Introduction to Numerical Methods
with examples in Javascript

D.V. Fedorov
Department of Physics and Astronomy
Aarhus University
8000 Aarhus C, Denmark

[1]http://en.wikipedia.org/wiki/GPL
[2]http://en.wikipedia.org/wiki/CC-BY-SA

Preface

This book evolved from lecture notes developed over several years of teaching numerical methods at the University of Aarhus. It contains short descriptions of the most common numerical methods together with program examples written in Javascript. The latter was chosen simply because the it seems concise and intuitive to me. The program examples are not tested or optimized in any way other than to fit on one page of the book.

The text of the book is free as in freedom. You are permitted to copy and redistribute the book in original or modified form either gratis or for a fee. However, you must attribute the original author(s) and pass the same freedoms to all recipients of your copies[3].

2010
Dmitri Fedorov

[3] see the GPL or CC-BY-SA licenses for more details.

Contents

Chapter 1

Linear equations

A system of linear equations is a set of linear algebraic equations generally written in the form

$$\sum_{j=1}^{n} A_{ij}x_j = b_i \ , \ i = 1 \dots m \ , \tag{1.1}$$

where $x_1, x_2, \dots, x_n$ are the unknown variables, $A_{11}, A_{12}, \dots, A_{mn}$ are the (constant) coefficients of the system, and $b_1, b_2, \dots, b_m$ are the (constant) right-hand side terms.

The system can be written in matrix form as

$$A\mathbf{x} = \mathbf{b} \ . \tag{1.2}$$

where $A \doteq \{A_{ij}\}$ is the $m \times n$ matrix of the coefficients, $\mathbf{x} \doteq \{x_j\}$ is the size-n column-vector of the unknown variables, and $\mathbf{b} \doteq \{b_i\}$ is the size-m column-vector of right-hand side terms.

Systems of linear equations occur regularly in applied mathematics. Therefore the computational algorithms for finding solutions of linear systems are an important part of numerical methods.

A system of non-linear equations can often be approximated by a linear system, a helpful technique (called *linearization*) in creating a mathematical model of an otherwise a more complex system.

If $m = n$, the matrix A is called *square*. A square system has a unique solution if A is invertible.

1.1 Triangular systems and back-substitution

An efficient algorithm to solve a square system of linear equations numerically is to transform the original system into an equivalent *triangular system*,

$$T\mathbf{y} = \mathbf{c} \ , \tag{1.3}$$

where T is a *triangular matrix*: a special kind of square matrix where the matrix elements either below or above the main diagonal are zero.

An upper triangular system can be readily solved by *back substitution*:

$$y_i = \frac{1}{T_{ii}}\left(c_i - \sum_{k=i+1}^{n} T_{ik}y_k\right) , \ i = n, \dots, 1 \ . \tag{1.4}$$

For the lower triangular system the equivalent procedure is called *forward substitution.*

Note that a diagonal matrix – that is, a square matrix in which the elements outside the main diagonal are all zero – is also a triangular matrix.

1.2 Reduction to triangular form

Popular algorithms for transforming a square system to triangular form are *LU decomposition* and *QR decomposition.*

1.2.1 LU decomposition

LU decomposition is a factorization of a square matrix into a product of a lower triangular matrix L and an upper triangular matrix U,

$$A = LU \; . \tag{1.5}$$

The linear system $A\mathbf{x} = \mathbf{b}$ after LU-decomposition of the matrix A becomes $LU\mathbf{x} = \mathbf{b}$ and can be solved by first solving $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ and then $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ with two runs of forward and backward substitutions.

If A is a $n \times n$ matrix, the condition (1.5) is a set of n^2 equations,

$$\sum_{k=1}^{n} L_{ik}U_{kj} = A_{ij} \; , \tag{1.6}$$

for $n^2 + n$ unknown elements of the triangular matrices L and U. The decomposition is thus not unique.

Usually the decomposition is made unique by providing extra n conditions e.g. by the requirement that the elements of the main diagonal of the matrix L are equal one, $L_{ii} = 1$, $i = 1 \dots n$. The system (1.6) can then be easily solved row after row using e.g. the *Doolittle algorithm*,

```
for i = 1 to n :
  L_ii = 1
  for j = 1 to i − 1 :
    L_ij = ( A_ij − Σ_{k<j} L_ik U_kj ) / U_jj
  for j = i to n :
    U_ij = A_ij − Σ_{k<i} L_ik U_kj
```

1.2.2 QR decomposition

QR decomposition is a factorization of a matrix into a product of an orthogonal matrix Q, such that $Q^TQ = 1$ (where ^{T} denotes transposition), and a right triangular matrix R,

$$A = QR \; . \tag{1.7}$$

QR-decomposition can be used to convert the linear system $A\mathbf{x} = \mathbf{b}$ into the triangular form

$$R\mathbf{x} = Q^T\mathbf{b}, \tag{1.8}$$

which can be solved directly by back-substitution.

QR-decomposition can also be performed on non-square matrices with few long columns. Generally speaking a rectangular $n \times m$ matrix A can be represented as a product, $A = QR$, of an orthogonal $n \times m$ matrix Q, $Q^T Q = 1$, and a right-triangular $m \times m$ matrix R.

QR decomposition of a matrix can be computed using several methods, such as Gram-Schmidt orthogonalization, Householder transformations, or Givens rotations.

Gram-Schmidt orthogonalization

Gram-Schmidt orthogonalization is an algorithm for orthogonalization of a set of vectors in a given inner product space. It takes a linearly independent set of vectors $A = \{\mathbf{a}_1, \dots, \mathbf{a}_m\}$ and generates an orthogonal set $Q = \{\mathbf{q}_1, \dots, \mathbf{q}_m\}$ which spans the same subspace as A. The algorithm is given as

```
for i = 1 to m
  q_i ← a_i/||a_i||  (normalization)
  for j = i+1 to m
    a_j ← a_j − ⟨a_j, q_i⟩q_i  (orthogonalization)
```

where $\langle \mathbf{a}, \mathbf{b} \rangle$ is the inner product of two vectors, and $\|\mathbf{a}\| = \sqrt{\langle \mathbf{a}, \mathbf{a} \rangle}$ is the vector's norm. This variant of the algorithm, where all remaining vectors $\mathbf{a}_j$ are made orthogonal to $\mathbf{q}_i$ as soon as the latter is calculated, is considered to be numerically stable and is referred to as *stabilized* or *modified*.

Stabilized Gram-Schmidt orthogonalization can be used to compute QR decomposition of a matrix A by orthogonalization of its column-vectors $\mathbf{a}_i$ with the inner product

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^T \mathbf{b} \equiv \sum_{k=1}^{n} (\mathbf{a})_k (\mathbf{b})_k \,, \tag{1.9}$$

where n is the length of column-vectors $\mathbf{a}$ and $\mathbf{b}$, and $(\mathbf{a})_k$ is the kth element of the column-vector.

```
input:  matrix A = {a_1,...,a_m}  (destroyed)
output:  matrices R, Q = {q_1,...,q_m}:  A = QR
for i = 1...m
  R_ii = (a_i^T a_i)^(1/2)
  q_i = a_i/R_ii
  for j = i+1...m
    R_ij = q_i^T a_j
    a_j = a_j − q_i R_ij
```

The factorization is unique under requirement that the diagonal elements of R are positive. For a $n \times m$ matrix the complexity of the algorithm is $O(m^2 n)$.

1.3 Determinant of a matrix

LU- and QR-decompositions allow $O(n^3)$ calculation of the determinant of a square matrix. Indeed, for the LU-decomposition,

$$\det A = \det LU = \det L \det U = \det U = \prod_{i=1}^{n} U_{ii} \,. \tag{1.10}$$

For the QR-decomposition

$$\det A = \det QR = \det Q \det R \,. \tag{1.11}$$

Since Q is an orthogonal matrix $(\det Q)^2 = 1$ and therefore

$$|\det A| = |\det R| = \left|\prod_{i=1}^{n} R_{ii}\right| \,. \tag{1.12}$$

1.4 Matrix inverse

The inverse A^{-1} of a square $n \times n$ matrix A can be calculated by solving n linear equations $A\mathbf{x}_i = \mathbf{z}_i$, $i = 1 \dots n$, where $\mathbf{z}_i$ is a column where all elements are equal zero except for the element number i, which is equal one. The matrix made of columns $\mathbf{x}_i$ is apparently the inverse of A.

1.5 JavaScript implementations

```
function qrdec(A){ // QR-decomposition A=QR of matrix A
  var m=A.length, dot = function(a,b){
    var s=0; for(var i in a) s+=a[i]*b[i]; return s;}
  var R=[[0 for (i in A)] for (j in A)];
  var Q=[[A[i][j] for (j in A[0])] for(i in A)]; //Q is a copy of A
  for(var i=0;i<m;i++){
    var e=Q[i], r=Math.sqrt(dot(e,e));
    if(r==0) throw "qrdec: singular matrix"
    R[i][i]=r;
    for(var k in e) e[k]/=r; //normalization
    for(var j=i+1;j<m;j++){
      var q=Q[j], s=dot(e,q);
      for(var k in q) q[k]-=s*e[k]; //orthogonalization
      R[j][i]=s; } }
  return [Q,R]; } //end qrdec
```

```
function qrback(Q,R,b){ // QR-backsubstitution
// input: matrices Q,R, array b; output: array x such that QRx=b
  var m = Q.length, c = new Array(m), x = new Array(m);
  for(var i in Q){ // c = Q^T b
    c[i]=0; for(var k in b) c[i]+=Q[i][k]*b[k]; }
  for(var i=m-1;i>=0;i--){ // backsubstitution
    for(var s=0, k=i+1;k<m;k++) s+=R[k][i]*x[k];
    x[i]=(c[i]-s)/R[i][i]; }
  return x; } // end qrback
```

```
function inverse(A){ // calculates inverse of matrix A
  var [Q,R]=qrdec(A);
  return [qrback(Q,R,[(k == i?1:0) for(k in A)]) for(i in A)]; } //
      end inverse
```

Chapter 2

Interpolation

In practice one often meets a situation where the function of interest, $f(x)$, is only given as a discrete set of n tabulated points, $\{x_i, y_i = f(x_i) \mid i = 1 \dots n\}$, as obtained for example by sampling, experimentation, or expensive numerical calculations.

Interpolation means constructing a (smooth) function, called *interpolating function*, which passes exactly through the given points and hopefully approximates the tabulated function in between the tabulated points. Interpolation is a specific case of *curve fitting* in which the fitting function must go exactly through the data points.

The interpolating function can be used for different practical needs like estimating the tabulated function between the tabulated points and estimating the derivatives and integrals involving the tabulated function.

2.1 Polynomial interpolation

Polynomial interpolation uses a polynomial as the interpolating function. Given a table of n points, $\{x_i, y_i\}$, one can construct a polynomial $P^{(n-1)}(x)$ of the order $n-1$ which passes exactly through the points. This polynomial can be intuitively written in the *Lagrange form*,

$$P^{(n-1)}(x) = \sum_{i=1}^{n} y_i \prod_{k\neq i}^{n} \frac{x - x_k}{x_i - x_k} \ . \tag{2.1}$$

```
function pinterp(x,y,z){
   for(var s=0,i=0; i<x.length; i++){
      for(var p=1,k=0; k<x.length; k++)
         if(k!=i) p*=(z-x[k])/(x[i]-x[k])
      s+=y[i]*p}
   return s }
```

Higher order interpolating polynomials are susceptible to the *Runge phenomenon* – erratic oscillations close to the end-points of the interval as illustrated on Fig. 2.1. This problem can be avoided by using only the nearest few points instead of all the points in the table (local interpolation) or by using spline interpolation.

