Introduction to Scientific Computation

W. E. Schiesser
Iacocca D307
111 Research Drive
Lehigh University
Bethlehem, PA 18015

(610) 758-4264 (office)
(610) 758-5057 (fax)

wes1@lehigh.edu

http://www.lehigh.edu/~wes1/wes1.html

Course Orientation and Content

http://www.lehigh.edu/~wes1/apci/28jan00.tex

1. Scientific computation
2. Quality software and software libraries
3. Programming languages
4. Numerical considerations
   (a) Computer word length
   (b) Roundoff and truncation
   (c) Conditioning
   (d) Convergence and error tolerances
5. Interpolation
   (a) Polynomials
   (b) Splines
6. Integration
   (a) Gaussian quadrature
   (b) Splines
   (c) Adaptive
7. Differentiation
   (a) Finite differences
   (b) Symbolic computation
8. References and software

In this course, we will emphasize:

- Terminology
- Modeling of engineering systems
- Mathematical forms resulting from the modeling
- Established algorithms (not the detailed underlying mathematics, but the principal conclusions)
- Computational resources (e.g., software libraries) for the application of the algorithms to the mathematical forms

Why LaTeX (.tex files)?

Latex:

- Has excellent mathematical formatting capabilities (better than the equation editor of MS Word, for example).
- Is entirely ASCII-based, and therefore LaTeX files can be composed and read with an ASCII editor and can easily be transmitted over the Internet.
- Is in the public domain.
- Is the established text formatter for scientific publishing (rather than MS Word, for example). Most major scientific publishers now specify LaTeX files as the preferred format for publication.
- Produces relatively small files (the file for all of the notes to be discussed today is under 70,000 bytes).

As an incidental point, LaTex code is easily written using Scientific Workplace (SW); SW also has a full Maple interface, and the results of Maple computations can be embedded in LaTeX documents.

Scientific Computation

Scientific computing is the collection of tools, techniques and theories required to solve on a computer mathematical models of problems in science and engineering.

The case for scientific computation:

- Reduced time
- Reduced cost
- Increasd spectrum of problem systems and parameters
- Enhancement of experimental results
  - Framework for interpretation of experimental data
  - Evaluation of model parameters

Golub and Ortega (1992)

Unique Features of Scientific Computation

Results cannot be quaranteed in advance

Example 1:

Consider the $2x2$ linear algebraic system:

$$\begin{aligned} 1.5x_1 + 3.5x_2 &= 2 \\ 3.0x_1 + 7.0x_2 &= 5 \end{aligned}$$

To eliminate x_2, multiply the first equation by 2 and subtract from the second equation

$$0 = 1$$

A rather astounding result! How did this happen?

The coefficient matrix is singular, i.e.,

$$\begin{vmatrix} 1.5 & 3.5 \\ 3.0 & 7.0 \end{vmatrix} = (1.5)(7.0) - (3.0)(3.5) = 0$$

Note that

$$2(\text{row } 1) = \text{row } 2$$

i.e., the two rows are *linearly dependent.*

Thus, the $2x2$ system is *singular*. Addtionally, because of the meaningless result $(0 = 1)$, it is *inconsistent* (does not have a solution).

- Another closely related system is

$$1.5x_1 + 3.5x_2 = 2.5$$
$$3.0x_1 + 7.0x_2 = 5$$

which leads to

$$0 = 0$$

Thus, we at least arrive at something that makes sense mathematically. However, the coefficient matrix is still singular so that we cannot solve for unique values of x_1 and x_2. This system is therefore termed *consistent* but *singular.*

Since these equations are essentially the same, we can really only use one or the other. Thus, we in effect have one equation and two unknowns, and we therefore must select one unknown arbitrarily, then solve for the other unknown. A (nonunique) solution to this system is

$$x_1 = c$$
$$x_2 = (2.5 - 1.5c)/3.5$$

where c is an arbitrary constant.

- More typical situation:

$$1.499999x_1 + 3.5x_2 = 2$$
$$3.0x_1 + 7.0x_2 = 5$$

Solving for x_1 by eliminating x_2,

$$(2)(1.499999)x_1 - 3.0x_1 = 1$$

or

$$(2.999998 - 3.0)x_1 = 1$$

Our computer had better carry at least seven figures in its calculations. But this is all that single precision Fortran, for example, can do (with 32-bit arithmetic).

The system is termed *nearly singular*, *ill-conditioned* or *numerically singular* with 32-bit arithmetic.

Thus, the precision determined by the word length of the computer in scientific calculations can be very important, and we should have some quantitative measure of the precision of any new computer that we use (to be addressed subsequently).

For example, with single precision Fortran, if the first coefficient had been 1.49999999, the problem would have been singular, and a solution could not be calculated. However, with double precision Fortran (64-bit arithmetic), we could still expect a solution with an accuracy of several (i.e., $4-5$) significant figures.

- We conclude that depending on the characteristics of the problem, and the precision of the computer arithmetic, we may not be able to compute a solution to even a $2x2$ linear algebraic system.
- The problems with these small ($2x2$) systems were rather obvious. But similar problems with $1000x1000$ systems, for example, will not be so obvious, yet can be just as troublesome.
- More generally, we may not be able to produce a "solution" because there is none (the problem is not properly posed mathematically), or the mathematical model is so ill-conditioned that a numerical solution is precluded.

- Ideally, the software that we develop and use should include some safeguards against such potential conceptual and numerical problems, but in general we cannot count on this. In other words, there are no guarantees of a useful result in scientific computation.
- Typically, by a process of trial and error, we develop a numerical method that is efficient and accurate, but this is an experimental process, and the success of this effort is directly tied to the characteristics of the problem at hand.
- This contrasts with calculations that never fail (e.g., financial calculations). Also, having a faster computer might help, but it does not guarantee success. Understanding the source of the computational difficulties, and choosing an effective algorithm is more important than raw speed.
- Fortunately, the iterative process in scientific computation generally leads to a successful result, but this is a function of the experience and insight of the analyst.

Example 2:

Consider the third order polynomial

$$f(x) = x^3 - 2x - 5 = 0$$

This polynomial may be the first application of Newton's method, since Newton used it to illustrate his method (with a geometric argument, but not calculus!).

If we apply Newton's method

$$x^{i+1} = x^i - \frac{f(x^i)}{df(x^i)/dx}, i = 0, 1, \cdots$$

(where i is an iteration index or counter) and we make the choice of the initial starting point a root of

$$3x^2 - 2 = 0$$

or

$$x^0 = \pm\sqrt{2/3}$$

the method will fail because this is a *singular point* (the *Jacobian matrix*, or in this case, the first derivative, is singular). Also, if we choose a starting point near the singular point, the method will most likely fail, i.e., the system is *near-singular*.

Thus, we face the same difficulties as with the linear ($2x2$) system. The choice of an initial point that is not singular or near-singular becomes increasingly difficult as the size of the problem increases (more equations), and we should have a test in our software for the *condition number* of the Jacobian matrix.

- In conclusion, mathematical computation, in general, is not a completely straightforward process.
- Additionally, the mathematical forms are numerous, e.g.,
  - Linear algebra and eigenvalue problems
  - Nonlinear algebraic and transcendental equations
  - 1-D and multidimensional integrals
  - Ordinary differential equations (ODEs)
  - Partial differential equations (PDEs)
  - Optimization (e.g., max/min) problems
- All of these mathematical forms require some knowledge of at least basic algorithms and software.
- The field of scientific computation is large, challenging and rapidly moving; not surprisingly, professional numerical analysts devote full time effort to keeping abreast of this field.

How do we arrive at these mathematical forms when applying mathematical analysis of engineering systems? Here's an example:

A tubular reactor, as depicted in the accompanying diagram, has a significant heat effect so that cooling at the wall is being considered to remove the heat of reaction.

