Yet Another
Introduction to Numerical Methods

version 13.05

D.V. Fedorov

[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. The book contains short descriptions of some of the most common numerical methods together with illustrational implementations of the discussed algorithms mostly in the C programming language. The implementations are solely for educational purposes, they are not thoroughly tested and are not tuned in any way other than to fit nicely on a single page of the book; some simple optimizations were often sacrificed to conciseness. For serious calculations one should use the GNU Scientific Library.

2013
Dmitri Fedorov

Contents

Chapter 1

Interpolation

1.1 Introduction

In practice one often meets a situation where the function of interest, $f(x)$, is only represented by a discrete set of tabulated points,

$$\{x_i, y_i = f(x_i) \mid i = 1 \dots n\},$$

obtained, for example, by sampling, experimentation, or extensive numerical calculations.

Interpolation means constructing a (smooth) function, called *interpolating function* or *interpolant*, 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.

1.2 Polynomial interpolation

Polynomial interpolation uses a polynomial as the interpolating function. Given a table of n points, $\{x_i, y_i\}$, where no two x_i are the same, one can construct a polynomial $P(x)$ of the order $n-1$ which passes exactly through the points: $P(x_i) = y_i$. This polynomial can be intuitively written in the *Lagrange form*,

$$P(x) = \sum_{i=1}^{n} y_i \prod_{k \neq i}^{n} \frac{x - x_k}{x_i - x_k} \,. \tag{1.1}$$

The interpolating polynomial always exists and is unique.

Higher order interpolating polynomials are susceptible to the *Runge's phenomenon*: erratic oscillations close to the end-points of the interval (see Figure 1.1).

Table 1.1: Polynomial interpolation in C

```
double polinterp(int n, double *x, double *y, double z) {
  double s=0;
  for(int i=0;i<n;i++) {
    double p=1;
    for(int k=0;k<n;k++) if(k!=i) p*=(z-x[k])/(x[i]-x[k]);
    s+=y[i]*p; }
  return s; }
```

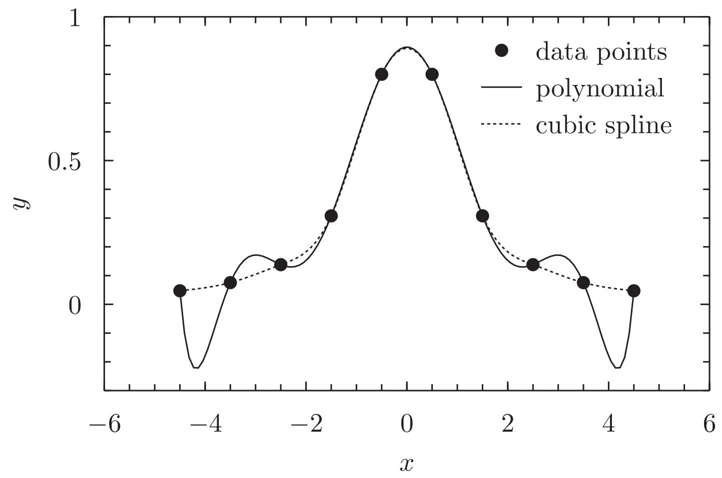

Figure 1.1: Lagrange interpolating polynomial, solid line, showing the Runge's phenomenon: large oscillations at the edges. For comparison the dashed line shows a cubic spline.

1.3 Spline interpolation

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

$$S(x) = S_i(x) \;, \text{ if } x \in [x_i, x_{i+1}] \;, \; i = 1, \dots, n-1 \tag{1.2}$$

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,

$$S_i(x_i) = y_i \;, \; S_i(x_{i+1}) = y_{i+1} \;, \; i = 1, \dots, n-1 \;, \tag{1.3}$$

together with its $k-1$ derivatives,

$$\left.\begin{array}{r} S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}) \;, \\ S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}) \;, \\ \vdots \\ S_i^{(k-1)}(x_{i+1}) = S_{i+1}^{(k-1)}(x_{i+1}) \;. \end{array}\right| \; i = 1, \dots, n-2 \tag{1.4}$$

Continuity conditions (1.3) and (1.4) make $kn + n - 2k$ linear equations for the $(n-1)(k+1) = kn+n-k-1$ coefficients of $n-1$ polynomials (1.2) 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 has a nice feature that it (sort of) minimizes the total curvature of the interpolating function. This makes the cubic splines look good.

Quadratic splines—continuous with the first derivative—are not nearly as good as cubic splines in most respects. In particular they might oscillate unpleasantly when a quick change in the tabulated function is followed by a period where the function is nearly a constant. Cubic splines are somewhat less susceptible to such oscillations.

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

1.3.1 Linear interpolation

Linear interpolation is a spline with linear polynomials. The continuity conditions (1.3) can be satisfied by choosing the spline in the (intuitive) form

$$S_i(x) = y_i + p_i(x - x_i) \ , \tag{1.5}$$

where

$$p_i = \frac{\Delta y_i}{\Delta x_i} \ , \ \ \Delta y_i \doteq y_{i+1} - y_i \ , \ \ \Delta x_i \doteq x_{i+1} - x_i \ . \tag{1.6}$$

Table 1.2: Linear interpolation in C

```c
#include<assert.h>
double linterp(int n, double* x, double* y, double z){
  assert(z>=x[0] && z<=x[n-1] && n>1);
  int i=0, j=n-1; /* binary search: */
  while(j-i>1){int m=(i+j)/2; if(z>x[m]) i=m; else j=m;}
  return y[i]+(y[i+1]-y[i])/(x[i+1]-x[i])*(z-x[i]);
}
```

Note that the search of the interval $[x_i \le x \le x_{i+1}]$ in an ordered array $\{x_i\}$ should be done with the *binary search* algoritm (also called *half-interval search*): the point x is compared to the middle element of the array, if it is less than the middle element, the algorithm repeats its action on the sub-array to the left of the middle element, if it is greater, on the sub-array to the right. When the remaining sub-array is reduced to two elements, the interval is found. The average number of operations for a binary search is $O(\log n)$.

1.3.2 Quadratic spline

Quadratic spline is made of second order polynomials, conveniently written in the form

$$S_i(x) = y_i + p_i(x - x_i) + c_i(x - x_i)(x - x_{i+1}) \Big|_{i=1,\dots,n-1} , \qquad (1.7)$$

which identically satisfies the continuity conditions

$$S_i(x_i) = y_i \ , \ S_i(x_{i+1}) = y_{i+1} \Big|_{i=1,\dots,n-1} . \qquad (1.8)$$

Substituting (1.7) into the derivative continuity condition,

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}) \Big|_{i=1,\dots,n-2} , \qquad (1.9)$$

gives $n-2$ equations for $n-1$ unknown coefficients c_i,

$$p_i + c_i \Delta x_i = p_{i+1} - c_{i+1} \Delta x_{i+1} \Big|_{i=1,\dots,n-2} . \qquad (1.10)$$

One coefficient can be chosen arbitrarily, for example $c_1 = 0$. The other coefficients can now be calculated recursively,

$$c_{i+1} = \frac{1}{\Delta x_{i+1}} (p_{i+1} - p_i - c_i \Delta x_i) \Big|_{i=1,\dots,n-2} . \qquad (1.11)$$

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

$$c_i = \frac{1}{\Delta x_i} (p_{i+1} - p_i - c_{i+1} \Delta x_{i+1}) \Big|_{i=n-2,\dots,1} . \qquad (1.12)$$

In practice, unless you know what your c_1 (or c_{n-1}) is, it is better to run both recursions and then average the resulting c's. This amounts to first running the forward-recursion from $c_1 = 0$ and then the backward recursion from $\frac{1}{2}c_{n-1}$.

The optimized form (1.7) of the quadratic spline can aslo be written in the ordinary form, suitable for differentiation and integration

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 \ , \text{ where } b_i = p_i - c_i \Delta x_i \ . \qquad (1.13)$$

An implementation of quadratic spline in C is listed in Table 1.3.2

1.3.3 Cubic spline

Cubic splines are made of third order polynomials,

$$S_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \ . \qquad (1.14)$$

Table 1.3: Quadratic spline in C

```
#include <stdlib.h>
#include <assert.h>
typedef struct {int n; double *x, *y, *b, *c;} qspline;
qspline* qspline_alloc(int n,double* x,double* y){
  //builds quadratic spline
  qspline *s = (qspline*)malloc(sizeof(qspline)); //spline
  s->b = (double*)malloc((n-1)*sizeof(double)); // b_i
  s->c = (double*)malloc((n-1)*sizeof(double)); // c_i
  s->x = (double*)malloc(n*sizeof(double)); //copy of x_i
  s->y = (double*)malloc(n*sizeof(double)); //copy of y_i
  s->n = n; for(int i=0;i<n;i++){s->x[i]=x[i]; s->y[i]=y[i];}
  int i; double p[n-1], h[n-1]; //VLA from C99
  for(i=0;i<n-1;i++){h[i]=x[i+1]-x[i]; p[i]=(y[i+1]-y[i])/h[i];}
  s->c[0]=0; //recursion up:
  for(i=0;i<n-2;i++)s->c[i+1]=(p[i+1]-p[i]-s->c[i]*h[i])/h[i+1];
  s->c[n-2]/=2; //recursion down:
  for(i=n-3;i>=0;i--)s->c[i]=(p[i+1]-p[i]-s->c[i+1]*h[i+1])/h[i];
  for(i=0;i<n-1;i++)s->b[i]=p[i]-s->c[i]*h[i];
  return s; }
double qspline_eval(qspline *s, double z){ //evaluates s(z)
  assert(z>=s->x[0] && z<=s->x[s->n-1]);
  int i=0, j=s->n-1; //binary search:
  while(j-i>1){int m=(i+j)/2; if(z>s->x[m]) i=m; else j=m;}
  double h=z-s->x[i];
  return s->y[i]+h*(s->b[i]+h*s->c[i]); }//inerpolating polynomial
void qspline_free(qspline *s){ //free the allocated memory
  free(s->x); free(s->y); free(s->b); free(s->c); free(s); }
```

This form automatically satisfies the first half of continuity conditions (1.3): $S_i(x_i) = y_i$. The second half of continuity conditions (1.3), $S_i(x_{i+1}) = y_{i+1}$, and the continuity of the first and second derivatives (1.4) give a set of equations,

$$
\begin{aligned}
y_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= y_{i+1} \ , \ i = 1, \ldots, n-1 \\
b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \ , \ i = 1, \ldots, n-2 \\
2c_i + 6d_i h_i &= 2c_{i+1} \ , \ i = 1, \ldots, n-2
\end{aligned}
\tag{1.15}
$$

where

$$h_i = x_{i+1} - x_i \ . \tag{1.16}$$

The set of equations (1.15) 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 given as vanishing second derivative at the end-points,

$$S''(x_1) = S''(x_n) = 0 \;, \tag{1.17}$$

which gives

$$\begin{aligned} c_1 &= 0 \;, \\ c_{n-1} + 3d_{n-1}h_{n-1} &= 0 \;. \end{aligned} \tag{1.18}$$

Solving the first two equations in (1.15) for c_i and d_i gives[1]

$$\begin{aligned} c_i h_i &= -2b_i - b_{i+1} + 3p_i \;, \\ d_i h_i^2 &= b_i + b_{i+1} - 2p_i \;, \end{aligned} \tag{1.19}$$

where $p_i \doteq \frac{\Delta y_i}{h_i}$. The natural conditions (1.18) and the third equation in (1.15) then produce the following tridiagonal system of n linear equations for the n coefficients b_i,

$$\begin{aligned} 2b_1 + b_2 &= 3p_1 \;, \\ b_i + (2\frac{h_i}{h_{i+1}} + 2)b_{i+1} + \frac{h_i}{h_{i+1}}b_{i+2} &= 3(p_i + p_{i+1}\frac{h_i}{h_{i+1}}) \Big|_{i=1,\dots,n-2} \;, \\ b_{n-1} + 2b_n &= 3p_{n-1} \;, \end{aligned} \tag{1.20}$$

or, in the matrix form,

$$\begin{pmatrix} D_1 & Q_1 & 0 & 0 & \dots \\ 1 & D_2 & Q_2 & 0 & \dots \\ 0 & 1 & D_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 1 & D_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} B_1 \\ \vdots \\ \vdots \\ B_n \end{pmatrix} \tag{1.21}$$

where the elements D_i at the main diagonal are

$$D_1 = 2 \;,\; D_{i+1} = 2\frac{h_i}{h_{i+1}} + 2 \Big|_{i=1,\dots,n-2} \;,\; D_n = 2 \;, \tag{1.22}$$

the elements Q_i at the above-main diagonal are

$$Q_1 = 1 \;,\; Q_{i+1} = \frac{h_i}{h_{i+1}} \Big|_{i=1,\dots,n-2} \;, \tag{1.23}$$

and the right-hand side terms B_i are

$$B_1 = 3p_1 \;,\; B_{i+1} = 3\left(p_i + p_{i+1}\frac{h_i}{h_{i+1}}\right) \Big|_{i=1,\dots,n-2} \;, B_n = 3p_{n-1} \;. \tag{1.24}$$