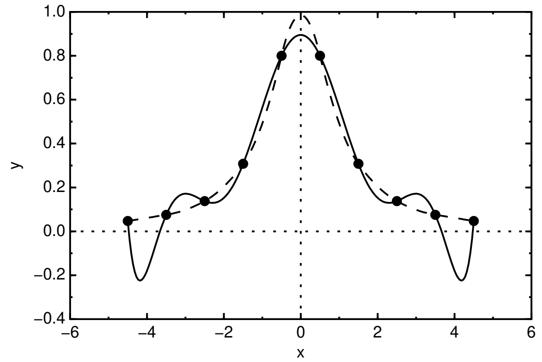

Figure 2.1: Lagrange interpolating polynomial, solid line, showing the Runge phenomenon: large oscillations at the end-points. Dashed line shows a quadratic spline.

2.2 Spline interpolation

Spline interpolation uses a *piecewise polynomial* $S(x)$, called *spline*, as the interpolating function,

$$S(x) = S_i(x) \text{ if } x \in [x_i, x_{i+1}] \,, \tag{2.2}$$

where $S_i(x)$ is a polynomial of a given order k.

The spline of the order $k \geq 1$ can be made continuous at the tabulated points,

$$\begin{aligned} S_i(x_i) &= y_i &&, \; i = 1 \dots n-1 \\ S_i(x_{i+1}) &= y_{i+1} &&, \; i = 1 \dots n-1 \,, \end{aligned} \tag{2.3}$$

together with its $k-1$ derivatives,

$$\begin{aligned} S_i'(x_{i+1}) &= S_{i+1}'(x_{i+1}) &&, \; i = 1 \dots n-2 \\ S_i''(x_{i+1}) &= S_{i+1}''(x_{i+1}) &&, \; i = 1 \dots n-2 \\ &\dots \end{aligned} \tag{2.4}$$

Continuity conditions (2.3) and (2.4) make $kn + n - 2k$ linear equations for the $(n-1)(k+1) = kn + n - k - 1$ coefficients in $n-1$ polynomials (2.2) of the order k. The missing $k-1$ conditions can be chosen (reasonably) arbitrarily.

The most popular is the cubic spline, where the polynomials $S_i(x)$ are of third order. The cubic spline is a continuous function together with its first and second derivatives. The cubic spline also has a nice feature that it (sort of) minimizes the total curvature of the interpolating function. This makes the cubic splines look good.

Quadratic spline, which is continuous together with its first derivative, is not nearly as good as the cubic spline in most respects. Particularly it might oscillate unpleasantly when a quick change in the tabulated function is followed by a period where the function is nearly a constant. The cubic spline is less susceptible to such oscillations.

Linear spline is simply a *polygon* drawn through the tabulated points.

2.2.1 Linear interpolation

If the spline polynomials are linear the spline is called *linear interpolation*. The continuity conditions (2.3) can be satisfied by choosing the spline as

$$S_i(x) = y_i + \frac{\Delta y_i}{\Delta x_i}(x - x_i) \ , \tag{2.5}$$

where

$$\Delta y_i \equiv y_{i+1} - y_i \ , \ \ \Delta x_i \equiv x_{i+1} - x_i \ . \tag{2.6}$$

2.2.2 Quadratic spline

Quadratic splines are made of second order polynomials, conveniently chosen in the form

$$S_i(x) = y_i + \frac{\Delta y_i}{\Delta x_i}(x - x_i) + a_i(x - x_i)(x - x_{i+1}), \tag{2.7}$$

which identically satisfies the continuity conditions (2.3).

Substituting (2.7) into the continuity condition for the first derivative (2.4) gives $n-2$ equations for $n-1$ unknown coefficients a_i,

$$\frac{\Delta y_i}{\Delta x_i} + a_i \Delta x_i = \frac{\Delta y_{i+1}}{\Delta x_{i+1}} - a_{i+1}\Delta x_{i+1} \ . \tag{2.8}$$

One coefficient can be chosen arbitrarily, for example $a_1 = 0$. The other coefficients can now be calculated recursively,

$$a_{i+1} = \frac{1}{\Delta x_{i+1}} \left(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i} - a_i \Delta x_i \right) . \tag{2.9}$$

Alternatively, one can choose $a_{n-1} = 0$ and make the inverse recursion

$$a_i = \frac{1}{\Delta x_i} \left(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i} - a_{i+1} \Delta x_{i+1} \right) . \tag{2.10}$$

In practice, unless you know what your a_1 (or a_{n-1}) is, it is better to run both recursions and then average the resulting a's.

2.2.3 Cubic spline

Cubic splines are made of third order polynomials written e.g. in the form

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \ , \tag{2.11}$$

which automatically satisfies the upper half of continuity conditions (2.3). The other half of continuity conditions (2.3) and the continuity of the first and second derivatives (2.4) give

$$\begin{aligned} y_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= y_{i+1} \ , \ i = 1, \ldots, n-1 \\ b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \ , \ i = 1, \ldots, n-2 \\ 2c_i + 6d_i h_i &= 2c_{i+1} \ , \ i = 1, \ldots, n-2 \end{aligned} \tag{2.12}$$

where

$$h_i = x_{i+1} - x_i \ . \tag{2.13}$$

The set of equations (2.12) is a set of $3n-5$ linear equations for the $3(n-1)$ unknown coefficients $\{a_i, b_i, c_i \mid i = 1, \ldots, n-1\}$. Therefore two more equations should be added to the set to find the coefficients. If the two extra equations are also linear, the total system is linear and can be easily solved.

The spline is called *natural* if the extra conditions are vanishing second derivative at the end-points,

$$S''(x_1) = S''(x_n) = 0 \; , \tag{2.14}$$

which gives

$$\begin{aligned} c_1 &= 0 \; , \\ c_{n-1} + 3d_{n-1}h_{n-1} &= 0 \; . \end{aligned} \tag{2.15}$$

Solving the first two equations in (2.12) for c_i and d_i gives[1]

$$\begin{aligned} c_i h_i &= -2b_i - b_{i+1} + 3p_i \; , \\ d_i h_i^2 &= b_i + b_{i+1} - 2p_i \; , \end{aligned} \tag{2.16}$$

where $p_i \equiv \frac{\Delta y_i}{h_i}$. The natural conditions (2.15) and the third equation in (2.12) then produce the following tridiagonal system of n linear equations for the n coefficients b_i,

$$\begin{aligned} 2b_1 + b_2 &= 3p_1 \; , \\ b_i + (2\frac{h_i}{h_{i+1}} + 2)b_{i+1} + \frac{h_i}{h_{i+1}}b_{i+2} &= 3(p_i + p_{i+1}\frac{h_i}{h_{i+1}}) \; , \; i = 1, n-2 \\ b_{n-1} + 2b_n &= 3p_{n-1} \; , \end{aligned} \tag{2.17}$$

or, in the matrix form,

$$\begin{pmatrix} D_1 & Q_1 & 0 & 0 & \cdots \\ 1 & D_2 & Q_2 & 0 & \cdots \\ 0 & 1 & D_3 & Q_3 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \cdots & \cdots & 0 & 1 & D_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ \vdots \\ B_n \end{pmatrix} \tag{2.18}$$

where the elements D_i at the main diagonal are

$$D_1 = 2 \; ; \; D_{i+1} = 2\frac{h_i}{h_{i+1}} + 2 \; , \; i = 1, \ldots, n-2 \; ; \; D_n = 2 \; , \tag{2.19}$$

the elements Q_i at the above-main diagonal are

$$Q_1 = 1 \; ; \; Q_{i+1} = \frac{h_i}{h_{i+1}} \; , \; i = 1, \ldots, n-2 \; , \tag{2.20}$$

and the right-hand side terms B_i are

$$B_1 = 3p_1 \; ; \; B_{i+1} = 3(p_i + p_{i+1}\frac{h_i}{h_{i+1}}) \; , \; i = 1, \ldots, n-2 \; ; B_n = 3p_{n-1} \; . \tag{2.21}$$

[1] introducing an auxiliary coefficient b_n

This system can be solved by one run of Gauss elimination and then a run of back-substitution. After a run of Gaussian elimination the system becomes

$$\begin{pmatrix} \tilde{D}_1 & Q_1 & 0 & 0 & \dots \\ 0 & \tilde{D}_2 & Q_2 & 0 & \dots \\ 0 & 0 & \tilde{D}_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 0 & \tilde{D}_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \tilde{B}_1 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{pmatrix} , \tag{2.22}$$

where

$$\tilde{D}_1 = D_1 \; ; \; \tilde{D}_i = D_i - Q_{i-1}/D_{i-1} \; , \; i = 2, \dots, n \tag{2.23}$$

and

$$\tilde{B}_1 = B_1 \; ; \; \tilde{B}_i = B_i - B_{i-1}/D_{i-1} \; , \; i = 2, \dots, n \tag{2.24}$$

The triangular system (2.22) can be solved by a run of back-substitution,

$$b_n = \frac{1}{\tilde{D}_n}\tilde{B}_n \; ; \; b_i = \frac{1}{\tilde{D}_i}(\tilde{B}_i - Q_i b_{i+1}) \; , \; i = n-1, \dots, 1 \; . \tag{2.25}$$

2.3 Other forms of interpolation

Other forms of interpolation can be constructed by choosing a different class of interpolating functions, for example, rational function interpolation, trigonometric interpolation, wavelet interpolation etc.

Sometimes not only the value of the function is given at the tabulated points, but also the derivative. This extra information can be taken advantage of when constructing the interpolation function.

Interpolation of a function in more than one dimension is called *multivariate interpolation*. In two dimension one of the easiest methods is the *bi-linear interpolation* where the function in each tabulated rectangle is approximated as a product of two linear functions,

$$f(x, y) \approx (ax + b)(cy + d) \; , \tag{2.26}$$

where the constants a, b, c, d are obtained from the condition that the interpolating function is equal the tabulated values at the nearest four tabulated points.

Chapter 3

Linear least squares

A system of linear equations is considered *overdetermined* if there are more equations than unknown variables. If all equations of an overdetermined system are linearly independent, the system has no exact solution.

A *linear least-squares problem* is the problem of finding an approximate solution to an overdetermined system. It often arises in applications where a theoretical model is fitted to experimental data.

3.1 Linear least-squares problem

Consider a linear system

$$A\mathbf{c} = \mathbf{b} \,, \tag{3.1}$$

where A is an $n \times m$ matrix, $\mathbf{c}$ is an m-component vector of unknown variables and $\mathbf{b}$ is an n-component vector of the right-hand side terms. If the number of equations n is larger than the number of unknowns m, the system is overdetermined and generally has no solution.

However, it is still possible to find an approximate solution – the one where $A\mathbf{c}$ is only approximately equal $\mathbf{b}$, in the sence that the Euclidean norm of the difference between $A\mathbf{c}$ and $\mathbf{b}$ is minimized,

$$\min_{\mathbf{c}} \|A\mathbf{c} - \mathbf{b}\|^2 \,. \tag{3.2}$$

The problem (3.2) is called the linear least-squares problem and the vector $\mathbf{c}$ that minimizes $\|A\mathbf{c} - \mathbf{b}\|^2$ is called the *least-squares solution*.