Consequently, significant radial heat and concentration profiles develop. The reactor is therefore modeled in terms of radial position, axial position and time; since there are three independent variables, we will use PDEs.

Our objective in this analayis is to determine what comes out of the reactor, i.e., the average concentrations and temperature, so that we can determine if it achieves the required operating performance. To do this, we construct a mathematical model.

The material balance for an incremental element of length Δz is (see accompanying diagram)

$$2\pi r\Delta r\Delta z\frac{\partial c_a}{\partial t} = 2\pi r\Delta z q_m|_r - 2\pi(r+\Delta r)\Delta z q_m|_{r+\Delta r}$$
$$+2\pi r\Delta r v c_a|_{z-\Delta z} - 2\pi r\Delta r v c_a|_z$$
$$-2\pi r\Delta r\Delta z k_r c_a^2$$

Division by $2\pi r\Delta r\Delta z$ and minor rearrangement gives

$$\frac{\partial c_a}{\partial t} = -\frac{(r+\Delta r)q_m|_{r+\Delta r} - rq_m|_r}{r\Delta r} - v(\frac{c_a|_z - c_a|_{z-\Delta z}}{\Delta z}) - k_r c_a^2$$

or in the limit $r \to 0, \Delta z \to 0$,

$$\frac{\partial c_a}{\partial t} = -\frac{1}{r}\frac{\partial(rq_m)}{\partial r} - v\frac{\partial c_a}{\partial z} - k_r c_a^2$$

If we now assume Fick's first law for the flux with a constant diffusivity, D

$$q_m = -D\frac{\partial c_a}{\partial r}$$

we obtain

$$\frac{\partial c_a}{\partial t} = \frac{D}{r}\frac{\partial(r\partial c_a\partial r)}{\partial r} - v\frac{\partial c_a}{\partial z} - k_r c_a^2$$

or

$$\frac{\partial c_a}{\partial t} = D(\partial^2 c_a\partial r^2 + \frac{1}{r}\partial c_a\partial r) - v\frac{\partial c_a}{\partial z} - k_r c_a^2 1 \qquad (1)$$

Equation (1) is the required material balance for $c_a(r, z, t)$.

The energy balance for the incremental section is

$$2\pi r\Delta r\Delta z\rho C_p\frac{\partial T}{\partial t} = 2\pi r\Delta z q_h|_r - 2\pi(r+\Delta r)\Delta z q_h|_{r+\Delta r}$$
$$+2\pi r\Delta r v\rho C_p T|_{z-\Delta z} - 2\pi r\Delta r v\rho C_p T|_z$$
$$-\Delta H 2\pi r\Delta r\Delta z k_r c_a^2$$

Division by $2\pi r\Delta r\Delta z\rho C_p$ gives

$$\frac{\partial T}{\partial t} = -\frac{(r+\Delta r)q_h|_{r+\Delta r} - rq_h|_r}{r\Delta r\rho C_p} - v\frac{(T|_z - T|_{z-\Delta z})}{\Delta z} - \frac{\Delta H k_r}{\rho C_p}c_a^2$$

or in the limit $\Delta r \to 0, \Delta z \to 0$,

$$\frac{\partial T}{\partial t} = -\frac{1}{\rho C_p r}\frac{\partial(rq_h)}{\partial r} - v\frac{\partial T}{\partial z} - \frac{\Delta H k_r}{\rho C_p}c_a^2$$

If we now assume Fourier's first law for the flux with a constant conductivity, k

$$q_h = -k\frac{\partial T}{\partial r}$$

we obtain

$$\frac{\partial T}{\partial t} = \frac{k}{\rho C_p r}\frac{\partial(r\partial T\partial r)}{\partial r} - v\frac{\partial T}{\partial z} - \frac{\Delta H k_r}{\rho C_p}c_a^2$$

or

$$\frac{\partial T}{\partial t} = \frac{k}{\rho C_p}(\partial^2 T\partial r^2 + \frac{1}{r}\partial T\partial r) - v\frac{\partial T}{\partial z} - \frac{\Delta H k_r}{\rho C_p}c_a^2 2 \qquad (2)$$

Eq. (2) is the required energy balance for $T(r, z, t)$.

The reaction rate constant, k_r, is given by

$$k_r = k_0 e^{-E/(RT)} 3 \tag{3}$$

The variables and parameters of eqs. (1), (2) and (3) are summarized in the following table (in cgs units)

$Reactantconcentration$	c_a	$Solutiontoeq.(1)$
$Temperature$	T	$Solutiontoeq.(2)$
$Reactionrateconstant$	k_r	$Fromeq.(3)$
$Time$	t	
$Radialposition$	r	
$Axialposition$	z	
$Enteringconcentration$	c_{a0}	0.01
$Enteringtemperature$	T_0	305
$Walltemperature$	T_w	$305, 375$
$Reactorradius$	r_0	2
$Reactorlength$	z_l	100
$Linearvelocity$	v	1.0
$Massdiffusivity$	D	0.1
$Thermaldiffusivity$	$k/(\rho C_p)$	0.1
$Liquiddensity$	ρ	1.0
$Liquidspecificheat$	C_p	0.5
$Heatofreaction$	ΔH	$-10,000$
$Specificrateconstant$	k_0	1.5×10^9
$Activiationenergy$	E	$15,000$
$Gasconstant$	R	1.987

Eq. (1) requires one initial condition (IC), two boundary conditions (BCs) in r and one BC in z

$$c_a(r,z,0) = 04 \tag{4}$$

$$\frac{\partial c_a(0,z,t)}{\partial r} = 0, \frac{\partial c_a(r_0,z,t)}{\partial r} = 05)(6 \tag{5}$$

$$c_a(r,0,t) = c_{a0}7 \tag{6}$$

Eq. (2) requires also requires one IC, two BCs in r and one BC in z

$$T(r,z,0) = T_0 8 \tag{7}$$

$$\frac{\partial T(0,z,t)}{\partial r} = 0, T(r_0,z,t) = T_w 9)(10 \tag{8}$$

$$T(r,0,t) = T_0 11 \tag{9}$$

The solution to this problem (and in general, to problems in ODEs and PDEs) are the dependent variables ($c_a(r,z,t)$,$T(r,z,t)$) as a function of the independent variables (r,z,t).

Since we are intertested in the exiting conditions, we will compute $c_a(r,z_l,t), T(r,z_l,t)$. Also, there may be significant radial profiles at the exit, so we will also compute the integrals

$$c_{a,avg}(t) = \int_0^{r0} 2\pi r c_a(r,z_l,t)dr/(\pi r_0^2)12 \tag{10}$$

$$T_{avg}(t) = \int_0^{r0} 2\pi r T(r,z_l,t)dr/(\pi r_0^2)13 \tag{11}$$

Note that this problem includes 2-D PDEs, transcendental equations and integrals. Because of its complexity (number of equations and their nonlinearity), an analytical solution is precluded, and we must use numerical methods.

We will consider the numerical solution of this problem later in the section on PDEs, and we will observe that with $T_w = 305^o K$, essentially no reaction takes place. If the wall is heated so that $T_w = 375^o K$, significant reaction takes place, but there are no hotspots or a runaway reaction. Thus, the reactor should give a reasonable production of the product if the wall is heated, not cooled, as originally suggested.

The validity of these conclusions, of course, depends on the validity of the mathematical model and the numerical values of the parameters. But assuming these requirements are satisfied, we can gain valuable insights and useful information about the performance of the reactor from the computer-based study.