[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

$$\begin{pmatrix} \tilde{D}_1 & Q_1 & 0 & 0 & \dots \\ 0 & \tilde{D}_2 & Q_2 & 0 & \dots \\ 0 & 0 & \tilde{D}_3 & Q_3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \dots & \dots & 0 & 0 & \tilde{D}_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} \tilde{B}_1 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{pmatrix}, \tag{1.25}$$

where

$$\tilde{D}_1 = D_1 \, , \; \tilde{D}_i = D_i - Q_{i-1}/\tilde{D}_{i-1} \Big|_{i=2,\dots,n} \, , \tag{1.26}$$

and

$$\tilde{B}_1 = B_1 \, , \; \tilde{B}_i = B_i - \tilde{B}_{i-1}/\tilde{D}_{i-1} \Big|_{i=2,\dots,n} \, . \tag{1.27}$$

The triangular system (1.25) can be solved by a run of back-substitution,

$$b_n = \frac{1}{\tilde{D}_n}\tilde{B}_n \, , \; b_i = \frac{1}{\tilde{D}_i}(\tilde{B}_i - Q_i b_{i+1}) \Big|_{i=n-1,\dots,1} \, . \tag{1.28}$$

An implementation of quadratic spline in C is listed in Table 1.3.3

1.3.4 Akima sub-spline interpolation

Akima sub-spline is an interpolating function in the form of a piecewise cubic polynomial [1], similar to a cubic spline,

$$\mathcal{A}(x)\Big|_{x\in[x_i,x_{i+1}]} = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \doteq A_i(x)\, . \tag{1.29}$$

However, unlike the cubic spline, Akima sub-spline dispenses with the demand of maximal differentiability of the spline—in this case, the continuity of the second derivative—hence the name *sub-spline*. Instead of achieving maximal differentiability Akima sub-splines try to reduce the wiggling the ordinary splines are typically prone to (see figure 1.2).

First let us note that the coefficients $\{a_i, b_i, c_i, d_i\}$ in eq. (1.29) are determined by the values of the derivatives $\mathcal{A}'_i \doteq \mathcal{A}'(x_i)$ of the sub-spline through the continuity conditions for the sub-spline and its first derivative,

$$A_i(x_i) = y_i, \; A'_i(x_i) = \mathcal{A}'_i, \; A_i(x_{i+1}) = y_{i+1}, \; A'_i(x_{i+1}) = \mathcal{A}'_{i+1}. \tag{1.30}$$

Inserting (1.29) into (1.30) and solving for the coefficients gives

$$a_i = y_i, \; b_i = \mathcal{A}'_i, \; c_i = \frac{3p_i - 2\mathcal{A}'_i - \mathcal{A}'_{i+1}}{\Delta x_i}, \; d_i = \frac{\mathcal{A}'_i + \mathcal{A}'_{i+1} - 2p_i}{(\Delta x_i)^2}, \tag{1.31}$$

where $p_i \doteq \Delta y_i/\Delta x_i$, $\Delta y_i \doteq y_{i+1} - y_i$, $\Delta x_i \doteq x_{i+1} - x_i$.

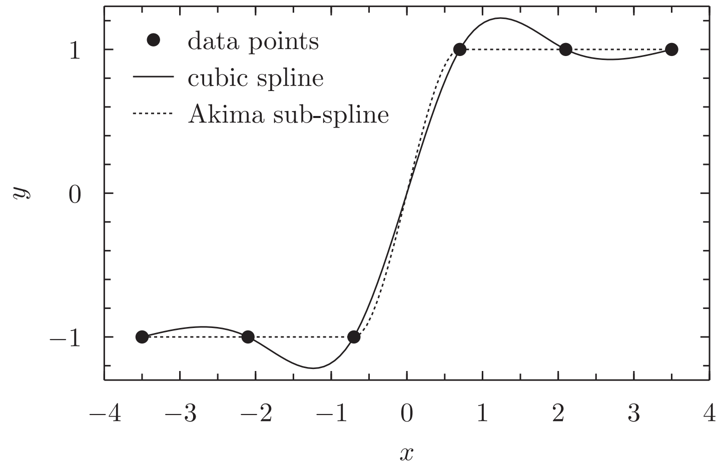

Figure 1.2: A cubic spline (solid line) showing the typical wiggles, compared to the Akima sub-spline (dashed line) where the wiggles are essentially removed.

In the ordinary qubic spline the derivatives $\mathcal{A}'_i$ are determined by the continuity condition of the second derivative of the spline. Sub-splines do without this continuity condition and can instead use the derivatives as free parameters to be chosen to satisfy some other condition.

Akima suggested to minimize the wiggling by choosing the derivatives as linear combinations of the nearest slopes,

$$\mathcal{A}'_i = \frac{w_{i+1}p_{i-1} + w_{i-1}p_i}{w_{i+1} + w_{i-1}} \,, \quad \text{if} \quad w_{i+1} + w_{i-1} \neq 0 \,, \tag{1.32}$$

$$\mathcal{A}'_i = \frac{p_{i-1} + p_i}{2} \,, \quad \text{if} \quad w_{i+1} + w_{i-1} = 0 \,, \tag{1.33}$$

where the weights w_i are given as

$$w_i = |p_i - p_{i-1}| \,. \tag{1.34}$$

The idea is that if three points lie close to a line, the sub-spline in this vicinity has to be close to this line. In other words, if $|p_i - p_{i-1}|$ is small, the nearby derivatives must be close to p_i.

The first two and the last two points need a special prescription, for example (naively) one can simply use

$$\mathcal{A}'_1 = p_1,\ \mathcal{A}'_1 = \frac{1}{2}p_1 + \frac{1}{2}p2,\ \mathcal{A}'_n = p_{n-1},\ \mathcal{A}'_{n-1} = \frac{1}{2}p_{n-1} + \frac{1}{2}p_{n-2}. \tag{1.35}$$

Table (1.5) shows a javascript implementation of this algorithm.

1.4 Other forms of interpolation

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

Sometimes not only the values of the function are tabulated but also the values of its derivative. This extra information can be taken advantage of when constructing the interpolation function.

1.5 Multivariate interpolation

Interpolation of a function in more than one varibale is called *multivariate interpolation*. The function of interest is represented as a set of discrete points in a multidimentional space. The points may or may not lie on a regular grid.

1.5.1 Nearest-neighbor interpolation

Nearest-neighbor interpolation approximates the value of the function at a non-tabulated point by the value at the nearest tabulated point, yielding a piecewise-constant interpolating function. It can be used for both regular and irregular grids.

1.5.2 Piecewise-linear interpolation

Piecewise-linear interpolation is used to interpolate functions of two variables tabulated on irregular grids. The tabulated 2D region is triangulated – subdivided into a set of non-intersecting triangles whose union is the original region. Inside each triangle the interpolating function $S(x, y)$ is taken in the linear form,

$$S(x, y) = a + bx + cy \ , \tag{1.36}$$

where the three constants are determined by the three conditions that the interpolating function is equal the tabulated values at the three vertices of the triangle.

1.5.3 Bi-linear interpolation

Bi-linear interpolation is used to interpolate functions of two variables tabulated on regular rectilinear 2D grids.

The interpolating function $B(x, y)$ inside each of the grid rectangles is taken as a product of two linear functions of x and y respectively,

$$B(x, y) = (\alpha + \beta x)(\gamma + \delta y) \doteq a + bx + cy + dxy \ , \tag{1.37}$$

where the four constants a, b, c, d are obtained from the four conditions that the interpolating function is equal the tabulated values at the four nearest tabulated points (which are the vertices of the given grid rectangle).

Table 1.4: Cubic spline in C

```
#include<stdlib.h>
#include<assert.h>
typedef struct {int n; double *x,*y,*b,*c,*d;} cspline;
cspline* cspline_alloc(int n, double *x, double *y)
{// builds natural cubic spline
  cspline* s = (cspline*)malloc(sizeof(cspline));
  s->x = (double*)malloc(n*sizeof(double));
  s->y = (double*)malloc(n*sizeof(double));
  s->b = (double*)malloc(n*sizeof(double));
  s->c = (double*)malloc((n-1)*sizeof(double));
  s->d = (double*)malloc((n-1)*sizeof(double));
  s->n = n; for(int i=0;i<n;i++){s->x[i]=x[i]; s->y[i]=y[i];}
  double h[n-1],p[n-1]; // VLA
  for(int i=0;i<n-1;i++) h[i]=x[i+1]-x[i];
  for(int i=0;i<n-1;i++) p[i]=(y[i+1]-y[i])/h[i];
  double D[n], Q[n-1], B[n];// building the tridiagonal system:
  D[0]=2; for(int i=0;i<n-2;i++)D[i+1]=2*h[i]/h[i+1]+2; D[n-1]=2;
  Q[0]=1; for(int i=0;i<n-2;i++)Q[i+1]=h[i]/h[i+1];
  for(int i=0;i<n-2;i++)B[i+1]=3*(p[i]+p[i+1]*h[i]/h[i+1]);
  B[0]=3*p[0]; B[n-1]=3*p[n-2]; //Gauss elimination :
  for(int i=1;i<n;i++){ D[i]-=Q[i-1]/D[i-1]; B[i]-=B[i-1]/D[i-1]; }
  s->b[n-1]=B[n-1]/D[n-1]; //back-substitution :
  for(int i=n-2;i>=0;i--) s->b[i]=(B[i]-Q[i]*s->b[i+1])/D[i];
  for(int i=0;i<n-1;i++){
    s->c[i]=(-2*s->b[i]-s->b[i+1]+3*p[i])/h[i];
    s->d[i]=(s->b[i]+s->b[i+1]-2*p[i])/h[i]/h[i];
  }
  return s;
}
double cspline_eval(cspline*s,double z){ // evaluation of spline
    assert(z>=s->x[0] && z<=s->x[s->n-1]);
    int i=0, j=s->n-1;// binary search for the interval for z :
    while(j-i>1){int m=(i+j)/2; if(z>s->x[m]) i=m; else j=m; }
    double h=z-s->x[i];// calculate the inerpolating spline :
    return s->y[i]+h*(s->b[i]+h*(s->c[i]+h*s->d[i]));
}
void cspline_free(cspline *s){ //free the allocated memory
  free(s->x);free(s->y);free(s->b);free(s->c);free(s->d);free(s);}
```

Table 1.5: Javascript implementation of Akima sub-spline.

```
Array.prototype.__iterator__=function()
  {for(var i=0;i<this.length;i++)yield i}

function akima(x,y){
  var n=x.length;
  var h=[(x[i+1]-x[i]) for(i in x) if(i<n-1)];
  var p=[(y[i+1]-y[i])/h[i] for(i in h)];
  var A=new Array(n), c=new Array(n-1), d=new Array(n-1);

  A[0]=p[0]; A[1]=(p[0]+p[1])/2;
  A[n-1]=p[n-2]; A[n-2]=(p[n-2]+p[n-3])/2;
  for(var i=2;i<n-2;i++)
  { w1=Math.abs(p[i+1]-p[i]); w2=Math.abs(p[i-1]-p[i-2]);
    if(w1+w2==0) A[i]=(p[i-1]+p[i])/2;
    else A[i]=(w1*p[i-1]+w2*p[i])/(w1+w2);
  }

  for(var i=0;i<n-1;i++)
  { c[i]=(3*p[i]-2*A[i]-A[i+1])/h[i];
    d[i]=(A[i+1]+A[i]-2*p[i])/h[i]/h[i];
  }

  var spline=function(z)
  { if(z<x[0] || z>x[n-1]) throw "spline: out of range";
    var i=0, j=n-1;
    while(j-i>1){
      var mid=Math.round( (i+j)/2 );
      if(z>x[mid]) i=mid; else j=mid;}
    var dx=z-x[i];
    return y[i]+dx*(A[i]+dx*(c[i]+dx*d[i]));
  }
  return spline;
}
```

Chapter 2

Numerical integration

2.1 Introduction

Numerical integration constitutes a broad family of algorithms to compute a numerical approximation to a definite (Riemann) integral.

Generally, the integral is approximated by a weighted sum of function values within the domain of integration,

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) . \tag{2.1}$$

Expression (2.1) is often referred to as *quadrature* (*cubature* for multidimensional integrals) or *rule*. The abscissas x_i (also called *nodes*) and the weights w_i of a quadrature are usually optimized—using one of a large number of different strategies—to suit a particular class of integration problems.

