0.1$.

```
void UnLoad (Array<double,1> y, Array<double,1>& r, Array<double,1>& v)
{
  for (int i = 0; i < N; i++)
    {
      r(i) = y(i);
      v(i) = y(N+i);
    }
}
```

8.7   Results

Figures 89 and 90 show the electron phase-space distributions evaluated at equally spaced times for a two-stream instability calculation performed with 2×10^4 electrons. It can be seen that the distribution initially takes the form of two uni-
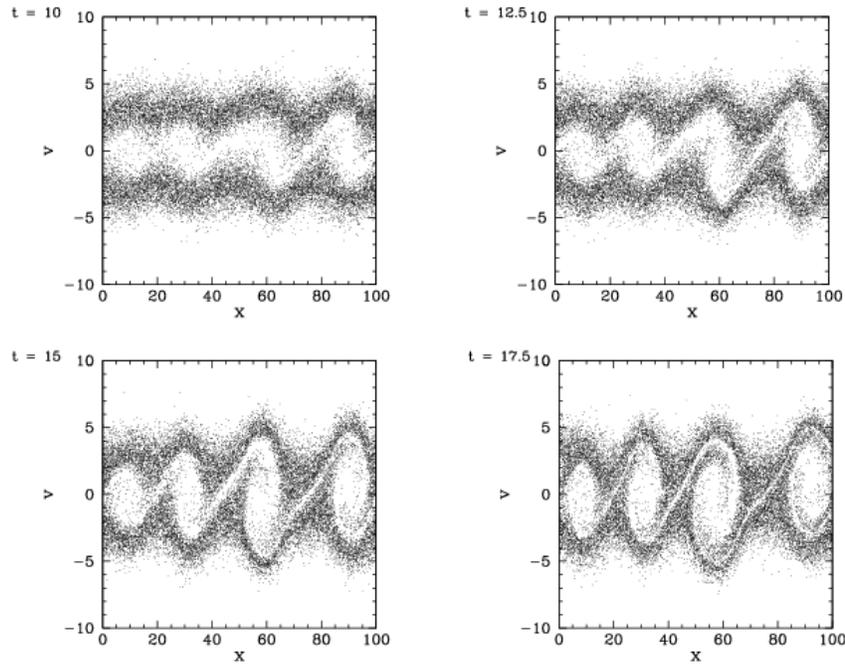
Figure 90: *The electron phase-space distribution evaluated at various times for a 1-dimensional two-stream instability calculation performed with* $N = 20000$, $J = 1000$, $L = 100$, $v_b = 3$, *and* $\delta t = 0.1$.

form bands, corresponding to two counter-streaming electron beams. However, as time progresses, the bands spontaneously develop structure which grows in magnitude and eventually converts the phase-space distribution into a set of connected vortices. In this final state, the electrons are basically bouncing backwards and forwards in a quasi-periodic electric potential generated by non-uniformities in the electron density. In other words, the instability effectively destroys the two beams. For this reason, the two-stream instability is of major concern in particle accelerators, which often consist of counter-propagating charged particle beams.

8.8 Discussion

Obviously, the ideas discussed above could be generalized in a fairly straight-forward manner to deal with the evolution of two and three-dimensional charged particle distributions. PIC codes have the advantage that they are reasonably straight-forward to write. Unfortunately, PIC codes also have a number of disadvantages. The first is that PIC codes suffer from high levels of statistical noise,

since they generally only deal with a relatively small number of particles (typically, $\leq 10^6$). Real physical systems do not exhibit anything like the same level of statistical noise, since they generally contain of order Avogadro's number ($\sim 10^{24}$) of interacting particles. Another problem with PIC codes is that they do not handle charged particle collisions very well. The reason for this is that there are generally a large number of particles in each cell (for practical reasons), and the short range Coulomb fields of these particles tend to cancel one another out (recall that the electric field is only calculated at the cell vertices).

9   Monte-Carlo Methods

9.1   Introduction

Numerical methods which make use of random numbers are called *Monte-Carlo methods*—after the famous casino. The obvious applications of such methods are in *stochastic physics*: *e.g.*, statistical thermodynamics. However, there are other, less obvious, applications: *e.g.*, the evaluation of multi-dimensional integrals.

9.2   Random Numbers

No numerical algorithm can generate a truly random sequence of numbers, However, there exist algorithms which generate repeating sequences of M (say) integers which are, to a fairly good approximation, randomly distributed in the range 0 to $M-1$. Here, M is a (hopefully) large integer. This type of sequence is termed *psuedo-random*.

The most well-known algorithm for generating psuedo-random sequences of integers is the so-called *linear congruental* method. The formula linking the nth and $(n+1)$th integers in the sequence is

$$I_{n+1} = (A\,I_n + C)\bmod M, \tag{9.1}$$

where A, C, and M are positive integer constants. The first number in the sequence, the so-called "seed" value, is selected by the user.

Consider an example case in which $A = 7$, $C = 0$, and $M = 10$. A typical sequence of numbers generated by formula (9.1) is

$$I = \{3, 1, 7, 9, 3, 1, \cdots\}. \tag{9.2}$$

Evidently, the above choice of values for A, C, and M is not a particularly good one, since the sequence repeats after only four iterations. However, if A, C, and M are properly chosen then the sequence is of maximal length (*i.e.*, of length M), and approximately randomly distributed in the range 0 to $M - 1$.

The function listed below is an implementation of the linear congruental method.

```
// random.cpp

// Linear congruential psuedo-random number generator.
// Generates psuedo-random sequence of integers in
//  range 0 .. RANDMAX.

#define RANDMAX 6074  // RANDMAX = M - 1

int random (int seed = 0)
{
  static int next = 1;
  static int A = 106;
  static int C = 1283;
  static int M = 6075;

  if (seed) next = seed;
  next = next * A + C;
  return next % M;
}
```

The keyword `static` in front of a local variable declaration indicates that the program should preserve the value of that variable between function calls. In other words, if the static variable `next` has the value 999 on exit from function `random` then the next time this function is called `next` will have exactly the same value. Note that the values of non-static local variables are not preserved between function calls. The `= 0` in the first line of function `random` is a default value for the argument `seed`. In fact, `random` can be called in one of two ways. Firstly, `random` can be called with no argument: *i.e.*, `random ():` in which case, `seed` is given the default value 0. Secondly, `random` can be called with an integer argument: *i.e.*, `random (n):` in which case, the value of `seed` is set to `n`. The first way of calling `random` just returns the next integer in the psuedo-random sequence. The second way seeds the sequence with the value n (*i.e.*, I_1 is set to n), and then returns the next integer in the sequence (*i.e.*, I_2). Note that the function prototype for `random` takes the form `int random (int = 0):` the `= 0` indicates that the argument is optional.

The above function returns a pseudo-random integer in the range 0 to `RANDMAX`
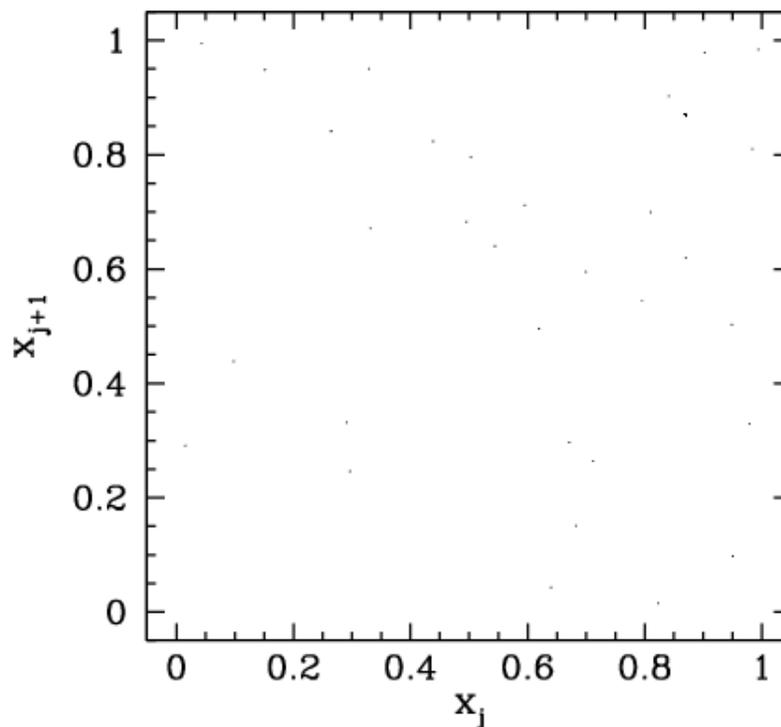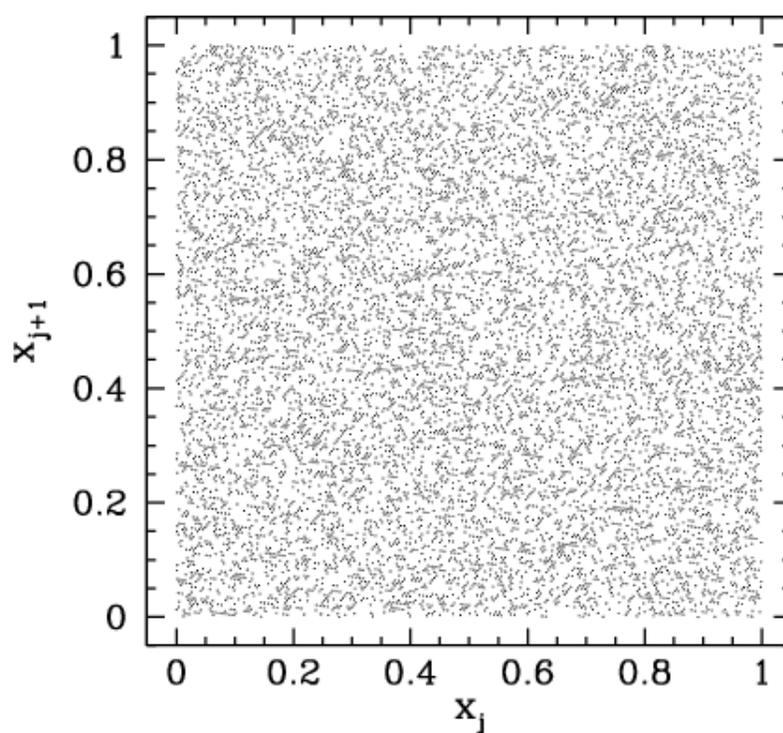
Figure 91: *Plot of* x_j *versus* x_{j+1} *for* $j = 1, 10000$. *Here, the* x_j *are random values, uniformly distributed in the range 0 to 1, generated using a linear congruental psuedo-random number generator characterized by* $A = 106$, $C = 1283$, *and* $M = 6075$.