3.2 Solution via QR-decomposition

The linear least-squares problem can be solved by QR-decomposition. The matrix A is factorized as $A = QR$, where Q is $n \times m$ matrix with orthogonal columns, $Q^T Q = 1$, and R is an $m \times m$ upper triangular matrix. The Euclidean norm $\|A\mathbf{c} - \mathbf{b}\|^2$ can then be rewritten as

$$\|A\mathbf{c}-\mathbf{b}\|^2 = \|QR\mathbf{c}-\mathbf{b}\|^2 = \|R\mathbf{c}-Q^T\mathbf{b}\|^2 + \|(1-QQ^T)\mathbf{b}\|^2 \geq \|(1-QQ^T)\mathbf{b}\|^2 \,. \tag{3.3}$$

The term $\|(1-QQ^T)\mathbf{b}\|^2$ is independent of the variables $\mathbf{c}$ and can not be reduced by their variations. However, the term $\|R\mathbf{c}-Q^T\mathbf{b}\|^2$ can be reduced down to zero by solving the $m\times m$ system of linear equations

$$R\mathbf{c}-Q^T\mathbf{b}=0\ . \tag{3.4}$$

The system is right-triangular and can be readily solved by back-substitution.

Thus the solution to the linear least-squares problem (3.2) is given by the solution of the triangular system (3.4).

3.3 Ordinary least-squares curve fitting

Ordinary (or linear) least-squares curve fitting is a problem of fitting n (experimental) data points $\{x_i, y_i\pm\Delta y_i\}$, where Δy_i are experimental errors, by a linear combination of m functions

$$F(x)=\sum_{k=1}^{m}c_k f_k(x)\ . \tag{3.5}$$

The objective of the least-squares fit is to minimize the square deviation, called χ^2, between the fitting function and the experimental data,

$$\chi^2=\sum_{i=1}^{n}\left(\frac{F(x_i)-y_i}{\Delta y_i}\right)^2\ . \tag{3.6}$$

Individual deviations from experimental points are weighted with their inverse errors in order to promote contributions from the more precise measurements.

Minimization of χ^2 with respect to the coefficiendt c_k in (3.5) is apparently equivalent to the least-squares problem (3.2) where

$$A_{ik}=\frac{f_k(x_i)}{\Delta y_i}\ ,\ b_i=\frac{y_i}{\Delta y_i}\ . \tag{3.7}$$

If $QR=A$ is the QR-decomposition of the matrix A, the formal least-squares solution is

$$\mathbf{c}=R^{-1}Q^T\mathbf{b}\ . \tag{3.8}$$

However in practice it is better to back-substitute the system $R\mathbf{c}=Q^T\mathbf{b}$.

3.3.1 Variances and correlations of fitting parameters

Suppose δy_i is a (small) deviation of the measured value of the physical observable from its exact value. The corresponding deviation δc_k of the fitting coefficient is then given as

$$\delta c_k=\sum_i\frac{\partial c_k}{\partial y_i}\delta y_i\ . \tag{3.9}$$

In a good experiment the deviations δy_i are statistically independent and distributed normally with the standard deviations Δy_i. The deviations (3.9) are then also distributed normally with *variances*,

$$\langle\delta c_k\delta c_k\rangle=\sum_i\left(\frac{\partial c_k}{\partial y_i}\Delta y_i\right)^2=\sum_i\left(\frac{\partial c_k}{\partial b_i}\right)^2\ . \tag{3.10}$$

The standard errors in the fitting coefficients are then given as the square roots of variances,

$$\Delta c_k = \sqrt{\langle \delta c_k \delta c_k \rangle} = \sqrt{\sum_i \left(\frac{\partial c_k}{\partial b_i} \right)^2} \,. \tag{3.11}$$

The variances are diagonal elements of the *covariance matrix*, Σ, made of *covariances*,

$$\Sigma_{kq} \equiv \langle \delta c_k \delta c_q \rangle = \sum_i \frac{\partial c_k}{\partial b_i} \frac{\partial c_q}{\partial b_i} \,. \tag{3.12}$$

Covariances $\langle \delta c_k \delta c_q \rangle$ are measures of to what extent the coefficients c_k and c_q change together if the measured values y_i are varied. The normalized covariances,

$$\frac{\langle \delta c_k \delta c_q \rangle}{\sqrt{\langle \delta c_k \delta c_k \rangle \langle \delta c_q \delta c_q \rangle}} \tag{3.13}$$

are called *correlations*.

Using (3.12) and (3.8) the covariance matrix can be calculated as

$$\Sigma = \left(\frac{\partial \mathbf{c}}{\partial \mathbf{b}} \right) \left(\frac{\partial \mathbf{c}}{\partial \mathbf{b}} \right)^T = R^{-1}(R^{-1})^T = (R^T R)^{-1} = (A^T A)^{-1} \,. \tag{3.14}$$

The square roots of the diagonal elements of this matrix provide the estimates of the errors of the fitting coefficients and the (normalized) off-diagonal elements are the estimates of their correlations.

3.4 JavaScript implementation

```
function lsfit(xs,ys,dys,funs){ // Linear least squares fit
// uses: qrdec, qrback, inverse
// input: data points {x,y,dy}; functions {funs}
// output: fitting coefficients c and covariance matrix S
  var dot = function(a,b) // a.b
    {let s=0;for(let i in a)s+=a[i]*b[i];return s}
  var ttimes = function(A,B) // A^T*B
    [[dot(A[r],B[c]) for(r in A)] for(c in B)];
  var A=[[funs[k](xs[i])/dys[i] for(i in xs)] for(k in funs)];
  var b=[ys[i]/dys[i] for(i in ys)];
  var [Q,R]=qrdec(A);
  var c=qrback(Q,R,b);
  var S=inverse(ttimes(R,R));
  return [c,S];
}
```

Chapter 4

Numerical integration

Numerical integration, also called *quadrature* for one-dimensional integrals and *cubature* for multi-dimensional integrals, is an algorithm to compute an approximation to a definite integral in the form of a finite sum,

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \,. \tag{4.1}$$

The abscissas x_i and the weights w_i in (4.1) are chosen such that the quadrature is particularly well suited for the given class class of functions to integrate. Different quadratures use different strategies of choosing the abscissas and weights.

4.1 Classical quadratures with equally spaced abscissas

Classical quadratures use predefined equally-spaced abscissas. A quadrature is called *closed* if the abscissas include the end-points of the interval or the mid-point (which becomes end-point after halving the interval). Otherwise it is called *open*. If the integrand is diverging at the end-points (or at the mid-point of the interval) the closed quadratures generally can not be used.

For an n-point classical quadrature the n free parameters w_i can be chosen such that the quadrature integrates exactly a set of n (linearly independent) functions $\{\phi_1(x), \dots, \phi_n(x)\}$ where the integrals

$$I_k \equiv \int_a^b \phi_k(x)dx \tag{4.2}$$

are known. This gives a set of equations, linear in w_i,

$$\sum_{i=1}^{n} w_i \phi_k(x_i) = I_k \;,\; k = 1 \dots n \,. \tag{4.3}$$

The weights w_i can then be determined by solving the linear system (4.3).

If the functions to be integrated exactly are chosen as polynomials $\{1, x, x^2, \dots, x^{n-1}\}$, the quadrature is called *Newton-Cotes quadrature*. An n-point Newton-Cotes quadrature can integrate exactly the first n terms of the function's Taylor expansion

$$f(a+t)=\sum_{k=0}^{\infty}\frac{f^{(k)}(a)}{k!}t^k\,. \tag{4.4}$$

The nth order term $\frac{f^{(n)}(a)}{n!}t^n$ will not be integrated exactly by an n-point quadrature and will then result[1] in the quadrature's error[2]

$$\epsilon_n\approx\int_0^h\frac{f^{(n)}(a)}{n!}t^n dt=\frac{f^{(n)}(a)}{n!(n+1)}h^{n+1}\,. \tag{4.6}$$

If the function is smooth and the interval h is small enough the Newton-Cotes quadrature can give a good approximation.

Here are several examples of closed and open classical quadratures:

$$\int_0^h f(x)dx \approx \tfrac{1}{2}h\left[f(0)+f(h)\right]\,, \tag{4.7}$$

$$\int_0^h f(x)dx \approx \tfrac{1}{6}h\left[f(0)+4f(\tfrac{1}{2}h)+f(h)\right]\,, \tag{4.8}$$

$$\int_0^h f(x)dx \approx \tfrac{1}{2}h\left[f(\tfrac{1}{3}h)+f(\tfrac{2}{3}h)\right]\,, \tag{4.9}$$

$$\int_0^h f(x)dx \approx \tfrac{1}{6}h\left[2f(\tfrac{1}{6}h)+f(\tfrac{2}{6}h)+f(\tfrac{4}{6}h)+2f(\tfrac{5}{6}h)\right]\,. \tag{4.10}$$

4.2 Quadratures with optimized abscissas

In quadratures with optimal abscissas, called *Gaussian quadratures*, not only weights w_i but also abscissas x_i are chosen optimally. The number of free parameters is thus $2n$ (n optimal abscissas and n weights) and one can chose $2n$ functions $\{\phi_1(x),\dots,\phi_{2n}(x)\}$ to be integrated exactly. This gives a system of $2n$ equations, linear in w_i and non-linear in x_i,

$$\sum_{i=1}^{n}w_i f_k(x_i)=I_k\,,k=1,\dots,2n, \tag{4.11}$$

where $I_k=\int_a^b f_k(x)dx$. The weights and abcissas can be determined by solving this system of equations.

Here is, for example, a two-point Gauss-Legendre quadrature rule [3]

$$\int_{-1}^{1}f(x)dx\approx f\left(-\sqrt{\tfrac{1}{3}}\right)+f\left(+\sqrt{\tfrac{1}{3}}\right)\,. \tag{4.13}$$

---

[1] Assuming that the integral is rescaled as

$$\int_a^b f(x)dx=\int_0^{h=b-a}f(a+t)dt\,. \tag{4.5}$$

[2] Actually the error is often one order in h higher due to symmetry of the the polynomials t^k with respect to reflections against the origin.

[3] assuming that the integral is rescaled as

$$\int_a^b f(x)dx=\int_{-1}^{1}\frac{b-a}{2}f\left(\frac{a+b}{2}+\frac{b-a}{2}t\right)dt\,. \tag{4.12}$$

The Gaussian quadratures are of order $2n-1$ compared to order $n-1$ for non-optimal abscissas. However, the optimal points generally can not be reused at the next iteration in an adaptive algorithm.

4.3 Reducing the error by subdividing the interval

The higher order quadratures, say $n > 10$, suffer from round-off errors as the weights w_i generally have alternating signs. Again, using high order polynomials is dangerous as they typically oscillate wildly and may lead to Runge phenomenon. Therefore if the error of the quadrature is yet too big for a sufficiently large n quadrature, the best strategy is to subdivide the interval in two and then use the quadrature on the half-intervals. Indeed, if the error is of the order h^k, the subdivision would lead to reduced error, $2\left(\frac{h}{2}\right)^k < h^k$, if $k > 1$.

4.4 Adaptive quadratures

Adaptive quadrature is an algorithm where the integration interval is subdivided into adaptively refined subintervals until the given accuracy goal is reached.