"Off-the-shelf" software for the solution of eqs. (1) to (13) is not available. Rather, we have to construct a code, perhaps using subroutines that will calculate terms and special parts of this set of equations. For example, we could use library routines to calculate the partial derivatives in the PDEs, and the integrals in eqs. (12) and (13). Thus, software construction and testing must, in general, precede the analysis of the problem system.

Quality Software and Software Libraries

Quality mathematical software is readily available with the following properties:

- Based on algorithms proven to be effective through use over time
- Written to include special cases and identify problems that might not occur to an analyst without the experience of the software author(s)
- Tranportable to most computers, and easily called, typically with a one line command or a single parameter in a call to a compiler
- Thoroughly tested and documented

There is also the consideration of software based on the use of a compiler with a national or international standard vs a commercial product whose existence is dependent on a single company.

The standard compilers for mathematical software include:

- Fortran 77/90/95/2000
- C
- C++ Includes object oriented programming (OOP)
- Java and extensions

Commercial products include:

- Maple

- Mathcad
- Mathematica
- Matlab
- Visual Fortran
- Visual Basic (VB)

Note that Visual Fortran and Visual Basic are not languages.

Collections/Libraries of mathematical routines:

- Forsythe et. al. (1977), Computer Methods for Computational Mathematics
- Kahaner et. al. (1989), Numerical Methods and Software
- Press et. al. (1992), Numerical Recipes (Fortran, C)
- IMSL (Fortran, C)
- NAG (Fortran, C; available at Air Products)
- Netlib (ORNL, AT&T, e.g., ODEPACK, DASSL)
- ACMS TOMS library
- NA Digest (sent weekly)

Also, there are combinations, e.g., VB/NAG using Fortran dll's.

As an example of what is available (taken from)

http://www.nag.co.uk/numeric/FLOLCH/mk18.html

Fortran Library Mark 18 Introduction

Chapter A00 - Library Identification
Chapter A02 - Complex Arithmetic
Chapter C02 - Zeros of Polynomials
Chapter C05 - Roots of Transcendental Equations
Chapter C06 - Summation of Series
Chapter D05 - Integral Equations
Chapter E01 - Interpolation
Chapter E02 - Curve and Surface Fitting
Chapter E04 - Minimizing or Maximizing a Function
Chapter F01 - Matrix Factorizations
Chapter F02 - Eigenvalue and Eigenvectors
Chapter F03 - Determinants
Chapter F04 - Simultaneous Linear Equations
Chapter F05 - Orthogonalisation
Chapter F06 - Linear Algebra Support Routines
Chapter F07 - Linear Equations (LAPACK)
Chapter F08 - Least Squares and Eigenvalue Problems
Chapter F11 - Sparse Linear Algebra
Chapter G01 - Simple Calculations and Statistical Data
Chapter G02 - Correlation and Regression Analysis
Chapter G03 - Multivariate Methods
Chapter G04 - Analysis of Variance
Chapter G05 - Random Number Generators
Chapter G07 - Univariate Estimation
Chapter G08 - Nonparametric Statistics
Chapter G10 - Smoothing in Statistics
Chapter G11 - Contingency Table Analysis
Chapter G12 - Survival Analysis

Chapter G13 - Time Series Analysis
Chapter H - Operations Research
Chapter M01 - Sorting
Chapter P01 - Error Trapping
Chapter S - Approximations of Special Functions
Chapter X01 - Mathematical Constants
Chapter X02 - Machine Constants
Chapter X03 - Inner Products
Chapter X04 - Input/Output Utilities
Chapter X05 - Date and Time Utilities

The Numerical Algorithms Group Ltd, Oxford UK, 1999

Thus, there are many sources and combinations, and standardization is therefore a problem in the sense that the analyst must select software items that are compatible, or can be made compatible with reasonable effort.

Programming Languages

- Fortran 77/90/95/2000
- C
- C++ Includes object oriented programming (OOP)
- Java and extensions

All have international standards that insure tranportability, but not necessarily longevity. For example, what happened to PASCAL, APL, ALGOL, to name a few? Longevity is an important issue in terms of preserving and extending the investment in programming over time.

Fortran, for all of its purported limitations, has preserved the investment made in it over an extraordinarily long time (for the computer field), and it is evolving, and thus improving so that it's lifetime is unpredictable, but certainly extended.

There is the basic question of what newer languages will do in scientific computing that Fortran cannot do, especially when considering new releases of Fortran.

The same questions of longevity and investments in programming can be applied to reliance on a commercial product that depends on the viability of one company.

Numerical considerations

1. Computer word length
2. Roundoff and truncation
3. Conditioning
4. Convergence and error tolerances

We considered briefly the effect of computer word length on the solution of a $2x2$ linear system. The system

$$\begin{aligned} 1.499999x_1 + 3.5x_2 &= 2 \\ 3.0x_1 + 7.0x_2 &= 5 \end{aligned}$$

is ill-conditioned if the computer carries seven significant figures (corresponding to 32 bits). This conclusion follows from the subtraction $2.999998 - 3.0$ when solving for x_1.
The system

$$\begin{aligned} 1.49999999x_1 + 3.5x_2 &= 2 \\ 3.0x_1 + 7.0x_2 &= 5 \end{aligned}$$

is numerically singular for a computer with seven figure arithmetic.

In general, the precision of the numerical calculations as determined by the computer word length is an important factor in determining the accuracy of the final computed result. In fact, the single most important number that characterizes the arithmetic performed by a specific computer with specifc software is the machine epsilon or unit roundoff, defined as the smallest number *eps* such that

$$1 + eps > 1$$

To determine the value of the machine epsilon, consider the following small Fortran program

```
      program epssp
      eps=1.0
      do while(1.0+eps.gt.1.0)
         eps=eps/2.0
      end do
      eps=2.0*eps
      write(*,1)eps
 1    format(/,' eps = ',e10.3)
      stop
      end
```

The output from this program is eps = 0.119E-06. Thus, single precision Fortran has a precision of about one part in 10^7.

This might seem more than adequate for scientific and engineering applications, e.g., rarely are experimental data better than one part in 10^4. However, there are at least two reasons why better precision is required:

- In solving large sets of complicated equations, $10^6, 10^9, 10^{12}$ ... floating point operations are performed, each with a small roundoff, and these errors can accumulate, and may eventually invalidate the computed result. The best protection against this accumulated error is to use higher precision arithmetic.
- The problem may be poorly conditioned as we observed, even with a $2x2$ linear algebraic system (the equations in linear re-

gression are well known for being poorly conditioned). Again, the way around this problem may be greater precision (although better algorithms will probably be more effective).

Thus, if we consider the same program written in double precision Fortran,

```
      program epsdp
      implicit double precision (a-h,o-z)
      eps=1.0d0
      do while(1.0d0+eps.gt.1.0d0)
         eps=eps/2.0d0
      end do
      eps=2.0d0*eps
      write(*,1)eps
 1    format(/,' eps = ',d10.3)
      stop
      end
```

The output from this program is eps = 0.222D-15. Thus, double precision Fortran has a precision of about one part in 10^{16}, which experience has demonstrated is usually adequate for most applications in science and engineering.

The follow program does the same calculation in Matlab:

```
      eps=1.0;
      while(1.0+eps>1.0)
         eps=eps/2.0;
      end
      eps=2.0*eps;
      eps
      end
```

The output from this program is ans = 2.2204e-16. Thus, double precision Fortran and Matlab have about the same precision.