(where `RANDMAX` takes the value $M - 1$). In order to obtain a random variable x, uniformly distributed in the range 0 to 1, we would write

```
x = double (random ()) / double (RANDMAX);
```

Now if x is truly random then there should be no correlation between successive values of x. Thus, a good way of testing our random number generator is to plot x_j versus x_{j+1} (where x_j corresponds to the jth number in the psuedo-random sequence) for many different values of j. For a good random number generator, the plotted points should densely fill the unit square. Moreover, there should be no discernible pattern in the distribution of points.

Figure 91 shows a correlation plot for the first 10000 x_j–x_{j+1} pairs generated using a linear congruental psuedo-random number generator characterized by
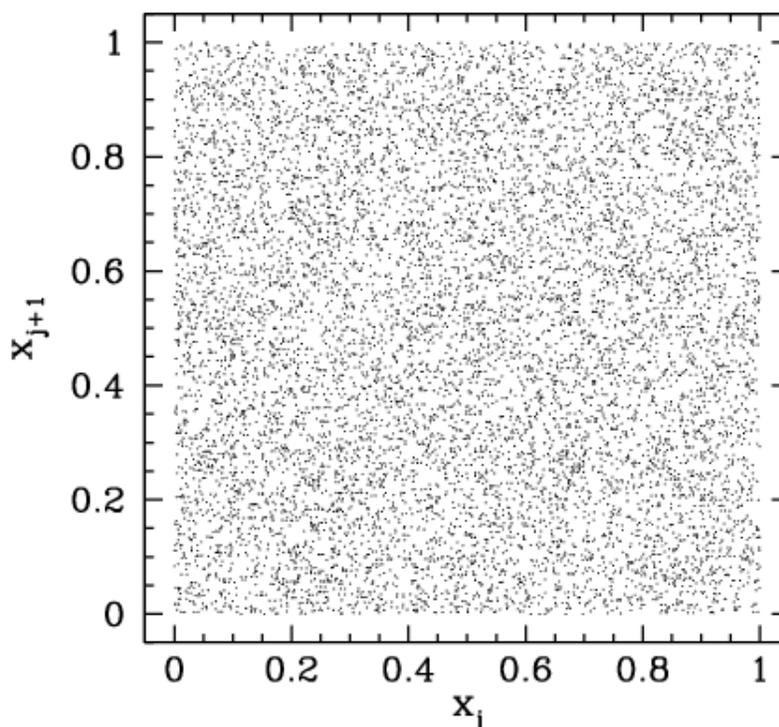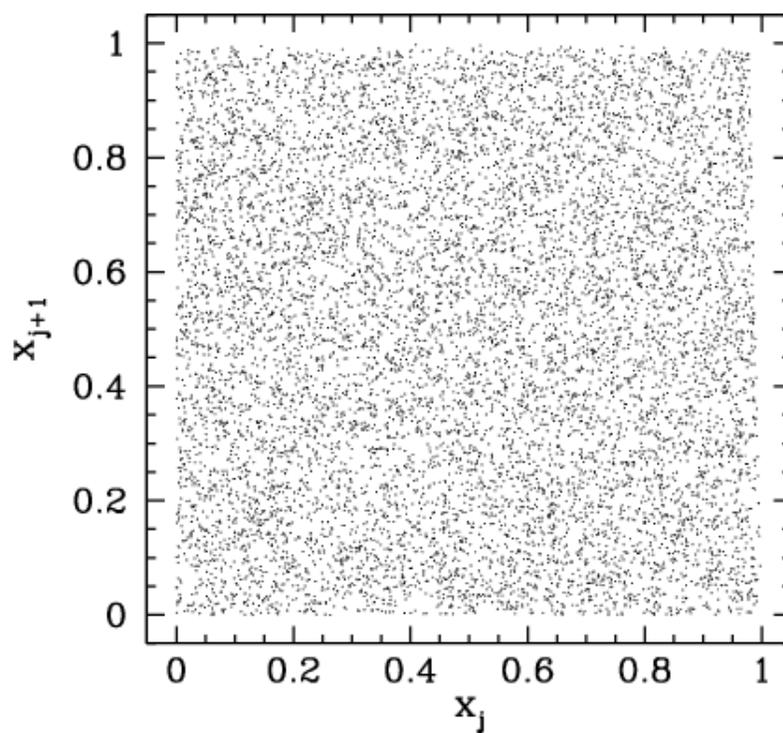
Figure 92: *Plot of x_j versus x_{j+1} for $j = 1, 10000$. Here, the x_j are random values, uniformly distributed in the range 0 to 1, generated using a linear congruental psuedo-random number generator characterized by $A = 107$, $C = 1283$, and $M = 6075$.*

$A = 106$, $C = 1283$, and $M = 6075$. It can be seen that this is a poor choice of values for A, C, and M, since the pseudo-random sequence repeats after a few iterations, yielding x_j values which do not densely fill the interval 0 to 1.

Figure 92 shows a correlation plot for the first 10000 x_j–x_{j+1} pairs generated using a linear congruental psuedo-random number generator characterized by $A = 107$, $C = 1283$, and $M = 6075$. It can be seen that this is a far better choice of values for A, C, and M, since the pseudo-random sequence is of maximal length, yielding x_j values which are fairly evenly distributed in the range 0 to 1. However, if we look carefully at Fig. 92, we can see that there is a slight tendency for the dots to line up in the horizontal and vertical directions. This indicates that the x_j are not quite randomly distributed: *i.e.*, there is some correlation between successive x_j values. The problem is that M is too low: *i.e.*, there is not a sufficiently wide selection of different x_j values in the interval 0 to 1.

Figure 93: *Plot of x_j versus x_{j+1} for $j = 1, 10000$. Here, the x_j are random values, uniformly distributed in the range 0 to 1, generated using a linear congruental psuedo-random number generator characterized by $A = 1103515245$, $C = 12345$, and $M = 32768$.*

Figure 93 shows a correlation plot for the first 10000 x_j–x_{j+1} pairs generated using a linear congruental psuedo-random number generator characterized by $A = 1103515245$, $C = 12345$, and $M = 32768$. The clumping of points in this figure indicates that the x_j are again not quite randomly distributed. This time the problem is integer overflow: *i.e.*, the values of A and M are sufficiently large that $A\,I_n > 10^{32} - 1$ for many integers in the pseudo-random sequence. Thus, the algorithm (9.1) is not being executed correctly.

Integer overflow can be overcome using *Schrange's algorithm*. If $y = (A\,z)\,\mathrm{mod}\,M$ then

$$y = \begin{cases} A\,(z\,\mathrm{mod}\,q) - r\,(z/q) & \text{if } y > 0 \\ A\,(z\,\mathrm{mod}\,q) - r\,(z/q) + M & \text{otherwise} \end{cases}, \qquad (9.3)$$

where $q = M/A$ and $r = M\%A$. The so-called *Park and Miller* method for generating a pseudo-random sequence corresponds to a linear congruental method characterized by the values $A = 16807$, $C = 0$, and $M = 2147483647$. The function listed below implements this method, using Schrange's algorithm to avoid integer overflow.

```
// random.cpp

// Park and Miller's psuedo-random number generator.

#define RANDMAX 2147483646 // RANDMAX = M - 1

int random (int seed = 0)
{
  static int next = 1;
  static int A = 16807;
  static int M = 2147483647;   // 2^31 - 1
  static int q = 127773;       // M / A
  static int r = 2836;         // M % A

  if (seed) next = seed;
  next = A * (next % q) - r * (next / q);
  if (next < 0) next += M;
  return next;
}
```

Figure 94 shows a correlation plot for the first 10000 x_j–x_{j+1} pairs generated using Park & Miller's method. We can now see no pattern whatsoever in the

Figure 94: *Plot of x_j versus x_{j+1} for $j = 1, 10000$.   Here, the x_j are random values, uniformly distributed in the range 0 to 1, generated using Park & Miller's psuedo-random number generator.*

plotted points. This indicates that the x_j are indeed randomly distributed in the range 0 to 1. From now on, we shall use Park & Miller's method to generate all the psuedo-random numbers needed in our investigation of Monte-Carlo methods.

9.3 Distribution Functions

Let $P(x)\,dx$ represent the probability of finding the random variable x in the interval x to $x + dx$. Here, $P(x)$ is termed a *probability density*. Note that $P = 0$ corresponds to no chance, whereas $P = 1$ corresponds to certainty. Since it is certain that the value of x lies in the range $-\infty$ to $+\infty$, probability densities are subject to the normalizing constraint

$$\int_{-\infty}^{+\infty} P(x)\,dx = 1. \tag{9.4}$$

Suppose that we wish to construct a random variable x which is uniformly distributed in the range x_1 to x_2. In other words, the probability density of x is

$$P(x) = \begin{cases} 1/(x_2 - x_1) & \text{if } x_1 \leq x \leq x_2 \\ 0 & \text{otherwise} \end{cases}. \tag{9.5}$$

Such a variable is constructed as follows

```
x = x1 + (x2 - x1) * double (random ()) / double (RANDMAX);
```

There are two basic methods of constructing non-uniformly distributed random variables: *i.e.*, the *transformation method* and the *rejection method*. We shall examine each of these methods in turn.

Let us first consider the transformation method. Let $y = f(x)$, where f is a known function, and x is a random variable. Suppose that the probability density of x is $P_x(x)$. What is the probability density, $P_y(y)$, of y? Our basic rule is the conservation of probability:

$$|P_x(x)\,dx| = |P_y(y)\,dy|. \tag{9.6}$$

In other words, the probability of finding x in the interval x to $x + dx$ is the same as the probability of finding y in the interval y to $y + dy$. It follows that

$$P_y(y) = \frac{P_x(x)}{|f'(x)|}, \tag{9.7}$$

where $f' = df/dx$.

For example, consider the *Poisson distribution*:

$$P_y(y) = \begin{cases} e^{-y} & \text{if } 0 \leq y \leq \infty \\ 0 & \text{otherwise} \end{cases}. \tag{9.8}$$

Let $y = f(x) = -\ln x$, so that $|f'| = 1/x$. Suppose that

$$P_x(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}. \tag{9.9}$$

It follows that

$$P_y(y) = \frac{1}{|f'|} = x = e^{-y}, \tag{9.10}$$

with $x = 0$ corresponding to $y = \infty$, and $x = 1$ corresponding to $y = 0$. We conclude that if

```
x = double (random ()) / double (RANDMAX);
y = - log (x);
```

then y is distributed according to the Poisson distribution.