The best quadrature algorithm for a given problem depends on several factors, in particular on the integrand. Different classes of integrands generally require different quadratures for the most effective calculation.

A popular numerical integration library is QUADPACK [14]. It includes general purpose routines—like QAGS, basen on an adaptive Gauss–Kronrod quadrature with acceleration—as well as a number of specialized routines. The GNU scientific library [4] (GSL) implements most of the QUADPACK routines and in addition includes a modern general-purpose adaptive routine CQUAD based on Clenshaw-Curtis quadratures [9].

In the following we shall consider some of the popular numerical integration algorithms.

2.2 Rectangle and trapezium rules

In mathematics, the *Reimann integral* is generally defined in terms of *Riemann sums* [15]. If the integration interval $[a, b]$ is partitioned into n subintervals,

$$a = t_0 < t_1 < t_2 < \cdots < t_n = b \; . \tag{2.2}$$

the Riemann sum is defined as

$$\sum_{i=1}^{n} f(x_i)\Delta x_i \; , \tag{2.3}$$

where $x_i \in [t_{i-1}, t_i]$ and $\Delta x_i = t_i - t_{i-1}$. Geometrically a Riemann sum can be interpreted as the area of a collection of adjucent rectangles with widths Δx_i and heights $f(x_i)$.

The Rieman integral is defined as the limit of a Riemann sum as the *mesh*—the length of the largest subinterval—of the partition approaches zero. Specifically, the number denoted as

$$\int_a^b f(x)dx \tag{2.4}$$

is called the Riemann integral, if for any $\epsilon > 0$ there exists $\delta > 0$ such that for any partition (2.2) with $\max \Delta x_i < \delta$ we have

$$\left| \sum_{i=1}^{n} f(x_i)\Delta x_i - \int_a^b f(x)dx \right| < \epsilon \; . \tag{2.5}$$

A definite integral can be interpreted as the net signed area bounded by the graph of the integrand.

Now, the n-point *rectangle quadrature* is simply the Riemann sum (2.3),

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(x_i)\Delta x_i \; , \tag{2.6}$$

where the node x_i is often (but not always) taken at the middle of the corresponding subinterval, $x_i = t_{i-1} + \frac{1}{2}\Delta x_i$, and the subintervals are often (but not always) chosen equal, $\Delta x_i = (b-a)/n$. Geometrically the n-point rectangle rule is an approximation to the integral given by the area of a collection of n adjucent equal rectangles whose heights are determined by the values of the function (at the middle of the rectangle).

An n-point *trapezium rule* uses instead a collection of trapezia fitted under the graph,

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{f(t_{i-1}) + f(t_i)}{2}\Delta x_i \; . \tag{2.7}$$

Importantly, the trapezium rule is the average of two Riemann sums,

$$\sum_{i=1}^{n} \frac{f(t_{i-1}) + f(t_i)}{2}\Delta x_i = \frac{1}{2}\sum_{i=1}^{n} f(t_{i-1})\Delta x_i + \frac{1}{2}\sum_{i=1}^{n} f(t_i)\Delta x_i \; . \tag{2.8}$$

Rectangle and trapezium quadratures both have the important feature of closely following the very mathematical definition of the integral as the limit of the Riemann sums. Therefore—disregarding the round-off errors—these two rules cannot fail if the integral exists.

For certain partitions of the interval the rectangle and trapezium rules coincide. For example, for the nodes

$$x_i = a + (b-a)\frac{i-\frac{1}{2}}{n}\,, \quad i = 1, \dots, n \tag{2.9}$$

both rules give the same quadrature with equal weights, $w_i = (b-a)/n$,

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=1}^{n} f\left(a + (b-a)\frac{i-\frac{1}{2}}{n}\right)\,. \tag{2.10}$$

Rectangle and trapezium quadratures are rarely used on their own—because of the slow convergence—but they often serve as the basis for more advanced quadratures, for example adaptive quadratures and variable transformation quadratures considered below.

2.3 Quadratures with regularly spaced abscissas

A quadrature (2.1) with n predefined nodes x_i has n free parameters: the weights w_i. A set of n parameters can generally be tuned to satisfy n conditions. The archetypal set of conditions in quadratures is that the quadrature integrates exactly a set of n functions,

$$\{\phi_1(x), \dots, \phi_n(x)\}\,. \tag{2.11}$$

This leads to a set of n equations,

$$\sum_{i=1}^{n} w_i \phi_k(x_i) = I_k \Big|_{k=1,\dots,n}\,, \tag{2.12}$$

where the integrals

$$I_k \doteq \int_a^b \phi_k(x)dx \tag{2.13}$$

are assumed to be known. Equations (2.12) are linear in w_i and can be easily solved.

Since integration is a linear operation, the quadrature will then also integrate exactly any linear combination of functions (2.11).

A popular choice for predefined nodes is a *closed* set—that is, including the end-points of the interval—of evenly spaced abscissas,

$$x_i = a + \frac{i-1}{n-1}(b-a)\Big|_{i=1,\dots,n}\,. \tag{2.14}$$

However, in practice it often happens that the integrand has an integrable sinfularity at one or both ends of the interval. In this case one can choose an *open set* of equidistant nodes,

$$x_i = a + \frac{i-\frac{1}{2}}{n}(b-a)\Big|_{i=1,\dots,n} . \tag{2.15}$$

The set of functions to be integrated exactly is generally chosen to suite the properties of the integrands at hand: the integrands must be well represented by linear combinations of the chosen functions.

2.3.1 Classical quadratures

Suppose the integrand can be well represented by the first few terms of its Taylor series,

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k , \tag{2.16}$$

where $f^{(k)}$ is the k-th derivative of the integrand. This is often the case for analytic—that is, infinitely differentiable—functions. For such integrands one can obviously choose polynomials

$$\{1, x, x^2, \dots, x^{n-1}\} \tag{2.17}$$

as the set of functions to be integrated exactly.

This leads to the so called *classical quadratures*: quadratures with regularly spaced abscissas and *polynomials* as exactly integrable functions.

An n-point classical quadrature integrates exactly the first n terms of the function's Taylor expansion (2.16). The x^n order term will not be integrated exactly and will lead to an error of the quadrature. Thus the error E_n of the n-point classical quadrature is on the order of the integral of the x^n term in (2.16),

$$E_n \approx \int_a^b \frac{f^{(n)}(a)}{n!}(x-a)^n dx = \frac{f^{(n)}(a)}{(n+1)!} h^{n+1} \propto h^{n+1} , \tag{2.18}$$

where $h = b - a$ is the length of the integration interval. A quadrature with the error of the order h^{n+1} is often called a *degree-n* quadrature.

If the integrand is smooth enough and the length h is small enough a classical quadrature with not so large n can provide a good approximation for the integral. However, for large n the weights of classical quadratures tend to have alternating signs, which leads to large round-off errors, which in turn negates the potentially higher accuracy of the quadrature. Again, if the integrand violates the assumption of Taylor expansion—for example by having an integrable singularity inside the integration interval—the higher order quadratures may perform poorly.

Table 2.1: Maxima script to calculate analytically the weights of an n-point classical quadrature with predefined abscissas in the interval $[0, 1]$.

```
n: 8; xs: makelist((i-1)/(n-1),i,1,n); /* nodes: adapt to your needs */
ws: makelist(concat(w,i),i,1,n);
ps: append([1],makelist(x^i,i,1,n-1)); /* polynomials */
fs: makelist(buildq([i:i,ps:ps],lambda([x],ps[i])),i,1,n);
integ01: lambda([f],integrate(f(x),x,0,1));
Is: maplist(integ01,fs); /* calculate the integrals */
eq: lambda([f],lreduce("+",maplist(f,xs)*ws));
eqs: maplist(eq,fs)-Is; /* build equations */
solve(eqs,ws); /* solve for the weights */
```

Classical quadratures are mostly of historical interest nowadays. Alternative methods—such as quadratures with optimized abscissas, adaptive, and variable transformation quadratures—are more stable and accurate and are normally preferred to classical quadratures.

Classical quadratures with equally spaced abscissas—both closed and open sets—are generally referred to as *Newton-Cotes quadratures*. An interested reader can generate Newton—Cotes quadratures of any degree n using the Maxima script in Table (2.1).

2.4 Quadratures with optimized abscissas

In quadratures with optimized abscissas not only the weights w_i but also the abscissas x_i are chosen optimally. The number of free parameters is thus $2n$ and one can choose a set of $2n$ functions,

$$\{\phi_1(x), \dots, \phi_{2n}(x)\} , \tag{2.19}$$

to be integrated exactly. This gives a system of $2n$ equations, linear in w_i and non-linear in x_i,

$$\sum_{i=1}^{n} w_i \phi_k(x_i) = I_k \Big|_{k=1,\dots,2n} , \tag{2.20}$$

where again

$$I_k \doteq \int_a^b \phi_k(x) dx . \tag{2.21}$$

The weights and abscissas of the quadrature can be determined by solving this system of equations[1].

[1]Here is, for example, an $n = 2$ quadrature with optimized abscissas,

$$\int_{-1}^{1} f(x)dx \approx f\left(-\sqrt{\tfrac{1}{3}}\right) + f\left(+\sqrt{\tfrac{1}{3}}\right) . \tag{2.22}$$

Although quadratures with optimized abcissas are generally of much higher order, $2n-1$ compared to $n-1$ for non-optimal abscissas, the optimal points generally can not be reused at the next iteration in an adaptive algorithm.

2.4.1 Gauss quadratures

Gauss quadratures deal with a slightly more general form of integrals,

$$\int_a^b \omega(x)f(x)dx \;, \tag{2.23}$$

where $\omega(x)$ is a positive weight function. For $\omega(x) = 1$ the problem is the same as considered above. Popular choices of the weight function include $\omega(x) = (1-x^2)^{\pm 1/2}$, $\exp(-x)$, $\exp(-x^2)$ and others. The idea is to represent the integrand as a product $\omega(x)f(x)$ such that all the difficulties go into the weight function $\omega(x)$ while the remaining factor $f(x)$ is smooth and well represented by polynomials.

An N-point *Gauss quadrature* is a quadrature with optimized abcissas,

$$\int_a^b \omega(x)f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i) \;, \tag{2.24}$$

which integrates exactly a set of $2N$ polynomials of the orders $1, \ldots, 2N-1$ with the given weight $\omega(x)$.

Fundamental theorem

There is a theorem stating that there exists a set of polynomials $p_n(x)$, orthogonal on the interval $[a, b]$ with the weight function $\omega(x)$,

$$\int_a^b \omega(x)p_n(x)p_k(x) \propto \delta_{nk} \;. \tag{2.25}$$

Now, one can prove that the optimal nodes for the N-point Gauss quadrature are the roots of the polynomial $p_N(x)$,

$$p_N(x_i) = 0 \;. \tag{2.26}$$

The idea behind the proof is to consider the integral

$$\int_a^b \omega(x)q(x)p_N(x)dx = 0 \;, \tag{2.27}$$

where $q(x)$ is an arbitrary polynomial of degree less than N. The quadrature should represent this integral exactly,

$$\sum_{i=1}^{N} q(x_i)p_N(x_i) = 0 \; . \tag{2.28}$$

Apparently this is only possible if x_i are the roots of p_N ■.

Calculation of nodes and weights

A neat algorithm—usually refered to as Golub-Welsch [8] algorithm—for calculation of the nodes and weights of a Gauss quadrature is based on the symmetric form of the three-term reccurence relation for orthogonal polynomials,

$$xp_{n-1}(x) = \beta_n p_n(x) + \alpha_n p_{n-1}(x) + \beta_{n-1}p_{n-2}(x) \; , \tag{2.29}$$

where $p_{-1}(x) \doteq 0$, $p_1(x) \doteq 1$, and $n = 1, \ldots, N$. This reccurence relation can be written in the matrix form,

$$x\mathbf{p}(x) = \mathbf{J}\mathbf{p}(x) + \beta_N p_N(x)\mathbf{e}_N \; , \tag{2.30}$$

where $\mathbf{p}(x) \doteq \{p_0(x), \ldots, p_{N-1}(x)\}^T$, $\mathbf{e}_N = \{0, \ldots, 0, 1\}^T$, and the tridiagonal matrix $\mathbf{J}$ — usually refered to as *Jacobi matrix* or *Jacobi operator* — is given as

$$\mathbf{J} = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \beta_3 & \\ & & \ddots & \ddots & \\ & & & \beta_{N-1} & \alpha_N \end{pmatrix} . \tag{2.31}$$

Substituting the roots x_i of p_N—that is, the set $\{x_i \mid p_N(x_i) = 0\}$—into the matrix equation (2.30) leads to eigenvalue problem for the Jacobi matrix,

$$\mathbf{J}\mathbf{p}(x_i) = x_i\mathbf{p}(x_i) \; . \tag{2.32}$$

Thus, the nodes of an N-point Gauss quadrature—the roots of the polynomial p_N—are the eigenvalues of the Jacobi matrix J and can be calculated by a standard diagonalization[2] routine ■.

