Computer Science 127

Introduction to Fortran & Scientific Programming

J Kiefer
April 2006

Table of Contents

I. Introduction

A. Numerical Methods or Numerical Analysis

1. Numerical Analysis

a. Definition

"Concerned with solving mathematical problems by the operations of arithmetic." That is, we manipulate ($+ / -, \times, \div$, etc.) numerical values rather than derive or manipulate analytical mathematic expressions ($\frac{d}{dx}, \int dx, e^x, x^b, \ln x$, etc.).

We will be dealing always with approximate values rather than exact formulæ.

b. History

Recall the definition of a derivative in Calculus:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = g(x),$$

where $\Delta f = f(x_2) - f(x_1)$ and $\Delta x = x_2 - x_1$. We will work it backwards, using $\frac{df}{dx} \cong \frac{\Delta f}{\Delta x}$.

In fact, before Newton and Leibnitz invented Calculus, the numerical methods were the methods. Mathematical problems were solved numerically or geometrically, e.g., Kepler and Newton with their orbits and gravity. Many of the numerical methods still used today were developed by Newton and his predecessors and contemporaries.

They, or their "computers," performed numerical calculations by hand. That's one reason it could take Kepler so many years to formulate his "Laws" of planetary orbits. In the 19th and early 20th centuries adding machines were used, mechanical and electric. In business, also, payroll and accounts were done by "hand."

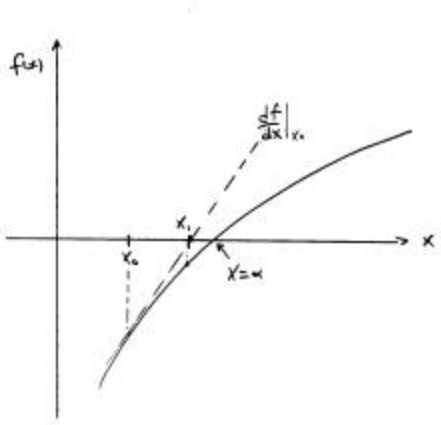
Today, we use automatic machines to do the arithmetic, and the word *computer* no longer refers to a person, but to the machine. The machines are cheaper and faster than people, however, they still have to be told what to do, and when to do it—computer programming.

2. Newton's Method for Solving a Nonlinear Equation—an example

a. Numerical solution

Let's say we want to evaluate the cube root of 467. That is, we want to find a value of x such that $x^3 = 467$. Put another way, we want to find a *root* of the following equation:

$$f(x) = x^3 - 467 = 0.$$



If $f(x)$ were a straight line, then $f(x_1) = f(x_o) + \frac{df(x = x_o)}{dx} = 0$. In fact, $f(x_1) \neq 0$, but let's say that $f(x_1) \cong 0$ and solve for x_1.

$$x_1 = x_o + \frac{f(x_1) - f(x_o)}{\frac{df(x_o)}{dx}} \cong x_o - \frac{f(x_o)}{f'(x_o)}.$$

Note that we are using $f'(x_o) = \frac{df(x = x_o)}{dx}$.

Having now obtained a new estimate for the root, we repeat the process to obtain a sequence of estimated roots which we hope converges on the exact or correct root.

$$x_2 \cong x_1 - \frac{f(x_1)}{f'(x_1)}$$

$$x_3 \cong x_2 - \frac{f(x_2)}{f'(x_2)}$$

etc.

In our example, $f(x) = x^3 - 467$ and $f'(x) = 3x^2$. If we take our *initial guess* to be $x_o = 6$, then by *iterating* the formula above, we generate the following table:

i	x	$f(x)$	$f'(x)$
0	6	-251	108
1	8.324	109.7718	207.8706
2	7.796	6.8172	182.3316
3	7.759	0.108	0.0350

$$x_1 \cong x_o - \frac{f(x_o)}{f'(x_{1o})} = 6 - \frac{-251}{108} = 8.32407$$

$$x_2 \cong x_1 - \frac{f(x_1)}{f'(x_1)} = 8.32407 - \frac{109.7768}{207.8706} = 7.79597$$

$$x_3 \cong x_2 - \frac{f(x_2)}{f'(x_2)} = 7.79597 - \frac{6.817273}{182.33156} = 7.75858$$

[Note: The pocket calculator has a (y^x) button, but a computer may do $x \cdot x \cdot x$ to get x^3.]

b. Analytical solution

How might we solve for the cube root of 467 analytically or symbolically? Take logarithms.

$$x^3 = 467$$

$$3 \ln x = \ln 467$$

$$\ln x = \frac{1}{3} \ln 467$$

$$x = e^{\frac{\ln 467}{3}} = 7.758402264\ldots$$

We used the (ln) button on our pocket calculator, followed by the (e^x) button. In earlier times, we'd have used log tables. But, whence cometh those tables and how does the calculator evaluate ln 467 or $e^{2.0488}$?

3. Series

$$\ln x = (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 + \cdots$$

$$\sin x = x - \frac{x_3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

The *infinite* series are exact. However, in practice we always keep a finite number of terms. In principle, we can achieve arbitrary precision, if we have the necessary patience. Pocket calculators and computer programs add up enough terms in a series to achieve a specified precision, say 8 or 16 significant digits.

4. Error

In this context, the term *error* does not refer to a mistake. Rather, it refers to the idea of deviation or of uncertainty. Every measured value is uncertain, according to the precision of the measuring instrument. Every computed value is uncertain, according to the number of significant digits carried along or according to the number of terms retained in the summation of a series. Consequently, all numerical solutions are approximate.

Oftentimes, in discussing an example problem, the correct exact solution is known, so it is possible to determine how an approximate numerical solution deviates from that exact solution. Indeed, algorithms are often tested by applying them to problems having known exact solutions. However, in real life, we don't know the correct exact solution. We can't know how far our approximate solutions deviate from the correct exact unknown solution. In other words, we have to approximate the solution to a problem, but also we can only estimate the error.

Fortunately, we have means of estimating error. A goodly portion of the discussion in a Numerical Methods textbook is devoted to rigorous estimation of error. In this course, we won't concern ourselves with a detailed discussion of error analysis. Nonetheless, we want to be always aware of the error issue, keeping in mind at least qualitatively the limitations of a numerical solution. From time to time in the paragraphs that follow some aspects of the error involved with a particular algorithm will be briefly discussed.

B. Programming

The computer carries out the tedious arithmetic, but it must be told what to do. That is the function of a computer program. A program may be written in one of any number of programming languages, however there are certain features or issues that all languages have in common.

1. Program Design

a. Stages
Conception—define the problem
Develop the algorithm—map out or outline the solution
Code—write the program
Debug & verify—trace the program; perform trial runs with known results; correct logical & syntax errors

b. Building blocks
Sequential operations—instructions done one after the other in a specified order
Branching operations—selecting alternative sequences of operations
Looping operations—repeating subsets of operations
I/O operations—reading and writing data

2. Branching

a. Simple yes or no—select between just 2 alternative actions

b. Nested branches—a sequence of decisions or branches; decision tree

c. Select case—more than two alternative actions

3. Loops

a. Counted loop—a section of code is executed a specified number of times

b. Conditional loop—a section of code is iterated until a specified condition is met

c. "Infinite' loop—the condition for ending the loop never is encountered, so the program never ends

4. I/O

a. Input—keyboard or data file

b. Output—monitor, output file, printer; numbers, text, graphics

5. Precision Issues

a. Binary

The computer does its arithmetic with binary numbers, that is, base-2. E.g., 0, 1, 10, 11, 100, 101, 110, 111, etc. We are accustomed to working and thinking with base-10 numbers. In producing the machine language code (the "executable") and carrying out calculations, all numerical values are translated from base-10 to base-2 then back again for output. Usually, we don't need to care about this. However, it can be a source of *loss of precision* in our numerical values because the machine stores values with only finite precision.

b. Precision

A single binary digit (0 or 1) is called a *bit*. Eight bits make up a *byte*. Within the machine, the unit of information that is transferred at one time to/from the CPU and main memory is called a *word*. The size of a word, or the *word length*, varies from one machine to another. Typically, it'll be from 4 to 64 bits. A 4-byte word contains 32 bits, etc.

One *memory cell* or *memory location* holds one or more words. Let's say it's one word, or 4 bytes. Whatever information (number) is stored in one such memory cell must be expressible as a string of 32 bits and no more. For instance, a non-terminating binary fraction will be *truncated*, e.g., $(0.1)_{10} = (0.00011001100110011\ldots)_2$. Only 32 digits will be stored in memory. When translated back into decimal, the number will be $(0.09999997)_{10}$, not $(0.1)_{10}$. Similarly, the finite precision places a limit on the largest and the smallest numerical value that can be stored in a memory cell.

In the back of our minds, we always remain aware of the physical limitations of the machine.

6. Debugging

When syntax errors are all eliminated, the program may very well run smoothly to completion. Perhaps it produces results which are clearly absurd; perhaps the results appear quite plausible. A programmer must always take steps to convince itself that the program is working correctly; the temptation to assume must be resisted.

One of the most insidious assumptions is that the program is doing what the programmer intended it to do. Perhaps, a typing error has produced a statement that has no syntax error, but does a different operation from that intended. Perhaps the logical sequence of steps written by the programmer doesn't accomplish the task intended by the programmer. This why *program tracing* is so important, why it is essential to insert print statements all through the program to display the intermediate values of variables, why it is essential to check and double check such things as argument lists and dimensions and the values of indices—checking not what the programmer intended, but what the program actually does.

The other, almost easier, aspect of debugging involves applying the program to a problem whose solution is already known. It also involves repeating a numerical solution with different values of various parameters such as step size and convergence tolerance. It involves comparing a numerical solution for consistency with previous experience.

II. Fortran

A. Constants and Variables [Chapters 2 & 8]

In Fortran, there are five types of data: *integer*, *real*, *complex*, *character* and *logical*.

1. Constants

Constants are values that do not change.

a. Integers

An *integer* is a +/- whole number, such as 8 or –379 or 739238. The maximum number of digits allowed is machine specific, depending on the word length of the machine. Integer constants are never displayed with a decimal point. In contrast to some other programming languages, Fortran treats numbers with decimal points differently from numbers without decimal points.

b. Real numbers

Fortran uses the term *real* to refer to numbers that may have a fractional part such as 65.4 or 0.00873, not to refer to a number whose imaginary part is zero. A real number always has a decimal point. Again, the largest and smallest allowed numerical value is machine dependent. A real constant is stored in exponential or scientific format—as a real mantissa < 1 and an integer exponent: 0.7368×10^{14}. The constant may be displayed in either exponential or decimal form: 37.67×10^{-3} or 0.03767.

c. Complex constants

In Fortran, the term *complex* refers to a number having both a real and an imaginary part. A complex constant is stored in the form of two real constants, in two separate memory cells—one for the real part and one for the imaginary part, in keeping with the mathematical definition of a complex number as an *ordered pair of numbers*.

d. Character constants

Character constants are also known as *character strings*. The character string is a sequence of alphanumeric characters enclosed in single quotes: 'Now is the time for all…' or '3' or 'x = '. Notice that '3' is a character constant while 3 is an integer constant.

e. Logical constants

There are two *logical* constants: .TRUE. and .FALSE. Notice the leading and ending periods. Logical constants and logical operators are enclosed by periods.

2. Variables

Numerical values are stored in memory cells. Each memory cell is assigned a unique address so that the program may refer to each cell. With constants, the contents of the memory cells do not change. However, the contents of a cell associated with a variable may change as the program executes.

a. Variable names and memory cells

In the Fortran program, each variable is given a name. That name is associated uniquely with one memory cell (or two in the cases of complex and double precision variables). The machine maintains a reference table containing every constant and variable name along with the memory address(es) assigned to each.

b. Data types

A variable is defined or *declared* to be of a particular *data type*, and stores numerical values of that type only. The major data types in Fortran are *integer*, *real*, *double precision*, *complex*, *character*, and *logical*. Double precision data has twice the number of digits as normal real or *single precision* data. Therefore, a double precision value occupies two memory cells. Mismatched data will be translated into the data type of the variable it's being stored in. For instance, if we attempt to store the value 45.678 in an integer variable, the fractional part will be truncated, so the value becomes 45. Likewise, an integer such as 567 becomes a real value (567.0) if stored in a real variable.

c. Assignment statements

The program instruction for storing a numerical value in a particular memory cell is called an *assignment statement*. Commonly, such an instruction is represented symbolically as $386 \rightarrow$ jot. In English, this says "store the integer value 386 in the memory cell associated with the variable name *jot*."

In Fortran, the symbol for the assignment operation is the equal sign and the line in the program code would be *jot = 386*. Keep in mind that is not the same meaning as the mathematical statement of equality. *jot = 386* does not mean "jot equals 386." The *arithmetic assignment statements* in a Fortran program resemble mathematical equations, but they are not equations. They are instructions for the machine to carry out certain arithmetical operations and store the result in a specified variable.

d. Variable names

There are restrictions on what string of characters may be used as a *variable name*. Originally, the variable name was restricted to no more than 6 characters. Some implementations of Fortran allow longer variable names. Only alphanumeric characters are allowed. The first character of the name must be a letter. Usually, no distinction is made between upper and lower case—Fortran is *case insensitive*.