The transformation method requires a differentiable probability distribution function. This is not always practical. In such cases, we can use the rejection method instead.

Suppose that we desire a random variable y distributed with density $P_y(y)$ in the range y_{min} to y_{max}. Let $P_{y\,max}$ be the maximum value of $P(y)$ in this range (see Fig. 95). The rejection method is as follows. The variable y is sampled randomly in the range y_{min} to y_{max}. For each value of y we first evaluate $P_y(y)$. We next generate a random number x which is uniformly distributed in the range 0 to
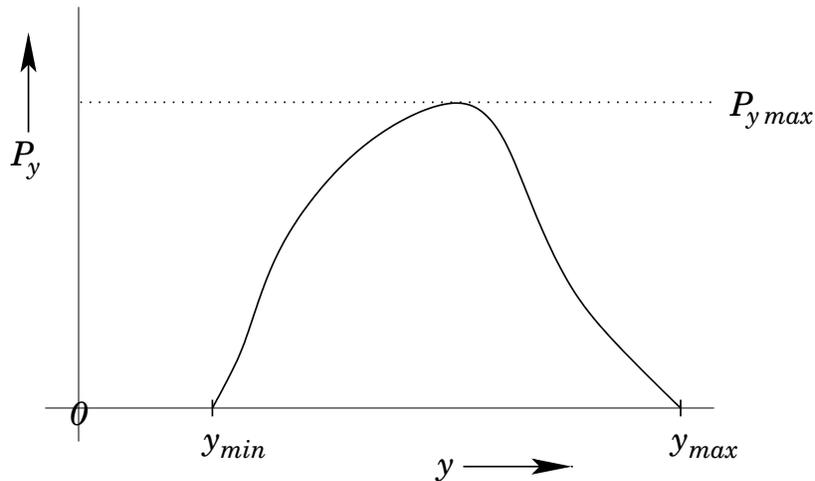
Figure 95: *The rejection method.*

$P_{y\,max}$. Finally, if $P_y(y) < x$ then we reject the y value; otherwise, we keep it. If this prescription is followed then y will be distributed according to $P_y(y)$.

As an example, consider the Gaussian distribution:

$$P_y(y) = \frac{\exp[(y - \bar{y})^2/2\,\sigma^2]}{\sqrt{2\pi}\,\sigma}, \tag{9.11}$$

where $\bar{y}$ is the mean value of y, and σ is the standard deviation. Let

$$y_{min} = \bar{y} - 4\,\sigma, \tag{9.12}$$

$$y_{max} = \bar{y} + 4\,\sigma, \tag{9.13}$$

since there is a negligible chance that y lies more than 4 standard deviations from its mean value. It follows that

$$P_{y\,max} = \frac{1}{\sqrt{2\pi}\,\sigma}, \tag{9.14}$$

with the maximum occurring at $y = \bar{y}$. The function listed below employs the rejection method to return a random variable distributed according to a Gaussian distribution with mean `mean` and standard deviation `sigma`:

```
// gaussian.cpp
```

```
// Function to return random variable distributed
```

293

```
// according to Gaussian distribution with mean mean
// and standard deviation sigma.

#define RANDMAX 2147483646

int random (int = 0);

double gaussian (double mean, double sigma)
{
  double ymin = mean - 4. * sigma;
  double ymax = mean + 4. * sigma;
  double Pymax = 1. / sqrt (2. * M_PI) / sigma;

  // Calculate random value uniformly distributed
  //  in range ymin to ymax
  double y = ymin + (ymax - ymin) * double (random ()) / double (RANDMAX);

  // Calculate Py
  double Py = exp (- (y - mean) * (y - mean) / 2. / sigma / sigma) /
    sqrt (2. * M_PI) / sigma;

  // Calculate random value uniformly distributed in range 0 to Pymax
  double x = Pymax * double (random ()) / double (RANDMAX);

  // If x > Py reject value and recalculate
  if (x > Py) return gaussian (mean, sigma);
  else return y;
}
```

Figure 96 illustrates the performance of the above function. It can be seen that the function successfully returns a random value distributed according to the Gaussian distribution.

9.4   Monte-Carlo Integration

Consider a one-dimensional integral: $\int_{x_l}^{x_h} f(x)\, dx$. We can evaluate this integral numerically by dividing the interval x_l to x_h into N identical subdivisions of width

$$h = \frac{x_h - x_l}{N}. \tag{9.15}$$
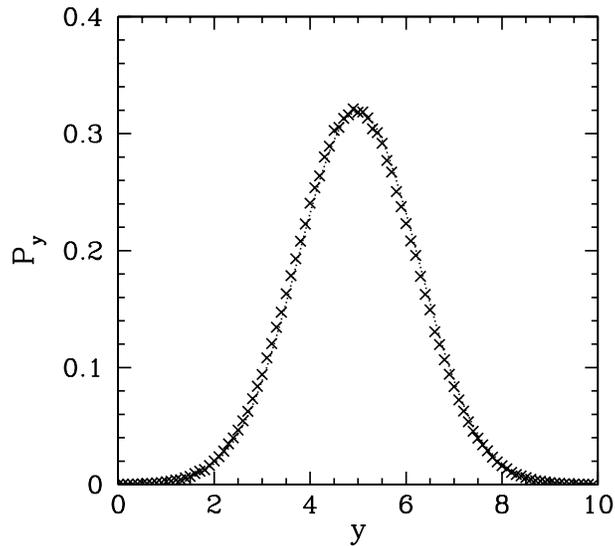
Figure 96: *A million values returned by function* gaussian *with* mean = 5. *and* sigma = 1.25. *The values are binned in 100 bins of width 0.1. The figure shows the number of points in each bin divided by a suitable normalization factor. A Gaussian curve is shown for comparison.*

Let x_i be the *midpoint* of the ith subdivision, and let $f_i = f(x_i)$. Our approximation to the integral takes the form

$$\int_{x_l}^{x_h} f(x)\,dx \simeq \sum_{i=1}^{N} f_i\,h \qquad (9.16)$$

This integration method—which is known as the *midpoint method*—is not particularly accurate, but is very easy to generalize to multi-dimensional integrals.

What is the error associated with the midpoint method? Well, the error is the product of the error per subdivision, which is $O(h^2)$, and the number of subdivisions, which is $O(h^{-1})$. The error per subdivision follows from the linear variation of $f(x)$ within each subdivision. Thus, the overall error is $O(h^2) \times O(h^{-1}) = O(h)$. Since, $h \propto N^{-1}$, we can write

$$\int_{x_l}^{x_h} f(x)\,dx \simeq \sum_{i=1}^{N} f_i\,h + O(N^{-1}). \qquad (9.17)$$

Let us now consider a two-dimensional integral. For instance, the area enclosed by a curve. We can evaluate such an integral by dividing space into iden-

tical squares of dimension h, and then counting the number of squares, N (say), whose midpoints lie within the curve. Our approximation to the integral then takes the form

$$A \simeq N\,h^2. \tag{9.18}$$

This is the two-dimensional generalization of the midpoint method.

What is the error associated with the midpoint method in two-dimensions? Well, the error is generated by those squares which are intersected by the curve. These squares either contribute wholly or not at all to the integral, depending on whether their midpoints lie within the curve. In reality, only those parts of the intersected squares which lie within the curve should contribute to the integral. Thus, the error is the product of the area of a given square, which is $O(h^2)$, and the number of squares intersected by the curve, which is $O(h^{-1})$. Hence, the overall error is $O(h^2) \times O(h^{-1}) = O(h) = O(N^{-1/2})$. It follows that we can write

$$A = N\,h^2 + O(N^{-1/2}). \tag{9.19}$$

Let us now consider a three-dimensional integral. For instance, the volume enclosed by a surface. We can evaluate such an integral by dividing space into identical cubes of dimension h, and then counting the number of cubes, N (say), whose midpoints lie within the surface. Our approximation to the integral then takes the form

$$V \simeq N\,h^3. \tag{9.20}$$

This is the three-dimensional generalization of the midpoint method.

What is the error associated with the midpoint method in three-dimensions? Well, the error is generated by those cubes which are intersected by the surface. These cubes either contribute wholly or not at all to the integral, depending on whether their midpoints lie within the surface. In reality, only those parts of the intersected cubes which lie within the surface should contribute to the integral. Thus, the error is the product of the volume of a given cube, which is $O(h^3)$, and the number of cubes intersected by the surface, which is $O(h^{-2})$. Hence, the overall error is $O(h^3) \times O(h^{-2}) = O(h) = O(N^{-1/3})$. It follows that we can write

$$V = N\,h^3 + O(N^{-1/3}). \tag{9.21}$$

Let us, finally, consider using the midpoint method to evaluate the volume, V, of a d-dimensional hypervolume enclosed by a $(d-1)$-dimensional hypersurface. It is clear, from the above examples, that

$$V = N\,h^d + O(N^{-1/d}), \tag{9.22}$$

where N is the number of identical hypercubes into which the hypervolume is divided. Note the increasingly slow fall-off of the error with N as the dimensionality, d, becomes greater. The explanation for this phenomenon is quite simple. Suppose that $N = 10^6$. With $N = 10^6$ we can divide a unit line into (identical) subdivisions whose linear extent is 10^{-6}, but we can only divide a unit area into subdivisions whose linear extent is 10^{-3}, and a unit volume into subdivisions whose linear extent is 10^{-2}. Thus, for a fixed number of subdivisions the grid spacing (and, hence, the integration error) increases dramatically with increasing dimension.