Adaptive algorithms are usually built on pairs of quadrature rules (preferably using the same points), a higher order rule (e.g. 4-point-open) and a lower order rule (e.g. 2-point-open). The higher order rule is used to compute the approximation, Q, to the integral. The difference between the higher order rule and the lower order rule gives an estimate of the error, δQ. The integration result is accepted, if

$$\delta Q < \delta + \epsilon |Q| \,, \tag{4.14}$$

where δ is the absolute accuracy goal and ϵ is the relative accuracy goal of the integration.

Otherwise the interval is subdivided into two half-intervals and the procedure applies recursively to subintervals with the same relative accuracy goal ϵ and rescaled absolute accuracy goal $\delta/\sqrt{2}$.

The reuse of the function evaluations made at the previous step of adaptive integration is very important for the efficiency of the algorithm. The equally-spaced abscissas naturally provide for such a reuse.

4.5 Gauss-Kronrod quadratures

Gauss-Kronrod quadratures represent a compromise between equally spaced abscissas and optimal abscissas: n points are reused from the previous iteration (n weights as free parameters) and then m optimal points are added (m abscissas and m weights as free parameters). Thus the accuracy of the method is $n + 2m - 1$. There are several special variants of these quadratures fit for particular types of the integrands.

Table 4.1: Recursive adaptive integrator based on open-2/4 quadratures.

```
function adapt(f,a,b,acc,eps,oldfs){// adaptive integrator

var x=[1/6,2/6,4/6,5/6];// abscissas
var w=[2/6,1/6,1/6,2/6];// weights of higher order quadrature
var v=[1/4,1/4,1/4,1/4];// weights of lower order quadrature
var p=[1,0,0,1];// shows the new points at each recursion
var n=x.length, h=b-a;

if(typeof(oldfs)=="undefined") // first call?
  fs=[f(a+x[i]*h) for(i in x)]; //first call: populate oldfs
else{ // recursive call: oldfs are given
  fs = new Array(n);
  for(var k=0,i=0;i<n;i++){
    if(p[i]) fs[i]=f(a+x[i]*h); // new points
    else     fs[i]=oldfs[k++];}} // reuse of old points

for(var q4=q2=i=0;i<n;i++){
  q4+=w[i]*fs[i]*h; // higher order estimate
  q2+=v[i]*fs[i]*h;} // lower order estimate
var tol=acc+eps*Math.abs(q4) // required tolerance
var err=Math.abs(q4-q2)/3 // error estimate

if(err<tol) // are we done?
  return [q4, err] // yes, return integral and error
else{ // too big error, preparing the recursion
  acc/=Math.sqrt(2.) // rescale the absolute accuracy goal
  var mid=(a+b)/2
  var left=[fs[i]for(i in fs)if(i< n/2)] // store the left points
  var rght=[fs[i]for(i in fs)if(i>=n/2)] // store the right points
  var [ql,el]=adapt(f,a,mid,eps,acc,left) // dispatch two recursive
      calls
  var [qr,er]=adapt(f,mid,b,eps,acc,rght)
  return [ql+qr, Math.sqrt(el*el+er*er)] // return the grand
      estimates
  }
}
```

4.6 Integrals over infinite intervals

4.6.1 Infinite intervals

One way to calculate an integral over infinite interval is to transform it into an integral over a finite interval,

$$\int_{-\infty}^{+\infty} f(x)dx = \int_{-1}^{+1} f\left(\frac{t}{1-t^2}\right)\frac{1+t^2}{(1-t^2)^2}dt\ , \tag{4.15}$$

by the variable substitution

$$x = \frac{t}{1-t^2}\ ,\ dx = \frac{1+t^2}{(1-t^2)^2}dt\ ,\ t = \frac{\sqrt{1+4x^2}-1}{2x}\ . \tag{4.16}$$

The integral over finite interval can be evaluated by ordinary integration methods.

Alternatively,

$$\int_{-\infty}^{+\infty} f(x)dx = \int_0^1 \frac{dt}{t^2}\left(f\left(\frac{1-t}{t}\right) + f\left(-\frac{1-t}{t}\right)\right) . \tag{4.17}$$

4.6.2 Half-infinite intervals

An integral over a half-infinite interval can be transformed into an integral over a finite interval,

$$\int_a^{+\infty} f(x)dx = \int_0^1 f\left(a + \frac{t}{1-t}\right)\frac{1}{(1-t)^2}dt , \tag{4.18}$$

by the variable substitution

$$x = a + \frac{t}{1-t} , \; dx = \frac{1}{(1-t)^2}dt , \; t = \frac{x-a}{1+(x-a)} . \tag{4.19}$$

Similarly,

$$\int_{-\infty}^{a} f(x)dx = \int_{-1}^{0} f\left(a - \frac{t}{1+t}\right)\frac{-1}{(1+t)^2}dt , \tag{4.20}$$

Alternatively,

$$\int_a^{+\infty} f(x)dx = \int_0^1 f\left(a + \frac{1-t}{t}\right)\frac{dt}{t^2} \tag{4.21}$$

$$\int_{-\infty}^{b} f(x)dx = \int_0^1 f\left(b - \frac{1-t}{t}\right)\frac{dt}{t^2} \tag{4.22}$$

Chapter 5

Monte Carlo integration

Monte Carlo integration is a cubature where the points, at which the integrand is evaluated, are chosen randomly. Typically no assumptions are made about the smoothness of the integrand, not even that it is continuous.

Plain Monte Carlo algorithm distributes points uniformly throughout the integration region using either uncorrelated pseudo-random or correlated quasi-random sequences of points.

Adaptive algorithms, such as VEGAS and MISER, distribute points non-uniformly in an attempt to reduce integration error. They use correspondingly *importance* and *stratified* sampling.

5.1 Multi-dimensional integration

One of the problems in multi-dimensional integration is that the integration region Ω is often quite complicated, with the boundary not easily described by simple functions. However, it is usually much easier to find out whether a given point lies within the integration region or not. Therefore a popular strategy is to create an auxiliary rectangular volume V which contains the integration volume Ω and an auxiliary function F which coincides with the integrand inside the volume Ω and is equal zero outside. Then the integral of the auxiliary function over the (simple rectangular) auxiliary volume is equal the original integral.

Unfortunately, the auxiliary function is generally non-continuous at the boundary and thus the ordinary quadratures which assume continuous integrand will fail badly here while the Monte-Carlo quadratures will do just as good (or as bad) as with continuous integrand.

5.2 Plain Monte Carlo sampling

Plain Monte Carlo is a quadrature with random abscissas and equal weights ,

$$\int_V f(\mathbf{x})dV \approx w \sum_{i=1}^{N} f(\mathbf{x}_i) \;, \tag{5.1}$$

where $\mathbf{x}$ a point in the multi-dimensional integration space. One free parameter, w, allows one condition to be satisfied: the quadrature has to integrate exactly a constant function. This gives $w = V/N$,

$$\int_V f(\mathbf{x})dV \approx \frac{V}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) = V\langle f\rangle \ . \tag{5.2}$$

According to the *central limit theorem* the error estimate ϵ is close to

$$\epsilon = V\frac{\sigma}{\sqrt{N}} \ , \tag{5.3}$$

where σ is the variance of the sample,

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2 . \tag{5.4}$$

The $1/\sqrt{N}$ convergence of the error, typical for a random process, is quite slow.

Table 5.1: Plain Monte Carlo integrator

```
function plainmc(fun, a, b, N){
var randomx = function(a,b) //throws a random point inside
    inegration volume
  [a[i]+Math.random()*(b[i]-a[i]) for (i in a)];
var V=1; for(var i in a) V*=b[i]-a[i]; // V = integration
    volume
for(var sum=0,sum2=0,i=0;i<N;i++){ //main loop
  var f=fun(randomx(a,b)); // sampling the function
  sum+=f; sum2+=f*f} // accumulating statistics
var average =sum/N;
var variance=sum2/N-average*average;
var integral=V*average; // integral
var error=V*Math.sqrt(variance/N); // error
return [integral,error];
}//end plainmc
```

5.3 Importance sampling

Suppose that the points are distributed not uniformly but with some density $\rho(x)$: the number of points Δn in the volume ΔV around point x is given as

$$\Delta n = \frac{N}{V}\rho\Delta V, \tag{5.5}$$

where ρ is normalised such that $\int_V \rho dV = V$.

The estimate of the integral is then given as

$$\int_V f(\mathbf{x})dV \approx \sum_{i=1}^{N} f(\mathbf{x}_i)\Delta V_i = \sum_{i=1}^{N} f(\mathbf{x}_i)\frac{V}{N\rho(\mathbf{x}_i)} = V\left\langle \frac{f}{\rho}\right\rangle \ , \tag{5.6}$$

where

$$\Delta V_i = \frac{V}{N\rho(x_i)} \tag{5.7}$$

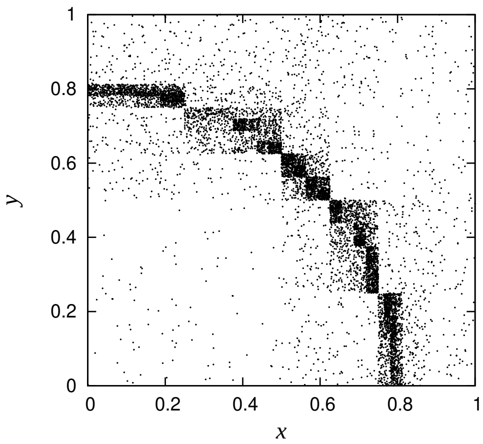

Figure 5.1: Stratified sample of a discontinuous function,
$f(x, y) = (x^2 + y^2 < 0.8^2)\ ?\ 1 : 0$

is the "volume per point" at the point x_i.

The corresponding variance is now given by

$$\sigma^2 = \left\langle \left(\frac{f}{\rho}\right)^2 \right\rangle - \left\langle \frac{f}{\rho} \right\rangle^2 . \tag{5.8}$$

Apparently if the ratio f/ρ is close to a constant, the variance is reduced.

It is tempting to take $\rho = |f|$ and sample directly from the function to be integrated. However in practice it is typically expensive to evaluate the integrand. Therefore a better strategy is to build an approximate density in the product form, $\rho(x, y, \ldots, z) = \rho_x(x)\rho_y(y)\ldots\rho_z(z)$, and then sample from this approximate density. A popular routine of this sort is called VEGAS. The sampling from a given function can be done using the *Metropolis algorithm* which we shall not discuss here.

5.4 Stratified sampling

Stratified sampling is a generalisation of the recursive adaptive integration algorithm to random quadratures in multi-dimensional spaces.

The ordinary "dividing by two" strategy does not work for multi-dimensions as the number of sub-volumes grows way too fast to keep track of. Instead one estimates along which dimension a subdivision should bring the most dividends and only subdivides along this dimension. Such strategy is called *recursive stratified sampling.* A simple variant of this algorithm is given in table 5.4.

In a stratified sample the points are concentrated in the regions where the variance of the function is largest, as illustrated on figure 5.4.