Unless otherwise declared, variables beginning with the letters *i* through *n* are assumed to be of the integer data type, while names beginning with *a – h* & *o – z* are assumed to be real. These assumptions are called *implicit data typing*. The implicit data typing is overridden by any explicit data type declaration.

B. Statements

A *statement* is a single instruction. There are several types of statements in Fortran.

non-executable	executable
declaration	assignment
external	if
dimension	goto
common	stop
end	do
parameter	i/o statements
data	return
format	call

1. Non-Executable

Non-executable statements are not *executed* or performed when the program is running. They are implemented during the *compiling* step, when the Fortran code is translated into the machine language. Usually, non-executable statements (except for END) are located at the top of the program code, or *program list*. Some statements must appear at the beginning of the list, others may appear at any place in the program.

a. Declaration

Declaration statements specify the data types of the variables.

b. Parameter

The PARAMETER statement in effect defines a variable name to be a constant.

c. Dimension and Common

These statements define what are called *subscripted variables*, which are like matrices.

d. Data

The DATA statement is used to give initial values to variables.

e. End

The END statement signals the end of a *program block* or *module*.

f. External

The EXTERNAL statement identifies a *subprogram* or module that is defined outside of the main program.

g. Format

The FORMAT statement specifies how output is to be displayed.

2. Executable

Executable statements are instructions that are carried out when the program is running.

a. Assignment

An assignment statement causes the value on the right side of the equal sign to be stored in the location identified with the variable name on the left side of the equal sign. The left side must always be a single variable name. The right side may be an expression or a constant or a single variable name. For instance,

ex = 47.0*sin(theta) why = sqrt(why) + 7.0 zed = 0.9805 que = zed

b. GoTo

A GOTO statement transfers control to a specified program statement. The GOTO may be conditional or unconditional. The statement may appear as two words (go to) or as one word (goto).

c. Do

The DO statement signals the beginning of a *do-loop*, which is a program block that is executed multiple times in a row.

d. Read, Print, Write

These statements are used to put data into or out of the program.

e. Stop

The STOP statement terminates execution of the program. A STOP statement may appear anywhere in the program and there may be more than one STOP statement.

f. Return

The RETURN statement appears in a subprogram or module and has the function of returning control to the calling program or module.

g. Call

The CALL statement transfers control to the particular kind of subprogram called a *subroutine*.

3. Keyboarding

Historically, Fortran statements were punched on computer cards (Holerinth cards), one statement per card. The physical limitations of those cards is still reflected in the restrictions placed on the keyboarding of Fortran statements:

i) the statement must lie entirely in columns 7 thru 72;
ii) column 6 is reserved for a character designating continuation of a statement;
iii) a *c* or *C* placed in column 1 designates a comment line;
iv) statement labels are placed in columns 1 through 5;
v) blank spaces within a statement are ignored.

C. Input and Output [Chapter 3]

Two kinds of input & output are defined: *list directed* and *formatted*.

1. List directed

a. Reading—free format

READ *, var1,var2,var3,…

The values are read from the keyboard—just keyboard the numbers delimited by commas or spaces and end with the ENTER (or RETURN) key. The numbers needn't be entered all on one line; however, each READ statement starts reading from a new line. The numbers can be entered in integer, decimal or exponential form. Character data or logical data can also be entered, if the corresponding variable has been so declared.

b. Printing—free format

PRINT, *,var1,var2,var3,…

Values are printed to the monitor, preformatted. The *print list* may also contain constants. Each PRINT statement begins a new line or a new *record*. If the record exceeds the width of the monitor screen, the record is continued on the next line.

2. Formatted I/O

With formatted I/O, we specify how the output is to appear: the spacing, number of digits displayed, etc.

a. Syntax of a FORMAT statement

sl format(ccc,specifier1,specifier2,…)

The sl is the statement label that identifies the format statement. The ccc stands for the *carriage control character*. The *specifier* (also known as an *edit descriptor*) is a code that specifies how a value is to be printed. There must be one specifier for every variable or constant in the print list.

b. Formatted output—using the edit descriptors

```
        PRINT sl, var1,var2,var3,…
 sl     FORMAT(1x,spec1,spec2,spec3,…)
```

The specifier or descriptor must match the data type of the variable in the order that the variables are listed, otherwise gibberish will be printed out.

If the field width (w) is not large enough, then a string of asterisks (*) are printed. It's advisable to use E-format for all real variables when the program is being developed.

If the list of specifiers is shorter than the print list then the computer starts over at the beginning of the format list.

A FORMAT statement may be placed anywhere in the program module. Any number of output (PRINT or WRITE) statements may use the same FORMAT statement.

Formatted input can be used also, but why bother?

c. Edit descriptors

Each printed value is said to occupy a certain *field*, that is, a certain number of columns. In the following table, w = the width of the field and d = the number of digits to display.

edit descriptor	description
Iw	integer value
Fw.d	real value in decimal form
Ew.d	real value in exponential form
Dw.d	double precision value in exponential form
Gw.d	real value in "general purpose" form
Aw	character value
rx	an r number of blank spaces
Tc	tab to column c
TRs	tab right by as s number of spaces
TLs	tab left by an s number of spaces
/	start a new line or record
r( )	repeat ( ) r times
' text '	character strings

3. File I/O [Chapter 9]

a. Opening and Closing units

OPEN(n,file='filename') and CLOSE(n)

In this context, the word unit refers to *I/O unit* or *device*. An I/O device might be the monitor, the keyboard, a disk file, a punched card reader, a punched card puncher, a teletypewriter, a line printer, a computer port, and so on. Most commonly, it'll be the monitor, keyboard or a disk file. Each device has to be given a unit number (n) and a name (filename). That is the purpose of the OPEN statement.

b. Reading

Each READ statement starts with the next new record or line. There is free format reading

READ(n,*) var1,var2,var3,...

and formatted reading

READ(n,sl) var1,var2,var3,…

We don't usually bother with formatted input. However, some commercially produced programs require formatted input.

c. Writing

Each WRITE statement starts a new record. Again, there is free format writing

WRITE(n,*) var1,var2,var3,…

and formatted writing

WRITE(n,sl) var1,var2,var3,…

sl FROMAT(1x,……)

In contrast to reading, we normally do use formatted writing so that output is displayed in an attractive and legible form.

d. Data file issues

i) sequential vs. direct access

Most often input files are read line by line from the top to the bottom. This is referred to as *sequential access*. The program cannot go back and forth within the data file. In a *direct access* file, specified records are accessed in any order, usually identified by a record number. Direct access is also known as *random access*.

ii) open statement parameters

There are some additional parameters that may be used in an OPEN statement.

ERR=sl	transfers control to statement sl if an I/O error occurs
IOSTAT=integer variable name	stores the value of the error code IOSTAT
ACCESS	'sequential' or 'random access'

iii) read statement parameters

There are some additional parameters that may be used in a READ statement.

ERR=sl1	transfers control to statement sl1 is a read error occurs.
END=sl2	transfers control if the end of the data file is encountered.

The END parameter is particularly useful when reading a data file whose length is unknown. If the END parameter is not present, the program will stop if an *end of file* is encountered.

iv) data files

Data files are plain text files. So, for instance, if you use a word processor to create an input file, be sure to save it as plain text. Likewise, output files can be subsequently edited with a plain text editor, such as NotePad. Of course, plain text editors prefer to attach the .txt extension. A data file can have any 3-letter extension you please. A Fortran source file, which is also a plain text file, must have the .for extension.

D. Functions and Subprograms [pages 55 –57 & Chapter 7]

In Fortran, program modules are called *functions* and *subprograms.* There are several types of program modules.

1. Functions

a. Intrinsic functions

A mathematical function, such as e^x, is evaluated by summing a series. One could write ones own subprogram to add up the series expansions of e^x, $\sin(x)$, $\ln(x)$ or $\sqrt{x}$, etc. However, some common functions are already done in Fortran. Those are the *intrinsic* or built-in functions such as SQRT(x), EXP(x), SIN(x) and so on.

b. Statement function

A *statement function* is a one-line subprogram defined by the programmer. It's a non-executable statement, so it must appear at the top of the program module, before the first executable statement and following the DIMENSION, COMMON, DATA, and DECLARATION statements. A statement function must have at least one dummy argument. It may have several.

```
FUNC(X) = 37.0*X + TAN(X)
```

Later in the program, the function is invoked just like an intrinsic function, thusly Z = FUNC(B). The function may have any name not being used as a variable name. If a statement function is given the same name as an intrinsic function, it will supercede the intrinsic function.

c. Function subprogram

A *function subprogram* is a multiline user-defined function. The function subprogram is self-contained in that it must have its own type declaration statements, its own dimensioning statements, its own data statements, and so on. Rather than a STOP statement, the function subprogram must have at least one RETURN statement, which has the effect of returning control to the program module calling the function. There may be STOP statements in a function subprogram. The function subprogram returns to the calling module a single value that is stored in a variable name of the appropriate data type. The name of that variable must be the same as the name of the function. The function subprogram must have at least one dummy argument. However, that dummy argument need not actually be used to pass data to the function. In the calling module, the function subprogram is invoked in the same manner as an intrinsic function.

d. Subroutine

The *subroutine* is really a complete independent program. It may have STOP statements, but like the function subprogram it must have at least one RETURN statement. A subroutine may return to the calling module any number of values, not just a single one. It may return no values at all, but simply carry out some task such as printing output. Information may be conveyed to the subroutine through an argument list and/or through COMMON statements. The subroutine may have no arguments at all. A subroutine is invoked by the CALL statement.

Fortran sidelight #0

Statement labels and GoTo statements and DO statements [page 70s; 101 – 103; Chapter 5]

There are no line numbers in Fortran. Any program statement may be given a *statement label* or a *statement number*. The statement label is used to refer to a program statement within the program. Statement labels must be unique and must appear in the first 5 columns of the line. However, they need not be in any particular order.

A *GoTo* statement is an *unconditional transfer of control* as in *GoTo 304*, which means that the statement labeled 304 will be executed next, no matter what. The GoTo statement must include a statement label, pointing to an executable line which appears in the program.

A DO statement begins a Do Loop. There are two forms of Do Loop. One makes use of a statement label to define the end of the code to be iterated, the other form uses the End Do statement for the same purpose. For example

```
      Do 400 i = 1,10                Do i = 1,10
      .                              .
      .                              .
      .                              .
400   Continue                       EndDo
```

It is permitted to transfer out of a Do Loop, but not into one. Do Loops can be nested.

III. Numerical Solution of Nonlinear Equations

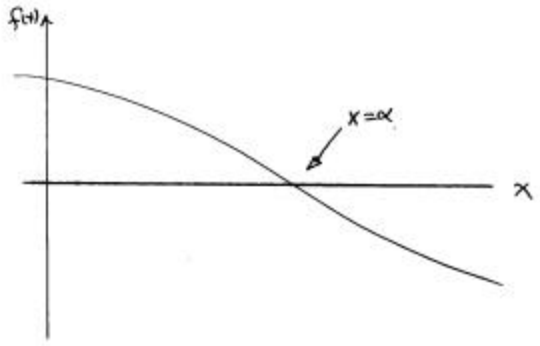
A. Non-Linear Equations—one at a time

There are closed form solutions for quadratic and even 3rd degree polynomial equations. Higher degree polynomials can sometimes be factored. However, in general there is no closed form analytical solution to non-linear equations.

1. The Problem

a. Roots & zeroes

We seek to find x such that $f(x) = 0$ or perhaps such that $f(x) = g(x)$. In the latter case, we merely set $h(x) = f(x) - g(x) = 0$. We are looking for a *root* of the equation $f(x) = 0$ or a *zero* of the function $f(x)$.



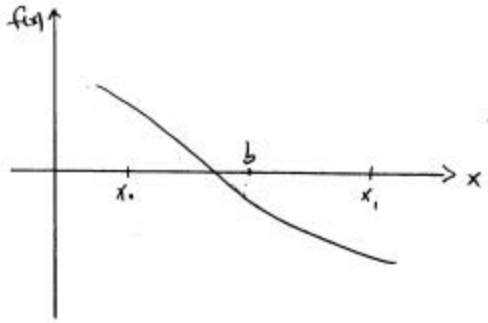
b. Graphical solution

Plot $f(x)$ vs. x—observe where the graph crosses the x-axis or plot $f(x)$ and $g(x)$ vs. x and observe where the two curves intersect. A graph won't give a precise root, but we can use the graph to choose an initial estimate of the root.

2. Bisection

a. Setup

For brevity, say $f_o = f(x_o)$ and $f_1 = f(x_1)$, etc. Say further that $x = a$ is the desired root. The graph shows us that $f_o \cdot f_1 < 0$ because $f(x)$ crosses the x-axis between $[x_o, x_1]$.



b. Algorithm

Let us find the midpoint of $[x_o, x_1]$, and call it b.

i) $b = \dfrac{x_o + x_1}{2}$ and then $f_b = f(b)$

ii) Does $f_b \cong 0$? If so, quit 'cause $a \cong b$.

iii) If not, then

if $f_o \cdot f_b < 0$, then set $x_o = b$ and $f_o = f_b$

or

if $f_b \cdot f_1 < 0$, then set instead $x_1 = b$ and $f_1 = f_b$.

iv) Is $|x_1 - x_o| \le e$? If so, quit and set $a = \dfrac{x_o + x_1}{2}$.

v) If not, then repeat beginning with step (i).