Let us now consider the so-called *Monte-Carlo method* for evaluating multi-dimensional integrals. Consider, for example, the evaluation of the area, A, enclosed by a curve, C. Suppose that the curve lies wholly within some simple domain of area A', as illustrated in Fig. 97. Let us generate N' points which are *randomly* distributed throughout A'. Suppose that N of these points lie within curve C. Our estimate for the area enclosed by the curve is simply

$$A = \frac{N}{N'}\,A'. \tag{9.23}$$

What is the error associated with the Monte-Carlo integration method? Well, each point has a probability $p = A/A'$ of lying within the curve. Hence, the determination of whether a given point lies within the curve is like the measurement of a random variable x which has two possible values: 1 (corresponding to the point being inside the curve) with probability p, and 0 (corresponding to the point being outside the curve) with probability $1 - p$. If we make N' measurements of x (*i.e.*, if we scatter N' points throughout A') then the number of points lying within the curve is

$$N = \sum_{i=1,N'} x_i, \tag{9.24}$$
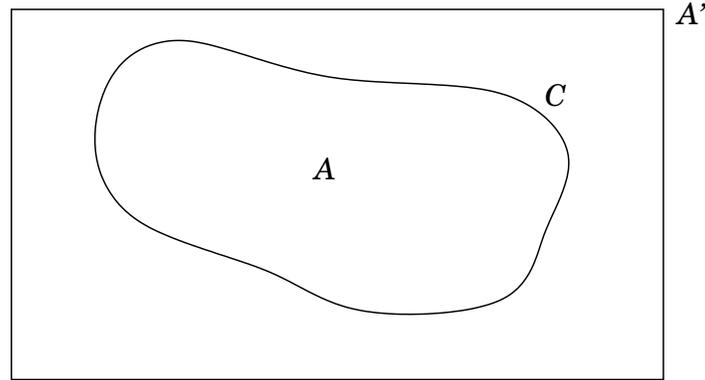
Figure 97: *The Monte-Carlo integration method.*

where x_i denotes the ith measurement of x. Now, the mean value of N is

$$\bar{N} = \sum_{i=1,N'} \bar{x} = N' \bar{x}, \tag{9.25}$$

where

$$\bar{x} = 1 \times p + 0 \times (1 - p) = p. \tag{9.26}$$

Hence,

$$\bar{N} = N' p = N' \frac{A}{A'}, \tag{9.27}$$

which is consistent with Eq. (9.23). We conclude that, on average, a measurement of N leads to the correct answer. But, what is the scatter in such a measurement? Well, if σ represents the *standard deviation* of N then we have

$$\sigma^2 = \overline{(N - \bar{N})^2}, \tag{9.28}$$

which can also be written

$$\sigma^2 = \sum_{i,j,=1,N'} \overline{(x_i - \bar{x})(x_j - \bar{x})}. \tag{9.29}$$

However, $\overline{(x_i - \bar{x})(x_j - \bar{x})}$ equals $\overline{(x - \bar{x})^2}$ if $i = j$, and equals zero, otherwise, since successive measurements of x are *uncorrelated*. Hence,

$$\sigma^2 = N' \overline{(x - \bar{x})^2}. \tag{9.30}$$

Now,

$$\overline{(x - \bar{x})^2} = \overline{(x^2 - 2x\bar{x} + \bar{x}^2)} = \overline{x^2} - \bar{x}^2, \tag{9.31}$$

and

$$\overline{x^2} = 1^2 \times p + 0^2 \times (1-p) = p. \tag{9.32}$$

Thus,

$$\overline{(x-\bar{x})^2} = p - p^2 = p\,(1-p), \tag{9.33}$$

giving

$$\sigma = \sqrt{N'\,p\,(1-p)}. \tag{9.34}$$

Finally, since the likely values of N lie in the range $N = \bar{N} \pm \sigma$, we can write

$$N = N'\,\frac{A}{A'} \pm \sqrt{N'\,\frac{A}{A'}\left(1 - \frac{A}{A'}\right)}. \tag{9.35}$$

It follows from Eq. (9.23) that

$$A = A'\,\frac{N}{N'} \pm \frac{\sqrt{A\,(A'-A)}}{\sqrt{N'}}. \tag{9.36}$$

In other words, the error scales like $(N')^{-1/2}$.

The Monte-Carlo method generalizes immediately to d-dimensions. For instance, consider a d-dimensional hypervolume V enclosed by a $(d-1)$-dimensional hypersurface A. Suppose that A lies wholly within some simple hypervolume V'. We can generate N' points randomly distributed throughout V'. Let N be the number of these points which lie within A. It follows that our estimate for V is simply

$$V = \frac{N}{N'}\,V'. \tag{9.37}$$

Now, there is nothing in our derivation of Eq. (9.36) which depends on the fact that the integral in question is two-dimensional. Hence, we can generalize this equation to give

$$V = V'\,\frac{N}{N'} \pm \frac{\sqrt{V\,(V'-V)}}{\sqrt{N'}}. \tag{9.38}$$

We conclude that the error associated with Monte-Carlo integration *always* scales like $(N')^{-1/2}$, irrespective of the dimensionality of the integral.

We are now in a position to compare and contrast the midpoint and Monte-Carlo methods for evaluating multi-dimensional integrals. In the midpoint method,
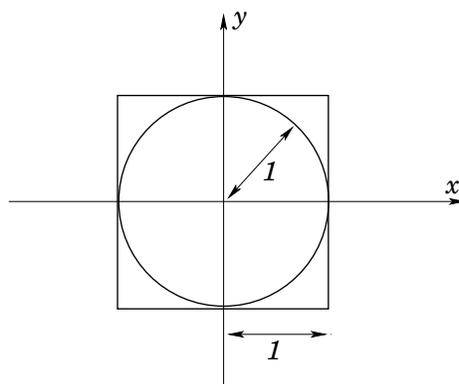
Figure 98: *Example calculation: volume of unit-radius 2-dimensional sphere enclosed in a close-fitting 2-dimensional cube.*

we fill space with an *evenly spaced* mesh of N (say) points (*i.e.*, the midpoints of the subdivisions), and the overall error scales like $N^{-1/d}$, where d is the dimensionality of the integral. In the Monte-Carlo method, we fill space with N (say) *randomly distributed* points, and the overall error scales like $N^{-1/2}$, irrespective of the dimensionality of the integral. For a one-dimensional integral ($d = 1$), the midpoint method is *more efficient* than the Monte-Carlo method, since in the former case the error scales like N^{-1}, whereas in the latter the error scales like $N^{-1/2}$. For a two-dimensional integral ($d = 2$), the midpoint and Monte-Carlo methods are both *equally efficient*, since in both cases the error scales like $N^{-1/2}$. Finally, for a three-dimensional integral ($d = 3$), the midpoint method is *less efficient* than the Monte-Carlo method, since in the former case the error scales like $N^{-1/3}$, whereas in the latter the error scales like $N^{-1/2}$. We conclude that for a sufficiently high dimension integral the Monte-Carlo method is always going to be more efficient than an integration method (such as the midpoint method) which relies on a uniform grid.

Up to now, we have only considered how the Monte-Carlo method can be employed to evaluate a rather special class of integrals in which the integrand function can only take the values 0 or 1. However, the Monte-Carlo method can easily be adapted to evaluate more general integrals. Suppose that we wish to evaluate $\int f \, dV$, where f is a general function and the domain of integration is of arbitrary dimension. We proceed by randomly scattering N points throughout the integration domain and calculating f at each point. Let x_i denote the ith point.
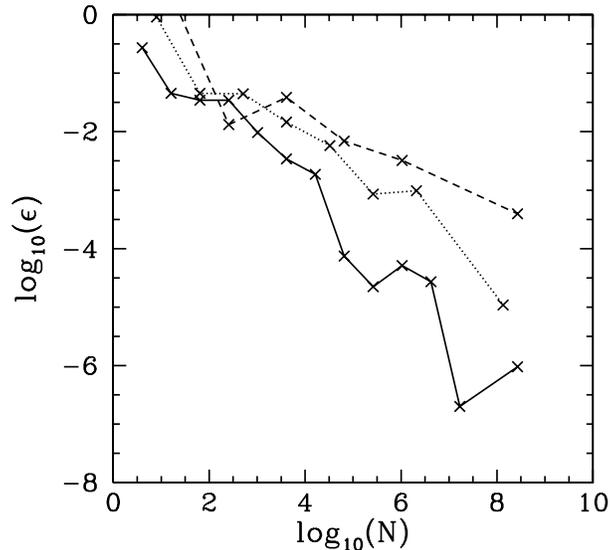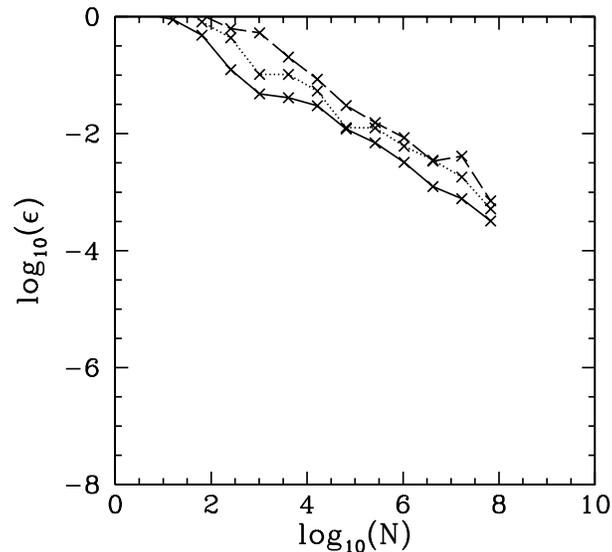
Figure 99: *The integration error, ϵ, versus the number of grid-points, N, for three integrals evaluated using the midpoint method. The integrals are the area of a unit-radius circle (solid curve), the volume of a unit-radius sphere (dotted curve), and the volume of a unit-radius 4-sphere (dashed curve).*

The Monte-Carlo approximation to the integral is simply

$$\int f \, dV = \frac{1}{N} \sum_{i=1,N} f(x_i) + O\left(\frac{1}{\sqrt{N}}\right). \tag{9.39}$$

We end this section with an example calculation. Let us evaluate the volume of a unit-radius d-dimensional sphere, where d runs from 2 to 4, using both the midpoint and Monte-Carlo methods. For both methods, the domain of integration is a cube, centred on the sphere, which is such that the sphere just touches each face of the cube, as illustrated in Fig. 98.

Figure 99 shows the integration error associated with the midpoint method as a function of the number of grid-points, N. It can be seen that as the dimensionality of the integral increases the error falls off much less rapidly as N increases.

Figure 100 shows the integration error associated with the Monte-Carlo method as a function of the number of points, N. It can be seen that there is very little change in the rate at which the error falls off with increasing N as the dimensionality of the integral varies. Hence, as the dimensionality, d, increases the

301

Figure 100: *The integration error, ϵ, versus the number of points, N, for three integrals evaluated using the Monte-Carlo method. The integrals are the area of a unit-radius circle (solid curve), the volume of a unit-radius sphere (dotted curve), and the volume of a unit-radius 4-sphere (dashed curve).*

Monte-Carlo method eventually wins out over the midpoint method.

9.5   The Ising Model

Ferromagnetism arises when a collection of atomic spins align such that their associated magnetic moments all point in the same direction, yielding a net magnetic moment which is macroscopic in size. The simplest theoretical description of ferromagnetism is called the *Ising model*. This model was invented by Wilhelm Lenz in 1920: it is named after Ernst Ising, a student of Lenz who chose the model as the subject of his doctoral dissertation in 1925.