Table 5.2: Recursive stratified sampling

```
sample N random points with plain Monte Carlo;
estimate the average and the error;
if the error is acceptable :
  return the average and the error;
else :
  for each dimension :
    subdivide the volume in two along the dimension;
    estimate the sub-variances in the two sub-volumes;
  pick the dimension with the largest sub-variance;
  subdivide the volume in two along this dimension;
  dispatch two recursive calls to each of the sub-volumes;
  estimate the grand average and grand error;
  return the grand average and grand error;
```

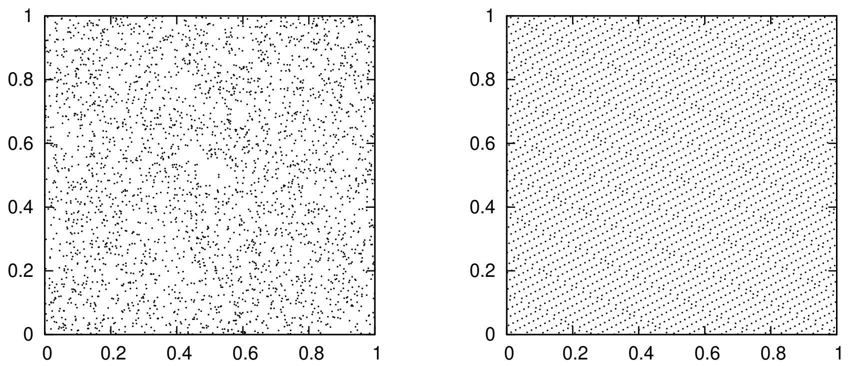

Figure 5.2: Typical distributions of pseudo-random (left) and quasi-random (right) points in two dimensions.

5.5 Quasi-random (low-discrepancy) sampling

Pseudo-random sampling has high discrepancy[1] – it typically creates regions with hight density of points and other regions with low density of points, as illustrated on fig. 5.2. With pseudo-random sampling there is a finite probability that all the N points would fall into one half of the region and none into the other half.

Quasi-random sequences avoid this phenomenon by distributing points in a highly correlated manner with a specific requirement of low discrepancy, see fig. 5.2 for an example. Quasi-random sampling is like a computation on a grid where the grid constant must not be known in advance as the grid is ever gradually refined and the points are always distributed uniformly over the region. The computation can be stopped at any time.

By placing points more evenly than at random, the quasi-random sequences try to improve the $1/\sqrt{N}$ convergence rate of pseudo-random sampling.

The central limit theorem does not work in this case as the points are not statistically independent. Therefore the variance can not be used as an estimate

[1]discrepancy is a measure of how unevenly the points are distributed over the region.

of the error. The error estimation is actually not trivial. In practice one can employ two different sequences and use their difference as the error estimate.

Quasi-random sequences can be roughly divided into *lattice rules* and *digital nets* (see e.g. arXiv:1003.4785 [math.NA] and references therein).

5.5.1 Lattice rules

In the simplest incarnation a lattice rule can be defined as follows.

Let α_i, $i = 1, \ldots, d$, (where d is the dimension of the integration space) be a set of cleverly chosen irrational numbers, like square roots of prime numbers. Then the kth point (in the unit volume) of the sequence is given as

$$\mathbf{x}^{(k)} = \{\mathrm{frac}(k\alpha_1), \ldots, \mathrm{frac}(k\alpha_d)\} , \tag{5.9}$$

where $\mathrm{frac}(x)$ is the fractional part of x.

A problem with this method is that a high accuracy arithmetics (e.g. `long double`) might be needed in order to generate a reasonable amount of quasi-random numbers.

Chapter 6

Ordinary differential equations

6.1 Introduction

Many scientific problems can be formulated in terms of a system of *ordinary differential equations* (ODE),

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}) \;, \tag{6.1}$$

with an initial condition

$$\mathbf{y}(x_0) = \mathbf{y}_0 \;, \tag{6.2}$$

where $\mathbf{y}' \equiv d\mathbf{y}/dx$, and the boldface variables $\mathbf{y}$ and $\mathbf{f}(x, \mathbf{y})$ are generally understood as column-vectors.

6.2 Runge-Kutta methods

Runge-Kutta methods are one-step methods for numerical integration of ODE (6.1). The solution $\mathbf{y}$ is advanced from the point x_0 to $x_1 = x_0 + h$ using a one-step formula

$$\mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}, \tag{6.3}$$

where $\mathbf{y}_1$ is the approximation to $\mathbf{y}(x_1)$, and $\mathbf{k}$ is a cleverly chosen (vector) constant. The Runge-Kutta methods are distinguished by their *order*: a method has order p if it can integrate exactly an ODE where the solution is a polynomyal of order p. In other words, if the error of the method is $O(h^{p+1})$ for small h.

The first order Runge-Kutta method is the Euler's method,

$$\mathbf{k} = \mathbf{f}(x_0, \mathbf{y}_0) \;. \tag{6.4}$$

Second order Runge-Kutta methods advance the solution by an auxiliary evaluation of the derivative, e.g. the *mid-point method*,

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \;, \\ \mathbf{k}_{1/2} &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \;, \\ \mathbf{k} &= \mathbf{k}_{1/2} \;, \end{aligned} \tag{6.5}$$

or the *two-point method*,

$$\begin{aligned}
\mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0), \\
\mathbf{k}_1 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_0), \\
\mathbf{k} &= \frac{1}{2}(\mathbf{k}_0 + \mathbf{k}_1)\ . \qquad (6.6)
\end{aligned}$$

These two methods can be combined into a third order method,

$$\mathbf{k} = \frac{1}{6}\mathbf{k}_0 + \frac{4}{6}\mathbf{k}_{1/2} + \frac{1}{6}\mathbf{k}_1\ . \qquad (6.7)$$

The most commont is the fourth-order method, which is called *RK4* or simply *the Runge-Kutta method*,

$$\begin{aligned}
\mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0)\ , \\
\mathbf{k}_1 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0)\ , \\
\mathbf{k}_2 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_1)\ , \\
\mathbf{k}_3 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_2)\ , \\
\mathbf{k} &= \tfrac{1}{6}(\mathbf{k}_0 + 2\mathbf{k}_1 + 2\mathbf{k}_2 + \mathbf{k}_3)\ . \qquad (6.8)
\end{aligned}$$

Higher order Runge-Kutta methods have been devised, with the most famous being the Runge-Kutta-Fehlberg fourth/fifth order method, *RKF45*, implemented in the renowned `rkf45.f` Fortran routine.

6.3 Multistep methods

Multistep methods try to use the information about the function gathered at the previous steps. They are generally not *self-starting* as there are no previous points at the start of the integration.

6.3.1 A two-step method

Given the previous point, $(x_{-1}, \mathbf{y}_{-1})$, in addition to the current point $(x_0, \mathbf{y}_0)$, the sought function $\mathbf{y}$ can be approximated in the vicinity of the point x_0 as

$$\bar{\mathbf{y}}(x) = \mathbf{y}_0 + \mathbf{y}'_0 \cdot (x - x_0) + \mathbf{c} \cdot (x - x_0)^2, \qquad (6.9)$$

where $\mathbf{y}'_0 = \mathbf{f}(x_0, \mathbf{y}_0)$ and the coefficient $\mathbf{c}$ is found from the condition $\mathbf{y}(x_{-1}) = \mathbf{y}_{-1}$,

$$\mathbf{c} = \frac{\mathbf{y}_{-1} - \mathbf{y}_0 + \mathbf{y}'_0 \cdot (x_0 - x_{-1})}{(x_0 - x_{-1})^2}. \qquad (6.10)$$

The value of the function at the next point, x_1, can now be estimated as $\bar{\mathbf{y}}(x_1)$ from (6.9).

6.4 Predictor-corrector methods

Predictor-corrector methods use extra iterations to improve the solution. For example, the two-point Runge-Kutta method (6.6) is as actually a predictor-corrector method, as it first calculates the *prediction* $\tilde{\mathbf{y}}_0$ for $\mathbf{y}(x_0)$,

$$\tilde{\mathbf{y}}_1 = \mathbf{y}_0 + h\mathbf{f}(x_0, \mathbf{y}_0)\ , \qquad (6.11)$$

and then uses this prediction in a *correction* step,

$$\tilde{\tilde{\mathbf{y}}}_1 = \mathbf{y}_0 + h\frac{1}{2}\big(\mathbf{f}(x_0, \mathbf{y}_0) + \mathbf{f}(x_1, \tilde{\mathbf{y}}_1)\big) \tag{6.12}$$

Similarly, one can use the two-step approximation (6.9) as a predictor, and then improve it by one order with a correction step, namely

$$\bar{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \mathbf{d} \cdot (x - x_0)^2 (x - x_{-1}). \tag{6.13}$$

The coefficient $\mathbf{d}$ can be found from the condition $\bar{\bar{\mathbf{y}}}'(x_1) = \bar{\mathbf{f}}_1$, where $\bar{\mathbf{f}}_1 = \mathbf{f}(x_1, \bar{\mathbf{y}}(x_1))$,

$$\mathbf{d} = \frac{\bar{\mathbf{f}}_1 - \mathbf{y}_0' - 2\mathbf{c} \cdot (x_1 - x_0)}{2(x_1 - x_0)(x_1 - x_{-1}) + (x_1 - x_0)^2}. \tag{6.14}$$

Equation (6.13) gives a better estimate, $\mathbf{y}_1 = \bar{\bar{\mathbf{y}}}(x_1)$, of the function at the point x_1.

In this context the formula (6.9) is referred to as *predictor*, and (6.13) as *corrector*. The difference between the two gives an estimate of the error.

6.5 Step size control

6.5.1 Error estimate

The error δy of the integration step for a given method can be estimated e.g. by comparing the solutions for a full-step and two half-steps (the *Runge principle*),

$$\delta y \approx \frac{y_{\text{two\_half\_steps}} - y_{\text{full\_step}}}{2^p - 1}, \tag{6.15}$$

where p is the order of the algorithm used. It is better to pick formulas where the full-step and two half-step calculations share the evaluations of the function $\mathbf{f}(x, \mathbf{y})$.

Another possibility is to make the same step with two methods of different orders, the difference between the solutions providing an estimate of the error.

In a predictor-corrector method the correction itself can serve as the estimate of the error.

Table 6.1: Runge-Kutta mid-point stepper with error estimate.

```
function rkstep(f,x,y,h){ // Runge-Kutta midpoint step with error
    estimate
  var k0  = f(x,y) // derivatives at x0
  var y12 = [y[i]+k0[i]*h/2 for(i in y)] // half-step
  var k12 = f(x+h/2,y12) // derivatives at half-step
  var y1  = [y[i]+k12[i]*h for(i in y)] // full step
  var deltay  = [(k12[i]-k0[i])*h/2 for(i in y)] // error estimate
  return [y1, deltay]
}
```

6.5.2 Adaptive step size control

Let *tolerance* τ be the maximal accepted error consistent with the required absolute, δ, and relative, ϵ, accuracies to be achieved in the integration of an ODE,

$$\tau = \epsilon\|\mathbf{y}\| + \delta \,, \tag{6.16}$$

where $\|\mathbf{y}\|$ is the “norm” of the column-vector $\mathbf{y}$.