Programming in double precision Fortran is straightforward and generally requires only:

- Declaring all floating (real) variables as double precision, e.g., IMPLICIT DOUBLE PRECISION (A-H,O-Z)
- Converting all constants to double precision, e.g., $1.0D0$
- Declaring all in-line functions as double precision, e.g., DSIN (in place of SIN)

Other numbers that characterize the arithmetic of a computer are:

- The largest number the computer can accommodate (which defines the *dynamic range*)
- The smallest number a computer can accommodate that is different than zero

All of the key numbers are determined by the computer word length (number of bits assigned to each word), and how many bits are assigned to the mantissa and exponent. Typically, for a 32-bit word,

- 23 bits for mantissa
- 1 bit for sign
- 8 bits for exponent

In the case of Fortran, some typical numbers are

Precision	Bits per word	Machine epsilon	Dynamic range
Single	32	10^{-7}	$10^{-37} - 10^{37}$
Double	64	10^{-14}	$10^{-300} - 10^{300}$

To conclude, the calculation of the machine epsilon is easy (consider the preceding programs), and is probably the first thing we should do when using a new computer/software.

We have considered briefly **computational roundoff**, due to the finite wordlength e.g., what happens in the last figure of the subtraction $2.999998 - 3.0$ will clearly have a major effect on the answer (but we should not be working at this threshold of precision).

Roundoff can also occur during storage of the number. For example, if two numbers with n figures are multiplied, the full accuracy product is of length $2n$. However, when this product is stored, only the first n figures can be retained, and the least n figures are therefore dropped.

Finally, accumulation of roundoff over many calculations can lead to computational failure unless we carry enough significant figures (use a sufficiently long word length).

Truncation error, as the name applies results from using a truncated series representation of a function. For example, consider, the well-known infinite series expansion of the exponential function around $x = 0$

$$e^x = 1 + x + x^2/2! + x^3/3! + \cdots$$

A series representation is required since the computer can only do arithmetic, e.g., addition, subtraction, multiplication, division. If we truncate this series (since the computer cannot sum an infinite series),

$$e^x \approx 1 + x + x^2/2! + x^3/3!$$

Note that this truncated series is a third order polynomial. Generally, polynomial approximations are used throughout scientific computation, and they can be considered as truncated series approximations of the functions of interest.

If we consider the Taylor series expansion of e^x about the point a

$$e^x = e^a + e^a(x-a) + e^a(x-a)^2/2! + e^a(x-a)^3/3! + \cdots$$

or with $\Delta x = x - a$

$$e^x = e^a(1 + \Delta x + \Delta x^2/2! + \Delta x^3/3! + \cdots)$$

Thus, we can consider the third order polynomial approximation as

$$e^x \approx e^a(1 + \Delta x + \Delta x^2/2! + \Delta x^3/3! + O(\Delta x^4))$$

where $O(\Delta x^4)$ is interpreted as *of order* Δx^4. The order of the approximation is a common way of indicating its truncation error.

Concerning the **condition**, we observed that the system

$$1.49999999x_1 + 3.5x_2 = 2$$
$$3.0x_1 + 7.0x_2 = 5$$

is so ill-conditioned for a 32-bit computer, that the calculation of a solution will fail. One way we might check for a numerically singular system is to use the definition, i.e., effectively zero determinant.

However, the determinant is actually a poor practical measure of the condition of a linear algebraic system because it scales poorly. What might appear to be a small determinant (e.g., 10^{-40}) may actually be for a well-conditioned system. Thus, condition numbers have been developed that are a better reflection of the actual condition of the linear algebraic system.

For example, if the condition number, *cond*, is calculated, and if

$$cond > 1/eps$$

then the system is numerically singular, and the computer with the given *eps* will not be able to solve the linear algebraic system.

The preceding discussion of condition number pertains to linear algebraic systems. More generally, we can consider the condition number as the sensitivity to the problem system parameters. If the condition number is high, the system is very sensitive to the values of the parameters, and therefore a solution to the model equations may be precluded. For example, extreme sensitivity of differential equations to the initial conditions is a condition for chaos.

To conclude the discussion of the first three topics

1. Computer word length
2. Roundoff and truncation
3. Conditioning

consider what is available in the NAG library. The following chapters contain one or two line listings for each routine, with a total of more than 800 lines. Therefore, only the first listing or a small number of listings in each chapter is indicated below:

Chapter F01 - Matrix Factorizations
F01ABF - Inverse of real symmetric positive-definite matrix using iterative refinement (simplified parameter list)

Chapter F02 - Eigenvalue and Eigenvectors
F02BBF - Selected eigenvalues and eigenvectors of real symmetric matrix (Black Box)

Chapter F03 - Determinants
F03AAF - Determinant of real matrix (Black Box)

Chapter F04 - Simultaneous Linear Equations
F04AAF - Solution of real simultaneous linear equations with multiple right-hand sides (Black Box)

Chapter F05 - Orthogonalisation
F05AAF - Gram-Schmidt orthogonalisation of n vectors of order m

Chapter F06 - Linear Algebra Support Routines
F06BLF - Compute quotient of two real scalars, with overflow flag

Chapter F07 - Linear Equations (LAPACK)
F07ADF - LU factorization of real m by n matrix
F07AGF - Estimate condition number of real matrix, matrix already factorized by F07ADF (SGECON/DGECON)

Chapter F08 - Least Squares and Eigenvalue Problems
F08AEF - QR factorization of real general rectangular matrix

Chapter F11 - Sparse Linear Algebra
F11BAF - Real sparse nonsymmetric linear systems, set-up for F11BBF

Chapter X02 - Machine Constants
X02AHF - Largest permissible argument for SIN and COS
X02AJF - Machine precision
X02AKF - Smallest positive model number
X02ALF - Largest positive model number

These chapters contain many routines pertaining to linear algebra, including the calculation of the condition of the coefficient matrix, and basic constants that determine the performance of the routines (e.g., the precision reflected in the machine epsilon). A more complete discussion of linear algebra will be given later.

Considering the final topic in this section, **Convergence and error tolerances** the convergence of an iterative procedure, such as Newton's method, is of course required to arrive at a solution with acceptable accuracy. For example,

$$x^{i+1} = x^i - \frac{f(x^i)}{df(x^i)/dx}, i = 0, 1, \cdots$$

is continued until

$$\left| x^j - x^{j-1} \right| < \epsilon$$

The question, is what is a reasonable value for ϵ?

The choice of an error tolerance to determine when the iteration should be terminated, is crucial, and is one of the most common causes for failures in scientific computation.

Error tolerances are of two types: *absolute* and *relative*. Some considerations in selecting these are illustrated with the following examples:

$Variable$	$Range$	$Typeoferror$	$Errortolerance$
$temperature(^oK)$	$300-400$	abs	0.1
$concentration$	$0-1$	abs	0.001

For temperature, an error tolerance of 0.1 is reasonable, but it is too large for concentration. A tolerance of 0.001 is appropriate for concentration, but it is too stringent for temperature. Thus, using an absolute error tolerance for variables with widely different magnitudes causes computational problems.

The solution to this problem is generally to use a relative tolerance. For example,

$Variable$	$Range$	$Typeoferror$	$Errortolerance$
$temperature(^oK)$	$300-400$	rel	0.001
$concentration$	$0-1$	rel	0.001

A tolerance of 0.001 is appropriate for both variables. However, a relative error tolerance is not defined for a variable when it has a zero value.

Thus, a tolerance that is a combination of the two is often used, e.g.,

$$tol = relerr * y + abserr$$

tol is then compared with the estimated error from the numerical algorithm to determine if the solution has achieved a reasonable accuracy.

As another example of the distinction between absolute and relative error, in the least squares analysis of experimental data, if the data are measured with constant absolute error, the error will be large for small values of the measured variable. If the data are measured with constant relative error, the actual error is small for small values of the measured data. Thus, depending on the type of error, the reliability of the least squares fit will be significantly different for small values of the measured variable.

In summary, the careful selection of an error tolerance is often the determining factor for successfully computing a numerical solution. The thoughtless selection of a tolerance, which is tempting particularly when using a library routine, generally leads to a failed calculation.