Consider N atoms in the presence of a z-directed magnetic field of strength H. Suppose that all atoms are identical spin-1/2 systems. It follows that either $s_i = +1$ (spin up) or $s_i = -1$ (spin down), where s_i is (twice) the z-component

of the ith atomic spin. The total energy of the system is written:

$$E = -J \sum_{<ij>} s_i \, s_j - \mu H \sum_{i=1,N} s_i. \qquad (9.40)$$

Here, $<ij>$ refers to a sum over *nearest neighbour* pairs of atoms. Furthermore, J is called the *exchange energy*, whereas μ is the atomic *magnetic moment*. Equation (9.40) is the essence of the Ising model.

The physics of the Ising model is as follows. The first term on the right-hand side of Eq. (9.40) shows that the overall energy is lowered when neighbouring atomic spins are aligned. This effect is mostly due to the *Pauli exclusion principle*. Electrons cannot occupy the same quantum state, so two electrons on neighbouring atoms which have parallel spins (*i.e.*, occupy the same orbital state) cannot come close together in space. No such restriction applies if the electrons have anti-parallel spins. Different spatial separations imply different electrostatic interaction energies, and the exchange energy, J, measures this difference. Note that since the exchange energy is *electrostatic* in origin, it can be quite large: *i.e.*, $J \sim 1$ eV. This is far larger than the energy associated with the direct magnetic interaction between neighbouring atomic spins, which is only about 10^{-4} eV. However, the exchange effect is very short-range; hence, the restriction to nearest neighbour interaction is quite realistic.

Our first attempt to analyze the Ising model will employ a simplification known as the *mean field approximation*. The energy of the ith atom is written

$$e_i = -\frac{J}{2} \sum_{k=1,z} s_k \, s_i - \mu H \, s_i, \qquad (9.41)$$

where the sum is over the z nearest neighbours of atom i. The factor $1/2$ is needed to ensure that when we sum to obtain the total energy,

$$E = \sum_{i=1,N} e_i, \qquad (9.42)$$

we do not count each pair of neighbouring atoms twice.

We can write

$$e_i = -\mu \, H_{\text{eff}} \, s_i, \qquad (9.43)$$

where

$$H_{eff} = H + \frac{J}{2\mu} \sum_{k=1,z} s_k. \tag{9.44}$$

Here, H_{eff} is the *effective magnetic field*, which is made up of two components: the external field, H, and the internal field generated by neighbouring atoms.

Consider a single atom in a magnetic field H_m. Suppose that the atom is in thermal equilibrium with a heat bath of temperature T. According to the well-known Boltzmann distribution, the mean spin of the atom is

$$\bar{s} = \frac{e^{+\beta\mu H_m} - e^{-\beta\mu H_m}}{e^{+\beta\mu H_m} + e^{-\beta\mu H_m}}, \tag{9.45}$$

where $\beta = 1/kT$, and k is the Boltzmann constant. The above expression follows because the energy of the "spin up" state ($s = +1$) is $-\mu H_m$, whereas the energy of the "spin down" state ($s = -1$) is $+\mu H_m$. Hence,

$$\bar{s} = \tanh(\beta\,\mu\,H_m). \tag{9.46}$$

Let us assume that all atoms have *identical* spins: *i.e.*, $s_i = \bar{s}$. This assumption is known as the "mean field approximation". We can write

$$H_{eff} = H + \frac{z\,J\,\bar{s}}{2\mu}. \tag{9.47}$$

Finally, we can combine Eqs. (9.46) and (9.47) (identifying H_m and H_{eff}) to obtain

$$\bar{s} = \tanh\{\beta\,\mu\,H + \beta\,z\,J\,\bar{s}/2\}. \tag{9.48}$$

Note that the heat bath in which a given atom is immersed is simply the rest of the atoms. Hence, T is the temperature of the atomic array. It is helpful to define the *critical temperature*,

$$T_c = \frac{z\,J}{2\,k}, \tag{9.49}$$

and the *critical magnetic field*,

$$H_c = \frac{k\,T_c}{\mu} = \frac{z\,J}{2\,\mu}. \tag{9.50}$$

Equation (9.48) reduces to

$$\bar{s} = \tanh\left\{ \frac{T_c}{T}\left(\frac{H}{H_c} + \bar{s} \right) \right\}. \tag{9.51}$$

The above equation cannot be solved analytically. However, it is fairly easily to solve numerically using the following iteration scheme:

$$\bar{s}_{i+1} = \tanh\left\{ \frac{T_c}{T}\left(\frac{H}{H_c} + \bar{s}_i \right) \right\}. \tag{9.52}$$

The above formula is iterated until $\bar{s}_{i+1} \rightarrow \bar{s}_i$.

It is helpful to define the *net magnetization*,

$$M = \mu \sum_{i=1,N} s_i = \mu\, N\, \bar{s}, \tag{9.53}$$

the net energy,

$$E = \sum_{i=1,N} e_i = -N\, k\, T_c \left(\frac{H}{H_c} + \bar{s} \right) \bar{s}, \tag{9.54}$$

and the *heat capacity*,

$$C = \frac{dE}{dT}. \tag{9.55}$$

Figures 102, 101, and 103 show the net magnetization, net energy, and heat capacity calculated from the iteration formula (9.52) in the absence of an external magnetic field (*i.e.*, with $H = 0$). It can be seen that below the critical (or "Curie") temperature, T_c, there is *spontaneous magnetization*: *i.e.*, the exchange effect is sufficiently large to cause neighbouring atomic spins to spontaneously align. On the other hand, thermal fluctuations completely eliminate any alignment above the critical temperature. Moreover, at the critical temperature there is a *discontinuity* in the first derivative of the energy, E, with respect to the temperature, T. This discontinuity generates a downward jump in the heat capacity, C, at $T = T_c$. The sudden loss of spontaneous magnetization as the temperature exceeds the critical temperature is a type of *phase transition*.

Now, according to the conventional classification of *phase transitions*, a transition is *first-order* if the energy is discontinuous with respect to the order parame-
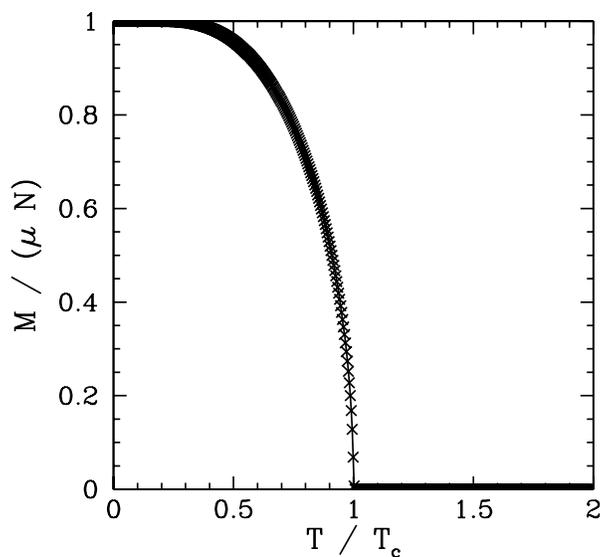
Figure 101: *The net magnetization, M, of a collection of N ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Calculation performed using the mean field approximation.*
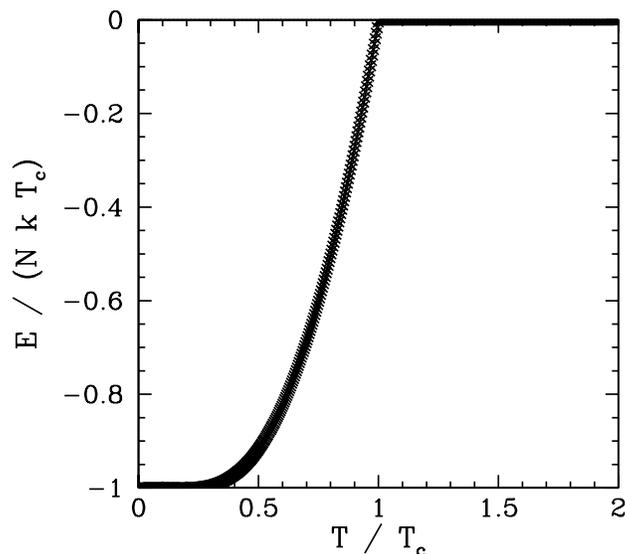
Figure 102: *The net energy, E, of a collection of N ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Calculation performed using the mean field approximation.*
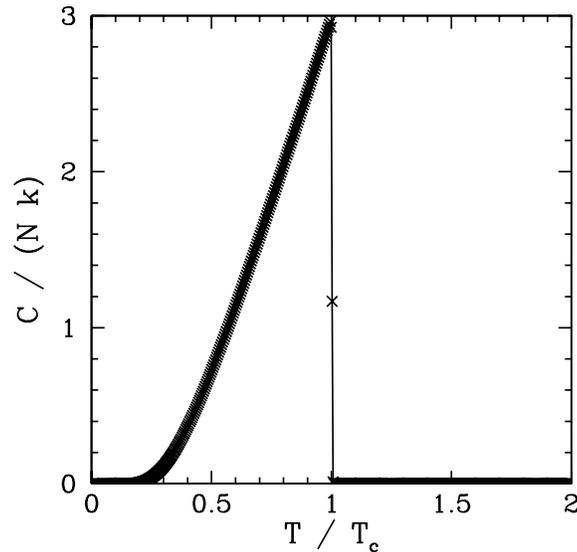
Figure 103: *The heat capacity, C, of a collection of N ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Calculation performed using the mean field approximation.*

ter (*i.e.*, in this case, the temperature), and *second-order* if the energy is continuous, but its first derivative with respect to the order parameter is discontinuous, *etc.* We conclude that the loss of spontaneous magnetization in a ferromagnetic material as the temperature exceeds the critical temperature is a second-order phase transition.

In order to see an example of a first-order phase transition, let us examine the behaviour of the magnetization, M, as the external field, H, is varied at constant temperature, T.