Suppose the inegration is done in n steps of size h_i such that $\sum_{i=1}^{n} h_i = b - a$. Under assumption that the errors at the integration steps are random and independent, the step tolerance τ_i for the step i has to scale as the square root of the step size,

$$\tau_i = \tau\sqrt{\frac{h_i}{b-a}}. \tag{6.17}$$

Then, if the error e_i on the step i is less than the step tolerance, $e_i \le \tau_i$, the total error E will be consistent with the total tolerance τ,

$$E \approx \sqrt{\sum_{i=1}^{n} e_i^2} \le \sqrt{\sum_{i=1}^{n} \tau_i^2} = \tau\sqrt{\sum_{i=1}^{n} \frac{h_i}{b-a}} = \tau \,. \tag{6.18}$$

In practice one uses the current values of the function $\mathbf{y}$ in the estimate of the tolerance,

$$\tau_i = (\epsilon\|\mathbf{y}_i\| + \delta)\sqrt{\frac{h_i}{b-a}} \tag{6.19}$$

The step is accepted if the error is smaller than tolerance. The next step-size can be estimated according to the empirical prescription

$$h_{\text{new}} = h_{\text{old}} \times \left(\frac{\tau}{e}\right)^{\text{Power}} \times \text{Safety}, \tag{6.20}$$

where Power ≈ 0.25, Safety ≈ 0.95. If the error e_i is larger than tolerance τ_i the step is rejected and a new step with the new step size (6.20) is attempted.

Table 6.2: An ODE driver with adaptive step size control.

```
function rkdrive(f,a,b,y0,acc,eps,h) { //ODE driver:
//integrates y'=f(x,y) with absolute accuracy acc and relative
    accuracy eps
//from a to b with initial condition y0 and initial step h
//storing the results in arrays xlist and ylist
  var norm=function(v) Math.sqrt(v.reduce(function(a,b)a+b*b,0));
  var x=a, y=y0, xlist=[a], ylist=[y0];
  while(x<b){
    if(x+h>b) h=b-x // the last step has to land on "b"
    var [y1,dy]=rkstep(f, x, y, h);
    var err=norm(dy), tol=(norm(y1)*eps+acc)*Math.sqrt(h/(b-a));
    if(err<tol){x+=h; y=y1; xlist.push(x); ylist.push(y);}//accept
        the step
    if(err>0) h*=Math.pow(tol/err,0.25)*0.95; else h*=2;//new step
  }//end while
  return [xlist,ylist];
}// end rkdrive
```

Chapter 7

Nonlinear equations

7.1 Introduction

Non-linear equations or *root-finding* is a problem of finding a set of n variables $\{x_1, \dots, x_n\}$ which satisfy n equations

$$f_i(x_1, ..., x_n) = 0 \;,\; i = 1, \dots, n \;, \tag{7.1}$$

where the functions f_i are generally non-linear.

7.2 Newton's method

Newton's method (also reffered to as Newton-Raphson method, after Isaac Newton and Joseph Raphson) is a root-finding algorithm that uses the first term of the Taylor series of the functions f_i to linearise the system (7.1) in the vicinity of a suspected root. It is one of the oldest and best known methods and is a basis of a number of more refined methods.

Suppose that the point $\mathbf{x} \equiv \{x_1, \dots, x_n\}$ is close to the root. The Newton's algorithm tries to find the step $\Delta\mathbf{x}$ which would move the point towards the root, such that

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = 0 \;,\; i = 1, \dots, n \;. \tag{7.2}$$

The first order Taylor expansion of (7.2) gives a system of linear equations,

$$f_i(\mathbf{x}) + \sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k} \Delta x_k = 0 \;,\; i = 1, \dots, n \;, \tag{7.3}$$

or, in the matrix form,

$$J\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}), \tag{7.4}$$

where $\mathbf{f}(\mathbf{x}) \equiv \{f_1(\mathbf{x}), \dots, f_n(\mathbf{x})\}$ and J is the matrix of partial derivatives[1],

$$J_{ik} \equiv \frac{\partial f_i}{\partial x_k} \;, \tag{7.5}$$

[1] in practice if derivatives are not available analytically one uses finite differences

$$\frac{\partial f_i}{\partial x_k} \approx \frac{f_i(x_1, \dots, x_{k-1}, x_k + \delta x, x_{k+1}, \dots, x_n) - f_i(x_1, \dots, x_k, \dots, x_n)}{\delta x}$$

with $\delta x \ll s$ where s is the typical scale of the problem at hand.

called the *Jacobian matrix*.

The solution $\Delta\mathbf{x}$ to the linear system (7.4) gives the approximate direction and the step-size towards the solution.

The Newton's method converges quadratically if sufficiently close to the solution. Otherwise the full Newton's step $\Delta\mathbf{x}$ might actually diverge from the solution. Therefore in practice a more conservative step $\lambda\Delta\mathbf{x}$ with $\lambda < 1$ is usually taken. The strategy of finding the optimal λ is referred to as *line search*.

It is typically not worth the effort to find λ which minimizes $\|\mathbf{f}(\mathbf{x}+\lambda\Delta\mathbf{x})\|$ exactly, since $\Delta\mathbf{x}$ is only an approximate direction towards the root. Instead an inexact but quick minimization strategy is usually used, like the *backtracking line search* where one first attempts the full step, $\lambda = 1$, and then backtracks, $\lambda \leftarrow \lambda/2$, until either the condition

$$\|\mathbf{f}(\mathbf{x}+\lambda\Delta\mathbf{x})\| < \left(1-\frac{\lambda}{2}\right)\|\mathbf{f}(\mathbf{x})\| \tag{7.6}$$

is satisfied, or λ becomes too small.

7.3 Broyden's quasi-Newton method

The Newton's method requires calculation of the Jacobian at every iteration. This is generally an expensive operation. Quasi-Newton methods avoid calculation of the Jacobian matrix at the new point $\mathbf{x}+\delta\mathbf{x}$, instead trying to use certain approximations, typically rank-1 updates.

Broyden algorithm estimates the Jacobian $J+\delta J$ at the point $\mathbf{x}+\delta\mathbf{x}$ using the finite-difference approximation,

$$(J+\delta J)\delta\mathbf{x} = \delta\mathbf{f} \,, \tag{7.7}$$

where $\delta\mathbf{f} \equiv \mathbf{f}(\mathbf{x}+\delta\mathbf{x}) - \mathbf{f}(\mathbf{x})$ and J is the Jacobian at the point $\mathbf{x}$.

The matrix equation (7.7) is under-determined in more than one dimension as it contains only n equations to determine n^2 matrix elements of δJ. Broyden suggested to choose δJ as a rank-1 update, linear in $\delta\mathbf{x}$,

$$\delta J = \mathbf{c}\,\delta\mathbf{x}^T \,, \tag{7.8}$$

where the unknown vector $\mathbf{c}$ can be found by substituting (7.8) into (7.7), which gives

$$\delta J = \frac{\delta\mathbf{f} - J\delta\mathbf{x}}{\|\delta\mathbf{x}\|^2}\delta\mathbf{x}^T \,. \tag{7.9}$$

7.4 Javascript implementation

```
load('../linear/qrdec.js'); load('../linear/qrback.js');

function newton(fs,x,acc,dx){//Newton's root-finding method
  var norm=function(v)Math.sqrt(v.reduce(function(s,e)s+e*e,0));
  if(acc==undefined)acc=1e-6
  if(dx==undefined)dx=1e-3
  var J = [[0 for(i in x)] for(j in x)]
  var minusfx=[-fs[i](x) for (i in x)]
  do{
```

```
    for(i in x) for(k in x){// calculate Jacobian
      x[k]+=dx
      J[k][i]=(fs[i](x)+minusfx[i])/dx
      x[k]-=dx }
    var [Q,R]=qrdec(J), Dx=qrback(Q,R,minusfx)// Newton's step
    var s=2
    do{ // simple backtracking linesearch
      s=s/2;
      var z=[x[i]+s*Dx[i] for(i in x)]
      var minusfz=[-fs[i](z) for(i in x)]
    }while(norm(minusfz)>(1-s/2)*norm(minusfx) && s>1./128)
    minusfx=minusfz; x=z; // step done
  }while(norm(minusfx)>acc)
  return x;
}//end newton
```

Chapter 8

Optimization

Optimization is a problem of finding the minimum (or the maximum) of a given real (non-linear) function $F(\mathbf{p})$ of an n-dimensional argument $\mathbf{p} \equiv \{x_1, \dots, x_n\}$.

8.1 Downhill simplex method

The *downhill simplex method* (also called Nelder-Mead method or amoeba method) is a commonnly used nonlinear optimization algorithm. The minimum of a function in an n-dimensional space is found by transforming a simplex (a polytope of $n{+}1$ vertexes) according to the function values at the vertexes, moving it downhill until it converges towards the minimum.

To introduce the algorithm we need the following definitions:

- Simplex: a figure (polytope) represented by $n{+}1$ points, called vertexes, $\{\mathbf{p}_1, \dots, \mathbf{p}_{n+1}\}$ (where each point $\mathbf{p}_k$ is an n-dimensional vector).
- Highest point: the vertex, $\mathbf{p}_{\mathrm{hi}}$, with the largest value of the function: $f(\mathbf{p}_{\mathrm{hi}}) = \max_{(k)} f(\mathbf{p}_k)$.
- Lowest point: the vertex, $\mathbf{p}_{\mathrm{lo}}$, with the smallest value of the function: $f(\mathbf{p}_{\mathrm{lo}}) = \min_{(k)} f(\mathbf{p}_k)$.
- Centroid: the center of gravity of all points, except for the highest: $\mathbf{p}_{\mathrm{ce}} = \frac{1}{n}\sum_{(k\neq\mathrm{hi})} \mathbf{p}_k$

The simplex is moved downhill by a combination of the following elementary operations:

1. Reflection: the highest point is reflected against the centroid, $\mathbf{p}_{\mathrm{hi}} \to \mathbf{p}_{\mathrm{re}} = \mathbf{p}_{\mathrm{ce}} + (\mathbf{p}_{\mathrm{ce}} - \mathbf{p}_{\mathrm{hi}})$.
2. Expansion: the highest point reflects and then doubles its distance from the centroid, $\mathbf{p}_{\mathrm{hi}} \to \mathbf{p}_{\mathrm{ex}} = \mathbf{p}_{\mathrm{ce}} + 2(\mathbf{p}_{\mathrm{ce}} - \mathbf{p}_{\mathrm{hi}})$.
3. Contraction: the highest point halves its distance from the centroid, $\mathbf{p}_{\mathrm{hi}} \to \mathbf{p}_{\mathrm{co}} = \mathbf{p}_{\mathrm{ce}} + \frac{1}{2}(\mathbf{p}_{\mathrm{hi}} - \mathbf{p}_{\mathrm{ce}})$.
4. Reduction: all points, except for the lowest, move towards the lowest points halving the distance. $\mathbf{p}_{k\neq\mathrm{lo}} \to \frac{1}{2}(\mathbf{p}_k + \mathbf{p}_{\mathrm{lo}})$.