The weights can be obtained considering N integrals,

$$\int_a^b \omega(x)p_n(x)dx = \delta_{n0}\int_a^b \omega(x)dx \, , \; n = 0, \ldots, N-1 \, . \tag{2.33}$$

[2]A symmetric tridiagonal matrix can be diagonalized very effectively using the QR/RL algorithm.

Applying our quadrature gives the matrix equation,

$$\mathbf{P}\mathbf{w} = \mathbf{e}_1 \int_a^b \omega(x)dx \; , \tag{2.34}$$

where $\mathbf{w} \doteq \{w_1, \ldots, w_N\}^T$, $\mathbf{e}_1 = \{1, 0, \ldots, 0\}^T$, and

$$\mathbf{P} \doteq \begin{pmatrix} p_0(x_1) & \ldots & p_0(x_N) \\ p_1(x_1) & \ldots & p_1(x_N) \\ \ldots & \ldots & \ldots \\ p_{N-1}(x_1) & \ldots & p_{N-1}(x_N) \end{pmatrix} . \tag{2.35}$$

Equation (2.34) is linear in w_i and can be solved directly. However, if diagonalization of the Jacobi matrix provided the normalized eigenvectors, the weigths can be readily obtained using the following method.

The matrix $\mathbf{P}$ apparently consists of non-normalized column eigenvectors of the matrix $\mathbf{J}$. The eigenvectors are orthogonal and therefore $\mathbf{P}^T\mathbf{P}$ is a diagonal matrix with positive elements. Multiplying (2.34) by $\mathbf{P}^T$ and then by $(\mathbf{P}^T\mathbf{P})^{-1}$ from the left gives

$$\mathbf{w} = (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T\mathbf{e}_1 \int_a^b \omega(x)dx \; . \tag{2.36}$$

From $p_0(x) = 1$ it follows that $\mathbf{P}^T\mathbf{e}_1 = \{1, \ldots, 1\}^T$ and therefore

$$w_i = \frac{1}{(\mathbf{P}^T\mathbf{P})_{ii}} \int_a^b \omega(x)dx \; . \tag{2.37}$$

Let the matrix $\mathbf{V}$ be the set of the normalized column eigenvectors of the matrix $\mathbf{J}$. The matrix $\mathbf{V}$ is then connected with the matrix $\mathbf{P}$ through the normalization equation,

$$\mathbf{V} = \sqrt{(\mathbf{P}^T\mathbf{P})^{-1}}\mathbf{P} \; . \tag{2.38}$$

Therefore, again taking into account that $p_0(x) = 1$, equation (2.37) can be written as

$$w_i = (V_{1i})^2 \int_a^b \omega(x)dx \; \blacksquare. \tag{2.39}$$

Example: Gauss-Legendre quadrature

Gauss-Legendre quadrature deals with the weight $\omega(x) = 1$ on the interval $[-1, 1]$. The associated polynomials are Legendre polynomials $\mathcal{P}_n(x)$, hence the name. Their reccurence relation is usually given as

$$(2n-1)x\mathcal{P}_{n-1}(x) = n\mathcal{P}_n(x) + (n-1)\mathcal{P}_{n-2}(x) \, . \tag{2.40}$$

Table 2.2: An Octave function which calculates the nodes and weights of the N-point Gauss-Legendre quadrature and then integrates a given function.

```
function Q = gauss_legendre(f,a,b,N)
beta = .5./sqrt(1-(2*(1:N-1)).^(-2)); % reccurence relation
J = diag(beta,1) + diag(beta,-1);     % Jacobi matrix
[V,D] = eig(J);                       % diagonalization of J
x = diag(D); [x,i] = sort(x);         % sorted nodes
w = V(1,i).^2*2;                      % weights
Q = w*f((a+b)/2+(b-a)/2*x)*(b-a)/2;   % integral
endfunction;
```

Rescaling the polynomials (preserving $p_0(x) = 1$) as

$$\sqrt{2n+1}\mathcal{P}_n(x) = p_n(x) \tag{2.41}$$

reduces this reccurence relation to the symmetric form (2.29),

$$xp_{n-1}(x) = \frac{1}{2}\frac{1}{\sqrt{1-(2n)^{-2}}}p_n(x) + \frac{1}{2}\frac{1}{\sqrt{1-(2(n-1))^{-2}}}p_{n-2}(x) \ . \tag{2.42}$$

Correspondingly, the coefficients in the matrix $\mathbf{J}$ are

$$\alpha_n = 0 \ , \quad \left\{ \beta_n = \frac{1}{2}\frac{1}{\sqrt{1-(2n)^{-2}}} \ \middle| \ n = 1, \ldots, N-1 \right\} \ . \tag{2.43}$$

The problem of finding the nodes and the weights of the N-point Gauss-Legendre quadrature is thus reduced to the eigenvalue problem for the Jacobi matrix with coefficients (2.43).

As an illustration of this algorithm Table (2.2) shows an Octave function which calculates the nodes and the weights of the N-point Gauss-Legendre quadrature and then integrates a given function.

2.4.2 Gauss-Kronrod quadratures

Generally, the error of a numerical integration is estimated by comparing the results from two rules of different orders. However, for ordinary Gauss quadratures the nodes for two rules of different orders almost never coinside. This means that one can not reuse the points of the lower order rule when calculating the hihger order rule.

Gauss-Kronrod algorithm [11] remedies this inefficiency. The points inherited from the lower order rule are reused in the higher order rule as predefined nodes (with n weights as free parameters), and then m more optimal points are added (m abscissas and m weights as free parameters). The order of the method is $n + 2m - 1$. The lower order rule becomes *embedded*—that is, it uses a subset of the nodes—into the higher order rule. On the next iteration the procedure is repeated.

Patterson [13] has tabulated nodes and weigths for several sequences of embedded Gauss-Kronrod rules.

2.5 Adaptive quadratures

Higher order quadratures 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's phenomenon. Therefore, if the error of the quadrature is yet too large for a quadrature with sufficiently large n, 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, $2\,(h/2)^k < h^k$, if $k > 1$.

An *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 – a higher order rule,

$$Q = \sum_i w_i f(x_i), \tag{2.44}$$

where w_i are the weights of the higher order rule and Q is the higher order estimate of the integral, and a lower order rule,

$$q = \sum_i v_i f(x_i), \tag{2.45}$$

where v_i are the weights of the lower order rule and q is the the lower order estimate of the integral. The difference between the higher order rule and the lower order rule gives an estimate of the error,

$$\delta Q = |Q - q| \,. \tag{2.46}$$

The integration result is accepted, if the error δQ is smaller than tolerance,

$$\delta Q < \delta + \epsilon |Q| \,, \tag{2.47}$$

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

If the error estimate is larger than tolerance, 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 points x_i are usually chosen such that the two quadratures use the same points, and that the points can be reused in the subsequent recursive steps. 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.

As an example, Table 2.3 shows an implementation of the described algorithm using

$$x_i = \left\{\frac{1}{6}, \frac{2}{6}, \frac{4}{6}, \frac{5}{6}\right\} \text{ (easily reusable points)}, \tag{2.48}$$

$$w_i = \left\{\frac{2}{6}, \frac{1}{6}, \frac{1}{6}, \frac{2}{6}\right\} \text{ (trapezium rule)}, \tag{2.49}$$

$$v_i = \left\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right\} \text{ (rectangle rule)}. \tag{2.50}$$

During recursion the function values at the points #2 and #3 are inherited from the previous step and need not be recalculated.

The points and weights are cited for the rescaled integration interval $[0, 1]$. The transformation of the points and weights to the original interval $[a, b]$ is given as

$$\begin{aligned} x_i &\rightarrow a + (b - a)x_i \,, \\ w_i &\rightarrow (b - a)w_i \,. \end{aligned} \tag{2.51}$$

This implementation calculates directly the Riemann sums and can therefore deal with integrable singularities, although rather inefficiently.

More efficient adaptive routines keep track of the subdivisions of the interval and the local errors [9]. This allows detection of singularities and switching in their vicinity to specifically tuned quadratures. It also allows better estimates of local and global errors.

2.6 Variable transformation quadratures

The idea behind *variable transformation quadratures* is to apply the given quadrature—either with optimimized or regularly spaced nodes—not to the original integral, but to a variable transformed integral [12],

$$\int_a^b f(x)dx = \int_{t_a}^{t_b} f\big(g(t)\big)g'(t)dt \approx \sum_{i=1}^{N} w_i f\big(g(t_i)\big)g'(t_i)\,, \tag{2.52}$$

where the transformation $x = g(t)$ is chosen such that the transformed integral better suits the given quadrature. Here g' denotes the derivative and $[t_a, t_b]$ is the corresponding interval in the new variable.

For example, the Gauss-Legendre quadrature assumes the integrand can be well represented with polynomials and performs poorly on integrals with integrable singularities like

$$I = \int_0^1 \frac{1}{2\sqrt{x}} dx\,. \tag{2.53}$$

Table 2.3: Recursive adaptive integrator in C

```
#include<math.h>
#include<assert.h>
#include<stdio.h>
double adapt24(double f(double),double a, double b,
             double acc, double eps, double f2, double f3, int nrec)
{ assert(nrec<1000000);
  double f1=f(a+(b-a)/6), f4=f(a+5*(b-a)/6);
  double Q=(2*f1+f2+f3+2*f4)/6*(b-a), q=(f1+f4+f2+f3)/4*(b-a);
  double tolerance=acc+eps*fabs(Q), error=fabs(Q-q);
  if(error < tolerance) return Q;
  else {
    double Q1=adapt24(f,a,(a+b)/2,acc/sqrt(2.),eps,f1,f2,nrec+1);
    double Q2=adapt24(f,(a+b)/2,b,acc/sqrt(2.),eps,f3,f4,nrec+1);
    return Q1+Q2; }
}
double adapt(double f(double),double a,double b,
           double acc,double eps)
{ double f2=f(a+2*(b-a)/6),f3=f(a+4*(b-a)/6); int nrec=0;
  return adapt24(f,a,b,acc,eps,f2,f3,nrec);
}
int main() //uses gcc nested functions
{ int calls=0; double a=0,b=1,acc=0.001,eps=0.001;
  double f(double x){calls++; return 1/sqrt(x);}; //nested function
  double Q=adapt(f,a,b,acc,eps); printf("Q=%g calls=%d\n",Q,calls);
  return 0 ;
}
```

However, a simple varibale transformation $x = t^2$ removes the singularity,

$$I = \int_0^1 dt \,, \tag{2.54}$$

and the Gauss-Legendre quadrature for the transformed integral gives exact result. The price is that the transformed quadrature performs less effectively on smooth functions.

Some of the popular variable transformation quadratures are Clenshaw-Curtis [3], based on the transformation

$$\int_{-1}^{1} f(x)dx = \int_0^{\pi} f(\cos\theta)\sin\theta d\theta \,, \tag{2.55}$$

and "tanh-sinh" quadrature [12], based on the transformation

$$\int_{-1}^{1} f(x)dx = \int_{-\infty}^{\infty} f\left(\tanh\left(\frac{\pi}{2}\sinh(t)\right)\right)\frac{\pi}{2}\frac{\cosh(t)}{\cosh^2\left(\frac{\pi}{2}\sinh(t)\right)}dt \,. \tag{2.56}$$

Generally, the equally spaced trapezium rule is used after the transformation.

2.7 Infinite intervals

One way to calculate an integral over infinite interval is to transform it by a variable sustitution into an integral over a finite interval. The latter can then be evaluated by ordinary integration methods. Table 2.4 lists several of such transformation.