It is well also to count the iterations and to place a limit on the number of iterations that will be performed. Otherwise, the program could be trapped in an infinite loop. Also, it is well to test for the cases $f_o \cdot f_b > 0$ and $f_1 \cdot f_b > 0$. It may be that the function does not cross the x-axis between f_o and f_1, or crosses more than once.

3. Newton's Method or the Newton-Raphson Method

a. Taylor's series

Any well-behaved function can be expanded in a Taylor's series:

$$f(x) = f(x_o) + (x - x_o) f'(x_o) + (x - x_o)^2 \frac{f''(x_o)}{2!} + (x - x_o)^3 \frac{f'''(x_o)}{3!} + \cdots.$$

Let's say that x is "close" to x_o and keep just the first two terms.

$$f(x) \approx f(x_o) + (x - x_o) f'(x_o)$$

We want to solve for x such that $f(x) = 0$.

$$f(x_o) + (x - x_o) f'(x_o) = 0$$

$$x = x_o - \frac{f(x_o)}{f'(x_o)}$$

In effect we have approximated $f(x)$ by a straight line; x is the intercept of that line with the x-axis. It may or may not be a good approximation for the root a.

b. Algorithm

i) choose an initial estimate, x_i

ii) compute $f(x_i)$ and $f'(x_i)$

iii) compute the new estimate: $x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$

iv) return to step (ii) with $i = i + 1$

c. Comments

It turns out that if the initial estimate of the root is a good one, then the method is guaranteed to converge, and rapidly. Even if the estimate is not so good, the method will converge to a root—maybe not the one we anticipated.

Also, if there is a $f' = 0$ point nearby the method can have trouble. It's always a good thing to graph $f(x)$ first.

4. Secant Method

a. Finite differences

A *finite difference* is merely the difference between two numerical values.

$$\Delta x = x_2 - x_1 \text{ or } \Delta x = x_{i+1} - x_i$$

Derivatives are approximated by *divided differences*.

$$f'(x) \cong \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = \frac{\Delta f}{\Delta x}$$

We may regard this divided difference as an estimate of f' at x_i or at x_{i+1} or at the midpoint between x_i and x_{i+1}.

b. The Secant method

We simply replace f' by the divided difference in the Newton-Raphson formula:

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}.$$

Notice the indices: $i + 1$, i, $i - 1$. With the Secant Method, we don't use a functional form for f'. We do have to carry along two values of f, however.

Care must be taken that $|f(x_i) - f(x_{i-1})|$ not be too small, which would cause an overflow error by the computer. This may occur if $f(x_i) \approx f(x_{i-1})$ due to the finite precision of the machine. This may also give a misleading result for the convergence test of $|f(x_i) - f(x_{i-1})|$. To avoid that, we might use the *relative deviation* to test for convergence.

$$\frac{|f(x_i) - f(x_{i-1})|}{|f(x_i)|} \le e$$

c. Compare and contrast

Both the Newton-Raphson and Secant Methods locate just one root at a time.

Newton: requires evaluation of f and of f' at each step; converges rapidly.

Secant: requires evaluation only of f at each step; converges less rapidly.

5. Hybrid Methods

A hybrid method combines the use in one program of two or more specific methods. For instance, we might use bisection to locate a root roughly, then use the Secant Method to compute the root more precisely. For instance, we might use bisection to locate multiple roots of an equation, then use Newton-Raphson to refine each one.

B. Systems of Nonlinear Equations

Consider a system of *n* nonlinear equations with *n* unknowns.

$$f_1(x_1, x_2, x_3, \ldots, x_n) = 0$$
$$f_2(x_1, x_2, x_3, \ldots, x_n) = 0$$
$$\vdots$$
$$f_n(x_1, x_2, x_3, \ldots, x_n) = 0$$

1. Newton-Raphson

a. Matrix notation

Let's write the system of equations as a matrix equation.

$$\bar{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} = 0$$

The unknowns form a column matrix also. $\bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$. We might write the system of equations compactly as $\bar{f}(\bar{x}) = 0$.

b. The Method

The Newton-Raphson method for simultaneous equations involves evaluating the derivative matrix, $\overline{F}$, whose elements are defined to be $F_{ij} = \dfrac{\partial f_i}{\partial x_j}$. If the inverse $\overline{F}^{-1}$ exists, then we can generate a sequence of approximations for the roots of functions $\{f_i\}$.

$$\bar{x}_{k+1} = \bar{x}_k - \overline{F}^{-1}(\bar{x}_k) \cdot \bar{f}(\bar{x}_k)$$

At each step, all the partial derivatives must be evaluated and the $\overline{F}$ matrix inverted. The iteration continues until <u>all</u> the $f_i \cong 0$. If the inverse matrix does not exist, then the method fails. If the number of equations, *n*, is more than a handful, the method becomes very cumbersome and time consuming.

2. Implicit Iterative Methods

The Newton-Raphson method is an iterative method in the sense that it generates a sequence of successive approximations by repeating, or iterating, the same formula. However, the term *iterative method* as commonly used refers to a particular class of algorithms which might more descriptively be called *implicit iterative methods*. Such algorithms occur in many numerical contexts as we'll see in subsequent sections of this course. At this point, we apply the approach to the system of simultaneous nonlinear equations.

a. General form

Let $\bar{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ be the solution matrix to the equation $\bar{f}(\bar{x}) = 0$. I.e., $\bar{f}(\bar{a}) = 0$. Now, solve algebraically each $f_i(\bar{x}) = 0$ for x_i. This creates a new set of equations, $x_i = F_i(\bar{x}')$, where $\bar{x}'$ refers to the set of unknowns $\{x_j\}$ excluding x_i. Algebraically, this looks funny, because each unknown is expressed in terms of all the other unknowns, hence the term *implicit*. Of course, what we really mean is

$$\bar{x}_{k+1} = \bar{F}(\bar{x}_k) .$$

Alternatively, in terms of matrix elements, the equations take the form

$$x_{i,k+1} = F_i(x_{1,k}, x_{2,k}, \ldots x_{n,k}) .$$

b. Algorithm

In a program, the iterative method is implemented thusly:

i) choose an initial guess, $\bar{x}_o$

ii) compute $\bar{x}_1 = \bar{F}(\bar{x}_o)$

iii) test $\bar{f}(\bar{x}_1) \cong 0$

iv) if yes, set $\bar{a} = \bar{x}_1$ and exit

v) if no, compute $\bar{x}_2 = \bar{F}(\bar{x}_1)$, etc.

c. Convergence

We hope that $\lim_{k\to\infty} \bar{x}_k = \bar{a}$. For what conditions will this be true? Consider a region R in the space of $\{x_i\}$ such that $|x_j - a_j| \le h$ for $1 \le j \le n$ and suppose that for $\bar{x}$ in R there is a positive number m such that $\sum_{j=1}^{n} \left| \frac{\partial F_i(\bar{x})}{\partial x_j} \right| \le m$. Then, it "can be shown" that if $\bar{x}_o$ lies in R, the iterative method will converge. What does this mean, practically? It means that if the initial guess, $\bar{x}_o$, is "close enough" to $\bar{a}$, then the method will converge to $\bar{a}$ after some number, k, of iterations. Big deal.

Fortran Sidelight #1

IF statements [Chapter 4]

There are two varieties of IF statements. The one liner, and the IF-THEN-ELSE block. In both cases, a decision on what action to take next is made on the basis of some logical relation.

A logical relation is a statement which may be true or false. Logical or relational operators [.OR. .AND. .LE. .LT. .GE. .GT. .NE. .EQ.] are used to form logical relations. For instance, the statement *sqrt(x).eq.3* is a logical relation. If the square root of *x* is 3, then the relation has the logical value .TRUE. If *x* = 16, though, then the relation has the logical value .FALSE.

One-liners

IF(*logical relation*) *action statement*

If the *logical relation* in the parentheses is .TRUE. then the *action statement* is executed. If the *logical relation* is .FALSE. the *action statement* is not executed. In either case, execution continues with the line following the IF statement, unless the *action statement*, when executed, redirects program control to another statement. In fact, such redirection of control is a common use of a one liner IF. The *action statement* may be any Fortran executable statement such as assignment, I/O, or GoTo but not a DO statement or another IF statement.

IF-THEN-ELSE block

The one liner is limited to a single action statement when the logical relation is .TRUE. The IF-THEN-ELSE construction allows more flexibility is setting up alternative blocks of program statements.

```
IF( logical relation ) THEN
        { block A }
ELSE
        { block B }
END IF
```

In this case, if the *logical relation* is .TRUE., then the program code Block A is executed. If the *logical relation* is .FALSE., then the code Block B is executed. Each code block may be a single executable statements or many executable statements. Once Block A or B is executed, the program continues with the statement following the END IF statement.

Additional logical branches may be nested within the IF-THEN-ELSE. Thus:

IF(*logical relation* 1) THEN
    { *block* A }
ELSE IF(*logical relation* 2) THEN
    { *block* B }
ELSE IF(*logical relation* 3 } THEN
    { *block* C }
ELSE
    { *block* D }
END IF

Obviously, it's easy to become tangled up in these nested ELSEs.

IV. Linear Algebra

A. Matrix Arithmetic

The use of matrix notation to represent a system of simultaneous equations was introduced in section III-B-1 above, mainly for the sake of brevity. In solving simultaneous linear equations, matrix operations are central. There follows, therefore, a brief review of the salient properties of matrices. Fuller discussion of the properties of matrices may be found in various texts, particularly Linear Algebra texts.

1. Matrices

A matrix is an n x m array of numbers. In these notes a matrix is symbolized by a letter with a line on top, $\overline{B}$; n is the number of rows and m is the number of columns. If $n = m$, the matrix is said to be a *square matrix*. If the matrix has only one column(row) it is said to be a *column(row) matrix*. The jth *element* in the ith row of a matrix is indicated by subscripts, b_{ij}. Mathematically, an entity like a matrix is defined by a list of properties and operations, for instance the rules for adding or multiplying two matrices. Also, matrices can be regarded as one way to represent members of a group in Group Theory.

$$\overline{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} \qquad \overline{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

2. Addition & Subtraction

a. Definition

The addition is carried out by adding the respective matrix elements.

$$\overline{C} = \overline{A} + \overline{B}$$
$$c_{ij} = a_{ij} + b_{ij}$$

b. Rules

The sum of two matrices is also a matrix. Only matrices having the same number of rows and the same number of columns may be added. Matrix addition is commutative and associative.

$$\overline{A} + \overline{B} = \overline{B} + \overline{A} \qquad (\overline{A} + \overline{B}) + \overline{C} = \overline{A} + (\overline{B} + \overline{C})$$

3. Multiplication

a. Definition

$$\overline{C} = \overline{A}\,\overline{B}$$
$$c_{ij} = \sum_k a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + a_{i3} b_{3j} + \cdots$$

b. Rules

The product of two matrices is also a matrix. The number of elements in a <u>row</u> of $\overline{A}$ must equal the number of elements in a <u>column</u> of $\overline{B}$. Matrix multiplication is <u>not</u> commutative.

$$\overline{A}\,\overline{B} \neq \overline{B}\,\overline{A}$$

A matrix may be multiplied by a constant, thusly: $c_{ij} = q \cdot a_{ij}$. The result is also a matrix.

4. Inverse Matrix

a. Unit matrix

The *unit matrix* is a square matrix with the diagonal elements equal to one and the off-diagonal elements all equal to zero. Here's a 3x3 unit matrix:

$$\overline{U} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

b. Inverse

The *inverse* of a matrix, $\overline{B}$, (denoted $\overline{B}^{-1}$) is a matrix such that $\overline{B}\,\overline{B}^{-1} = \overline{B}^{-1}\overline{B} = \overline{U}$. The inverse of a particular matrix may not exist, in which case the matrix is said to be singular.

The solution of a system of simultaneous equations in effect is a problem of evaluating the inverse of a square matrix.

Fortran Sidelight #2

Dimensions, Arrays, and Matrices [Chapter 6; pages 223 – 224]

In Fortran, a matrix is called a *dimensioned variable*, or an *array*, or a *subscripted variable*. The DIMENSION statement specifies the size of an array and the number of indices or subscripts.

Consider a two dimensional matrix, $\overline{B}$. An element of that matrix might be written as b_{ij}. In the Fortran code this becomes *b(i,j).* An element of a one-dimensional matrix $\overline{x}$ (either a *row* or *column matrix*) is represented by *x(i)*.

For instance, the following DIMENSION statement sets up one square matrix and two column matrices:

```
DIMENSION b(20,20), x(20), c(13)
```

Notice that the array names are delimited by commas. The numbers in the parentheses are upper limits on the ranges of the indices. Therefore, both the indices of the variable *b* range from 1 to 20. The index of variable *c* ranges from 1 to 13. Later versions of Fortran allow array indices to be negative or zero: DIMENSION b(0:19,0:19), x(-3,16). Referring to an index value outside the range specified in the DIMENSION statement can lead to "unpredictable results."

The DIMENSION statement appears at the top of a program module, preceding any executable statements. There may be more than one DIMENSION statement. The array names may be listed in any order. Dimensioning is not global, so any program module that uses array variables must have its own DIMENSION statement(s).

In Fortran, the addition or multiplication of matrices must be spelled out with DO loops. In other words, there are no array operations.