Interpolation

1. Polynomials
2. Splines

Frequently in scientific and engineering analysis, we start with a set of numbers (e.g., experimental data, output from a complicated mathematical model), and we may then wish to perform some mathematical analysis on the numbers (e.g., approximate mathematical function, differentiation, integration).

But this implies that we have a mathematical function, when in fact, we have only numbers. Thus, is the mathematical analysis possible?

Yes, we can still do a mathematical analysis if we first develop a functional representation or approximation of the numbers, that is, develop a function that approximately fits the numbers in some sense. In this discussion, we will consider standard and spline polynomials (although there are many other possibilities).

Polynomials are selected because they are: (1) quite general and (2) can be manipulated mathematically relatively easily.

Consider a set of three data pairs

x	$f(x)$
x_1	$f(x_1)$
x_2	$f(x_2)$
x_3	$f(x_3)$

We can consider a functional approximation of these of the form

$$\begin{aligned} f(x) &= (x-x_2)(x-x_3)a_1 \\ &+(x-x_1)(x-x_3)a_2 \\ &+(x-x_1)(x-x_2)a_3 1 \end{aligned} \tag{12}$$

where a_1, a_2 and a_3 are constants to be determined. If we substitute $x = x_1$ in eq. (1),

$$f(x_1) = (x_1 - x_2)(x_1 - x_3)a_1 + 0a_2 + 0a_3$$

we can solve for a_1,

$$a_1 = \frac{f(x_1)}{(x_1 - x_2)(x_1 - x_3)}$$

Similarly, substituting $x = x_2$ in eq. (1) gives

$$a_2 = \frac{f(x_2)}{(x_2 - x_1)(x_2 - x_3)}$$

and finally, substituting $x = x_3$ in eq. (1) gives

$$a_3 = \frac{f(x_3)}{(x_3 - x_1)(x_3 - x_2)}$$

Substituting these coefficients in eq. (1), we arrive at the *second order Lagrange interpolation polynomial*

$$\begin{aligned} f(x) &= \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}f(x_1) \\ &+\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}f(x_2) \\ &+\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}f(x_3)2 \end{aligned} \tag{13}$$

Note that eq. (2) has three important properties:

- It passes through the points $(x_1, f(x_1))$,$(x_2, f(x_2))$,$(x_3, f(x_3))$ exactly, i.e., it returns the original data exactly.
- It can be applied to unequally spaced data, i.e., for $(x_1 - x_2) \neq (x_1 - x_3) \neq (x_2 - x_3)$.
- Since it is a polynomial, it can be easily integrated, differentiatiated, and generally manipulated relatively easily mathematically.

However, there is no guarantee that it is well behaved between the data points, and of course, it is limited to just three data pairs.

If we have more than three data pairs, we can still use a polynomial. For example, if we start with the model

$$y(x) = a_0 + a_1 x + a_2 x^2$$

then a sum of squared errors for n data pairs gives

$$E(a_0, a_1 a_2) = \sum_{i=1}^{n} [y_i - y(x_i)]^2 = \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2)]^2$$

To minimize E

$$\frac{\partial E}{\partial a_0} = 2 \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2)](-1) = 0$$

$$\frac{\partial E}{\partial a_1} = 2 \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2)](-x_i) = 0$$

$$\frac{\partial E}{\partial a_2} = 2 \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2)](-x_i^2) = 0$$

or

$$a_0 \Sigma_{i=1}^n 1 + a_1 \Sigma_{i=1}^n x_i + a_2 \Sigma_{i=1}^n x_i^2 = \Sigma_{i=1}^n y_i$$
$$a_0 \Sigma_{i=1}^n x_i + a_1 \Sigma_{i=1}^n x_i^2 + a_2 \Sigma_{i=1}^n x_i^3 = \Sigma_{i=1}^n x_i y_i$$
$$a_0 \Sigma_{i=1}^n x_i^2 + a_1 \Sigma_{i=1}^n x_i^3 + a_2 \Sigma_{i=1}^n x_i^4 = \Sigma_{i=1}^n x_i^2 y_i$$

which is a $3x3$ linear system for the three coefficients a_0, a_1, a_2. The number of data pairs, n, is arbitrary, and the resulting third order polynomial will pass through the data 11smoothly" (in the least squares sense).

Other models are possible, for example:

$$y(x) = \sum_{j=0}^{m} a_j x^j (polynomialofdegree$$

m)

$$y(x) = a_1 x + a_2 x^2 (linear, includestheorigin)$$

$$y(x) = a_0 + a_1 x + a_2 e^{b_0 x} (linear,$$

b$_0$ fixed)

$$y(x) = a_0 + a_1 x + a_2 e^{b_0 x} (nonlinear,$$

b$_0$ variable)
For the last model, we would have to solve nonlinear equations (to be considered subsequently).

Splines

Splines are piecewise polynomials, selected to ensure continuity not only in the dependent variable where the pieces connect, but also in their derivatives. For example, when using cubic splines of the form

$$f(x) = a_0 + a_1(x - x_i) + a_2(x - x_i)^2 + a_3(x - x_i)^3$$

we require that two splines coming together at point x_i have continuity in $f(x), df(x)/dx$ and $d^2f(x)/dx^2$. These three continuity requirements are applied at each of the points, $x_1, x_2, \cdots, x_N$ to evaluate the coefficients a_1, a_2 and a_3 at each of the points (also, we have $a_0 = f(x_i)$ to ensure the splines go through the given points). This procedure leads to a system of tridiagonal equations that is solved for a_1, a_2, a_3 at each of the points, $x_1, x_2, \cdots, x_N$.

The solution of the tridiagonal system can be done very efficiently because the coefficient matrix is:

- Tridiagonal (so storage is only $3xn$)
- Symmetric
- Nonsingular
- Well conditioned
- Diagonally dominant (pivoting not required with Gaussian elimination)

Thus, updating the spline as the calculation proceeds can be done efficiently, e.g., in the solution of PDEs.

NAG includes the following routines for functional approximation:

Chapter E01 - Interpolation

E01AAF - Interpolated values, Aitken's technique, unequally spaced data, one variable

E01ABF - Interpolated values, Everett's formula, equally spaced data, one variable

E01AEF - Interpolating functions, polynomial interpolant, data may include , derivative values, one variable

E01BAF - Interpolating functions, cubic spline interpolant, one variable

E01BEF - Interpolating functions, monotonicity-preserving, piecewise cubic Hermite, one variable

E01BFF - Interpolated values, interpolant computed by E01BEF, function only, one variable

E01BGF - Interpolated values, interpolant computed by E01BEF, function and 1st derivative, one variable

E01BHF - Interpolated values, interpolant computed by E01BEF, definite integral, one variable

E01DAF - Interpolating functions, fitting bicubic spline, data on rectangular grid

E01RAF - Interpolating functions, rational interpolant, one variable

E01RBF - Interpolated values, evaluate rational interpolant computed by E01RAF, one variable

E01SAF - Interpolating functions, method of Renka and Cline, two variables

E01SBF - Interpolated values, evaluate interpolant computed by E01SAF, two variables

E01SEF - Interpolating functions, modified Shepard's method, two variables

E01SFF - Interpolated values, evaluate interpolant computed by E01SEF, two variables

E01SGF - Interpolating functions, modified Shepard's method, two variables

E01SHF - Interpolated values, evaluate interpolant computed by E01SGF, functiona nd first derivatives, two variables

E01TGF - Interpolating functions, modified Shepard's method, three variables

E01THF - Interpolated values, evaluate interpolant computed by E01TGF, function and first derivatives, three variables