Table 2.4: Variable transformations reducing infinite interval integrals into integrals over finite intervals.

$$\int_{-\infty}^{+\infty} f(x)dx = \int_{-1}^{+1} f\left(\frac{t}{1-t^2}\right)\frac{1+t^2}{(1-t^2)^2}dt\ , \tag{2.57}$$

$$\int_{-\infty}^{+\infty} f(x)dx = \int_{0}^{1} \left(f\left(\frac{1-t}{t}\right)+f\left(-\frac{1-t}{t}\right)\right)\frac{dt}{t^2}\ , \tag{2.58}$$

$$\int_{a}^{+\infty} f(x)dx = \int_{0}^{1} f\left(a+\frac{t}{1-t}\right)\frac{1}{(1-t)^2}dt\ , \tag{2.59}$$

$$\int_{a}^{+\infty} f(x)dx = \int_{0}^{1} f\left(a+\frac{1-t}{t}\right)\frac{dt}{t^2}\ , \tag{2.60}$$

$$\int_{-\infty}^{a} f(x)dx = \int_{-1}^{0} f\left(a-\frac{t}{1+t}\right)\frac{-1}{(1+t)^2}dt\ , \tag{2.61}$$

$$\int_{-\infty}^{a} f(x)dx = \int_{0}^{1} f\left(a-\frac{1-t}{t}\right)\frac{dt}{t^2}\ . \tag{2.62}$$

Chapter 3

Monte Carlo integration

3.1 Introduction

Monte Carlo integration is a quadrature (cubature) where the nodes are chosen randomly. Typically no assumptions are made about the smoothness of the integrand, not even that it is continuous.

Monte Carlo algorithms are particularly suited for multi-dimensional integrations where one of the problems is that the integration region, Ω, often has quite complicated boundary which can not be 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 encompasses the integration volume Ω, and an auxiliary function which coincides with the integrand inside the volume Ω and is equal zero outside. Then the integral of the auxiliary function over the auxiliary volume is equal the original integral.

However, the auxiliary function is generally non-continuous at the boundary; thus ordinary quadratures which assume continuity of the integrand are bound to have difficulties here. One the contrary the Monte-Carlo quadratures will do just as good (or as bad) as with continuous integrands.

A typical implementation of a Monte Carlo algorithm integrates the given function over a rectangular volume, specified by the coordinates of its "lower-left" and "upper-right" vertices, assuming the user has provided the encompassing volume with the auxiliary function.

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

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

Yet another strategy to reduce the error is to use correlated quasi-random sequences.

The GNU Scientific Library, GSL, implements a plain Monte Carlo integration algorithm; a stratified sampling algorithm, MISER; an importance sampling algorithm, VEGAS; and a number of quasi-random generators.

3.2 Plain Monte Carlo sampling

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

$$\int_V f(\mathbf{x})dV \approx w \sum_{i=1}^{N} f(\mathbf{x}_i) \ , \tag{3.1}$$

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

$$\int_V f(\mathbf{x})dV \approx \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{x}_i) \doteq V\langle f\rangle \ . \tag{3.2}$$

Under the assumptions of the *central limit theorem* the error of the integration can be estimated as

$$\text{error} = V\frac{\sigma}{\sqrt{N}} \ , \tag{3.3}$$

where σ is the variance of the sample,

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2 \ . \tag{3.4}$$

The familiar $1/\sqrt{N}$ convergence of a random walk process is quite slow: to reduce the error by a factor 10 requires 100-fold increase in the number of sample points.

Expression (3.3) provides only a statistical estimate of the error, which is not a strict bound; random sampling may not uncover all the important features of the function, resulting in an underestimate of the error.

A simple implementation of the plain Monte Carlo algorithm is shown in Table 3.1.

3.3 Importance sampling

Suppose the points are distributed not uniformly but with some density $\rho(\mathbf{x})$. That is, the number of points Δn in the volume ΔV around point $\mathbf{x}$ is given as

$$\Delta n = \frac{N}{V}\rho(\mathbf{x})\Delta V, \tag{3.5}$$

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

Table 3.1: Plain Monte Carlo integrator

```
#include <math.h>
#include <stdlib.h>
#define RND ((double)rand()/RAND_MAX)
void randomx(int dim, double *a, double *b, double *x)
{ for(int i=0;i<dim;i++) x[i]=a[i]+RND*(b[i]-a[i]); }

void plainmc(int dim, double *a, double *b,
double f(double* x),int N, double* result, double* error)
{ double V=1; for(int i=0;i<dim;i++) V*=b[i]-a[i];
  double sum=0, sum2=0, fx, x[dim];
  for(int i=0;i<N;i++){ randomx(dim,a,b,x); fx=f(x);
                        sum+=fx; sum2+=fx*fx; }
  double avr = sum/N, var = sum2/N-avr*avr;
  *result = avr*V; *error = sqrt(var/N)*V;
}
```

The estimate of the integral is then given as

$$\int_V f(\mathbf{x})dV \approx \sum_{i=1}^{N} f(\mathbf{x}_i)\Delta V_i = \sum_{i=1}^{N} f(\mathbf{x}_i)\frac{V}{N\rho(\mathbf{x}_i)} = V\left\langle \frac{f}{\rho}\right\rangle , \tag{3.6}$$

where

$$\Delta V_i = \frac{V}{N\rho(x_i)} \tag{3.7}$$

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

The corresponding variance is now given by

$$\sigma^2 = \left\langle \left(\frac{f}{\rho}\right)^2 \right\rangle - \left\langle \frac{f}{\rho}\right\rangle^2 . \tag{3.8}$$

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 integrand. However in practice evaluations of the integrand are typically expensive. 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.

3.4 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-dimensional integrations as the number of sub-volumes grows way too fast to keep track

Table 3.2: Recursive stratified sampling algorithm

```
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;
```

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 presented in Table 3.2.

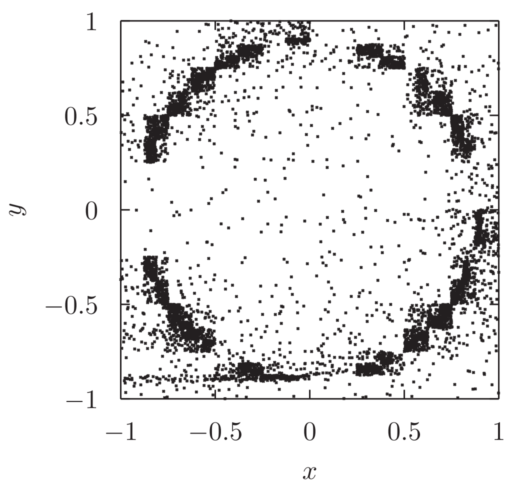

Figure 3.1: Stratified sample of a discontinuous function $f(x,y) = \begin{cases} 1 \text{ if } x^2+y^2<0.9^2 \\ 0 \text{ otherwise} \end{cases}$ built with the algorithm in Table 3.2.

In a stratified sample the points are concentrated in the regions where the variance of the function is largest, see an illustration in Figure 3.1.

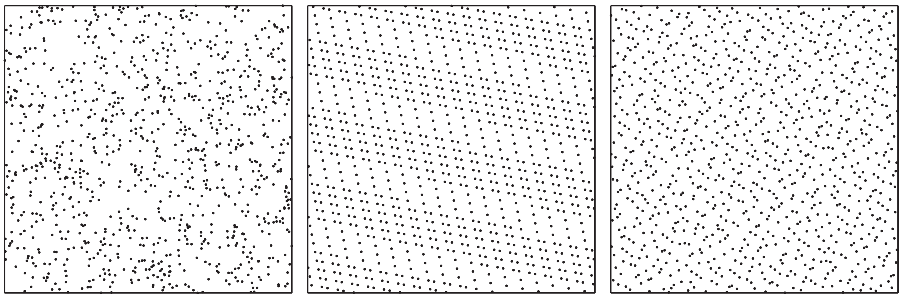

Figure 3.2: Typical distributions of pseudo-random points (left), and quasi-random low-discrepancy points: lattice (center) and base-2/3 Halton (right) sequences. The first thousand points are plotted in each case.

3.5 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, see an illustration on Figure 3.2 (left). 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 Figure 3.2 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 on the $1/\sqrt{N}$ convergence rate of pseudo-random sampling.

The central limit theorem does not apply in this case as the points are not statistically independent. Therefore the variance can not be used as an estimate of the error. The error estimation is actually not trivial. In practice one can employ two different sequences and use the difference in the resulting integrals as an error estimate.

3.5.1 Van der Corput and Halton sequences

A *van der Corput sequence* is a low-discrepancy sequence over the unit interval. It is constructed by reversing the base-b representation of the sequence of natural numbers $(1, 2, 3, \dots)$. For example, the decimal van der Corput sequence begins as

$$0.1, 0.2, 0.3, \dots, 0.8, 0.9, 0.01, 0.11, 0.21, 0.31, \dots, 0.91, 0.02, 0.12, \dots \,. \quad (3.9)$$

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

In a base-b representation a natural number n with s digits $\{d_i \mid i = 1\dots s,\ 0 \le d_i < b\}$ is given as

$$n = \sum_{k=1}^{s} d_k b^{k-1} . \tag{3.10}$$

The corresponding base-b van der Corput number $q_b(n)$ is then given as

$$q_b(n) = \sum_{k=1}^{s} d_k b^{-k} . \tag{3.11}$$

Here is a C implementation of this algorithm,

```
double corput(int n, int base){
   double q=0, bk=(double)1/base;
   while(n>0){ q += (n % base)*bk; n /= base; bk /= base; }
   return q; }
```

The van der Corput numbers of any base are uniformly distributed over the unit interval. They also form a dense set in the unit interval: there exists a subsequence of the van der Corput sequence which converges to any given real number in $[0, 1]$.

The Halton sequence is a generalization of the van der Corput sequence to d-dimensional spaces. One chooses a set of coprime bases $b_1, \dots, b_d$ and then for each dimension i generates a van der Corput sequence with its own base b_i. The n-th Halton d-dimentional point $\mathbf{x}$ in the unit volume is then given as

$$\mathbf{x}_{b_1,\dots,b_d}(n) = \{q_{b_1}(n), \dots, q_{b_d}(n)\} . \tag{3.12}$$

Here is a C implementation which calls the `corput` function listed above,

```
#include<assert.h>
void halton(int n, int d, double *x){
  int base[]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67};
  int maxd=19; assert(d <= maxd);
  for(int i=0;i<d;i++) x[i]=corput(n,base[i]); }
```

3.5.2 Lattice rules

In the simplest incarnation a lattice rule can be defined as follows. Let the *generating vector* $\mathbf{z} = \{\alpha_i \mid i = 1, \dots, d\}$—where d is the dimension of the integration space—be a set of cleverly chosen irrational numbers. Then the n-th point (in the unit volume) of the sequence is given as

$$\mathbf{x}(n) = \mathrm{frac}(n\mathbf{z}) \equiv \{\mathrm{frac}(n\alpha_1), \dots, \mathrm{frac}(n\alpha_d)\} , \tag{3.13}$$

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

An implementation of this algorithm in C is given in Table 3.3 and an illustration of such sequence is shown on Figure 3.2 (center).

Table 3.3: Lattice low-disrepancy quasi-random sequence in C.

```
#include <math.h>
#include <assert.h>
#include <gc.h> // garbage collector
#define fracl(x) ((x)-floorl(x))
#define real long double
void lattice(int d, double *x){
  static int dim=0, n=-1; static real *alpha; int i;
  if(d<0){ // d<0 is the signal to (re-)initialize the lattice
    dim=-d; n=0; alpha=(real*)GC_MALLOC(dim*sizeof(real));
    for(i=0;i<dim;i++) alpha[i]=fracl(sqrtl((real)13*(i+1))); }
  else{
    assert(d==dim && n>=0);
    n++;for(i=0;i<dim;i++)x[i]=fracl(n*alpha[i]); }
}
```

3.6 Implementations

Table 3.4: Javascript implementation of the stratified sampling algorithm.

```
Array.prototype.__iterator__=function(){
  for(var i=0;i<this.length;i++) yield i;}

function strata(f,a,b,acc,eps,N,oldstat,V)
{
var randomx=function(a,b)[a[i]+Math.random()*(b[i]-a[i])for(i in a)]
var range =function(n) {for(var i=0;i<n;i++) yield i}
var stats =function(xs){
  var n=xs.length;
  var average=xs.reduce(function(a,b)a+b,0)/n;
  var variance=xs.reduce(function(a,b)a+b*b,0)/n-average*average;
  return [average,variance,n];
   }
if(typeof(N)=="undefined")N=42;
if(typeof(oldstat)=="undefined"){// first call: setting up old stats
  var V=1; for(let k in a) V*=(b[k]-a[k]);
  var xs=[randomx(a,b) for(i in range(N))];
  var ys=[f(xs[i]) for (i in xs)];
  var oldstat=stats(ys);
  }
var xs=[randomx(a,b) for(i in range(N))] // new points
var ys=[f(xs[i]) for(i in xs)]           // new function values
var [average,variance,]=stats(ys)  // average and variance
var [oaverage,ovariance,oN]=oldstat
var integ=V*(average*N+oaverage*oN)/(N+oN)
// integral and error
var error=V*Math.sqrt( (variance*N+ovariance*oN)/(N+oN)/(N+oN) )
if(error<acc+eps*Math.abs(integ)) return [integ,error]; // done
else{ // not done: need to dispatch a recursive call
  var vmax=-1, kmax=0
  for(let k in a){ // look in all dimensions with is best to bisect
    var [al,vl,nl]=
      stats([ys[i] for(i in xs)if(xs[i][k]< (a[k]+b[k])/2)])
    var [ar,vr,nr]=
      stats([ys[i] for(i in xs)if(xs[i][k]>=(a[k]+b[k])/2)])
    var v=Math.abs(al-ar) // take the one with largest variation
    if(v>vmax){ // remember the values
      vmax=v;kmax=k;
      var oldstatl=[al,vl,nl], oldstatr=[ar,vr,nr]
      }
  } // now dispatch two recursive calls
  var a2=a.slice(); a2[kmax]=(a[kmax]+b[kmax])/2
  var b2=b.slice(); b2[kmax]=(a[kmax]+b[kmax])/2
  var [i1,e1]=strata(f,a,b2,acc/1.414,eps,N,oldstatl,V/2)
  var [i2,e2]=strata(f,a2,b,acc/1.414,eps,N,oldstatr,V/2)
  return [i1+i2,Math.sqrt(e1*e1+e2*e2)] // return results
  }
}
```

Chapter 4

Ordinary differential equations

4.1 Introduction

Ordinary differential equations (ODE) are generally defined as differential equations in one variable where the highest order derivative enters linearly. Such equations invariably arise in many different contexts throughout mathematics and science as soon as changes in the phenomena at hand are considered, usually with respect to variations of certain parameters.