Multiply a column matrix by a square matrix

```
      DO 200 i=1,20
      c(i) = 0.0
      DO 100 j=1,20
  100 c(i) = c(i) + b(i,j)*x(j)
  200 CONTINUE
```

Multiply two square *n* x *n* matrices

```
      DO 300 i=1,n
      DO 300 j=1,n
      d(i,j) = 0.0
      DO 200 k=1,n
  200 d(i,j) = d(i,j) + a(i,k)*b(k,j)
  300 CONTINUE
```

Multiply a column matrix by a constant

```
      DO 100 j=1,n
  100 d(j) = que*d(j)
```

B. Simultaneous Linear Equations

1. The Problem

a. Simultaneous equations

We wish to solve a system of n linear equations in n unknowns.

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + \cdots b_{1n}x_n &= c_1 \\ b_{21}x_1 + b_{22}x_2 + \cdots b_{2n}x_n &= c_2 \\ &\vdots \\ b_{n1}x_1 + b_{n2}x_2 + \cdots b_{nn}x_n &= c_n \end{aligned}$$

where the $\{b_{ij}\}$ and the $\{c_i\}$ are constants.

b. Matrix notation

The system of equations can be written as a matrix multiplication.

$$\overline{B}\overline{x} = \overline{c}\text{ , where }\overline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\ \overline{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}\text{ and }\overline{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}.$$

When n is small ($n \le 40$, say) a direct or *one-step* method is used. For larger systems, iterative methods are preferred.

2. Gaussian Elimination

In a one-step approach, we seek to evaluate the inverse of the $\overline{B}$ matrix.

$$\overline{B}\overline{x} = \overline{c}$$
$$\overline{B}^{-1}\overline{B}\overline{x} = \overline{x} = \overline{B}^{-1}\overline{c}$$

The solution is obtained by carrying out the matrix multiplication $\overline{B}^{-1}\overline{c}$.

a. Elimination

You may have seen this in high school algebra. For brevity's sake, let's let n = 3.

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 &= c_1 \\ b_{21}x_1 + b_{22}x_2 + b_{23}x_3 &= c_2 \\ b_{31}x_1 + b_{32}x_2 + b_{33}x_3 &= c_3 \end{aligned}$$

In essence, we wish to eliminate unknowns from the equations by a sequence of algebraic steps.

normalization i) multiply eqn. 1 by $-\dfrac{b_{21}}{b_{11}}$ and add to eqn. 2; replace eqn. 2.

reduction ii) multiply eqn 1 by $-\dfrac{b_{31}}{b_{11}}$ and add to eqn. 3; replace eqn. 3.

$$b_{11}x_1 + b_{12}x_2 + b_{13}x_3 = c_1$$
$$b'_{22}x_2 + b'_{23}x_3 = c'_2$$
$$b'_{32}x_2 + b'_{33}x_3 = c'_3$$

iii) multiply eqn. 2 by $-\frac{b'_{32}}{b'_{22}}$ and add to eqn. 3; replace eqn. 3.

$$b_{11}x_1 + b_{12}x_2 + b_{13}x_3 = c_1$$
$$b'_{22}x_2 + b'_{23}x_3 = c'_2$$
$$b''_{33}x_3 = c''_3$$

We have eliminated x_1 and x_2 from eqn.3 and x_1 from eqn. 2.

back substitution iv) solve eqn. 3 for x_3, substitute in eqn. 2 & 1.
solve eqn. 2 for x_2, substitute in eqn. 1.
solve eqn. 1 for x_1.

b. Pivoting

Due to the finite number of digits carried along by the machine, we have to worry about the relative magnitudes of the matrix elements, especially the diagonal elements. In other words, the inverse matrix, $\overline{B}^{-1}$ may be effectively singular even if not actually so. To minimize this possibility, we commonly rearrange the set of equations to place the largest coefficients on the diagonal, to the extent possible. This process is called *pivoting*.

e.g.

$$37x_2 - 3x_3 = 4$$
$$19x_1 - 2x_2 + 48x_3 = 99$$
$$7x_1 + 0.6x_2 + 15x_3 = -9$$

rearrange

$$19x_1 - 2x_2 + 48x_3 = 99$$
$$37x_2 - 3x_3 = 4$$
$$7x_1 + 0.6x_2 + 15x_3 = -9$$

or

$$7x_1 + 0.6x_2 + 15x_3 = -9$$
$$37x_2 - 3x_3 = 4$$
$$19x_1 - 2x_2 + 48x_3 = 99$$

3. Matrix Operations

In preparation for writing a computer program, we'll cast the elimination and back substitution in the form of matrix multiplications.

a. Augmented matrix

$$\overline{A} = \left[\overline{B} : \overline{c}\right] = \begin{bmatrix} b_{11} & b_{12} & b_{13} & c_1 \\ b_{21} & b_{22} & b_{23} & c_2 \\ b_{31} & b_{32} & b_{33} & c_3 \end{bmatrix}$$

b. Elementary matrices

Each single step is represented by a single matrix multiplication.

The elimination steps:

$$\overline{S}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{b_{21}}{b_{11}} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \overline{S}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{b_{31}}{b_{11}} & 0 & 1 \end{bmatrix} \qquad \overline{S}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{b'_{32}}{b'_{22}} & 1 \end{bmatrix}$$

$$\overline{S}_3\overline{S}_2\overline{S}_1\overline{A} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & c_1 \\ 0 & b'_{22} & b'_{23} & c'_2 \\ 0 & 0 & b''_{33} & c''_3 \end{bmatrix}$$

The first back substitution step:

$$\overline{Q}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{b''_{33}} \end{bmatrix}$$

$$\overline{Q}_1\overline{S}_3\overline{S}_2\overline{S}_1\overline{A} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & c_1 \\ 0 & b'_{22} & b'_{23} & c'_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

This completes one cycle. Next we eliminate one unknown from the second row using

$$\overline{S}_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -b'_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\overline{S}_4\overline{Q}_1\overline{S}_3\overline{S}_2\overline{S}_1\overline{A} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & c_1 \\ 0 & b''_{22} & 0 & c''_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

$$\overline{Q}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{b''_{22}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\overline{Q}_2\overline{S}_4\overline{Q}_1\overline{S}_3\overline{S}_2\overline{S}_1\overline{A} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & c_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

This completes the second cycle. The final cycle is

$$\overline{S}_5 = \begin{bmatrix} 1 & 0 & -b_{13} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \overline{S}_6 = \begin{bmatrix} 1 & -b_{12} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \overline{Q}_3 = \begin{bmatrix} \frac{1}{b_{11}} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\overline{Q}_3\overline{S}_6\overline{S}_5\overline{Q}_2\overline{S}_4\overline{Q}_1\overline{S}_3\overline{S}_2\overline{S}_1 = \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

We identify the inverse matrix $\overline{B}^{-1} = \overline{Q}_3\overline{S}_6\overline{S}_5\overline{Q}_2\overline{S}_4\overline{Q}_1\overline{S}_3\overline{S}_2\overline{S}_1$. Notice that the order of the matrix multiplications is significant. Naturally, we want to automate this process, and generalize to *n* equations.

4. Gauss-Jordan Elimination

a. Inverse matrix

We might multiply all the elementary matrices together before multiplying by the augmented matrix. That is, carry out the evaluation of $\overline{B}^{-1}$, then perform $\overline{B}^{-1}\overline{A}$.

b. Algorithm

$$\left.\begin{array}{ll} a_{kj}^{k} = \dfrac{a_{kj}^{k-1}}{a_{kk}^{k-1}} & i = k \\ a_{ij}^{k} = a_{ij}^{k-1} - a_{ik}^{k-1} \cdot a_{kj}^{k} & i \neq k \end{array}\right\} \begin{array}{l} k = 1, n \\ i = 1, n \\ j = k, n+1 \end{array}$$

n = number of equations
k = index of the step or cycle
a_{ij} = elements of the original augmented matrix, $\overline{A}$.

For each value of k, do the $i = k$ line first.

c. Example

$n = 3$ and $n + 1 = 4$

$$4x_1 + x_2 + 2x_3 = 16$$
$$x_1 + 3x_2 + x_3 = 10$$
$$x_1 + 2x_2 + 5x_3 = 12$$

$$k = 0 \quad \overline{A} = \begin{bmatrix} 4 & 1 & 2 & 16 \\ 1 & 3 & 1 & 10 \\ 1 & 2 & 5 & 12 \end{bmatrix}$$

e.g., for $k = 1$, $i = 1$, $j = 1$ & $j = 4$

$$a_{11}^1 = \frac{a_{11}^0}{a_{11}^0} = 1 \qquad a_{14}^1 = \frac{a_{14}^0}{a_{11}^0} = \frac{16}{4} = 4$$

$$a_{21}^1 = a_{21}^0 - a_{21}^0 a_{11}^1 = 1 - 1 \cdot 1 = 0$$

$$k = 1 \quad \overline{A}' = \begin{bmatrix} 1 & \frac{1}{4} & \frac{1}{2} & 4 \\ 0 & \frac{11}{4} & \frac{1}{2} & 6 \\ 0 & \frac{7}{4} & \frac{9}{2} & 8 \end{bmatrix}$$

$$k = 2 \quad \overline{A}'' = \begin{bmatrix} 1 & 0 & \frac{5}{11} & \frac{38}{11} \\ 0 & 1 & \frac{2}{11} & \frac{24}{11} \\ 0 & 0 & \frac{46}{11} & \frac{46}{11} \end{bmatrix}$$

$$k = 3 \quad \overline{A}''' = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\overline{x} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

C. Iterative Methods

For n > about 40, the one-step methods take too long and accumulate too much round-off error.

1. Jacobi Method

a. Recursion formula

Each equation is solved for one of the unknowns.

$$x_1 = (c_1 - b_{12}x_2 - b_{12}x_3 - \cdots - b_{1n}x_n)\frac{1}{b_{11}}$$
$$x_2 = (c_2 - b_{21}x_1 - b_{23}x_3 - \cdots - b_{2n}x_n)\frac{1}{b_{22}}$$
$$\vdots$$
$$x_n = (c_n - b_{n1}x_1 - b_{n2}x_2 - \cdots - b_{nn-1}x_{n-1})\frac{1}{b_{nn}}$$

In short $x_i = \left(c_i - \sum_{\substack{j=1 \\ i \neq j}}^{n} b_{ij}x_j \right)\frac{1}{b_{ii}}$, i = 1, 2, 3, . . .,n.

Of course, we cannot have $b_{ii} = 0$ for any i. So before starting the iterative program, we may have to reorder the equations. Further, it can be shown that if $|b_{ii}| \geq b_{ij}$ for each i, then the method will converge, though it may be slowly. Here's an outline of the "showing."

The first iteration is: $\bar{x}^1 = -\bar{A}\,\bar{x}^0 + \bar{V}$

After several iterations, $\bar{x}^{k+1} = -\bar{A}\bar{x}^k + \bar{V}^k = -\underset{k+1}{\bar{A}}\cdots\underset{3}{\bar{A}}\,\underset{2}{\bar{A}}\,\underset{1}{\bar{A}}\,\bar{x}^0 + \underset{k+1}{\bar{A}}\cdots\underset{3}{\bar{A}}\,\underset{2}{\bar{A}}\bar{V} = \bar{A}^{k+1}\bar{x}^0 + \bar{A}^k\bar{V}$

We want $\lim_{k\to\infty} \bar{A}^{k+1}\bar{x}^0 = 0$, which will happen if $\frac{b_{ij}}{b_{ii}} \leq 1$.

b. Algorithm

We need four arrays: $\bar{x}^k$, $\bar{x}^{k+1}$, $\bar{B}$, and $\bar{c}$.

Firstly, select an initial guess (k = 0) $\bar{x}^0 = \begin{bmatrix} x_1^0 \\ x_2^0 \\ \vdots \\ x_n^0 \end{bmatrix}$.

Secondly, compute a new $\bar{x}$ (k + 1 = 1).

$$x_i^{k+1} = \left(c_i - \sum_{\substack{j=1 \\ i \neq j}}^{n} b_{ij}x_j^k \right)\frac{1}{b_{ii}}$$

Thirdly, test for convergence. $\frac{\left|x_i^{k+1} - x_i^k\right|}{\left|x_i^k\right|} \le e$. Notice that all the x_i must pass the test.

If all the x_i do not pass the test, then repeat until they do.

c. FORTRAN

Four arrays: xold(n), xnew(n), B(n,n), c(n), where n is the number of simultaneous equations.