Chapter E02 - Curve and Surface Fitting

E02ACF - Minimax curve fit by polynomials

E02ADF - Least-squares curve fit, by polynomials, arbitrary data points

E02AEF - Evaluation of fitted polynomial in one variable from Chebyshev series form (simplified parameter list)

E02AFF - Least-squares polynomial fit, special data points (including interpolation)

E02AGF - Least-squares polynomial fit, values and derivatives may be constrained, arbitrary data points,

E02AHF - Derivative of fitted polynomial in Chebyshev series form

E02AJF - Integral of fitted polynomial in Chebyshev series form

E02AKF - Evaluation of fitted polynomial in one variable, from Chebyshev series form

E02BAF - Least-squares curve cubic spline fit (including interpolation)

E02BBF - Evaluation of fitted cubic spline, function only

E02BCF - Evaluation of fitted cubic spline, function and derivatives

E02BDF - Evaluation of fitted cubic spline, definite integral

E02BEF - Least-squares cubic spline curve fit, automatic knot placement

E02CAF - Least-squares surface fit by polynomials, data on lines

E02CBF - Evaluation of fitted polynomial in two variables

E02DAF - Least-squares surface fit, bicubic splines

E02DCF - Least-squares surface fit by bicubic splines with automatic knot placement, data on rectangular grid

E02DDF - Least-squares surface fit by bicubic splines with automatic knot placement, scattered data

E02DEF - Evaluation of a fitted bicubic spline at a vector of points

E02DFF - Evaluation of a fitted bicubic spline at a mesh of points

E02GAF - L(1)-approximation by general linear function

E02GBF - L(1)-approximation by general linear function subject to linear inequality constraints

E02GCF - L(infinity)-approximation by general linear function

E02RAF - Pade-approximants

E02RBF - Evaluation of fitted rational function as computed by E02RAF

E02ZAF - Sort 2-D data into panels for fitting bicubic splines

Chapter E04 - Minimizing or Maximizing a Function
(routines not listed)

Integration

1. Polynomials
2. Gaussian quadrature
3. Splines
4. Adaptive

Returning to the second order Lagrange interpolation polynomial integration of $f(x)$,

$$I = \int_{x_1}^{x_3} f(x)dz$$

$$= \int_{x_1}^{x_3} \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} f(x_1) + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} f(x_3)dx$$

Integrating the second order polynomials, we arrive at a quadrature rule (numerical integration rule) for nonuniformly spaced data (which can be applied over successive sets of three points or two intervals).

For the special case of a uniform grid, $(x_2 - x_1) = (x_3 - x_2) = \Delta x$, I becomes

$$I = \int_{x_1}^{x_3} f(x)dx = \frac{\Delta x}{3}(f(x_1) + 4f(x_2) + f(x_3))$$

which is **Simpson's rule.**

Some important details about Simpson's rule:

- An even number of "panels" (odd number of grid points) is required. In other words, Simpson's rule can be extended to n points as

$$
\begin{aligned}
I = \int_{x_1}^{x_n} f(x)dx = & \frac{\Delta x}{3}(f(x_1) + 4f(x_2) + f(x_3)) \\
& + \frac{\Delta x}{3}(f(x_3) + 4f(x_4) + f(x_5)) \\
& + \cdots + \frac{\Delta x}{3}(f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)) \\
= & \frac{\Delta x}{3}(f(x_1) + 4f(x_2) + 2f(x_3) + 4f(x_4) + \cdots \\
& + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n))1
\end{aligned}
\tag{14}
$$

- Eq. (1) is surprisingly accurate. In fact, it is **exact for third order polynomials**, even thought it is derived using a second order polynomial approximation for the integrand (in general, for equally spaced points, quadrature formulas based on a polynomial of order $2n$ are exact for polynomials of order $2n+1$.

- Although eq. (1) is generally sufficiently accurate (this, of course, depends on Δx) it is based on equally spaced data (constant Δx).

Even more accurate integrals can be computed using unequally spaced data or points; the best example of this idea is **Gaussian quadrature**:

- The spacing of the points is selected to give an exact intergal for a polynomial of maximum order. Specifically, integrating a polynomial of order $n-1$ gives an integral that is exact for polynomials of order $2n-1$. For example, with $n=2$ $n-1=2-1=1$ is a linear approximation of the integrand, but the computed integral is exact for a polynomial of order $2n-1=4-1=3$!

- The spacing of the points generally corresponds to the roots of an orthogonal polynomial. The location of the points is determined by the number of points and the limits of integration, i.e., $a \leq x \leq b, 0 \leq x \leq \infty, -\infty \leq x \leq \infty$.

Gaussian quadrature, as other quadrature rules (rectangular, trapezoidal, Simpson's rules) is based on a weighted sum

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Now, however, we are going to choose the points x_i as well as the weights w_i to get a numerical integral that is exact for a polynomial of as high degree as possible (degree $2n-1$ when $f(x)$ is a polynomial of order $n-1$ - the basic idea of Gaussian quadrature). A short table of points and weights follows (Borse, 1997):

n	i	x_i	w_i
2	1	0.5773502692	1.0
3	0	0.0	0.8888888889
	1	0.7745966692	0.5555555556
5	0	0.0	0.5688888889
	1	0.5384693101	0.4786286705
	2	0.9061798459	0.2369268850

The points are in the interval $-1 \leq x_i \leq 1$ (they are symmetric around $x = 0$). Thus, the limits of the integral must be $a = -1$,$b = 1$). This can easily be achieved by a linear change of variables

$$I = \int_a^b f(x)dx = \int_{-1}^1 f(\lambda)d\lambda$$

where

$$x = \frac{b-a}{2}\lambda + \frac{b+a}{2}\lambda$$

As an example,

$$I = \int_0^2 \sin x dx$$

Then, $x = \lambda + 1$, and the integral in λ is

$$I = \int_{-1}^1 \sin(\lambda + 1)d\lambda$$

Then, by Gaussian quadrature for $n = 5$,

$$I \approx w_0 f(0) + w_1 \left[f(x_1) + f(-x_1)\right] + w_2 \left[f(x_2) + f(-x_2)\right]$$

$$= 0.5688888889 \sin(1.0)$$

$$+0.4786286705 \left[\sin(0.5384693101 + 1)\right] + \sin(-0.5384693101 + 1)$$

$$+0.2369268850 \left[\sin(0.9061798459 + 1)\right] + \sin(-0.9061798459 + 1)$$

$$= 1.4161467; exactanswer = 1 - \cos(2) = 1.4161468 \ldots$$

The method can also be applied to integrals with infinite limits by considering a product integrand

$$\int_a^b g(x)f(x)dx \approx \sum_{i=1}^n w_i f(x_i)$$

Then, the methods are summarized as

$Integraltype$	$g(x)$	$[a,b]$	$Methodname$
$\int_{-1}^{1} f(x)dx$	1	$(-1,1)$	$Gauss-Legendre$
$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}}dx$	$(1-x^2)^{-1/2}$	$(-1,1)$	$Gauss-Chebyshev$
$\int_{0}^{\infty} x^q e^{-x} f(x)dx$	$x^q e^{-x}$	$(0,\infty)$	$Gauss-Laguerre$
$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx$	e^{-x^2}	$(-\infty,\infty)$	$Gauss-Hermite$

The choice of a method is determined primarily by the limits of the integral. The name indicates the orthogonal polynomial for which the roots determine the points (we previously used Gauss-Legendre, and the tabulated values of x_i are the roots of the Legendre polynomial of order n).

Borse, G. J. (1997), *Numerical Methods with Matlab; A Resource for Scientists and Engineers*, PWS Publishing Company, Boston

We can also use the spline as an approximation for an integrand, that is **spline quadrature**

$$I = \int_{x_0}^{x_n} f(x)$$

with

$$f(x) = a_0 + a_1(x-x_i) + a_2(x-x_i)^2 + a_3(x-x_i)^3$$

The integration is easily carried out, and the spline coefficients can be evaluated by the solution of tridiagonal equations, as described previously.