Finally, here is a possible algorithm for the downhill simplex method:

```
repeat :
  find highest , lowest , and centroid points
  try reflection
  if f(reflected) < f(highest) :
    accept reflection
    if f(reflected) < f(lowest) :
      try expansion
      if f(expanded) < f(reflected) :
        accept expansion
  else:
    try contraction
    if f(contracted) < f(highest) :
      accept contraction
    else :
      do reduction
until converged (e.g. size(simplex)<tolerance)
```

8.2 Javascript implementation

```
function amoeba(F,s,acc){// s: inital simplex, F: function to
    minimize
var sum =function(xs)xs.reduce(function(s,x)s+x,0)
var norm=function(xs)Math.sqrt(xs.reduce(function(s,x)s+x*x,0))
var dist=function(as,bs)norm([(as[k]-bs[k])for(k in as)])
var size=function(s)norm([dist(s[i],s[0])for(i in s)if(i>0)])
var p=s[0], n=p.length, fs=[F(s[i]) for(i in s)] //vertexes
while(size(s)>acc){
  var h=0,l=0
  for(var i in fs){ //finding high and low points
    if(fs[i]>fs[h]) h=i
    if(fs[i]<fs[l]) l=i }
var pce=[sum([s[i][k] for(i in s) if(i!=h)])/n for(k in p)]//
    p_centroid
var pre=[pce[k]+(pce[k]-s[h][k]) for(k in p)], Fre=F(pre)  //
    p_reflected
var pex=[pce[k]+2*(pce[k]-s[h][k]) for(k in p)] //p_expanded
  if(Fre<fs[h]){ // accept reflection
    for(var k in p) s[h][k]=pre[k]; fs[h]=Fre
    if(Fre<fs[l]){
      var Fex=F(pex)
      if(Fex<Fre){ // expansion
        for(var k in p) s[h][k]=pex[k]; fs[h]=Fex }}}
  else{
    var pco=[pce[k]+.5*(pce[k]-s[h][k]) for(k in p)],Fco=F(pco)//
        contraction
    if(Fco<fs[h]){ // contraction
      for(var k in p) s[h][k]=pco[k]; fs[h]=Fco }
    else{ // reduction
      for(var i in s)if(i!=l){
        for(var k in p) s[i][k]=.5*(s[i][k]+s[l][k])
        fs[i]=F(s[i]) } } }
  }// end while
  return s[l]
}//end amoeba
```

Chapter 9

Eigenvalues and eigenvectors

9.1 Introduction

A non-zero column-vector $\mathbf{v}$ is called an *eigenvector* of a matrix A with an *eigenvalue* λ, if

$$A\mathbf{v} = \lambda\mathbf{v} \ . \tag{9.1}$$

If an $n \times n$ matrix A is real and symmetric, $A^T = A$, then it has n real eigenvalues $\lambda_1, \dots, \lambda_n$, and its (orthogonalized) eigenvectors $V = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ form a full basis,

$$VV^T = V^TV = \mathbf{1} \ , \tag{9.2}$$

in which the matrix is diagonal,

$$V^TAV = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \lambda_n \end{bmatrix} . \tag{9.3}$$

Matrix diagonalization means finding all eigenvalues and (optionally) eigenvectors of a matrix.

Eigenvalues and eigenvectors enjoy a maltitude of applications in different branches of science and technology.

9.2 Similarity transformations

Orthogonal transformations,

$$A \to Q^TAQ \ , \tag{9.4}$$

where $Q^TQ = \mathbf{1}$, and, generally, similarity transformations,

$$A \to S^{-1}AS \ , \tag{9.5}$$

preserve eigenvalues and eigenvectors. Therefore one of the strategies to diagonalize a matrix is to apply a sequence of similarity transformations (also called rotations) which (iteratively) turn the matrix into diagonal form.

9.2.1 Jacobi eigenvalue algorithm

Jacobi eigenvalue algorithm is an iterative method to calculate the eigenvalues and eigenvectors of a real symmetric matrix by a sequence of Jacobi rotations.

Jacobi rotation is an orthogonal transformation which zeroes a pair of the off-diagonal elements of a (real symmetric) matrix A,

$$A \to A' = J(p,q)^T A J(p,q) \; : \; A'_{pq} = A'_{qp} = 0 \; . \tag{9.6}$$

The orthogonal matrix $J(p,q)$ which eliminates the element A_{pq} is called the Jacobi rotation matrix. It is equal identity matrix except for the four elements with indices pp, pq, qp, and qq,

$$J(p,q) = \begin{array}{cc} \begin{bmatrix} 1 & & & & & \\ & \ddots & & & 0 & \\ & & \cos\phi & \cdots & \sin\phi & & \\ & & \vdots & \ddots & \vdots & & \\ & & -\sin\phi & \cdots & \cos\phi & & \\ & 0 & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} & \begin{array}{l} \\ \\ \leftarrow \text{row } p \\ \\ \leftarrow \text{row } q \\ \\ \\ \end{array} \end{array} . \tag{9.7}$$

Or explicitly,

$$\begin{aligned} & J(p,q)_{ij} = \delta_{ij} \; \forall \, ij \notin \{pq, qp, pp, qq\} \; ; \\ & J(p,q)_{pp} = \cos\phi = J(p,q)_{qq} \; ; \\ & J(p,q)_{pq} = \sin\phi = -J(p,q)_{qp} \; . \end{aligned} \tag{9.8}$$

After a Jacobi rotation, $A \to A' = J^T A J$, the matrix elements of A' become

$$\begin{aligned} A'_{ij} &= A_{ij} \; \forall \, i \neq p,q \wedge j \neq p,q \\ A'_{pi} &= A'_{ip} = cA_{pi} - sA_{qi} \; \forall \, i \neq p,q \; ; \\ A'_{qi} &= A'_{iq} = sA_{pi} + cA_{qi} \; \forall \, i \neq p,q \; ; \\ A'_{pp} &= c^2 A_{pp} - 2scA_{pq} + s^2 A_{qq} \; ; \\ A'_{qq} &= s^2 A_{pp} + 2scA_{pq} + c^2 A_{qq} \; ; \\ A'_{pq} &= A'_{qp} = sc(A_{pp} - A_{qq}) + (c^2 - s^2) A_{pq} \; , \end{aligned} \tag{9.9}$$

where $c \equiv \cos\phi$, $s \equiv \sin\phi$. The angle ϕ is chosen such that after rotation the matrix element A'_{pq} is zeroed,

$$\cot(2\phi) = \frac{A_{qq} - A_{pp}}{2A_{pq}} \; \Rightarrow \; A'_{pq} = 0 \; . \tag{9.10}$$

A side effect of zeroing a given off-diagonal element A_{pq} by a Jacobi rotation is that other off-diagonal elements are changed. Namely the elements of the rows and columns with indices equal to p and q. However, after the Jacobi rotation the sum of squares of all off-diagonal elemens is reduced. The algorithm repeatedly performs rotations until the off-diagonal elements become sufficiently small.

The convergence of the Jacobi method can be proved for two strategies for choosing the order in which the elements are zeroed:

1. *Classical method*: with each rotation the largest of the remaining off-diagonal elements is zeroed.

2. *Cyclic method*: the off-diagonal elements are zeroed in strict order, e.g. row after row.

Although the classical method allows the least number of rotations, it is typically slower than the cyclic method since searching for the largest element is an $O(n^2)$ operation. The count can be reduced by keeping an additional array with indexes of the largest elements in each row. Updating this array after each rotation is only an $O(n)$ operation.

A *sweep* is a sequence of Jacobi rotations applied to all non-diagonal elements. Typically the method converges after a small number of sweeps. The operation count is $O(n)$ for a Jacobi rotation and $O(n^3)$ for a sweep.

The typical convergence criterion is that the sum of absolute values of the off-diagonal elements is small, $\sum_{i<j} |A_{ij}| < \epsilon$, where ϵ is the required accuracy. Other criteria can also be used, like the largest off-diagonal element is small, $\max |A_{i<j}| < \epsilon$, or the diagonal elements have not changed after a sweep.

The eigenvectors can be calculated as $V = \mathbf{1}J_1J_2...$, where J_i are the successive Jacobi matrices. At each stage the transformation is

$$\begin{aligned} V_{ij} &\rightarrow V_{ij}\ ,\ j \neq p, q \\ V_{ip} &\rightarrow cV_{ip} - sV_{iq} \\ V_{iq} &\rightarrow sV_{ip} + cV_{iq} \end{aligned} \tag{9.11}$$

Alternatively, if only one (or few) eigenvector $\mathbf{v}_k$ is needed, one can instead solve the (singular) system $(A - \lambda_k)\mathbf{v} = 0$.

9.3 Power iteration methods

9.3.1 Power method

Power method is an iterative method to calculate an eigenvalue and the corresponding eigenvector using the iteration

$$\mathbf{x}_{i+1} = A\mathbf{x}_i\ . \tag{9.12}$$

The iteration converges to the eigenvector of the largest eigenvalue. The eigenvalue can be estimated using the *Rayleigh quotient*

$$\lambda[\mathbf{x}_i] = \frac{\mathbf{x}_i^T A\mathbf{x}_i}{\mathbf{x}_i^T\mathbf{x}_i} = \frac{\mathbf{x}_{i+1}^T\mathbf{x}_i}{\mathbf{x}_i^T\mathbf{x}_i}. \tag{9.13}$$

9.3.2 Inverse power method

The iteration with the inverse matrix

$$\mathbf{x}_{i+1} = A^{-1}\mathbf{x}_i \tag{9.14}$$

converges to the smallest eigenvalue of matrix A. Alternatively, the iteration

$$\mathbf{x}_{i+1} = (A - s)^{-1}\mathbf{x}_i \tag{9.15}$$

converges to an eigenvalue closest to the given number s.

9.3.3 Inverse iteration method

Inverse iteration method is the refinement of the inverse power method where the trick is not to invert the matrix in (9.15) but rather solve the linear system

$$(A - \lambda)\mathbf{x}_{i+1} = \mathbf{x}_i \tag{9.16}$$

using e.g. QR decomposition.

One can update the estimate for the eigenvalue using the Rayleigh quotient $\lambda[\mathbf{x}_i]$ after each iteration and get faster convergence for the price of $O(n^3)$ operations per QR-decomposition; or one can instead make more iterations (with $O(n^2)$ operations per iteration) using the same matrix $(A - \lambda)$. The optimal strategy is probably an update after several iterations.

9.4 JavaScript implementation

```
function jacobi(M){ // Jacobi diagonalization
// input: matrix M[][]; output: egenvalues E[], eigenvectors V[][];
  var V=[[(i==j?1:0) for (i in M)] for(j in M)]
  var A=M // in-place diagonalization, right triangle of M is
      destroyed
  var eps = 1e-12, rotated, sweeps=0;
  do{ rotated=0;
    for(var r=0;r<M.length;r++)for(var c=r+1;c<M.length;c++){//
        sweep
      if(Math.abs(A[c][r])>eps*(Math.abs(A[c][c])+Math.abs(A[r][r])
          )){
        rotated=1; rotate(r,c,A,V);}
      }sweeps++;//end sweep
    }while(rotated==1); //end do
  var E = [A[i][i] for (i in A)];
  return [E,V,sweeps];
} // end jacobi
```

```
function rotate(p,q,A,V){ // Jacobi rotation eliminating A_pq.
// Only upper triangle of A is updated.
// The matrix of eigenvectors V is also updated.
  if(q<p) [p,q]=[q,p]
  var n=A.length, app = A[p][p], aqq = A[q][q], apq = A[q][p];
  var phi=0.5*Math.atan2(2*apq, aqq-app);// could be done better
  var c=Math.cos(phi), s=Math.sin(phi);
  A[p][p] = c * c * app + s * s * aqq - 2 * s * c * apq;
  A[q][q] = s * s * app + c * c * aqq + 2 * s * c * apq;
  A[q][p]=0;
```

```
  for(var i=0;i<p;i++){
    var aip=A[p][i],aiq=A[q][i];
    A[p][i] = c*aip-s*aiq; A[q][i] = c*aiq+s*aip; }
  for(var i=p+1;i<q;i++){
    var api=A[i][p],aiq=A[q][i];
    A[i][p] = c*api-s*aiq; A[q][i] = c*aiq+s*api; }
  for(var i=q+1;i<n;i++){
    var api=A[i][p],aqi=A[i][q];
    A[i][p] = c*api-s*aqi; A[i][q] = c*aqi+s*api; }
  if(V!=undefined) //update eigenvectors
  for(var i=0;i<n;i++){
    var vip=V[p][i],viq=V[q][i];
    V[p][i] = c*vip-s*viq; V[q][i] = c*viq+s*vip; }
}//end rotate
```

Chapter 10

Power method and Krylov subspaces

10.1 Introduction

When calculating an eigenvalue of a matrix A using the power method, one starts with an initial random vector $\mathbf{b}$ and then computes iteratively the sequence $A\mathbf{b}, A^2\mathbf{b}, \ldots, A^{n-1}\mathbf{b}$ normalising and storing the result in $\mathbf{b}$ on each iteration. The sequence converges to the eigenvector of the largest eigenvalue of A.