Figures 104 and 105 show the magnetization, M, and energy, E, versus external field-strength, H, calculated from the iteration formula (9.52) at some constant temperature, T, which is less than the critical temperature, T_c. It can be seen that E is *discontinuous*, indicating the presence of a first-order phase transition. Moreover, the system exhibits *hysteresis*—meta-stable states exist within a certain range of H values, and the magnetization of the system at fixed T and H (within the aforementioned range) depends on its *past history*: *i.e.*, on whether H was increasing or decreasing when it entered the meta-stable range.
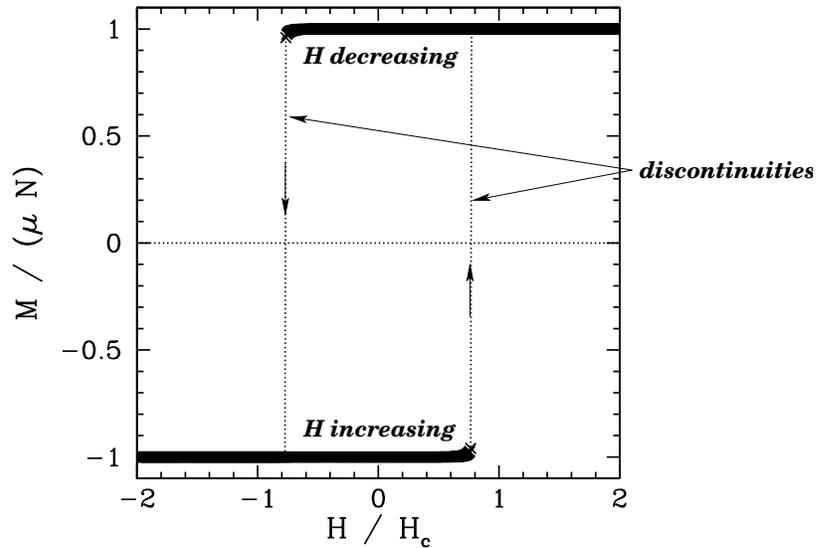
Figure 104: *The net magnetization, M, of a collection of* N *ferromagnetic atoms as a function of the external magnetic field, H, at constant temperature,* T < T$_c$. *Calculation performed using the mean field approximation.*
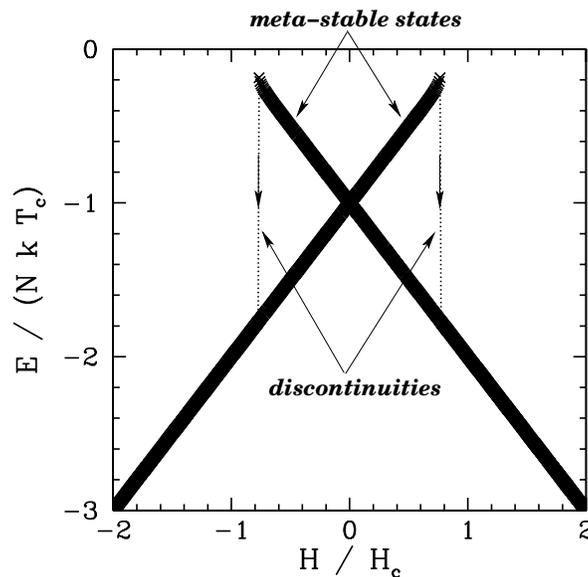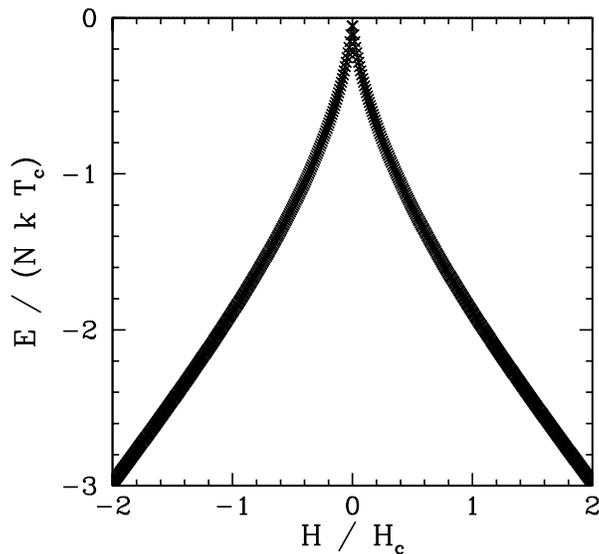

Figure 105: *The net energy, E, of a collection of* N *ferromagnetic atoms as a function of the external magnetic field, H, at constant temperature,* T < T$_c$. *Calculation performed using the mean field approximation.*
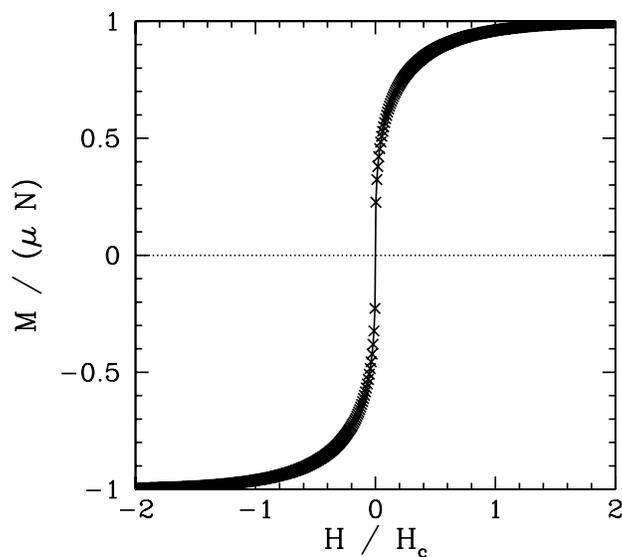
Figure 106: *The net magnetization, M, of a collection of* N *ferromagnetic atoms as a function of the external magnetic field,* H, *at constant temperature,* T $= T_c$. *Calculation performed using the mean field approximation.*

Figure 107: *The net energy,* E, *of a collection of* N *ferromagnetic atoms as a function of the external magnetic field,* H, *at constant temperature,* T $= T_c$. *Calculation performed using the mean field approximation.*
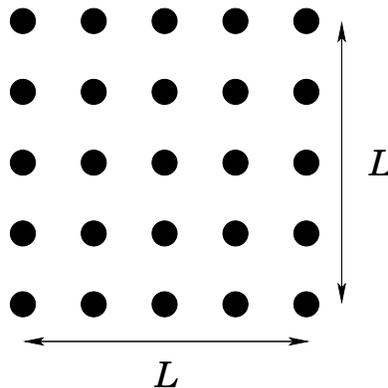
Figure 108: *A two-dimensional array of atoms.*

Figures 106 and 107 show the magnetization, M, and energy, E, versus external field-strength, H, calculated from the iteration formula (9.52) at a constant temperature, T, which is equal to the critical temperature, T_c. It can be seen that E is now *continuous*, and there are no meta-stable states. We conclude that first-order phase transitions and hysteresis only occur, as the external field-strength is varied, when the temperature lies below the critical temperature: *i.e.*, when the ferromagnetic material in question is capable of spontaneous magnetization.

The above calculations, which are based on the mean field approximation, correctly predict the existence of first- and second-order phase transitions when $H \neq 0$ and $H = 0$, respectively. However, these calculations get some of the details of the second-order phase transition wrong. In order to do a better job, we must abandon the mean field approximation and adopt a Monte-Carlo approach.

Let us consider a two-dimensional square array of atoms. Let L be the size of the array, and $N = L^2$ the number of atoms in the array, as shown in Fig. 108. The Monte-Carlo approach to the Ising model, which completely avoids the use of the mean field approximation, is based on the following algorithm:

- Step through each atom in the array in turn:
    - For a given atom, evaluate the change in energy of the system, ΔE, when the atomic spin is flipped.
    - If $\Delta E < 0$ then flip the spin.
    - If $\Delta E > 0$ then flip the spin with probability $P = \exp(-\beta\,\Delta E)$.

- Repeat the process many times until thermal equilibrium is achieved.

The purpose of the algorithm is to shuffle through all possible states of the system, and to ensure that the system occupies a given state with the Boltzmann probability: *i.e.*, with a probability proportional to $\exp(-\beta\,E)$, where E is the energy of the state.

In order to demonstrate that the above algorithm is correct, let us consider flipping the spin of the ith atom. Suppose that this operation causes the system to make a transition from state a (energy, E_a) to state b (energy, E_b). Suppose, further, that $E_a < E_b$. According to the above algorithm, the probability of a transition from state a to state b is

$$P_{a \to b} = \exp[-\beta\,(E_b - E_a)], \qquad (9.56)$$

whereas the probability of a transition from state b to state a is

$$P_{b \to a} = 1. \qquad (9.57)$$

In thermal equilibrium, the well-known *principal of detailed balance* implies that

$$P_a\,P_{a \to b} = P_b\,P_{b \to a}, \qquad (9.58)$$

where P_a is the probability that the system occupies state a, and P_b is the probability that the system occupies state b. Equation (9.58) simply states that in thermal equilibrium the rate at which the system makes transitions from state a to state b is equal to the rate at which the system makes reverse transitions. The previous equation can be rearranged to give

$$\frac{P_b}{P_a} = \exp[-\beta\,(E_b - E_a)], \qquad (9.59)$$

which is consistent with the Boltzmann distribution.

Now, each atom in our array has *four* nearest neighbours, except for atoms on the edge of the array, which have less than four neighbours. We can eliminate this annoying special behaviour by adopting *periodic boundary conditions*: *i.e.*, by identifying opposite edges of the array. Indeed, we can think of the array as existing on the surface of a torus.
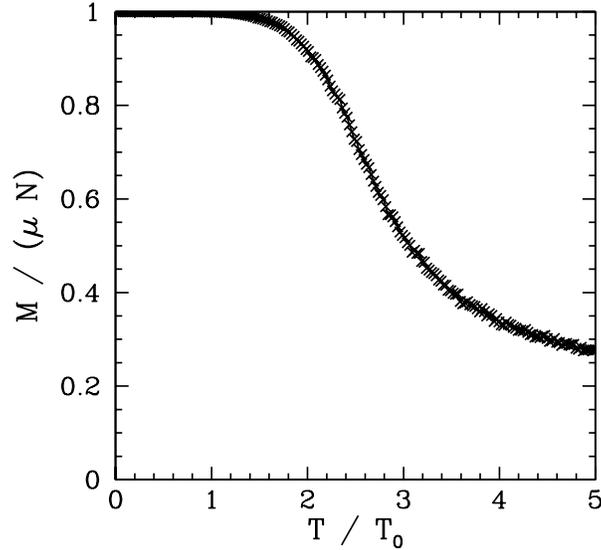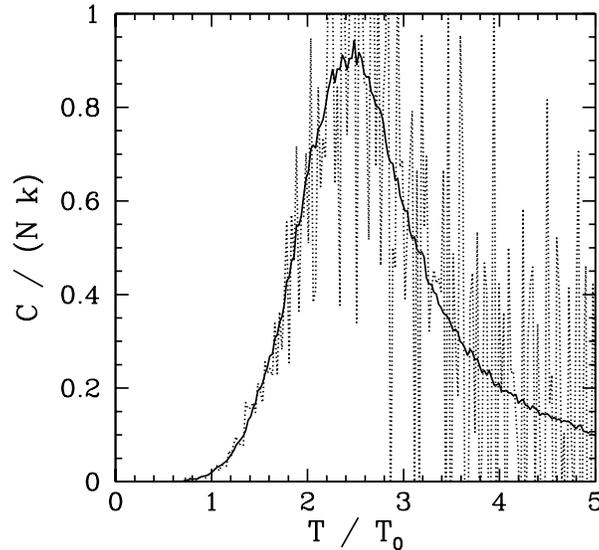
Figure 109: *The net magnetization, M, of a 5 × 5 array of ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Monte-Carlo simulation.*