Another approach to improving accuracy is to have the code sense the rate of change of $f(x)$, then insert points automatically in regions of high rate of change. This is termed **adaptive quadrature**. Consider for example the algorithm

```
1.              $Select h, I_0 = 0, i = 0$
2.              $If x_i > x_n stop$
3.              $I^p_{i+1} = I_i + f(x_i)h,$
4.              $\varepsilon = [f(x_{i+1}) - f(x_i)]\, h/2$
5.              $If \varepsilon > tol, h = h/2, goto 3$
6.  $If \varepsilon \leq tol, I^c_{i+1} = I^p_{i+1} + \varepsilon, i = i + 1, x_i = x_i + h goto 2$
```

This is an adaptive trapezoidal rule.

Adaptive quadrature is readily available in a series of library routines. For example, QUANC8 from Forsythe, Malcolm and Moler (1977) is named from Quadrature, Adaptive, Newton-Cotes 8-panel. A typical call to QUANC8 is

```
CALL QUANC8(FUN,A,B,ABSERR,RELERR,RESULT,ERREST,NOFUN,FLAG
```

where

FUN	External to compute the integrand (input)
A	Lower limit of integration (input)
B	Upper limit of integration (input)
ABSERR	Absolute error tolerance (input)
RELERR	Relative error tolerance (input)
RESULT	Computed integral (output)
ERREST	Estimated error for RESULT (output)

NOFUN	Number of integrand evaluations (output)
FLAG	Error flag indicating where calculation failed (output)

Matlab has an implementation of QUANC8, as well as an adaptive Simpson's rule routine:

- quad uses an adaptive, recursive Simpson's rule; the call is

  ```
  q = quad('function',a,b,tol)
  ```

- quad8 implements an adaptive recursive Newton-Cotes 8 panel rule (as in QUANC8); the call is

  ```
  q=quad8('function',a,b,tol)
  ```

The routines in NAG for quadrature are:

Chapter D01 - Quadrature
D01AHF - 1-D quadrature, adaptive, finite interval, strategy due to Patterson, suitable for well-behaved integrands

D01AJF - 1-D quadrature, adaptive, finite interval, strategy due to Piessens and deDoncker, allowing for badly-behaved integrands

D01AKF - 1-D quadrature, adaptive, finite interval, method suitable for oscillating functions

D01ALF - 1-D quadrature, adaptive, finite interval, allowing for singularities at user-specified break-points

D01AMF - 1-D quadrature, adaptive, infinite or semi-infinite interval

D01ANF - 1-D quadrature, adaptive, finite interval, weight function cos(wx) or sin(wx)

D01APF - 1-D quadrature, adaptive, finite interval, weight function with end-point singularities of algebraico-logarithmic type

D01AQF - 1-D quadrature, adaptive, finite interval, weight function 1/(x-c), Cauchy principal value (Hilbert transform)

D01ARF - 1-D quadrature, non-adaptive, finite interval with provision for indefinite integrals

D01ASF - 1-D quadrature, adaptive, semi-infinite interval, weight function cos(wx) or sin(wx)

D01ATF - 1-D quadrature, adaptive, finite interval, variant of D01AJF efficient on vector machines

D01AUF - 1-D quadrature, adaptive, finite interval, variant of D01AKF efficient on vector machines

D01BAF - 1-D Gaussian quadrature

D01BBF - Pre-computed weights and abscissae for Gaussian quadrature rules, restricted choice of rule

D01BCF - Calculation of weights and abscissae for Gaussian quadrature rules, general choice of rule

D01BDF - 1-D quadrature, non-adaptive, finite interval

D01DAF - 2-D quadrature, finite region

D01EAF - Multi-dimensional adaptive quadrature over hyper-rectangle, multiple integrands

D01FBF - Multi-dimensional Gaussian quadrature over hyper-rectangle

D01FCF - Multi-dimensional adaptive quadrature over hyper-rectangle

D01FDF - Multi-dimensional quadrature, Sag-Szekeres method, general product region or n-sphere

D01GAF - 1-D quadrature, integration of function defined by data values, Gill-Miller method

D01GBF - Multi-dimensional quadrature over hyper-rectangle, Monte Carlo method

D01GCF - Multi-dimensional quadrature, general product region, number-theoretic method

D01GDF - Multi-dimensional quadrature, general product region, number-theoretic method, variant of D01GCF efficient on vector machines

D01GYF - Korobov optimal coefficients for use in D01GCF or D01GDF, when number of points is prime

D01GZF - Korobov optimal coefficients for use in D01GCF or D01GDF, when number of points is product of two primes

D01JAF - Multi-dimensional quadrature over an n-sphere, allowing for badly-behaved integrands

D01PAF - Multi-dimensional quadrature over an n-simplex

Differentiation

1. Finite differences
2. Symbolic computation
3. Automatic differentiation (to be discussed later)

Returning to the Lagrange second order interpolation polynomial

$$\begin{aligned} f(x) = {} & \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} f(x_1) \\ & + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} f(x_2) \\ & + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} f(x_3) 2 \end{aligned} \tag{15}$$

we can differentiate eq. (2) to obtain an approximation to the derivative of $f(x)$

$$\begin{aligned} \frac{df(x)}{dx} = {} & \frac{f(x_1)}{(x_1-x_2)(x_1-x_3)} \{(x-x_2)+(x-x_3)\} \\ & + \frac{f(x_2)}{(x_2-x_1)(x_2-x_3)} \{(x-x_1)+(x-x_3)\} \\ & + \frac{f(x_3)}{(x_3-x_1)(x_3-x_2)} \{(x-x_1)+(x-x_2)\} \, 3 \end{aligned} \tag{16}$$

Eq. (3) could then be used to calculate the derivative at any $x_1 \leq x \leq x_3$. In particular,

$$\frac{df(x_1)}{dx} = \frac{f(x_1)}{(x_1-x_2)(x_1-x_3)} \{(x_1-x_2)+(x_1-x_3)\}$$

$$+\frac{f(x_2)}{(x_2-x_1)(x_2-x_3)}(x_1-x_3)$$

$$+\frac{f(x_3)}{(x_3-x_1)(x_3-x_2)}(x_1-x_2)4a \tag{17}$$

$$\frac{df(x_2)}{dx}=\frac{f(x_1)}{(x_1-x_2)(x_1-x_3)}(x_2-x_3)$$

$$+\frac{f(x_2)}{(x_2-x_1)(x_2-x_3)}\left\{(x_2-x_1)+(x_2-x_3)\right\}$$

$$+\frac{f(x_3)}{(x_3-x_1)(x_3-x_2)}(x_2-x_1)4b \tag{18}$$

$$\frac{df(x_3)}{dx}=\frac{f(x_1)}{(x_1-x_2)(x_1-x_3)}(x_3-x_2)$$

$$+\frac{f(x_2)}{(x_2-x_1)(x_2-x_3)}(x_3-x_1)$$

$$+\frac{f(x_3)}{(x_3-x_1)(x_3-x_2)}\left\{(x_3-x_1)+(x_3-x_2)\right\}4c \tag{19}$$

Note that the RHSs of eqs. (4a) to (4c) are just constants, and therefore eqs. (4) can be considered as a **three point differentiation matrix on a nonuniform grid**.