The set of vectors

$$\mathcal{K}_n = \left\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \ldots, A^{n-1}\mathbf{b}\right\} , \tag{10.1}$$

where $n < \text{rank}(A)$, is called the order-n *Krylov matrix*, and the subspace spanned by these vectors is called the order-n *Krylov subspace*. The vectors are not orthogonal but can be made so e.g. by Gram-Schmidt orthogonalisation.

For the same reason that $A^{n-1}\mathbf{b}$ approximates the dominant eigenvector one can expect that the other orthogonalised vectors approximate the eigenvectors of the n largest eigenvalues.

Krylov subspaces are the basis of several successful iterative methods in numerical linear algebra, in particular: Arnoldi and Lanczos methods for finding one (or a few) eigenvalues of a matrix; and GMRES (Generalised Minimum RESidual) method for solving systems of linear equations.

These methods are particularly suitable for large sparse matrices as they avoid matrix-matrix operations but rather multiply vectors by matrices and work with the resulting vectors and matrices in Krylov subspaces of modest sizes.

10.2 Arnoldi iteration

Arnoldi iteration is an algorithm where the order-n orthogonalised Krylov matrix Q_n of a matrix A is built using stabilised Gram-Schmidt process:

- start with a set $Q = \{\mathbf{q}_1\}$ of one random normalised vector $\mathbf{q}_1$
- repeat for $k = 2$ to n :

– make a new vector $\mathbf{q}_k = A\mathbf{q}_{k-1}$
– orthogonalise $\mathbf{q}_k$ to all vectors $\mathbf{q}_i \in Q$ storing $\mathbf{q}_i^\dagger \mathbf{q}_k \to h_{i,k-1}$
– normalise $\mathbf{q}_k$ storing $\|\mathbf{q}_k\| \to h_{k,k-1}$
– add $\mathbf{q}_k$ to the set Q

By construction the matrix H_n made of the elements h_{jk} is an upper Hessenberg matrix,

$$H_n = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,n} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,n} \\ 0 & h_{3,2} & h_{3,3} & \cdots & h_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n,n-1} & h_{n,n} \end{bmatrix} , \tag{10.2}$$

which is a partial orthogonal reduction of A into Hessenberg form,

$$H_n = Q_n^\dagger A Q_n \; . \tag{10.3}$$

The matrix H_n can be viewed as a representation of A in the Krylov subspace $\mathcal{K}_n$. The eigenvalues and eigenvectors of the matrix H_n approximate the largest eigenvalues of matrix A.

Since H_n is a Hessenberg matrix of modest size its eigenvalues can be relatively easily computed with standard algorithms.

In practice if the size n of the Krylov subspace becomes too large the method is restarted.

10.3 Lanczos iteration

Lanczos iteration is Arnoldi iteration for Hermitian matrices, in which case the Hessenberg matrix H_n of Arnoldi method becomes a tridiagonal matrix T_n.

The Lanczos algorithm thus reduces the original hermitian $N \times N$ matrix A into a smaller $n \times n$ tridiagonal matrix T_n by an orthogonal projection onto the order-n Krylov subspace. The eigenvalues and eigenvectors of a tridiagonal matrix of a modest size can be easily found by e.g. the QR-diagonalisation method.

In practice the Lanczos method is not very stable due to round-off errors leading to quick loss of orthogonality. The eigenvalues of the resulting tridiagonal matrix may then not be a good approximation to the original matrix. Library implementations fight the stability issues by trying to prevent the loss of orthogonality and/or to recover the orthogonality after the basis is generated.

10.4 Generalised minimum residual (GMRES)

GMRES is an iterative method for the numerical solution of a system of linear equations,

$$A\mathbf{x} = \mathbf{b} \, , \tag{10.4}$$

where the exact solution $\mathbf{x}$ is approximated by the vector $\mathbf{x}_n \in \mathcal{K}_n$ that minimises the residual $A\mathbf{x}_n - \mathbf{b}$ in the Krylov subspace $\mathcal{K}_n$ of matrix A,

$$\mathbf{x} \approx \mathbf{x}_n \leftarrow \min_{\mathbf{x} \in \mathcal{K}_n} \|A\mathbf{x} - \mathbf{b}\| \; . \tag{10.5}$$

Chapter 11

Fast Fourier transform

Fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT).

Computing DFT of N points in the naive way, using the definition, takes $O(N^2)$ arithmetic operations, while an FFT can compute the same result in only $O(N \log N)$ operations. The difference in speed can be substantial, especially for large data sets. This improvement made many DFT-based algorithms practical.

Since the inverse of a DFT is also a DFT any FFT algorithm can be used in for the inverse DFT as well.

The most well known FFT algorithms, like the Cooley-Tukey algorithm, depend upon the factorization of N. However, there are FFTs with $O(N \log N)$ complexity for all N, even for prime N.

11.1 Discrete Fourier Transform

For a set of complex numbers x_n, $n = 0, \dots, N-1$, the DFT is defined as a set of complex numbers c_k,

$$c_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \, , k = 0, \dots, N-1 \, . \tag{11.1}$$

The inverse DFT is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} c_k e^{+2\pi i \frac{nk}{N}} \, . \tag{11.2}$$

These transformations can be viewed as expansion of the vector x_n in terms of the orthogonal basis of vectors $e^{2\pi i \frac{kn}{N}}$,

$$\sum_{n=0}^{N-1} \left(e^{2\pi i \frac{kn}{N}}\right) \left(e^{-2\pi i \frac{k'n}{N}}\right) = N\delta_{kk'} \, . \tag{11.3}$$

The DFT represent the amplitude and phase of the different sinusoidal components in the input data x_n.

The DFT is widely used in different fields, like spectral analysis, data compression, solution of partial differential equations and others.

11.1.1 Applications

Data compression

Several lossy (that is, with certain loss of data) image and sound compression methods employ DFT as an approximation for the Fourier series. The signal is discretized and transformed, and then the Fourier coefficients of high/low frequencies, which are assumed to be unnoticeable, are discarded. The decompressor computes the inverse transform based on this reduced number of Fourier coefficients.

Partial differential equations

Discrete Fourier transforms are often used to solve partial differential equations, where the DFT is used as an approximation for the Fourier series (which is recovered in the limit of infinite N). The advantage of this approach is that it expands the signal in complex exponentials e^{inx}, which are eigenfunctions of differentiation: $\frac{d}{dx}e^{inx} = ine^{inx}$. Thus, in the Fourier representation, differentiation is simply multiplication by in.

A linear differential equation with constant coefficients is transformed into an easily solvable algebraic equation. One then uses the inverse DFT to transform the result back into the ordinary spatial representation. Such an approach is called a *spectral method*.

Convolution and Deconvolution

DFT can be used to efficiently compute convolutions of two sequences. A convolution is the pairwise product of elements from two different sequences, such as in multiplying two polynomials or multiplying two long integers.

Another example comes from data acquisition processes where the detector introduces certain (typically Gaussian) blurring to the sampled signal. A reconstruction of the original signal can be obtained by deconvoluting the acquired signal with the detector's blurring function.

11.2 Cooley-Tukey algorithm

In its simplest incarnation this algorithm re-expresses the DFT of size $N = 2M$ in terms of two DFTs of size M,

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-2\pi i \frac{mk}{M}} + e^{-2\pi i \frac{k}{N}} \sum_{m=0}^{M-1} x_{2m+1} e^{-2\pi i \frac{mk}{M}} \\ &= \begin{cases} c_k^{(\text{even})} + e^{-2\pi i \frac{k}{N}} c_k^{(\text{odd})} & , k < M \\ c_{k-M}^{(\text{even})} - e^{-2\pi i \frac{k-M}{N}} c_{k-M}^{(\text{odd})} & , k \geq M \end{cases} , \end{aligned} \tag{11.4}$$

where $c^{(\text{even})}$ and $c^{(\text{odd})}$ are the DFTs of the even- and odd-numbered sub-sets of x.

This re-expression of a size-N DFT as two size-$\frac{N}{2}$ DFTs is sometimes called the Danielson-Lanczos lemma. The exponents $e^{-2\pi i\frac{k}{N}}$ are called *twiddle factors*.

The operation count by application of the lemma is reduced from the original N^2 down to $2(N/2)^2 + N/2 = N^2/2 + N/2 < N^2$.

For $N = 2^p$ Danielson-Lanczos lemma can be applied recursively until the data sets are reduced to one datum each. The number of operations is then reduced to $O(N \ln N)$ compared to the original $O(N^2)$. The established library FFT routines, like FFTW and GSL, further reduce the operation count (by a constant factor) using advanced programming techniques like precomputing the twiddle factors, effective memory management and others.

11.3 Multidimensional DFT

For example, a two-dimensional set of data $x_{n_1n_2}$, $n_1 = 1 \dots N_1$, $n_2 = 1 \dots N_2$ has the discrete Fourier transform

$$c_{k_1k_2} = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} x_{n_1n_2} e^{-2\pi i\frac{n_1k_1}{N_1}} e^{-2\pi i\frac{n_2k_2}{N_2}} . \tag{11.5}$$

11.4 C implementation

```
#include<complex.h>
#include<tgmath.h>
#define PI 3.14159265358979323846264338327950288
void dft (int N, complex* x, complex* c, int sign){
  complex w = cexp(sign*2*PI*I/N);
  for(int k=0;k<N;k++){
    complex sum=0; for(int n=0;n<N;n++) sum+=x[n]*cpow(w,n*k);
    c[k]=sum/sqrt(N);}
}
void fft (int N, complex* x, complex* c, int sign){
  if(N%2==0){
    complex w = exp(sign*2*PI*I/N);
    int M=N/2;
    complex xo[M],co[M],xe[M],ce[M]; //VLA: compile with -std=c99
    for(int m=0;m<M;m++){xo[m]=x[2*m+1];xe[m]=x[2*m];}
    fft(M,xo,co,sign); fft(M,xe,ce,sign);
    for(int k=0;k<M;k++) c[k]=(ce[k]+cpow(w,k)*co[k])/sqrt(2);
    for(int k=M;k<N;k++) c[k]=(ce[k-M]+cpow(w,k)*co[k-M])/sqrt(2);
    }
  else dft(N,x,c,sign);
}
```

Index