Read the equations and initial guess

```
      read *,n
      do 97 i=1,n
 97   read*,(b(i,j),j=1,n),c(i)
      read *,(xold(i),i=1,n)
 98   do 99 i=1,n
 99   xnew(i) = xold(i)
```

Compute the new approximation

```
      do 100 i=1,n
      sum = 0.0
      do 50 j=1,n
      if( j.eq.i ) goto 50
      sum = sum +b(i,j)*xold(j)
 50   continue
      xnew(i) = ( c(i) – sum )/b(i,i)
100   continue
```

Test for convergence

```
      do 200 i=1,n
      if( abs(xnew(i)-xold(i))/abs(xold(i)) .gt. eps ) goto 98
200   continue
```

Put out the result

```
      print *, xnew
      stop
      end
```

2. Gauss-Seidel Method

The Gauss-Seidel Method hopes to speed up the convergence by using newly computed values of x_i at once, as soon as each is available. Thus, in computing xnew(12), for instance, the values

of xnew(1), xnew(2), . . ., xnew(11) are used on the right hand side of the formula. We still need to keep separate sets of xnew and xold in order to perform the convergence tests.

```
      read *,n
      do 97 i=1,n
 97   read*,(b(i,j),j=1,n),c(i)
      read *,(xold(i),i=1,n)
      do 99 i=1,n
 99   xnew(i) = xold(i)


 98   do 100 i=1,n
      sum = 0.0
      do 50 j=1,n
      if( j.eq.i ) goto 50
      sum = sum +b(i,j)*xnew(j)
 50   continue
      xnew(i) = ( c(i) – sum )/b(i,i)
100   continue

      do 200 i=1,n
      if( abs(xnew(i)-xold(i))/abs(xold(i)) .le. eps ) goto 200
      do 199 j=1,n
199   xold(j) = xnew(j)
      goto 98
200   continue
```

Fortran Sidelight #3

1. Subprograms [Chapter 7]

a. Functions

```
function name(argument list)
declaration statements
dimension/common statements
data statements
.
.
.
.
name =
return
end
```

A function subprogram is invoked just like the built-in or intrinsic functions.

```
x = name(argument list)
```

b. Subroutines

Subroutines are self-contained program modules.

```
subroutine name(argument list)
declarations statements
dimension/common statements
data statements
.
.
.
.
return
end
```

A subroutine is invoked by a CALL statement. call name(argument list)

2. Communication Among the Main and Subprograms

a. Argument lists [pages 219 – 221; 222]

Information is passed between program modules by argument lists. The variables in an argument list of a subprogram must match the argument list in the calling statement in number of variables and data types and in the order in which the variables are listed. However, the variables needn't have identical names in the separate modules.

b. Common blocks [pages 225 -228]

Unlike some other programming languages, Fortran variables are *local*—they exist only in the program modules in which they are declared or used. However, there is a way to create a list of global variables, the *common block*.

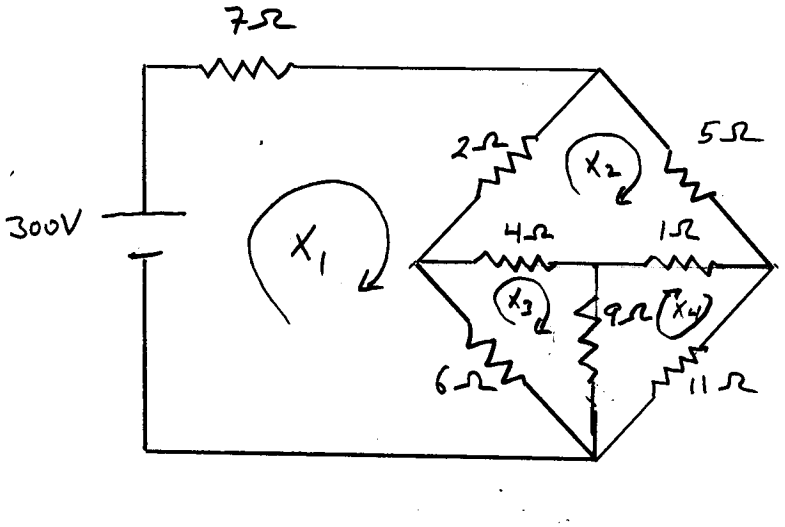
A common block is created by the COMMON statement: COMMON var1, var2, var3,. . ., varN. The variables in the common block will be available to all program modules that contain the COMMON statement. In other words, the COMMON statement must appear in every program module that needs access to those variables in the common block. On the other hand, the variable names in the COMMON statement need not be identical in the several program modules. The variables must be listed, though, in the same order and have the same data types, etc., in every occurrence of the COMMON statement. If a variable is passed to a subprogram (function or subroutine) via a COMMON statement, it is not also included in an argument list.

The dimensions of an array variable may be specified in a COMMON statement, in which case the same variable is not included in a DIMENSION statement.

D. Applications

A couple of cases in engineering that give rise to simultaneous linear equations.

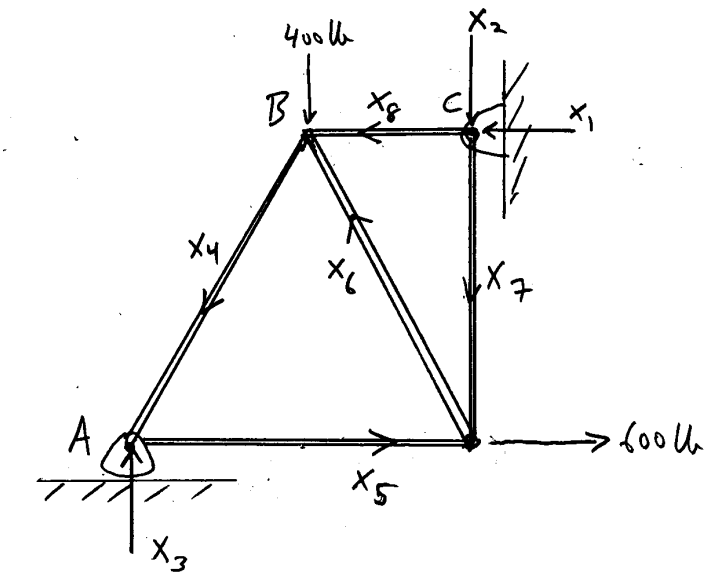
1. Electrical Circuit



$$(7+2+6)x_1 - 2x_2 - 6x_3 = 300$$
$$-2x_1 + (2+5+4+1)x_2 - 4x_3 - x_4 = 0$$
$$-6x_1 - 4x_2 + (4+9+6)x_3 - 9x_4 = 0$$
$$-x_2 - 9x_3 + (9+1+11)x_4 = 0$$

$$\overline{A} = \begin{bmatrix} 15 & -2 & -6 & 0 & 300 \\ -2 & 12 & -4 & -1 & 0 \\ -6 & -4 & 19 & -9 & 0 \\ 0 & -1 & -9 & 21 & 0 \end{bmatrix}; \quad \text{solution: } \overline{x} = \begin{bmatrix} 26.5 \\ 9.35 \\ 13.3 \\ 6.13 \end{bmatrix}$$

2. Truss System



$$\overline{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 600 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 3600 \\ 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 400 \\ 0 & 1 & 0 & -\frac{4}{5} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & \frac{3}{5} & 0 & 0 & 0 & -600 \\ 0 & 0 & 0 & 0 & \frac{4}{5} & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 600 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}; \quad \text{solution: } \overline{x} = \begin{bmatrix} 600 \\ -1000 \\ 600 \\ 1250 \\ 1083.33 \\ 866.67 \\ -600 \\ -600 \end{bmatrix}$$

V. Interpolation and Curve Fitting

Suppose one has a set of data pairs:

x	f
x_1	f_1
x_2	f_2
x_3	f_3
⋮	⋮
x_m	f_m

where f_i is the measured (or known) value of $f(x)$ at x_i. We would like to find a function that will approximate $f(x)$ for all x in a specified range. There are two basic approaches: interpolation and curve fitting.

A. Polynomial Interpolation

With *interpolation*, the approximating function passes through the data points. Commonly, the unknown $f(x)$ is approximated by a polynomial of degree n, $p_n(x)$, which is required to pass through all the data points, or a subset thereof.

1. Uniqueness

Theorem: Given $\{x_i\}$ and $\{f_i\}$, i = 1, 2, 3, . . ., n + 1, there exists one and only one polynomial of degree n or less which reproduces $f(x)$ exactly at the $\{x_i\}$.

Notes

i) There are many polynomials of degree > n which also reproduce the $\{f_i\}$.

ii) There is no guarantee that the polynomial $p_n(x)$ will accurately reproduce $f(x)$ for $x \neq x_i$. It will do so if $f(x)$ <u>is</u> a polynomial of degree n or less.

Proof: We require that $p_n(x) = f_i$ for all i = 1, 2, 3, . . ., n+1. This leads to a set of simultaneous linear equations

$$\begin{aligned} a_o + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n &= f_1 \\ a_o + a_1 x_2 + a_2 x_2^2 + \cdots + a_n x_2^n &= f_2 \\ &\vdots \\ a_o + a_1 x_{n+1} + a_2 x_{n+1}^2 + \cdots + a_n x_{n+1}^n &= f_{n+1} \end{aligned}$$

which we'd solve for the $\{a_i\}$. As long as no two of the $\{x_i\}$ are the same, the solution to such a set of simultaneous linear equations is unique.

The significance of uniqueness is that *no matter how* an interpolating polynomial is derived, as long as it passes through <u>all</u> the data points, it is <u>the</u> interpolating polynomial. There are many methods of deriving an interpolating polynomial. Here, we'll consider just one.

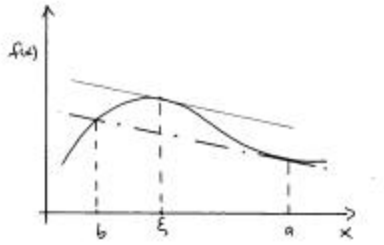
2. Newton's Divided Difference Interpolating Polynomial

a. Divided differences

The *first divided difference* is defined to be (notice the use of square brackets)

$$f[a,b] = \frac{f(a) - f(b)}{a - b}, \; a \neq b$$

If $f(x)$ is differentiable in the interval $[a,b]$, then there exists at least one point between a and b at which $\frac{df(x)}{dx} = f[a,b]$.



In practice, we would take a as close to b as we can (limited by the finite precision of the machine) and say that $f'(x) \approx f[a,b]$.

Higher order differences are defined as well:

order	**notation**	**definition**
0	$f[x_1]$	$f(x_1)$
1	$f[x_2, x_1]$	$\frac{f[x_2] - f[x_1]}{x_2 - x_1}$
2	$f[x_3, x_2, x_1]$	$\frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1}$
3	$f[x_4, x_3, x_2, x_1]$	$\frac{f[x_4, x_3, x_2] - f[x_3, x_2, x_1]}{x_4 - x_1}$
⋮	⋮	⋮
n	$f[x_{n+1}, x_n, \cdots, x_2, x_1]$	$\frac{f[x_{n+1}, x_n, \cdots, x_3, x_2] - f[x_n, x_{n-1}, \cdots, x_2, x_1]}{x_{n+1} - x_1}$

b. Newton's divided difference formula

Build the formula up step by step:

i) two data points (x_1,f_1) & (x_2,f_2). We wish to approximate $f(x)$ for $x_1 < x < x_2$.

As a first order approximation, we use a straight line ($p_1(x)$ so that

$$f[x, x_1] \cong f[x_2, x]$$

$$\frac{f(x) - f_1}{x - x_1} \cong \frac{f_2 - f(x)}{x_2 - x}$$

Solve for $f(x)$

$$f(x) \cong f_1 + (x - x_1) f[x_2, x_1] = p_1(x)$$

ii) Now, if $f(x)$ is a straight line, then $f(x) = p_1(x)$. If not, there is a remainder, R_1.

$$R_1(x) = f(x) - p_1(x) = f(x) - f_1 - (x - x_1) f[x_2, x_1] = (x - x_1)(x - x_2) f[x, x_2, x_1]$$

We don't know $f(x)$, so we cannot evaluate $f[x,x_2,x_1]$. However, if we had a third data point we could approximate $f[x,x_2,x_1] \cong f[x_3,x_2,x_1]$. Then we have a quadratic

$$f(x) \cong f_1 + (x-x_1) f[x_2,x_1] + (x-x_1)(x-x_2) f[x_3,x_2,x_1] = p_2(x).$$

iii) If $f(x)$ is not a quadratic polynomial, then there is still a remainder, R_2.

$$R_2(x) = f(x) - p_2(x)$$

To estimate R_2, we need a fourth data point and the next order divided difference. . .

$$f[x,x_3,x_2,x_1] \cong f[x_4,x_3,x_2,x_1]$$

iv) Jump to the generalization for $n + 1$ data points:

$$f(x) = p_n(x) + R_n(x), \text{ where}$$

$$p_n(x) = f[x_1] + (x-x_1) f[x_2,x_1] + (x-x_1)(x-x_2) f[x_3,x_2,x_1] + \cdots + (x-x_1)(x-x_2)(x-x_3)\cdots(x-x_n) f[x_{n+1},x_n,\cdots,x_2,x_1]$$

Notice that i) $p_3 = p_2 + (x-x_1)(x-x_2)(x-x_3) f[x_4,x_3,x_2,x_1]$, etc. and ii) the $(x - x_i)$ factors are also those of the previous term times one more factor.

c. Inverse interpolation

The NDDIP lends itself to inverse interpolation. That is, given $f(x)$, approximate x. In effect, we are solving $f(x) = 0$ when $f(x)$ is in the form of a table of data. Simply reverse the roles of the $\{f_i\}$ and the $\{x_i\}$.

$$x = p_n(f) = \sum_{i=2}^{n+1} f[f_1, f_2, \cdots, f_i] \prod_{j=1}^{i-1} \left(f(x) - f_j\right) + f[f_1]$$

Set $f(x) = 0$ and evaluate $x = p_n(0)$. In practice, with a Fortran program, one would just reverse the data columns and use the same code.

d. Example

The difference table is computed thusly:

```
      do 50 j=1,n+1
 50   diff(j,1) = f(j)
      do 200 j=2,n+1
      do 100 i=1,n+1-j+1
100   diff(i,j) = ( diff(i+1,j-1) – diff(i,j-1) )/( x(i+j-1) – x(i) )
200   continue
```

Divided Difference Table for $n = 6$

j	*x*	*f*	*f[ ]*	*f[,]*	*f[, ,]*	*f[, , , ,]*	*f[, , , , ,]*	*f[, , , , , ,]*
1	1	-1.5	0.5	1.667	-2.583	1.583	-0.727	0.27
2	2	-1	3	-3.5	2.167	-0.96	0.353	
3	2.5	0.5	-0.5	0.833	-0.233	0.1		
4	3	0.25	0.75	0.367	0.017			
5	4	1	1.3	0.4				
6	4.5	1.65	1.7					
7	5	2.5						

The sixth degree polynomial constructed from this table is

$$p_6(x) = f[x_1] + \sum_{i=2}^{7} f[x_1, x_2, \cdots, x_i] \prod_{j=1}^{i-1} (x - x_j).$$

Line by line, the Fortran might look like this:

```
fac = ex – x(1)
p0 = diff(1,1)
p1 = p0 + fac*diff(1,2)
fac = fac*(ex-x(2))
p2 = p1 + fac*diff(1,3)
fac = fac*(ex-x(3))
p3 = p2 + fac*diff(1,4)
fac = fac*(ex-x(4))
p4 = p3 + fac*diff(1,5))
fac = fac*(ex-x(5))
p5 = p4 + fac*diff(1,6)
fac = fac*(ex-x(6))
p6 = p5 + fac*diff(1,7)
```

Notice that we must use a different variable name for the argument x from the name used for the data array x(i).