If we consider the special case of a uniform grid, $\Delta x = x_2 - x_1 = x_3 - x_2$; $2\Delta x = x_3 - x_1$, eqs. (4) become

$$\frac{df(x_1)}{dx}=\frac{f(x_1)}{(x_1-x_2)(x_1-x_3)}\left\{(x_1-x_2)+(x_1-x_3)\right\}$$

$$+\frac{f(x_2)}{(x_2-x_1)(x_2-x_3)}(x_1-x_3)$$
$$+\frac{f(x_3)}{(x_3-x_1)(x_3-x_2)}(x_1-x_2)$$

$$=\frac{f(x_1)}{(-\Delta x)(-2\Delta x)}\left\{-\Delta x-2\Delta x\right\}$$
$$+\frac{f(x_2)}{(\Delta x)(-\Delta x)}(-2\Delta x)$$
$$+\frac{f(x_3)}{(2\Delta x)(\Delta x)}(-\Delta x)$$

$$=\frac{-3}{2\Delta x}f(x_1)$$
$$+\frac{4}{2\Delta x}f(x_2)$$
$$+\frac{-1}{2\Delta x}f(x_3)$$

$$=\frac{1}{2!\Delta x}\begin{pmatrix}-3 & 4 & -1\end{pmatrix}\mathbf{f}$$

$$\frac{df(x_2)}{dx}=\frac{f(x_1)}{(x_1-x_2)(x_1-x_3)}(x_2-x_3)$$
$$+\frac{f(x_2)}{(x_2-x_1)(x_2-x_3)}\left\{(x_2-x_1)+(x_2-x_3)\right\}$$
$$+\frac{f(x_3)}{(x_3-x_1)(x_3-x_2)}(x_2-x_1)$$

$$= \frac{f(x_1)}{(-\Delta x)(-2\Delta x)}(-\Delta x)$$

$$+\frac{f(x_2)}{(\Delta x)(-\Delta x)}(\Delta x - \Delta x)$$

$$+\frac{f(x_3)}{(2\Delta x)(\Delta x)}(\Delta x)$$

$$= -\frac{1}{2\Delta x}f(x_1) + 0f(x_2) + \frac{1}{2\Delta x}f(x_3)$$

$$= \frac{1}{2!\Delta x}\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}\mathbf{f}$$

$$\frac{df(x_3)}{dx} = \frac{f(x_1)}{(x_1 - x_2)(x_1 - x_3)}(x_3 - x_2)$$

$$+\frac{f(x_2)}{(x_2 - x_1)(x_2 - x_3)}(x_3 - x_1)$$

$$+\frac{f(x_3)}{(x_3 - x_1)(x_3 - x_2)}\left\{(x_3 - x_1) + (x_3 - x_2)\right\}$$

$$= \frac{f(x_1)}{(-\Delta x)(-2\Delta x)}(\Delta x)$$

$$+\frac{f(x_2)}{(\Delta x)(-\Delta x)}(2\Delta x)$$

$$+\frac{f(x_3)}{(2\Delta x)(\Delta x)}(2\Delta x + \Delta x)$$

$$= \frac{1}{2\Delta x}f(x_1) - \frac{4}{2\Delta x}f(x_2) + \frac{3}{2\Delta x}f(x_3)$$

$$= \frac{1}{2!\Delta x}\begin{pmatrix} 1 & -4 & 3 \end{pmatrix}\mathbf{f}$$

Combining results, we arrive at the three point, uniformly spaced differentiation matrix

$$\frac{d\mathbf{f}}{dx} = \frac{1}{2!\Delta x}\begin{bmatrix} -3 & 4 & -1 \\ -1 & 0 & 1 \\ 1 & -4 & 3 \end{bmatrix}\mathbf{f}+O(\Delta x^2)$$

This matrix will be used later in the numerical integration of PDEs.

A Fortran subroutine, weights.for, by Bengt Fornberg, is available for the convenient calculation of the weights in differentiation formulas. For example, the following fourth order, differentiation matrix is easily obtained using weights.for

$$\frac{d\mathbf{u}}{dx} =$$

$$\frac{1}{4!\Delta x}\begin{bmatrix} -50 & 96 & -72 & 32 & -6 & i=1 \\ -6 & -20 & 36 & -12 & 2 & i=2 \\ 2 & -16 & 0 & 16 & -2 & i=3,\cdots,N-2 \\ -2 & 12 & -36 & 20 & 6 & i=N-1 \\ 6 & -32 & 72 & -96 & 50 & i=N \end{bmatrix}\mathbf{u}$$

$$+O(\Delta x^4)$$

Note that in these two differentiation matrices, the weighting coefficients in any give row sum to zero, which is required to correctrly differentiate a constant (to zero).

Also, the second order matrix differentiates a second order polynomial exactly, and the fourth order matrix differentiates a fourth order matrix exactly. For example, using the second row of the second order matrix,

$$\frac{df(x)}{dx} = \frac{d\left(a_0 + a_1 x + a_2 x^2\right)}{dx}$$

$$= \frac{-1[a_0 + a_1(x - \Delta x) + a_2(x - \Delta x)^2]}{2\Delta x}$$

$$+ \frac{1[a_0 + a_1(x + \Delta x) + a_2(x + \Delta x)^2]}{2\Delta x}$$

$$= \frac{2a_1\Delta x + 4a_2 x\Delta x}{2\Delta x} = a_1 + 2a_2 x$$

as expected.

We will later use the formulas defined by these differentiation matrices for first order derivatives, as well as for second derivatives, in the numerical integration of PDEs.

Subroutine weights.for can be used to compute the finite difference (FD) weights for:

- Derivatives of any order
- Approximations of any order
- Any grid point
- Equal and unequally spaced grids

NAG has the following routine for the calculation of derivatives:

D04AAF - Numerical differentiation, derivatives up to order 14, function of one real variable

Symbolic computation (analytical differentiation) is also possible using any of a number of computer algebra systems (CAS) such as Maple and Mathematica. Symbolic (exact) mathematics of other types is also possible with these systems, partial differentiation, integration, linear algebra, etc.

As an example of symbolic differentiation with Maple, consider (Nicolaides and Walkington (1996))

```
>u:=x*tan(x*y);

                    u:=x tan(x y)

>diff(u,x);
                                          2
           tan(x y) + x (1 + tan(x y) ) y

>diff(u,y):
                 2               2
                x  (1 + tan( x y) )

>diff(u,x,y);
                 2         2
    2 (1 + tan(x y) ) x + 2 x  tan(x y)

               2
    (1 + tan(x y) ) y

>diff(x^2*y^3,y$3);
                        2
                     6 x
```

Briefly, here are some examples of integration:

```
Indefinite integration:

>int(sin(ln(x)),x);

           1/2 x sin(ln(x)) - cos(ln(x)))

Definite integration:

>int(x*ln(sin(x)),x=0..Pi);

                        2
                 - 1/2 Pi   ln(2)

Numerical integration:

>evalf(int(x*ln(sin(x)),x=0..Pi);

                   -3.420544232

>evalf(-Pi^2*ln(2)/2);
```

```
-3.420544233
```

The integration is done by adaptive quadrature for 10-figure accuracy (the default for all Maple calculations). This can lead to some very long computation times. If the accuracy is reduced to 4-5 figures, the integration can be much faster.

In other words, Maple has variable precision arithmetic, so in principle, calculations of arbitrary precision can be performed (at the price of greater computation time). Thus, Maple has, in effect, a variable machine epsilon.

Additionally, Maple and NAG have recently been combined. Matlab has a "NAG foundation toolbox" with about 50 basic NAG routines.

Nicolaides, R., and N. Walkington (1996), *Maple: A Comprehensive Introduction*, Cambridge University Press, Cambridge

References and Software

1. Forsythe, et al (1977)
2. Kahaner, et al (1989)
3. Golub, et al (1992)
4. Press, et al (1992)
5. NAG Library, Mark 18 Release