Systems of ordinary differential equations can be generally reformulated as systems of first-order ordinary differential equations,

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}) \ , \tag{4.1}$$

where $\mathbf{y}' \doteq d\mathbf{y}/dx$, and the variables $\mathbf{y}$ and the *right-hand side function* $\mathbf{f}(x, \mathbf{y})$ are understood as column-vectors. For example, a second order differential equation in the form

$$u'' = g(x, u, u') \tag{4.2}$$

can be rewritten as a system of two first-order equations,

$$\begin{cases} y_1' = y_2 \\ y_2' = g(x, y_1, y_2) \end{cases} , \tag{4.3}$$

using the variable substitution $y_1 = u$, $y_2 = u'$.

In practice ODEs are usually supplemented with boundary conditions which pick out a certain class or a unique solution of the ODE. In the following we shall mostly consider *initial value problems*: ODE with the boundary condition in the form of an initial condition at a given point a,

$$\mathbf{y}(a) = \mathbf{y}_0 \ . \tag{4.4}$$

The problem then is to find the value of the solution $\mathbf{y}$ at some other point b. Finding a solution to an ODE is often referred to as *integrating* the ODE.

An ODE integration algorighm typically advances the solution from the initial point a to the final point b in a number of discrete steps

$$\{x_0 \doteq a, x_1, \ldots, x_{n-1}, x_n \doteq b\}. \tag{4.5}$$

An efficient algorithm tries to integrate an ODE using as few steps as possible under the constraint of the given accuracy goal. For this purpose the algorthm should continuously adjust the step-size during the integration, using few larger steps in the regions where the solution is smooth and perhaps many smaller steps in more treacherous regions.

Typically, an adaptive step-size ODE integrator is implemented as two routines. One of them—called *driver*—monitors the local errors and tolerances and adjusts the step-sizes. To actually perform a step the driver calls a separate routine—the *stepper*—which advances the solution by one step, using one of the many available algorithms, and estimates the local error. The GNU Scientific Library, GSL, implements about a dozen of different steppers and a tunable adaptive driver.

In the following the chapter describes several of the popular driving algorithms and stepping methods for solving initial value ODE problems.

4.2 Error estimate

In an adaptive step-size algorithm the stepping routine must provide an estimate of the integration error, upon which the driver bases its strategy to determine the optimal step-size for a user-specified accuracy goal.

A stepping method is generally characterized by its *order* : a method has order p if it can integrate exactly an ODE where the solution is a polynomial of order p. In other words, for small h the error of the order-p method is $O(h^{p+1})$.

For sufficiently small steps the error δy of an integration step for a method of a given order p can be estimated by comparing the solution $\mathbf{y}_{\text{full\_step}}$ obtained with one full-step integration with a potentially more precise solution, $\mathbf{y}_{\text{two\_half\_steps}}$, obtained with two consecutive half-step integrations,

$$\delta\mathbf{y} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^{p+1} - 1} . \tag{4.6}$$

where p is the order of the algorithm used.

Indeed, if the step-size h is small, we can assume

$$\begin{aligned} \delta\mathbf{y}_{\text{full\_step}} &= Ch^{p+1} , & (4.7)\\ \delta\mathbf{y}_{\text{two\_half\_steps}} &= C\left(\frac{h}{2}\right)^{p+1} , & (4.8) \end{aligned}$$

where $\delta\mathbf{y}_{\text{full\_step}}$ and $\delta\mathbf{y}_{\text{two\_half\_steps}}$ are the errors of the full-step and two half-steps integrations, and C is an unknown constant. The two can be combined as

$$\begin{aligned}\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}} &= \delta\mathbf{y}_{\text{full\_step}} - \delta\mathbf{y}_{\text{two\_half\_steps}} \\ &= C\left(\frac{h}{2}\right)^{p+1}(2^{p+1}-1)\,, \end{aligned} \tag{4.9}$$

from which it immedliately follows that

$$C\left(\frac{h}{2}\right)^{p+1} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^{p+1}-1}\,. \tag{4.10}$$

One has, of course, to take the potentially more precise $\mathbf{y}_{\text{two\_half\_steps}}$ as the solution $\mathbf{y}$. Its error is then given as

$$\delta\mathbf{y} \doteq \delta\mathbf{y}_{\text{two\_half\_steps}} = C\left(\frac{h}{2}\right)^{p+1} = \frac{\mathbf{y}_{\text{full\_step}} - \mathbf{y}_{\text{two\_half\_steps}}}{2^{p+1}-1}\,, \tag{4.11}$$

which had to be demonstrated. This prescription is often referred to as the *Runge's principle*.

One drawback of the Runge's principle is that the full-step and the two half-step calculations generally do not share evaluations of the right-hand side function $\mathbf{f}(x, \mathbf{y})$, and therefore many extra evaluations are needed to estimate the error.

An alternative prescription for error estimation is to make the same step-size integration using two methods of *different orders*, with the difference between the two solutions providing the estimate of the error. If the lower order method mostly uses the same evaluations of the right-hand side function—in which case it is called *embedded* in the higher order method—the error estimate does not need additional evaluations.

Predictor-corrector methods are naturally of embedded type: the correction — which generally increases the order of the method — itself can serve as the estimate of the error.

4.3 Runge-Kutta methods

Runge-Kutta methods are one-step methods which advance the solution over the current step using only the information gathered from withing the step itself. The solution $\mathbf{y}$ is advanced from the point x_i to $x_{i+1} = x_i + h$, where h is the step-size, using a one-step formula,

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{k}, \tag{4.12}$$

where $\mathbf{y}_{i+1}$ is the approximation to $\mathbf{y}(x_{i+1})$, and the value $\mathbf{k}$ is chosen such that the method integrates exactly an ODE whose solution is a polynomial of the highest possible order.

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 polynomial of order p. In other words, for small h the error of the order-p method is $O(h^{p+1})$.

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

$$\mathbf{k} = \mathbf{f}(x_0, \mathbf{y}_0) \, . \tag{4.13}$$

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

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \, , \\ \mathbf{k}_{1/2} &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \, , \\ \mathbf{k} &= \mathbf{k}_{1/2} \, , \end{aligned} \tag{4.14}$$

or the *two-point method*, also called the *Heun's method*

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0), \\ \mathbf{k}_1 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_0), \\ \mathbf{k} &= \frac{1}{2}(\mathbf{k}_0 + \mathbf{k}_1) \, . \end{aligned} \tag{4.15}$$

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

$$\mathbf{k} = \frac{1}{6}\mathbf{k}_0 + \frac{4}{6}\mathbf{k}_{1/2} + \frac{1}{6}\mathbf{k}_1 \, . \tag{4.16}$$

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

$$\begin{aligned} \mathbf{k}_0 &= \mathbf{f}(x_0, \mathbf{y}_0) \, , \\ \mathbf{k}_1 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_0) \, , \\ \mathbf{k}_2 &= \mathbf{f}(x_0 + \tfrac{1}{2}h, \mathbf{y}_0 + \tfrac{1}{2}h\mathbf{k}_1) \, , \\ \mathbf{k}_3 &= \mathbf{f}(x_0 + h, \mathbf{y}_0 + h\mathbf{k}_2) \, , \\ \mathbf{k} &= \tfrac{1}{6}\mathbf{k}_0 + \tfrac{1}{3}\mathbf{k}_1 + \tfrac{1}{3}\mathbf{k}_2 + \tfrac{1}{6}\mathbf{k}_3 \, . \end{aligned} \tag{4.17}$$

A general Runge-Kutta method can be written as

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \sum_{i=1}^{s} b_i \mathbf{K}_i \, , \tag{4.18}$$

where

$$\begin{aligned} \mathbf{K}_1 &= h\mathbf{f}(x_n, \mathbf{y}_n) \, , \\ \mathbf{K}_2 &= h\mathbf{f}(x_n + c_2 h, \mathbf{y}_n + a_{21}\mathbf{K}_1) \, , \\ \mathbf{K}_3 &= h\mathbf{f}(x_n + c_3 h, \mathbf{y}_n + a_{31}\mathbf{K}_1 + a_{32}\mathbf{K}_2) \, , \\ &\vdots \\ \mathbf{K}_s &= h\mathbf{f}(x_n + c_s h, \mathbf{y}_n + a_{s1}\mathbf{K}_1 + a_{s2}\mathbf{K}_2 + \cdots + a_{s,s-1}\mathbf{K}_{s-1}) \, . \end{aligned} \tag{4.19}$$

The upper case $\mathbf{K}_i$ are simply the lower case $\mathbf{k}_i$ multiplied by the step-size h.

To specify a particular Runge-Kutta method one needs to provide the coefficients $\{a_{ij}|1 \le j < i \le s\}$, $\{b_i|i = 1..s\}$ and $\{c_i|i = 1..s\}$. The matrix $[a_{ij}]$ is called the Runge-Kutta matrix, while the coefficients b_i and c_i are known as the weights and the nodes. These data are usually arranged in the so called *Butcher's tableau*,

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} . \tag{4.20}$$

For example, the Butcher's tableau for the RK4 method is

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} . \tag{4.21}$$

4.3.1 Embeded methods with error estimates

The embedded Runge-Kutta methods—in addition to advancing the solution by one step—produce an estimate of the local error of the step. This is done by having two methods in the tableau, one with a certain order p and another one with order $p-1$. The difference bitween the two method gives the the estimate of the local error. The lower order method is *embedded* in the higher order method, that is, it uses the same $\mathbf{K}$-values. This allows a very effective estimate of the error.

The embedded lower order method is written as

$$\mathbf{y}^*_{n+1} = \mathbf{y}_n + \sum_{i=1}^{s} b^*_i \mathbf{K}_i \,, \tag{4.22}$$

where $\mathbf{K}_i$ are the same as for the higher order method. The error estimate is then given as

$$\mathbf{e}_n = \mathbf{y}_{n+1} - \mathbf{y}^*_{n+1} = \sum_{i=1}^{s} (b_i - b^*_i)\mathbf{K}_i \,. \tag{4.23}$$

The Butcher's tableau for this kind of method is extended by one row to give the values of b^*_i.

The simplest embedded methods are Heun-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \\ & 1 & 0 \end{array}\,, \tag{4.24}$$

and midpoint-Euler method,

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \\ & 1 & 0 \end{array}\,, \tag{4.25}$$

which both combine methods of orders 2 and 1. An implementation of the midpoint-Euler method in C is shown in Table 4.1.

Table 4.1: Embedded Runge-Kutta midpoint-Euler stepper with error estimate.

```
void rkstep12(void f(int n,double x,double* y,double* dydx),
int n, double x, double* y, double h, double* yout, double* err)
{ double k1[n],k2[n],ytmp[n]; //VLA: --std=c99
  f(n,x,y,k1);
  for(int i = 0;i<n;i++)  ytmp[i]=y[i]+1./2*h*k1[i];
  f(n,x+1./2*h,ytmp,k2);
  for(int i = 0;i<n;i++){ yout[i]=y[i]+h*k2[i];
                          err[i]=h*(k2[i]-k1[i])/2;}// optimistic
}
```

The *Bogacki-Shampine method* [2] combines methods of orders 3 and 2,

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 3/4 & 0 & 3/4 & & \\ 1 & 2/9 & 1/3 & 4/9 & \\ \hline & 2/9 & 1/3 & 4/9 & 0 \\ & 7/24 & 1/4 & 1/3 & 1/8 \end{array}\,. \tag{4.26}$$

Bogacki and Shampine argue that their method has better stability properties and actually outperforms higher order methods at lower accuracy goal calculations. This method has the FSAL—first same as last—property: the value $\mathbf{k}_4$ at one step equals $\mathbf{k}_1$ at the next step; thus only three function evaluations are needed per step. Table 4.2 shows a simple implementation which does not utilise this property for the sake of presentational clarity.