Of course, it's more general and flexible to use a DO loop.

```
      fac = 1.0
      p = diff(1,1)
      do 400 j=1,n
      fac = fac*(ex-x(j))
400   p = p + fac*diff(1,j+1)
```

e. Issues with high degree polynomials

If we have a large number of data points, 20 or 100 or 1000s, it does not pay to use the entire data table to create a 20 or 100 or 1000th degree polynomial. The greater the degree, the more often the p_n goes up and down between the data points. Our confidence that $f(x) \cong p_n(x)$ actually decreases. It's better to interpolate on subsets of the data using a p_3 or a p_4 using data points that surround the specified x. This process can be incorporated into the program.

B. Least Squares Fitting

Often, there are errors or uncertainties in the data values, $10.07 \pm 0.005\,\text{sec}$, for instance. Perhaps forcing the approximating function to pass through the data points is not the wisest approach.

An alternative approach is to assume a functional form for the unknown $f(x)$ and adjust it to "best fit" the uncertain data. A way to judge what is "best" is needed.

1. Goodness of Fit

The *method of least squares* uses a particular measure of *goodness of fit*.

a. Total squared error, E

First of all, never forget that the word error in this context means uncertainty. Now, let's say $\{x_i, f_i\}$ are the n+1 data values and $f(x)$ is the assumed function. Then E is defined to be

$$E = \sum_{i=1}^{n+1} \frac{1}{\sigma_i^2} \left(f_i - f(x_i) \right)^2$$

The $\{\sigma_i\}$ are weighting factors that depend on the nature of the uncertainties in the data $\{f_i\}$. For measured values, the $\sigma_i = \Delta f_i$, the experimental uncertainties. Often, we just take all the $\sigma_i = 1$, perhaps implying that the experimental uncertainties are all the same.. In that case,

$$E = \sum_{i=1}^{n+1} \left(f_i - f(x_i) \right)^2 .$$

b. Least squares fit

We wish to derive an $f(x)$ which minimizes E. That means taking the derivative of E with respect to each adjustable parameter in $f(x)$ and setting it equal to zero. We obtain a set of simultaneous linear equations with the adjustable parameters as the unknowns. These are called the *normal equations*.

2. Least Squares Fit to a Polynomial

Assume that $f(x) = a + bx + cx^2 + dx^3$.

a. Total squared error

$$E = \sum_{i=1}^{n+1} \frac{1}{\sigma_i^2} \left(f_i - a - bx_i - cx_i^2 - dx_i^3 \right)^2$$

We have four adjustable parameters: a, b, c, and d. Notice that, unlike the interpolating polynomial, there may be any number of data pairs, regardless of the number of parameters. Let's take all the $\sigma_i = 1$.

The partial derivative with respect to the adjustable parameters are

$$\frac{\partial E}{\partial a} = -2\sum_i \left(f_i - a - bx_i - cx_i^2 - dx_i^3\right)$$
$$\frac{\partial E}{\partial b} = -2\sum_i x_i\left(f_i - a - bx_i - cx_i^2 - dx_i^3\right)$$
$$\frac{\partial E}{\partial c} = -2\sum x_i^2\left(f_i - a - bx_i - cx_i^2 - dx_i^3\right)$$
$$\frac{\partial E}{\partial d} = -2\sum_i x_i^3\left(f_i - a - bx_i - cx_i^2 - dx_i^3\right)$$

b. Normal equations

Collect the like powers of x_i and set the derivatives equal to zero.

$$\sum_i f_i = a + b\sum_i x_i + c\sum_i x_i^2 + d\sum_i x_i^3$$
$$\sum_i x_i f_i = a\sum_i x_i + b\sum_i x_i^2 + c\sum_i x_i^3 + d\sum_i x_i^4$$
$$\sum_i x_i^2 f_i = a\sum_i x_i^2 + b\sum_i x_i^3 + c\sum_i x_i^4 + d\sum_i x_i^5$$
$$\sum_i x_i^3 f_i = a\sum_i x_i^3 + b\sum_i x_i^4 + c\sum_i x_i^5 + d\sum_i x_i^6$$

In terms of the matrix elements we used in solving simultaneous linear equations,

$$c_1 = \sum f_i \qquad b_{11} = 1$$
$$c_2 = \sum x_i f_i \qquad b_{12} = \sum x_i$$
$$c_3 = \sum x_i^2 f_i \qquad b_{21} = \sum x_i$$
$$c_4 = \sum x_i^3 f_i \qquad b_{22} = \sum x_i^2 \text{ , etc.}$$

The system is solved by any standard method, Gauss-Jordan, Gauss-Seidel, even by Cramer's method.

c. Accuracy of fit

We'd like to have some statistical measure of how good the fit between the $\{f_i\}$ and $f(x)$ is. This will depend on the relation between E and the $\{s_i^2\}$. Let's consider a quantity called ($N = n + 1$)

$$\mathrm{X}^2 = \sum_{i=1}^{N} \frac{\left(f(x_i) - f_i\right)^2}{s_i^2}.$$

If all $s_i = 1$, then $\mathrm{X}^2 = E$. Now, on another hand, if $s_i \approx \left|f(x_i) - f_i\right|$, then $\mathrm{X}^2 \approx N - g$, where g is the number of adjustable parameters and $N - g$ is the *number of degrees of freedom* in the mathematical model for the data. We'd like to see $\dfrac{\mathrm{X}^2}{N-g} \approx 1$ for a "good" fit, while

$\frac{X^2}{N-g} << 1$ indicates that the quality of the fit is ambiguous (sometimes called over fitted), and

$\frac{X^2}{N-g} >> 1$ indicates a "poor" fit.

3. Least Squares Fit to Non-polynomial Function

The process is similar when fitting to a function that is not a polynomial. For instance, say that

$$f(x) = a \ln x + b \cos x + ce^x .$$

We wish to fit this function to the data shown at right. In this case, $N = 10$ and $g = 3$. The adjustable parameters are a, b and c.

x_i	f_i
.24	0.23
.65	-0.26
.95	-1.10
1.24	-0.45
1.73	0.27
2.01	0.10
2.23	-0.29
2.52	0.24
2.77	0.56
2.99	1.00

$$E = X^2 = \sum_{i=1}^{10} \left(f_i - a \ln x_i - b \cos x_i - ce^{x_i} \right)^2$$

The normal equations are:

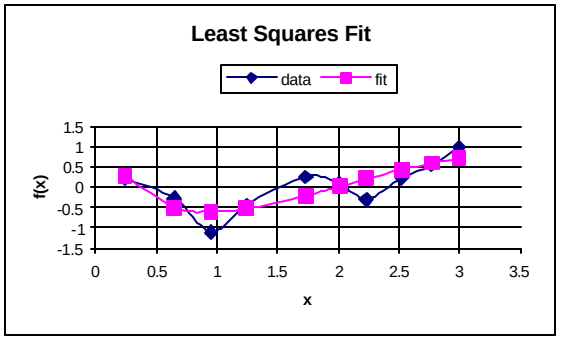
$$a\sum (\ln x_i)^2 + b\sum \ln x_i \cos x_i + c\sum \ln x_i e^{x_i} = \sum f_i \ln x_i$$
$$a\sum \ln x_i \cos x_i + b\sum (\cos x_i)^2 + c\sum \cos x_i e^{x_i} = \sum f_i \cos x_i$$
$$a\sum \ln x_i e^{x_i} + b\sum \cos x_i e^{x_i} + c\sum (e^{x_i})^2 = \sum f_i e^{x_i}$$

$$6.794a - 5.348b + 63.259c = 1.616$$
$$-5.347a + 5.108b - 49.009c = -2.383$$
$$63.259a - 49.009b + 1002.506c = 26.773$$

When solved by the Gauss-Jordan method, these yield

$$a = -1.041$$
$$b = -1.261$$
$$c = 0.031$$
$$f(x) = -1.041 \ln x - 1.261 \cos x + 0.031e^x$$
$$\frac{X^2}{N-g} = \frac{0.926}{7} << 1$$

The goodness of fit between these data and this function is ambiguous. A glance at a graph verifies that the fit is "iffy." [That's the technical term for it.]



VI. Integration

We wish to evaluate the following definite integral: $\int_a^b f(x)dx$.

We use numerical methods when

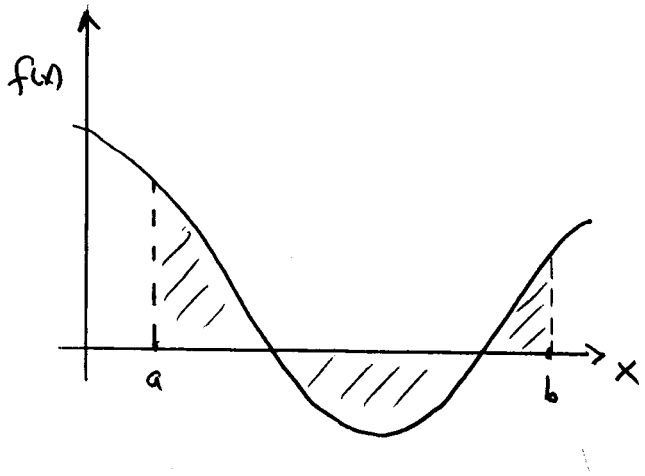
i) $f(x)$ is known analytically but is too complicated to integrate analytically or
ii) $f(x)$ is known only as a table of data.

A. Newton-Cotes Formulæ

1. Trapezoid Rule

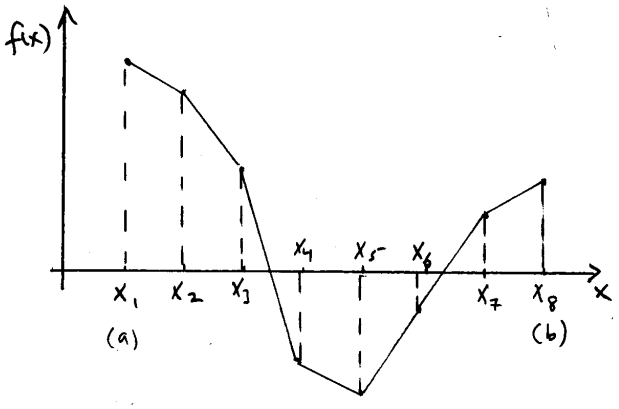
a. Graphs

Graphically, a definite integral is the area between the x-axis and the curve $f(x)$. Areas below the axis are negative; areas above the axis are positive.