It is helpful to define

$$T_0 = \frac{J}{k}. \tag{9.60}$$

Now, according to mean field theory,

$$T_c = \frac{z\,J}{2\,k} = 2\,T_0. \tag{9.61}$$

The evaluation of

$$C = \lim_{\Delta T \to 0} \frac{\Delta E}{\Delta T} \tag{9.62}$$

via the direct method is difficult due to statistical noise in the energy, E. Instead, we can make use of a standard result in equilibrium statistical thermodynamics:

$$C = \frac{\sigma_E^2}{k\,T^2}, \tag{9.63}$$

where σ_E is the standard deviation of fluctuations in E. Fortunately, it is fairly easy to evaluate σ_E: we can simply employ the standard deviation in E from step to step in our Monte-Carlo iteration scheme.

Figures 109–116 show magnetization and heat capacity versus temperature curves for L = 5, 10, 20, and 40 in the absence of an external magnetic field. In

Figure 110: *The heat capacity, C, of a 5 × 5 array of ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Monte-Carlo simulation. The solid curve shows the heat capacity calculated from Eq. (9.62), whereas the dotted curve shows the heat capacity calculated from Eq. (9.63).*

all cases, the Monte-Carlo simulation is iterated 5000 times, and the first 1000 iterations are discarded when evaluating σ_E (in order to allow the system to attain thermal equilibrium). The two-dimensional array of atoms is initialized in a fully aligned state for each different value of the temperature. Since there is no external magnetic field, it is irrelevant whether the magnetization, M, is positive or negative. Hence, M is replaced by $|M|$ in all plots.

Note that the M versus T curves generated by the Monte-Carlo simulations look very much like those predicted by the mean field model. The resemblance increases as the size, L, of the atomic array increases. The major difference is the presence of a magnetization "tail" for $T > T_c$ in the Monte-Carlo simulations: *i.e.*, in the Monte-Carlo simulations the spontaneous magnetization does not collapse to zero once the critical temperature is exceeded—there is a small lingering magnetization for $T > T_c$. The C versus T curves show the heat capacity calculated directly (*i.e.*, $C = \Delta E/\Delta T$), and via the identity $C = \sigma_E^2/k\,T^2$. The latter method of calculation is clearly far superior, since it generates significantly less statistical noise. Note that the heat capacity *peaks* at the critical temperature: *i.e.*, unlike
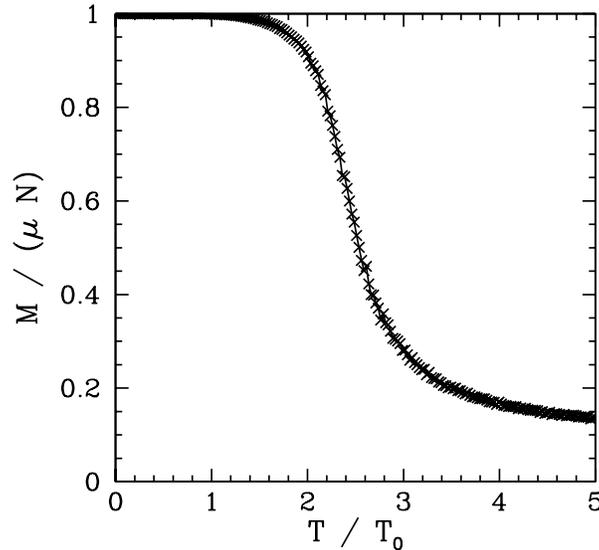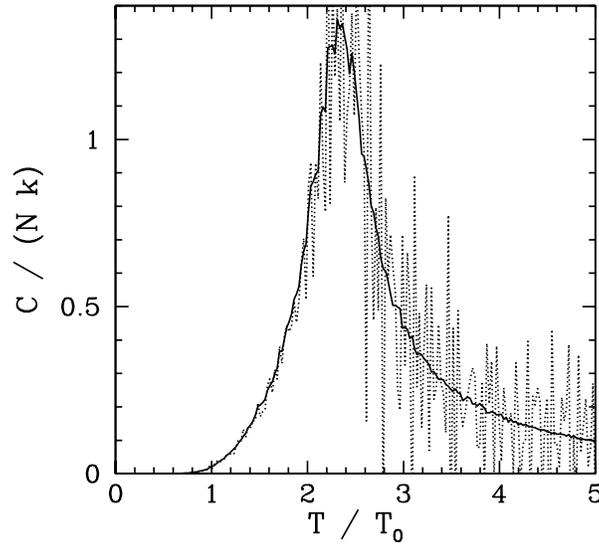
313

Figure 111: *The net magnetization, M, of a* 10×10 *array of ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Monte-Carlo simulation.*

the mean field model, C is not zero for $T > T_c$. This effect is due to the residual magnetization present when $T > T_c$.

Our best estimate for T_c is obtained from the location of the peak in the C versus T curve in Fig. 116. We obtain $T_c = 2.27\, T_0$. Recall that the mean field model yields $T_c = 2\, T_0$. The exact answer for a two-dimensional array of ferromagnetic atoms is

$$T_c = \frac{2\, T_0}{\ln(1 + \sqrt{2})} = 2.27\, T_0, \tag{9.64}$$

which is consistent with our Monte-Carlo calculations. The above analytic result was first obtained by Onsager in 1944.[39] Incidentally, Onsager's analytic solution of the 2-D Ising model is one of the most complicated and involved calculations in all of theoretical physics. Needless to say, no one has ever been able to find an analytic solution of the Ising model in more than two dimensions.

Note, from Figs. 110, 112, 114, and 116, that the height of the peak in the heat capacity curve at $T = T_c$ increases with increasing array size, L. Indeed, a close examination of these figures yields $C_{max}/N\, k = 0.95$ for $L = 5$, $C_{max}/N\, k = 1.34$

---

[39]L. Onsager, Phys. Rev. **65**, 117 (1944).

Figure 112: *The heat capacity,* C, *of a* 10 × 10 *array of ferromagnetic atoms as a function of the temperature,* T, *in the absence of an external magnetic field. Monte-Carlo simulation. The solid curve shows the heat capacity calculated from Eq. (9.62), whereas the dotted curve shows the heat capacity calculated from Eq. (9.63).*

for $L = 10$, $C_{max}/N\,k = 1.77$ for $L = 20$, and $C_{max}/N\,k = 2.16$ for $L = 40$. Figure 117 shows $C_{max}/N\,k$ plotted against $\ln L$ for $L = 5$, 10, 20, and 40. It can be seen that the points lie on a very convincing straight-line, which strongly suggests that

$$\frac{C_{max}}{k\,N} \propto \ln L. \tag{9.65}$$

Of course, for physical systems, $L \sim \sqrt{N_A} \sim 10^{12}$, where N_A is Avogadro's number. Hence, C is effectively *singular* at the critical temperature (since $\ln N_A \gg 1$), as sketched in Fig. 118. This observation leads us to revise our definition of a second-order phase transition. It turns out that actual discontinuities in the heat capacity almost never occur. Instead, second-order phase transitions are characterized by a *local quasi-singularity* in the heat capacity.

Recall, from Eq. (9.63), that the typical amplitude of energy fluctuations is proportional to the square-root of the heat capacity (*i.e.*, $\sigma_E \propto \sqrt{C}$). It follows that the amplitude of energy fluctuations becomes *extremely large* in the vicinity of a second-order phase transition.
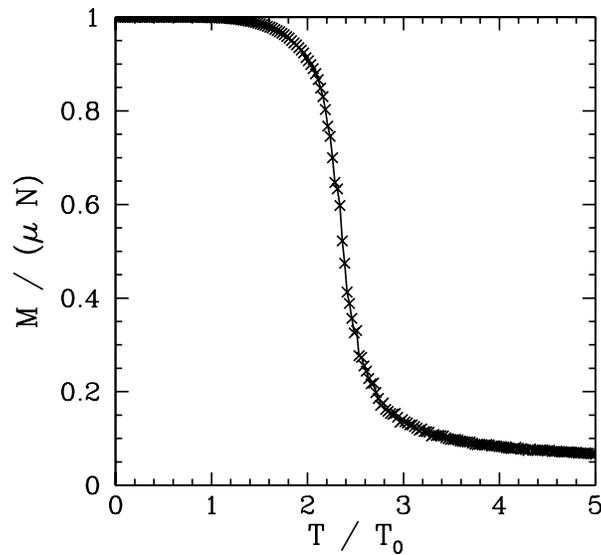
Figure 113: *The net magnetization, M, of a 20 × 20 array of ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Monte-Carlo simulation.*
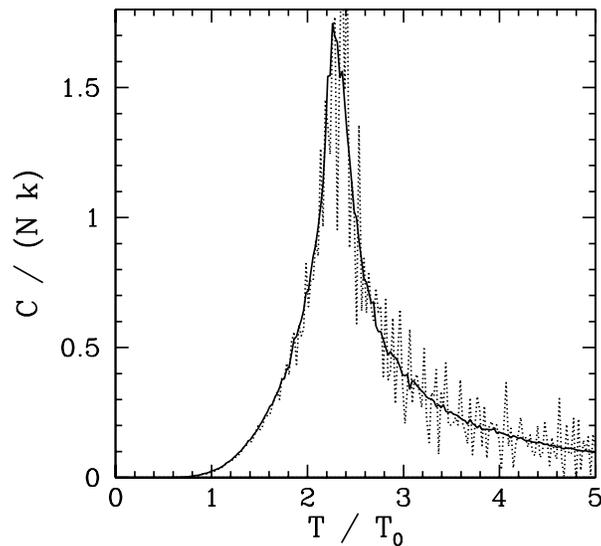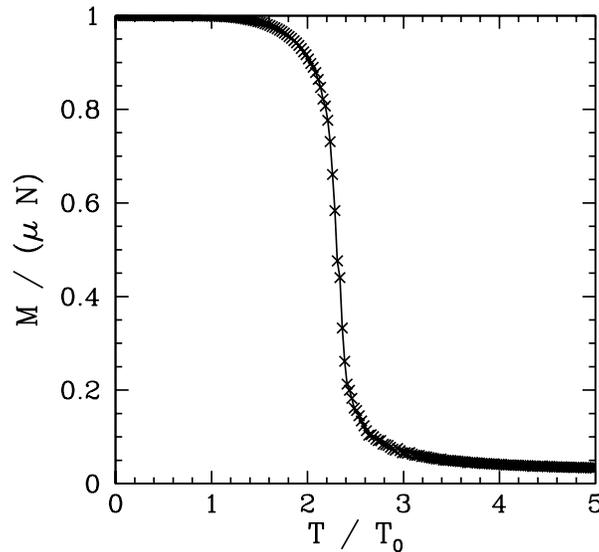