Table 4.2: Embedded Runge-Kutta Bogacki-Shampine stepper

```
void rkstep23( void f(int n,double x,double* y,double* dydx),
int n, double x, double* y, double h, double* yout, double* err)
{ double k1[n],k2[n],k3[n],k4[n],ytmp[n]; //VLA: -std=c99
  f(n,x,y,k1);
  for(int i=0;i<n;i++) ytmp[i]=y[i]+1./2*k1[i]*h;
  f(n,x+1./2*h,ytmp,k2);
  for(int i=0;i<n;i++) ytmp[i]=y[i]+3./4*k2[i]*h;
  f(n,x+3./4*h,ytmp,k3);
  for(int i=0;i<n;i++)
    ytmp[i]=y[i]+(2./9*k1[i]+1./3*k2[i]+4./9*k3[i])*h;
  f(n,x+h,ytmp,k4);
  for(int i = 0;i<n;i++)
  { yout[i]=y[i]+(2./9*k1[i]+1./3*k2[i]+4./9*k3[i])*h;
    ytmp[i]=y[i]+(7./24*k1[i]+1./4*k2[i]+1./3*k3[i]+1./8*k4[i])*h;
      err[i]=yout[i]-ytmp[i];
  }
}
```

The Runge-Kutta-Fehlberg method [5] — called *RKF45* — implemented in the renowned `rkf45` Fortran routine, has two methods of orders 5 and 4,

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1/4 & 1/4 & & & & & \\
3/8 & 3/32 & 9/32 & & & & \\
12/13 & 1932/2197 & -7200/2197 & 7296/2197 & & & \\
1 & 439/216 & -8 & 3680/513 & -845/4104 & & \\
1/2 & -8/27 & 2 & -3544/2565 & 1859/4104 & -11/40 & \\
\hline
 & 16/135 & 0 & 6656/12825 & 28561/56430 & -9/50 & 2/55 \\
 & 25/216 & 0 & 1408/2565 & 2197/4104 & -1/5 & 0
\end{array}
$$

4.4 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. The first step must be done with a one-step method like Runge-Kutta.

A number of multistep have been devised (and named after different mathematicians); we shall only consider a few simple ones here to get the idea of how it works.

4.4.1 Two-step method

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

$$\bar{\mathbf{y}}(x) = \mathbf{y}_i + \mathbf{y}'_i \cdot (x - x_i) + \bar{\mathbf{c}} \cdot (x - x_i)^2, \tag{4.27}$$

where $\mathbf{y}'_i = \mathbf{f}(x_i, \mathbf{y}_i)$ and the coefficient $\bar{\mathbf{c}}$ can be found from the condition $\bar{\mathbf{y}}(x_{i-1}) = \mathbf{y}_{i-1}$, which gives

$$\bar{\mathbf{c}} = \frac{\mathbf{y}_{i-1} - \mathbf{y}_i + \mathbf{y}'_i \cdot (x_i - x_{i-1})}{(x_i - x_{i-1})^2} \,. \tag{4.28}$$

The value of the function at the next point, $x_{i+1} \doteq x_i + h$, can now be estimated as $\bar{\mathbf{y}}(x_{i+1})$ from (4.27).

The error of this second-order two-step stepper can be estimated by a comparison with the first-order Euler's step, which is given by the linear part of (4.27). The correction term $\bar{\mathbf{c}}h^2$ can serve as the error estimate,

$$\delta\mathbf{y} = \bar{\mathbf{c}}h^2 \,. \tag{4.29}$$

4.4.2 Two-step method with extra evaluation

One can further increase the order of the approximation (4.27) by adding a third order term,

$$\bar{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \bar{\bar{\mathbf{d}}} \cdot (x - x_i)^2 (x - x_{i-1}) \,. \tag{4.30}$$

The coefficient $\bar{\bar{\mathbf{d}}}$ can be found from the matching condition at a certain point inside the invervarl, for example at half-step,

$$\bar{\bar{\mathbf{y}}}'(x_{1/2}) = \mathbf{f}(x_{1/2}, \bar{\mathbf{y}}(x_{1/2})) \doteq \bar{\mathbf{f}}_{1/2} \,, \tag{4.31}$$

where $x_{1/2} \doteq x_i + h/2$. This gives

$$\bar{\bar{\mathbf{d}}} = \frac{\bar{\mathbf{f}}_{1/2} - \mathbf{y}'_i - 2\bar{\mathbf{c}} \cdot (x_{1/2} - x_i)}{2(x_{1/2} - x_i)(x_{1/2} - x_{i-1}) + (x_{1/2} - x_i)^2} . \tag{4.32}$$

The error estimate at the point $x_{i+1} \doteq x_0 + h$ is again given as the difference between the higher and the lower order methods,

$$\delta\mathbf{y} = \bar{\bar{\mathbf{y}}}(x_{i+1}) - \bar{\mathbf{y}}(x_{i+1}) \,. \tag{4.33}$$

4.5 Predictor-corrector methods

A predictor-corrector method uses extra iterations to improve the solution. It is an algorithm that proceeds in two steps. First, the predictor step calculates a rough approximation of $\mathbf{y}(x + h)$. Second, the corrector step refines the initial approximation. Aditionally the corrector step can be repeated in the hope that this achieves an even better approximation to the true solution.

For example, the two-point Runge-Kutta method (4.15) is as actually a predictor-corrector method, as it first calculates the *prediction* $\tilde{\mathbf{y}}_{i+1}$ for $\mathbf{y}(x_{i+1})$,

$$\tilde{\mathbf{y}}_{i+1} = \mathbf{y}_i + h\mathbf{f}(x_i, \mathbf{y}_i) \,, \tag{4.34}$$

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

$$\check{\tilde{\mathbf{y}}}_{i+1} = \mathbf{y}_i + h\frac{1}{2}\left(\mathbf{f}(x_i, \mathbf{y}_i) + \mathbf{f}(x_{i+1}, \tilde{\mathbf{y}}_{i+1})\right) \,. \tag{4.35}$$

4.5.1 Two-step method with correction

Similarly, one can use the two-step approximation (4.27) as a predictor, and then improve it by one order with a correction step, namely

$$\check{\bar{\mathbf{y}}}(x) = \bar{\mathbf{y}}(x) + \check{\mathbf{d}} \cdot (x - x_i)^2 (x - x_{i-1}). \tag{4.36}$$

The coefficient $\check{\mathbf{d}}$ can be found from the condition $\check{\bar{\mathbf{y}}}'(x_{i+1}) = \bar{\mathbf{f}}_{i+1}$, where $\bar{\mathbf{f}}_{i+1} \doteq \mathbf{f}(x_{i+1}, \bar{\mathbf{y}}(x_{i+1}))$,

$$\check{\mathbf{d}} = \frac{\bar{\mathbf{f}}_{i+1} - \mathbf{y}'_i - 2\bar{\mathbf{c}} \cdot (x_{i+1} - x_i)}{2(x_{i+1} - x_i)(x_{i+1} - x_{i-1}) + (x_{i+1} - x_i)^2}. \tag{4.37}$$

Equation (4.36) gives a better estimate, $\mathbf{y}_{i+1} = \check{\bar{\mathbf{y}}}(x_{i+1})$, of the sought function at the point x_{i+1}. In this context the formula (4.27) serves as *predictor*, and (4.36) as *corrector*. The difference between the two gives an estimate of the error.

This method is equivalent to the two-step method with an extra evaluation where the extra evaluation is done at the full step.

4.6 Adaptive step-size control

Let *tolerance* τ be the maximal accepted error consistent with the required accuracy to be achieved in the integration of an ODE. 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 statistically uncorrelated, the local tolerance τ_i for the step i has to scale as the square root of the step-size,

$$\tau_i = \tau\sqrt{\frac{h_i}{b-a}}. \tag{4.38}$$

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

$$E \approx \sqrt{\sum_{i=1}^{n} e_i^2} \le \sqrt{\sum_{i=1}^{n} \tau_i^2} = \tau\sqrt{\sum_{i=1}^{n} \frac{h_i}{b-a}} = \tau \,. \tag{4.39}$$

The current step h_i is accepted if the local error e_i is smaller than the local tolerance τ_i, after which the next step is attempted with the step-size adjusted according to the following empirical prescription [4],

$$h_{i+1} = h_i \times \left(\frac{\tau_i}{e_i}\right)^{\text{Power}} \times \text{Safety}, \tag{4.40}$$

where Power ≈ 0.25 and Safety ≈ 0.95.

If the local error is larger than the local tolerance the step is rejected and a new step is attempted with the step-size adjusted according to the same prescription (4.40).

One simple prescription for the local tolerance τ_i and the local error e_i to be used in (4.40) is

$$\tau_i = (\epsilon\|\mathbf{y}_i\| + \delta)\sqrt{\frac{h_i}{b-a}}\,,\; e_i = \|\delta\mathbf{y}_i\|\,, \tag{4.41}$$

where δ and ϵ are the required absolute and relative precision and $\delta\mathbf{y}_i$ is the estimate of the integration error at the step i.

A more elaborate prescription considers components of the solution separately,

$$(\tau_i)_k = \big(\epsilon|(\mathbf{y}_i)_k| + \delta\big)\sqrt{\frac{h_i}{b-a}}\,,\; (\mathbf{e}_i)_k = |(\delta\mathbf{y}_i)_k|\,, \tag{4.42}$$

where the index k runs over the components of the solution. In this case the step acceptence criterion also becomes component-wise: the step is accepted, if

$$\forall k:\; (\mathbf{e}_i)_k < (\tau_i)_k\,. \tag{4.43}$$

The factor τ_i/e_i in the step adjustment formula (4.40) is then replaced by

$$\frac{\tau_i}{e_i} \to \min_k \frac{(\tau_i)_k}{(\mathbf{e}_i)_k}\,. \tag{4.44}$$

Yet another refinement is to include the derivatives $\mathbf{y}'$ of the solution into the local tolerance estimate, either overally,

$$\tau_i = \Big(\epsilon\alpha\|\mathbf{y}_i\| + \epsilon\beta\|\mathbf{y}'_i\| + \delta\Big)\sqrt{\frac{h_i}{b-a}}\,, \tag{4.45}$$

or commponent-wise,

$$(\tau_i)_k = \Big(\epsilon\alpha|(\mathbf{y}_i)_k| + \epsilon\beta|(\mathbf{y}'_i)_k| + \delta\Big)\sqrt{\frac{h_i}{b-a}}\,. \tag{4.46}$$

The weights α and β are chosen by the user.

Table (4.3) shows a simple implementation of the described algorithm.