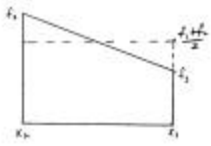


b. Trapezoids

The area "under" the curve might be approximated most simply by a series of trapezoids and triangles.



$$\frac{f_1+f_2}{2}(x_2-x_1)+\frac{f_2+f_3}{2}(x_3-x_2)+\cdots$$



Notice that $x_1 = a$ and that $x_8 = b$.

c. Interpolating polynomial

In effect, we are replacing the integrand, $f(x)$, by a straight line between each pair of points:

$$p_1(x) = f(x_{i-1}) + (x - x_{i-1})\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

This can be checked by integrating $p_1(x)$ analytically.

$$\int_{x_{i-1}}^{x_i} f_{i-1} + (x - x_{i-1})\frac{f_i - f_{i-1}}{x_i - x_{i-1}}\,dx = f_{i-1}(x_i - x_{i-1}) + \frac{f_i - f_{i-1}}{x_i - x_{i-1}}\frac{x^2}{2}\bigg|_{x_{i-1}}^{x_i} - x_{i-1}\frac{f_i - f_{i-1}}{x_i - x_{i-1}}x\bigg|_{x_{i-1}}^{x_i}$$

$$= x_i f_{i-1} - x_{i-1} f_{i-1} + \frac{x_i f_i}{2} - \frac{x_i f_{i-1}}{2} + \frac{x_{i-1} f_i}{2} - \frac{x_{i-1} f_{i-1}}{2} - x_{i-1} f_i + x_{i-1} f_{i-1}$$

$$= \frac{x_i f_{i-1}}{2} + \frac{x_i f_i}{2} - \frac{x_{i-1} f_i}{2} - \frac{x_{i-1} f_{i-1}}{2}$$

$$= \frac{1}{2}(x_i - x_{i-1})(f_i + f_{i-1}) \qquad \text{check.}$$

d. Implementation

For N data points spanning $[a,b]$, there are $N - 1$ trapezoids. $T = \sum_{i=2}^{N}(x_i - x_{i-1})\frac{f_i + f_{i-1}}{2}$

If the data are uniformly spaced, then $x_i - x_{i-1} = h$ for all i, and

$$T = \frac{h}{2}\sum_{i=2}^{N}(f_i + f_{i-1}) = h\left(\frac{f_1}{2} + \frac{f_N}{2} + \sum_{i=2}^{N-1} f_i\right).$$

The Fortran might look like this:

```
      n = 10
      T = 0.0
      do 100 i=1,n
100   T = T + (x(i)-x(i-1))*(f(i)+f(i-1))/2.0
```

2. Extension to Higher Order Formulæ

a. Forward difference interpolating polynomial

We'll take this opportunity to examine an alternative interpolating polynomial—the *Forward Difference Polynomial*.

Imagine we have a table of data pairs (x_i,f_i) which are uniformly spaced, with spacing h. The forward differences are just the familiar deltas.

first order: $\Delta f(x_i) = f(x_2) - f(x_1) = f_2 - f_1$

second order: $\Delta^2 f(x_1) = \Delta f(x_2) - \Delta f(x_1) = (f(x_3) - f(x_2)) - (f(x_2) - f(x_1))$

Notice that the differences $\Delta f(x_1)$ and $\Delta^2 f(x_1)$ are regarded as being evaluated at $x = x_1$. Hence the term *forward* difference.

Notice, too, that the forward differences are related to the divided differences simply by multiplying by the denominators.

$$\Delta f(x_1) = h \cdot f[x_2, x_1]$$
$$\Delta^2 f(x_1) = 2h^2 \cdot f[x_3, x_2, x_1]$$
$$\vdots$$
$$\Delta^n f(x_1) = n! h^n \cdot f[x_{n+1}, x_n, \cdots, x_2, x_1]$$

Now, let's expand the integrand $f(x)$ in a Taylor's Series about $x = x_1$. Further, to increase the element of confusion, let $a = \dfrac{x - x_1}{h}$ so that $x = x_1 + ah$.

$$f(x) = f(x_1) + a\Delta f(x_1) + \frac{a(a-1)}{2!}\Delta^2 f(x_1) + \frac{a(a-1)(a-2)}{3!}\Delta^3 f(x_1) + \cdots$$

Depending on how many terms are kept, this will give a polynomial in a or in x.

b. Simpson's rule

Any number of formulæ may be created by replacing the integrand, $f(x)$, with an interpolating polynomial of some specified degree. If $f(x) \approx p_1(x) = f(x_1) + a\Delta f(x_1)$, the Trapezoid Rule is recovered.

Perhaps $f(x)$ has some curvature, so a second degree interpolating polynomial may serve better.

$$\int_{x_1}^{x_3} f(x)dx \approx h\int_0^2 p_2(x_1 + ah)da = h\int_0^2 \left[f(x_1) + a\Delta f(x_1) + \frac{a(a-1)}{2}\Delta^2 f(x_1) \right] da$$
$$= h\left[2f(x_1) + 2\Delta f(x_1) + \frac{1}{3}\Delta^2 f(x_1) \right]$$

Expand the differences. . .

$$\int_{x_1}^{x_3} f(x)dx \approx h\left[2f(x_1) + 2f(x_2) - 2f(x_1) + \frac{1}{3}f(x_3) - \frac{2}{3}f(x_2) + \frac{1}{3}f(x_1) \right]$$
$$= \frac{h}{3}\left[f(x_1) + 4f(x_2) + f(x_3) \right]$$

This is Simpson's Rule, which integrates over segments of three data points (or two intervals of h) in one step.

c. Implementation

$$\int_{x_1}^{x_3} f(x)dx = \frac{h}{3}\left[f(x_1) + 4f(x_2) + f(x_3) \right]$$

$$\int_{x_3}^{x_5} f(x)dx = \frac{h}{3}\left[f(x_3) + 4f(x_4) + f(x_5)\right]$$

$$\vdots$$

$$\int_{x_{n-1}}^{x_{n+1}} f(x)dx = \frac{h}{3}\left[f(x_{n-1}) + 4f(x_n) + f(x_{n+1})\right]$$

Add ‘em up. . .

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[2\sum_{\substack{i=1 \\ \Delta i=2}}^{n+1} f(x_i) + 4\sum_{\substack{i=2 \\ \Delta i=2}}^{n} f(x_i) - f(a) - f(b)\right]$$

Caveats: i) the data points must be uniformly spaced.

ii) $n + 1$ must be odd, starting with 1 so that $n = \frac{b-a}{h}$ is even.

B. Numerical Integration by Random Sampling

1. Random Sampling

a. Pseudorandom numbers

Random numbers are a sequence of numbers, $(z_1, z_2, z_3, \cdots)$, lying in the interval (0,1). There is no pattern in the progression of the numbers, nor is any number in the sequence related to any other number by a continuous function. There are statistical tests for randomness in a sequence of numbers but we won't bother with them here.

The operation of a computer is deterministic, so truly random numbers cannot be generated by a computer program. However, sequences can be generated that appear to be random in that the sequence passes some of the statistical tests for randomness. Such a sequence of numbers is called *pseudorandom*.

Here is an algorithm for generating a sequence of pseudorandom numbers:

$$x_i = \mathrm{mod}(a \cdot x_{i-1} + c, m)$$

$$z_i = \frac{x_i}{m}$$

where a, c and m are integers and mod( ) is the modulus function. The pseudorandom number uniformly distributed in the interval (0,1) is z_i.

In Fortran, this looks like the following:

```
x = xo
do i=1,100
x1 = amod(a*x*c,em)
z = x1/em
x = x1
print *,z
end do
```

This process generates a sequence of numbers $\{z_i\}$ that have some properties of random numbers, but in fact the sequence repeats itself—it's periodic. The exact sequence depends on the initial value, x_o, called the *seed*. Usually, m is a large integer, commonly a power of 2. The numbers c and m can have no common factor (c can be zero) while a is a multiple of a prime factor of $m + 1$. The period of the sequence is m, which is why m needs to be large. For instance, we might take $m = 2^{31}$, $c = 0$ and $a = 16807$.

b. Intervals

Suppose we want our pseudorandom numbers to lie in the interval (a,b) rather than (0,1). This is easily done by scaling, or mapping onto the desired interval. Say $0 \le z \le 1$, then $y = (b-a) \cdot z + a$ will lie in the interval (a,b).

c. Distributions

The example random number generator mentioned above produces numbers *uniformly distributed* in (0,1). This means that is (0,1) were divided into equal subintervals, an equal

number of random numbers is expected in each of those subintervals. The probability of the next random number in the sequence falling in a particular subinterval is the same for all the subintervals spanning (0,1).

It is possible to form sequences of pseudorandom numbers which obey some other distribution function, such as Poisson or Gaussian, etc. We won't get into that here.

2. Samples of Random Sampling

a. Coin toss

We have two outcomes for each toss, of equal probability. We'll generate an integer, either 1 or 2, using a pseudorandom number generator.

z_i = a uniformly distributed pseudorandom number in (0,1)
j = int(2*z_i) + 1 = 1 or 2

We'll say that if j = 1, it's heads, if j = 2 it's tails.

b. Roll of a die

In this case we have six outcomes, of equal probability (we hope). So we need to produce an integer from 1 to 6.

j = int(6*z_i)+1 = 1, 2, 3, 4, 5 or 6

Now, if it is known that the die is loaded, then we use a different scheme, creating subintervals in (0,1) whose lengths reflect the relative probabilities of the faces of the die coming up. For instance, we might say that

z_i	j
$0 < z_i \leq 0.2$	1
$0.2 < z_i \leq 0.34$	2
$0.34 < z_i \leq 0.56$	3
$0.56 < z_i \leq 0.72$	4
$0.72 < z_i \leq 0.89$	5
$0.89 < z_i < 1$	6

3. Integration

Thinking again of the definite integral as an area under a curve, we envision a rectangle whose area is equal to the total area under the curve $f(x)$. The area of that equivalent rectangle is just the length of the integration interval (a,b) times the average value of the integrand over that interval. How to take that average? One way is to sample the integrand at randomly selected points.

a. One dimensional definite integrals

$\int_0^1 f(x)dx \cong \frac{1}{n}\sum_{i=1}^{n} f(x_i)$, where the $\{x_i\}$ form a pseudorandom sequence uniformly distributed in (0,1). Over some other interval, $\int_a^b f(x)dx \cong (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i)$, where $\{x_i\} \in (a,b)$.

Since we are just averaging over a list of numbers, the error is O[$\frac{1}{\sqrt{n}}$], just like the deviation of the mean.

example: $\int_0^1 \sin x\,dx$

$$\int_0^1 \sin x\,dx = \frac{1}{3}[\sin 0.00075 + \sin 0.01335 + \sin 0.3904] = 0.1313$$

$$\int_0^1 \sin x\,dx = \frac{1}{4}[\sin 0.00075 + \sin 0.01335 + \sin 0.3904 + \sin 0.8776] = 0.2910$$

$$\int_0^1 \sin x\,dx = \frac{1}{5}[\sin 0.00075 + \sin 0.01335 + \sin 0.3904 + \sin 0.8776 + \sin 0.0992] = 0.2524$$

⋮

The exact result is 0.460.

b. Multi-dimension integrals

The random sampling approach is particularly useful with 2- and 3-dimensional integrals. The other methods of numerical integration quickly become too messy to set up.

$$\int_0^1\int_0^1\int_0^1 f(x,y,z)dxdydz \cong \frac{1}{n}\sum_{i=1}^{n} f(x_i,y_i,z_i),$$

where (x_i,y_i,z_i) is an ordered triple, each member uniformly distributed on (0,1).

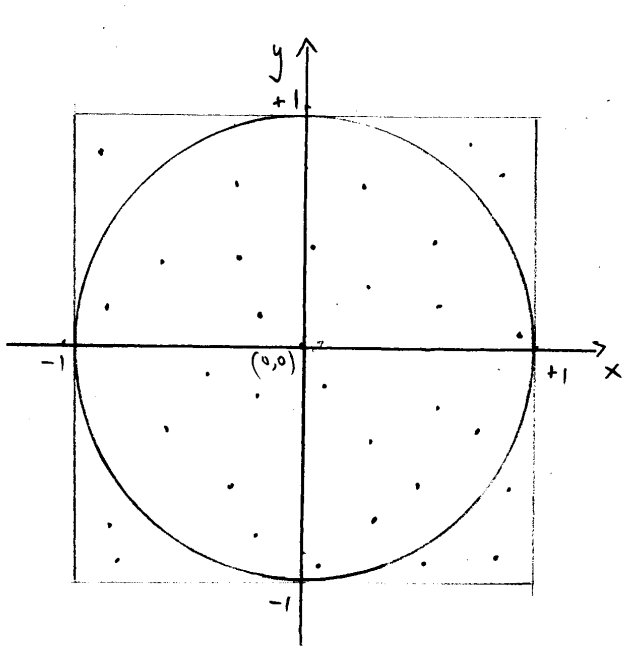
We may use three separate sequences of pseudorandom numbers or simply take numbers from one sequence three at a time.

c. Alternate integration regions

i) $\int_{a_z}^{b_z}\int_{a_y}^{b_y}\int_{a_x}^{b_x} f(x,y,z)dxdydz \cong (b_x-a_x)(b_y-a_y)(b_z-a_z)\frac{1}{n}\sum_{i=1}^{n} f(x_i,y_i,z_i)$

ii) Suppose the integration region is not rectangular. Then an extra step is needed, to test for and discard random points that fall outside the integration region.

e.g., a circle—discard points for which $x_i^2 + y_i^2 > 1$.



Why do it this way; to ensure that the points are uniformly distributed in all directions. If points are taken uniformly distributed in the radius, the points will be more widely spread the further out from the center they lie, not uniformly spread over the area of the circle.

example: compute the volume of a sphere of radius *R*. In this situation, the integrand is 1.

$$V = \int_0^R \int_0^{2p} \int_0^{p} \sin^2 q \, dq \, dj \, dr = \frac{4}{3} p R^3$$

Numerically,

$$V \cong \left(R-\left(-R\right)\right)\left(R-\left(-R\right)\right)\left(R-\left(-R\right)\right)\frac{1}{n} \sum_{\substack{i=1 \\ x_i^2+y_i^2+z_i^2 \le R^2}}^{m} 1 = \frac{(2R)^3}{n} m = \frac{m}{n} 8R^3 .$$

Notice this: the total number of random points generated is *n*. However, only *m* of those lie within the spherical volume. The spherical volume we obtain is equal to $\frac{m}{n}$ times the volume of a cube whose side is 2*R*. It's interesting to see what this fraction is.