Figure 114: *The heat capacity, C, of a 20 × 20 array of ferromagnetic atoms as a function of the temperature, T, in the absence of an external magnetic field. Monte-Carlo simulation. The solid curve shows the heat capacity calculated from Eq. (9.62), whereas the dotted curve shows the heat capacity calculated from Eq. (9.63).*

Figure 115: *The net magnetization,* M, *of a* 40×40 *array of ferromagnetic atoms as a function of the temperature,* T, *in the absence of an external magnetic field. Monte-Carlo simulation.*

Now, the main difference between our mean field and Monte-Carlo calculations is the existence of residual magnetization for $T > T_c$ in the latter case. Figures 119–123 show the magnetization pattern of a 40×40 array of ferromagnetic atoms, in thermal equilibrium and in the absence of an external magnetic field, calculated at various temperatures. It can be seen that for $T = 20\,T_0$ the pattern is essentially random. However, for $T = 5\,T_0$, small clumps appear in the pattern. For $T = 3\,T_0$, the clumps are somewhat bigger. For $T = 2.32\,T_0$, which is just above the critical temperature, the clumps are global in extent. Finally, for $T = 1.8\,T_0$, which is a little below the critical temperature, there is almost complete alignment of the atomic spins.

The problem with the mean field model is that it assumes that all atoms are situated in identical environments. Hence, if the exchange effect is not sufficiently large to cause global alignment of the atomic spins then there is no alignment at all. What actually happens when the temperature exceeds the critical temperature is that global alignment disappears, but local alignment (*i.e.*, clumping) remains. Clumps are only eliminated by thermal fluctuations once the temperature is significantly greater than the critical temperature. Atoms in the middle of the clumps are situated in a different environment than atoms on the clump
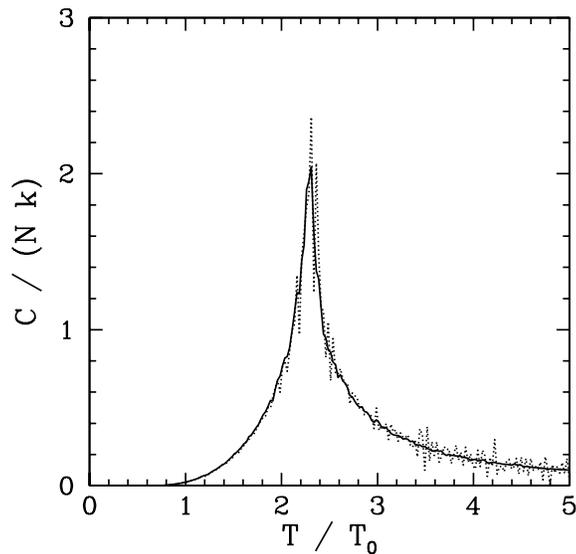
317

Figure 116: *The heat capacity,* C, *of a* 40×40 *array of ferromagnetic atoms as a function of the temperature,* T, *in the absence of an external magnetic field. Monte-Carlo simulation. The solid curve shows the heat capacity calculated from Eq. (9.62), whereas the dotted curve shows the heat capacity calculated from Eq. (9.63).*

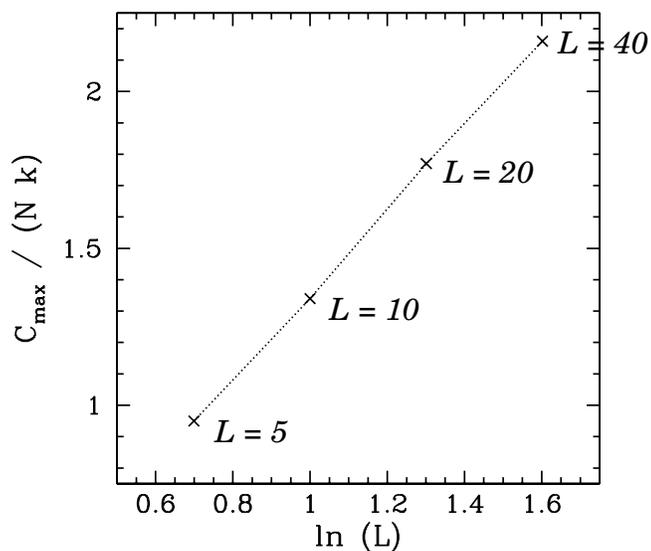boundaries. Hence, clumps cannot occur in the mean field model.

Figure 117: *The peak value of the heat capacity (normalized by* N k*) versus the logarithm of the array size for a two-dimensional array of ferromagnetic atoms in the absence of an external magnetic field. Monte-Carlo simulation.*
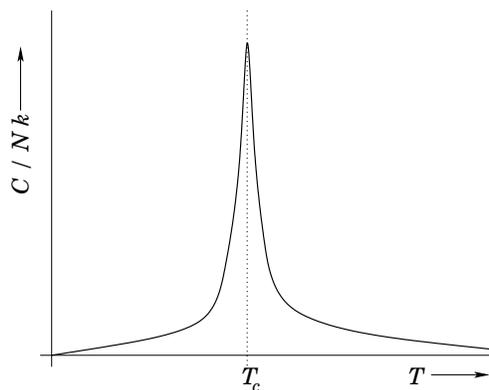
Figure 118: *A sketch of the expected variation of the heat capacity versus the temperature for a physical two-dimensional ferromagnetic system.*
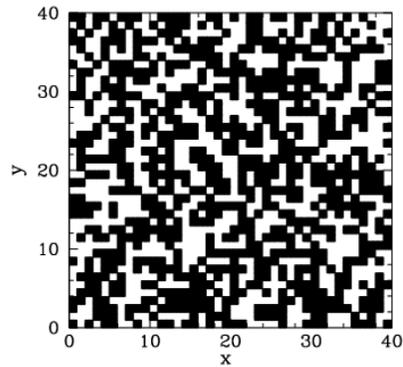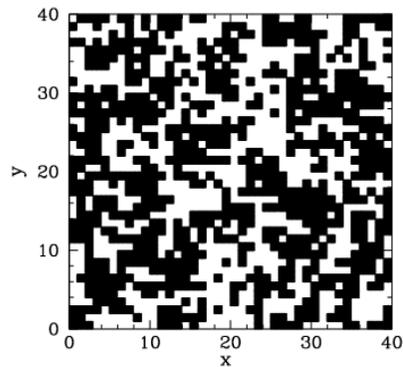
Figure 119: *Magnetization pattern of a* 40×40 *array of ferromagnetic atoms in thermal equilibrium and in the absence of an external magnetic field. Monte-Carlo calculation with* $\mathsf{T} = 20\,\mathsf{T}_0$. *Black/white squares indicate atoms magnetized in plus/minus z-direction, respectively.*

Figure 120: *Magnetization pattern of a* 40×40 *array of ferromagnetic atoms in thermal equilibrium and in the absence of an external magnetic field. Monte-Carlo calculation with* $\mathsf{T} = 5\,\mathsf{T}_0$. *Black/white squares indicate atoms magnetized in plus/minus z-direction, respectively.*

Figure 121: *Magnetization pattern of a* 40×40 *array of ferromagnetic atoms in thermal equilibrium and in the absence of an external magnetic field. Monte-Carlo calculation with* $T = 3\,T_0$. *Black/white squares indicate atoms magnetized in plus/minus z-direction, respectively.*
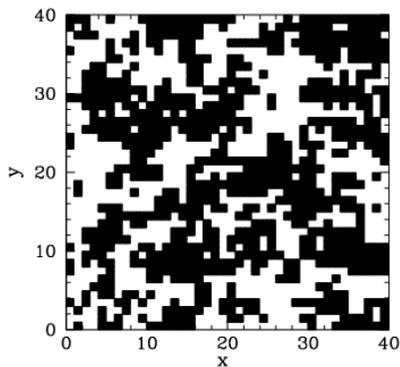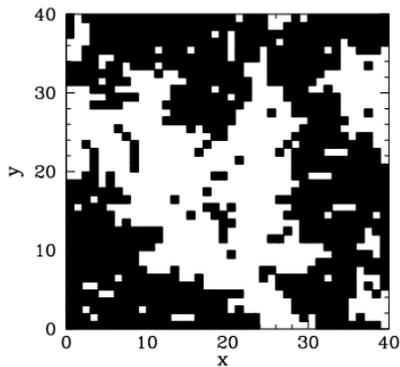

Figure 122: *Magnetization pattern of a* 40×40 *array of ferromagnetic atoms in thermal equilibrium and in the absence of an external magnetic field. Monte-Carlo calculation with* $T = 2.32\,T_0$. *Black/white squares indicate atoms magnetized in plus/minus z-direction, respectively.*
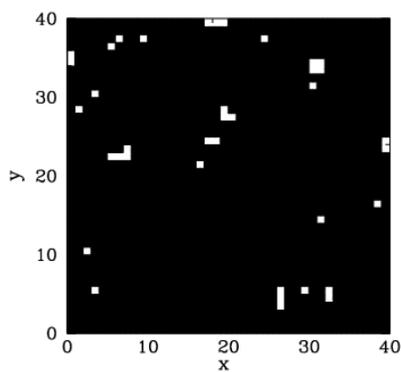
Figure 123: *Magnetization pattern of a* 40×40 *array of ferromagnetic atoms in thermal equilibrium and in the absence of an external magnetic field. Monte-Carlo calculation with* $T = 1.8\,T_0$. *Black/white squares indicate atoms magnetized in plus/minus z-direction, respectively.*