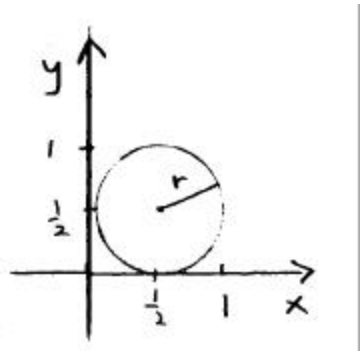
$$\frac{V_{sphere}}{V_{cube}} = \frac{\frac{4}{3} p R^3}{8R^3} = \frac{p}{6} = 0.52359\cdots .$$

The ratio $\frac{m}{n}$ should approach this constant as we generate more points and include them in the summation.

Another way to look at this $\frac{m}{n}$ issue is to say that $f(x) = 1$ when $x_i^2 + y_i^2 + z_i^2 \le R^2$ and 0 when $x_i^2 + y_i^2 + z_i^2 > R^2$. Then there is no distinction between n and m, and the summation is a sum of $n - m$ zeros and m ones.

d. Example

Evaluate $\iint_\Omega \sin\sqrt{\ln(x+y+1}\,dxdy$, where Ω is the region $\left(x-\frac{1}{2}\right)^2\left(y-\frac{1}{2}\right)^2 \le r^2$.



$$I = \iint_\Omega \sin\sqrt{\ln(x+y+1}\,dxdy \cong \left(r+\frac{1}{2}-\left(\frac{1}{2}-r\right)\right)^2 \frac{1}{n} \sum_{\left(x_i-\frac{1}{2}\right)^2\left(y_i-\frac{1}{2}\right)^2 \le r^2}^{m} f(x_i, y_i) \quad = 4r^2 \frac{1}{n} \sum_{\left(x_i-\frac{1}{2}\right)^2\left(y_i-\frac{1}{2}\right)^2 \le r^2}^{m} f(x_i, y_i)$$

[If you want to try it, for $r = 0.5$, $I = 0.57$.]

This is equivalent to averaging the integrand over a circular area, thusly

$$\iint_\Omega \sin\sqrt{\ln(x+y+1}\,dxdy \cong pr^2 \frac{1}{m} \sum_{\left(x_i-\frac{1}{2}\right)^2\left(y_i-\frac{1}{2}\right)^2 \le r^2}^{m} f(x_i, y_i) \quad .$$

Of course, often the shape of the region of integration isn't a simple rectangle or circle.

e. Fortran

```
      real*8 x,a,em,sum,ex,why,ax,ay,bx,by,r,r2
      f(x,y) = sin(log(x+y+1))
      n = 100
      r = 0.5
      r2 = r*r
      x = 256.
      em = 2.0**31
      a = 16807.
      sum = 0.0
      ax = 0.5 – r
      ay = 0.5 – r
      bx = r + 0.5
      by = r + 0.5
      do 500 i=1,n
      x = amod(a*x,em)
      ex = x/em*(bx-ax) + ax
      why = x/em*(by-ay) + ay
      If( (ex-0.5)*(ex-0.5)+(why-0.5)*(why-0.5) .gt. r2 ) goto 500
      sum = sum + f(ex,why)
 500  continue
      sum = sum*(by-ay)*(bx-ax)/n
      print *, sum
      stop
      end
```

VII. Ordinary Differential Equations

A. Linear First Order Equations

We seek to solve the following equation for $x(t)$: $\frac{dx}{dt} = f(x,t)$. There are analytical methods of solution: integration, separation of variables, infinite series, etc. In practice these may not be convenient or even possible. In such cases we resort to a numerical solution. The $x(t)$ takes the form of a table of data pairs $\{t_i,x_i\}$, rather than a function.

1. One Step Methods

a. Taylor's Series

Many numerical solutions derive from the Taylor's series expansion

$$x(t) = x(t_o) + (t-t_o)\frac{dx(t_o)}{dt} + \frac{(t-t_o)^2}{2!}\frac{d^2x(t_o)}{dt^2} + \cdots + \frac{(t-t_o)^p}{p!}\frac{d^px(t_o)}{dt^p} + \cdots.$$

We are given $\frac{dx}{dt} = f(x,t)$, so we could substitute this into the series thusly:

$$x(t) = x(t_o) + (t-t_o)f(x_o,t_o) + \frac{(t-t_o)^2}{2!}\frac{df(x_o,t_o)}{dt} + \cdots + \frac{(t-t_o)^p}{p!}\frac{d^{p-1}f(x_o,t_o)}{dt^{p-1}} + \cdots.$$

However, to obtain $\frac{df}{dt}$, $\frac{d^2f}{dt^2}$, $\frac{d^3f}{dt^3}$, etc., we have to use the chain rule.

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{dx}{dt}$$

$$\frac{d^2f}{dt^2} = \frac{\partial^2 f}{\partial t^2} + 2f\frac{\partial^2 f}{\partial x \partial t} + f^2\frac{\partial^2 f}{dx^2} + \frac{\partial f}{\partial x}\left[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial x}\right]$$

It's easy to see that this gets very messy rather quickly.

b. Euler's Method

Let's keep just the first two terms of the Taylor's series: $x(t) = x(t_o) + (t-t_o)f(x_o,t_o) + T_o$, where the T_o is the sum of all the terms we're dropping—call it the *truncation error*. In what follows, we will have to distinguish between the correct or exact solution, $x(t)$, and our approximate solution, x_i. We <u>hope</u> $x_i \cong x(t_i)$.

With the Euler Method, our algorithm is [given t_o, $x(t_o) = x_o$ and $f(x,t)$]

$$x_1 = x_o + (t_1 - t_o)f(x_o,t_o)$$
$$x_2 = x_1 + (t_2 - t_1)f(x_1,t_1)$$
$$\vdots$$
$$x_{i+1} = x_i + (t_{i+1} - t_i)f(x_i,t_i)$$
$$\vdots$$

example: $\frac{dx}{dt} = 13t$, with $t_o = 0$ and $x_o = 4$ and $(t_{i+1} - t_i) = h = 0.5$.

The algorithm is: $x_{i+1} = x_i + (t_{i+1} - t_i)(13t_i)$.

The first few steps in the numerical solution are shown in the following table.

i	t	x
0	0	4
1	.5	4
2	1	7.25
3	1.5	13.75
4	2	23.5
⋮	⋮	⋮

2. Error

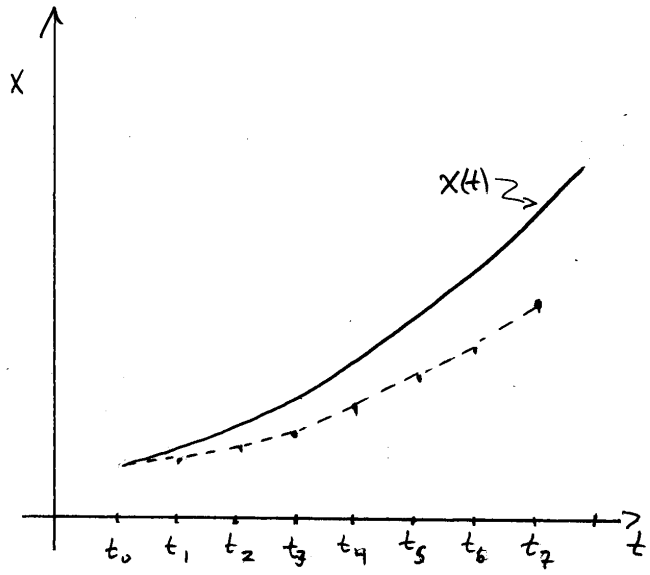
a. Truncation error

$$x_{i+1} = x_i + hf(x_i, t_i) + T_{i+1}$$

Not only do we not know what the exact solution is, we don't know how far the numerical solution deviates from the exact solution. In the case of a truncated Taylor's series, we can estimate the truncation error by evaluating the first term that is dropped. For Euler's formula, that's the third term of the series.

$$T_{i+1} \approx \frac{h^2}{2}\frac{df(x_i)}{dt} = \frac{h^2}{2} f'(x_i)$$

Here's a graph of both the exact (but unknown) and the numerical solutions.



The deviation from the exact $x(t)$ may tend to increase as the total truncation error accumulates from step to step, the further we get from the initial values (t_o, x_o). The lesson is—make h small.

b. Round-off error

Since the values are stored in finite precision, round-off error accumulates from step to step also. Therefore, in traversing an interval $o \le t \le b$, we'd like to have as few steps as possible. In other words, we want h to be large. Consequently, the two sources of error put competing pressure on our choice of step size, h. If we have some knowledge of $x(t)$, we may be able to achieve a balance between large and small step size. Otherwise, it's trial and error.

c. Higher order methods

The many numerical algorithms that have been developed over the years for solving differential equation seek to reduce the effect of truncation error by using more terms from the Taylor's series, or in some way correcting for the truncation error at each step. In that way, fewer, larger steps can be used.

B. Second Order Ordinary Differential Equations

$\frac{d^2x}{dt^2} = x'' = f(t, x, \frac{dx}{dt}) = f(t, x, x')$, with initial conditions $x(0) = x_o$ and $x'(0) = v_o$.

1. Reduction to a System of First Order Equations

a. New Variables

We start by introducing new variable names: $z_1 = t$; $z_2 = x$; $z_3 = x'$; $z_4 = x''$. The first three variables are the solutions to the following differential equations:

$$z_1' = 1$$
$$z_2' = x' = z_3$$
$$z_3' = x'' = z_4$$

These form a set of three simultaneous first order differential equations,

$$z_1' = 1$$
$$z_2' = z_3$$
$$z_3' = z_4 = f(z_1, z_2, z_3)$$

with the initial conditions $z_1(0) = 0$, $z_2(0) = x_o$ and $z_3(0) = v_o$ respectively.

b. Solution

Any method, such as Euler's, may now be applied to each first order equation in turn. Thusly:

$$z_{1,i+1} = z_{1,i} + h \cdot 1$$
$$z_{2,i+1} = z_{2,i} + h \cdot z_{3,i}$$
$$z_{3,i+1} = z_{3,i} + h \cdot f_i .$$

The Fortran code might look like this:

```
      z(1) = 0.0
      z(2) = xo
      z(3) = vo
      h = 0.01
      do 100 i=1,100
      z(1) = z(1) + h
      z(2) = z(2) +h*z(3)
      z(3) = z(3) + h*f(z(1),z(2),z(3))
      write(5,1000) z(1),z(2),z(3)
1000  format(1x,3e15.5)
 100  continue
```

c. Example

$$x'' = -x' - 9 + \cos(w \cdot t)$$
$$x(0) = x_o,\ x'(0) = v_o$$

In this case, $f(t, x, x') = -x' - 9 + \cos(w \cdot t)$, so the algorithm looks like

$$z_{1,i+1} = z_{1,i} + h \cdot 1$$
$$z_{2,i+1} = z_{2,i} + h \cdot z_{3,i}$$
$$z_{3,i+1} = z_{3,i} + h \cdot \left[-z_{3,i} - g + \cos(w \cdot z_{1,i}) \right].$$

2. Difference Equations

An alternative approach to second order ordinary differential equations is to replace the derivatives with finite differences. The differential equation is replaced by a *difference equation*.

a. Difference equation

Using forward divided differences, we obtain

$$x' = \frac{dx}{dt} \cong \frac{x_{i+1} - x_i}{h} \text{ and } x'' = \frac{d^2x}{dt^2} \cong \frac{1}{h}\left(\frac{x_{i+1} - x_i}{h} - \frac{x_i - x_{i-1}}{h} \right) = \frac{x_{i+1} - 2x_i + x_{i-1}}{h^2}.$$

Let's say that we have the second order differential equation

$$x'' = ax' + bx + ct + d.$$

The corresponding difference equation is

$$\frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} = a\left(\frac{x_{i+1} - x_i}{h} \right) + bx_i + ct_i + d.$$

The next step is to solve for the "latest" x.

$$x_{i+1} - 2x_i + x_{i-1} = ahx_{i+1} - ahx_i + bh^2 x_i + ch^2 t_i + dh^2$$
$$(1 - ah)x_{i+1} = (2 - ah + bh^2)x_i - x_{i-1} + ch^2 t_i + dh^2$$
$$x_{i+1} = \frac{1}{1 - ah}\left[(2 - ah + bh^2)x_i - x_{i-1} + ch^2 t_i + dh^2 \right]$$

The initial conditions are applied by setting $t_o = 0$, $x_0 = x_o$ and $x_{-1} = x_o - v_o h$.

b. Examples

i) $x'' = -g$

Here, $a = b = c = 0$ and $d = -g$.

$$x_{i+1} = 2x_i - x_{i-1} - gh^2$$

ii) $x'' = -x' - g$

This time, $a = -1$, $b = 0$, $c = 0$ and $d = -g$.

$$x_{i+1} = \frac{1}{1 + h}\left[(2 + h)x_i - x_{i-1} - gh^2 \right]$$

c. Discretization error

Replacing continuous derivatives with finite differences introduces what is known as *discretization error*. Implicitly, we are assuming a straight line between x_i and x_{i+1} and between x_i' and x_{i+1}' as well. There will always be some $\Delta = x_{i+1} - x(t_{i+1})$ at each step which will then accumulate over the sequence of steps in the numerical solution.