Table 4.3: An ODE driver with adaptive step-size control in C.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
int drive(
  void f(int n,double x,double* y,double* dydx),int n,double* xlist ,
  double** ylist ,double b,double h,double acc ,double eps ,int max,
  void step(void f(int n,double x,double* y,double* dydx),
  int n,double x,double* y,double h,double* yout ,double* dy)
  )
{
  int i=0; double a=xlist[0]; // starting point
  double dy[n],y1[n]; // will store delta y and y_{i+1}
  while(xlist[i]<b) // do until b is reached
  {
    double x=xlist[i], *y=ylist[i]; // current x and y
    if(x+h>b) h=b-x; // make sure we land on b
    step(f,n,x,y,h,y1,dy); // the step
    double sum=0; for(int k=0;k<n;k++) sum+=dy[k]*dy[k];
    double err=sqrt(sum);// local error
    sum=0; for(int k=0;k<n;k++) sum+=y1[k]*y1[k];
    double normy=sqrt(sum);
    double tol=(normy*eps+acc)*sqrt(h/(b-a)); // local tolerance:
    if(tol>err)//then accept step and store x and y
    {
      i++; if(i==max) return -i; // storage filled, abort :(
      xlist[i]=x+h; for(int k=0;k<n;k++)ylist[i][k]=y1[k];
    } // adjust the step:
    if(err>0) h=fmin(2*h,h*pow(tol/err,0.25)*0.95); else h = 2*h;
  }
  return i+1;
}
```

Chapter 5

Systems of linear equations

5.1 Introduction

A *system of linear equations* (or *linear system*) is a collection of linear equations involving the same set of unknown variables. A general system of n linear equations with m unknowns can be written as

$$\left\{\begin{array}{ccccccccc} A_{11}x_1 & + & A_{12}x_2 & + & \cdots & + & A_{1m}x_m & = & b_1 \\ A_{21}x_1 & + & A_{22}x_2 & + & \cdots & + & A_{2m}x_m & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ A_{n1}x_1 & + & A_{n2}x_2 & + & \cdots & + & A_{nm}x_m & = & b_n \end{array}\right. , \tag{5.1}$$

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

The system can be equivalently written in the matrix form,

$$\mathbf{A}\mathbf{x} = \mathbf{b} , \tag{5.2}$$

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

A solution to a linear system is a set of values for the variables $\mathbf{x}$ which satisfies all equations.

Systems of linear equations occur quite regularly in applied mathematics. Therefore 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.

5.2 Triangular systems

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

$$\mathbf{T}\mathbf{y} = \mathbf{c} \;, \tag{5.3}$$

where $\mathbf{T}$ is a *triangular matrix* – a special kind of square matrix where the matrix elements either below (lower triangular) or above (upper triangular) the main diagonal are zero.

Indeed, an upper triangular system $\mathbf{U}\mathbf{y} = \mathbf{c}$ can be easily solved by *back-substitution*,

$$y_i = \frac{1}{U_{ii}}\left(c_i - \sum_{k=i+1}^{n} U_{ik}y_k\right) \;,\; i = n, n-1, \ldots, 1 \;, \tag{5.4}$$

where one first computes $y_n = b_n/U_{nn}$, then substitutes *back* into the previous equation to solve for y_{n-1}, and repeats through y_1.

Here is a C-function implementing in-place[1] back-substitution[2]:

```
#include<matrix.h>
#include<assert.h>
void backsub(matrix *U, vector *b){
  assert(U->size1==U->size2 && U->size2==b->size);
  for(int i=b->size-1;i>=0;i--){
    double s=vector_get(b,i);
    for(int k=i+1;k<b->size;k++) s-=matrix_get(U,i,k)*vector_get(b,k);
    vector_set(b,i,s/matrix_get(U,i,i));
    }
}
```

For a lower triangular system $\mathbf{L}\mathbf{y} = \mathbf{c}$ the equivalent procedure is called *forward-substitution*,

$$y_i = \frac{1}{L_{ii}}\left(c_i - \sum_{k=1}^{i-1} U_{ik}y_k\right) \;,\; i = 1, 2, \ldots, n \;. \tag{5.5}$$

5.3 Reduction to triangular form

Popular algorithms for reducing a square system of linear equations to a triangular form are *LU-decomposition* and *QR-decomposition*.

[1]here *in-place* means the right-hand side $\mathbf{c}$ is replaced by the solution $\mathbf{y}$.

[2]the functions `vector_get`, `vector_set`, and `matrix_get` are assumed to implement fetching and setting the vector and matrix elements.

5.3.1 QR-decomposition

QR-decomposition is a factorization of a matrix into a product of an orthogonal matrix $\mathbf{Q}$, such that $\mathbf{Q}^\intercal\mathbf{Q} = 1$, where $^\intercal$ denotes transposition, and a right triangular matrix $\mathbf{R}$,

$$\mathbf{A} = \mathbf{QR} \, . \tag{5.6}$$

QR-decomposition can be used to convert a linear system $\mathbf{Ax} = \mathbf{b}$ into the triangular form (by multiplying with $\mathbf{Q}^\intercal$ from the left),

$$\mathbf{Rx} = \mathbf{Q}^\intercal\mathbf{b} \, , \tag{5.7}$$

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 $\mathbf{A}$ can be represented as a product, $\mathbf{A} = \mathbf{QR}$, of an orthogonal $n \times m$ matrix $\mathbf{Q}$, $\mathbf{Q}^\intercal\mathbf{Q} = 1$, and a right-triangular $m \times m$ matrix $\mathbf{R}$.

QR-decomposition of a matrix can be computed using several methods, such as Gram-Schmidt orthogonalization, Householder transformation [10], or Givens rotation [6].

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 $\mathbf{A} = \{\mathbf{a}_1, \dots, \mathbf{a}_m\}$ and generates an orthogonal set $\mathbf{Q} = \{\mathbf{q}_1, \dots, \mathbf{q}_m\}$ which spans the same subspace as $\mathbf{A}$. The algorithm is given as

```
for i = 1 to m :
  q_i ← a_i/||a_i||
  for j = i+1 to m :   a_j ← a_j − ⟨q_i|a_j⟩ q_i
```

where $\langle\mathbf{a}|\mathbf{b}\rangle$ is the inner product of two vectors, and $\|\mathbf{a}\| \doteq \sqrt{\langle\mathbf{a}|\mathbf{a}\rangle}$ 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 $\mathbf{A}$ by orthogonalization of its column-vectors $\mathbf{a}_i$ with the inner product

$$\langle\mathbf{a}|\mathbf{b}\rangle = \mathbf{a}^\intercal\mathbf{b} \equiv \sum_{k=1}^{n}(\mathbf{a})_k(\mathbf{b})_k \, , \tag{5.8}$$

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,

Table 5.1: QR-decomposition in C using stabilized Gram-Schmidt orthogonalization.

```
#include <matrix.h>
#include <math.h>
#include <stdio.h>
void qrdec(matrix* A, matrix* R){// A is replaced with Q, R is filled
  for(int i=0;i<A->size2;i++){
    vector* ai = matrix_get_column(A,i);
    double r = sqrt(vector_dot(ai,ai));
    matrix_set(R,i,i,r);
    vector_scale(ai,1/r);
    for(int j=i+1;j<A->size2;j++){
      vector* aj=matrix_get_column(A,j);
      double s=vector_dot(ai,aj);
      vector_add(aj,-s,ai);
      matrix_set(R,i,j,s);}}}
```

for $i = 1$ to m :
  $R_{ii} = \sqrt{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}$; $\mathbf{q}_i = \mathbf{a}_i/R_{ii}$
  for $j = i+1$ to m :
    $R_{ij} = \mathbf{q}_i^\mathsf{T}\mathbf{a}_j$; $\mathbf{a}_j = \mathbf{a}_j - \mathbf{q}_i R_{ij}$.

After orthogonalization the matrices $\mathbf{Q} = \{\mathbf{q}_1 \dots \mathbf{q}_m\}$ and $\mathbf{R}$ are the sought orthogonal and right-triangular factors of matrix $\mathbf{A}$.

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 $O(m^2 n)$.

Table (5.1) shows a C implementation of the stabilized Gram-Schmidt orthogonalization. The corresponding function to solve the equation

$$\mathbf{QRx} = \mathbf{b} \tag{5.9}$$

are shown in Table (5.2).

Householder transformation

A square matrix $\mathbf{H}$ of the form

$$\mathbf{H} = 1 - \frac{2}{\mathbf{u}^\mathsf{T}\mathbf{u}}\mathbf{u}\mathbf{u}^\mathsf{T} \tag{5.10}$$

is called *Householder matrix*, where the vector $\mathbf{u}$ is called a *Householder vector*. Householder matrices are symmetric and orthogonal,

$$\mathbf{H}^\mathsf{T} = \mathbf{H}\,,\ \mathbf{H}^\mathsf{T}\mathbf{H} = 1\,. \tag{5.11}$$

The transformation induced by the Householder matrix on a given vector a,

$$\mathbf{a} \to \mathbf{Ha}\,, \tag{5.12}$$

Table 5.2: C-functions to perform QR-backsubstitution of equation (5.9)

```
#include<matrix.h> // garbage collected matrixlibrary
#include<assert.h>
void backsub(matrix*,vector*);
vector* qrback(matrix*Q, matrix*R, vector*b){
  vector* x = matrixT_times_vector(Q,b);
  backsub(R,x);
  return x;
}
void qrback_inplace(matrix*Q, matrix*R, vector*b){
  assert(Q->size1==Q->size2);
  vector* x = qrback(Q,R,b);
  for(int i=0;i<b->size;i++)vector_set(b,i,vector_get(x,i));
}
```

is called a *Householder transformation* or *Householder reflection*. The transformation changes the sign of the affected vector's component in the $\mathbf{u}$ direction, or, in other words, makes a reflection of the vector about the hyperplane perpendicular to $\mathbf{u}$, hence the name.

Householder transformation can be used to zero selected components of a given vector $\mathbf{a}$. For example, one can zero all components but the first one, such that

$$\mathbf{Ha} = \gamma \mathbf{e}_1 \,, \tag{5.13}$$

where γ is a number and $\mathbf{e}_1$ is the unit vector in the first direction. The factor γ can be easily calculated,

$$\|\mathbf{a}\|^2 \doteq \mathbf{a}^\mathsf{T}\mathbf{a} = \mathbf{a}^\mathsf{T}\mathbf{H}^\mathsf{T}\mathbf{Ha} = (\gamma\mathbf{e}_1)^\mathsf{T}(\gamma\mathbf{e}_1) = \gamma^2 \,, \tag{5.14}$$

$$\Rightarrow \; \gamma = \pm\|a\| \,. \tag{5.15}$$

To find the Householder vector, we notice that

$$\mathbf{a} = \mathbf{H}^\mathsf{T}\mathbf{Ha} = \mathbf{H}^\mathsf{T}\gamma\mathbf{e}_1 = \gamma\mathbf{e}_1 - \frac{2(\mathbf{u})_1}{\mathbf{u}^\mathsf{T}\mathbf{u}}\mathbf{u} \,, \tag{5.16}$$

$$\Rightarrow \; \frac{2(\mathbf{u})_1}{\mathbf{u}^\mathsf{T}\mathbf{u}}\mathbf{u} = \gamma\mathbf{e}^1 - \mathbf{a} \,, \tag{5.17}$$

where $(\mathbf{u})_1$ is the first component of the vector $\mathbf{u}$. One usually chooses $(\mathbf{u})_1 = 1$ (for the sake of the possibility to store the other components of the Householder vector in the zeroed elements of the vector $\mathbf{a}$) and stores the factor

$$\frac{2}{\mathbf{u}^\mathsf{T}\mathbf{u}} \equiv \tau \tag{5.18}$$

separately. With this convention one readily finds τ from the first component of equation (5.17),

$$\tau = \gamma - (\mathbf{a})_1 \,. \tag{5.19}$$

where $(\mathbf{a})_1$ is the first element of the vector $\mathbf{a}$. For the sake of numerical stability the sign of γ has to be chosen opposite to the sign of $(\mathbf{a})_1$,

$$\gamma = -\text{sign}\left((\mathbf{a})_1\right) \|\mathbf{a}\| \,. \tag{5.20}$$

Finally, the Householder reflection, which zeroes all component of a vector $\mathbf{a}$ but the first, is given as

$$\mathbf{H} = 1 - \tau \mathbf{u}\mathbf{u}^\mathsf{T} \,,\ \tau = -\text{sign}((\mathbf{a})_1)\|\mathbf{a}\| - (\mathbf{a})_1 \,,\ (\mathbf{u})_1 = 1 \,,\ (\mathbf{u})_{i>1} = -\frac{1}{\tau}(\mathbf{a})_i \,. \tag{5.21}$$

Now, a QR-decomposition of an $n \times n$ matrix $\mathbf{A}$ by Householder transformations can be performed in the following way:

1. Build the size-n Householder vector $\mathbf{u}_1$ which zeroes the sub-diagonal elements of the first column of matrix $\mathbf{A}$, such that

$$\mathbf{H}_1\mathbf{A} = \left[\begin{array}{c|ccc} \star & \star & \dots & \star \\ \hline 0 & & & \\ \vdots & & \mathbf{A}_1 & \\ 0 & & & \end{array}\right] , \tag{5.22}$$

   where $\mathbf{H}_1 = 1 - \tau_1\mathbf{u}_1\mathbf{u}_1^\mathsf{T}$ and where $\star$ denotes (generally) non-zero matrix elements. In practice one does not build the matrix $\mathbf{H}_1$ explicitly, but rather calculates the matrix $\mathbf{H}_1\mathbf{A}$ in-place, consecutively applying the Householder reflection to columns the matrix $\mathbf{A}$, thus avoiding computationally expensive matrix-matrix operations. The zeroed sub-diagonal elements of the first column of the matrix $\mathbf{A}$ can be used to store the elements of the Householder vector $\mathbf{u}_1$ while the factor τ_1 has to be stored separately in a special array. This is the storage scheme used by LAPACK and GSL.

2. Similarly, build the size-$(n-1)$ Householder vector $\mathbf{u}_2$ which zeroes the sub-diagonal elements of the first column of matrix $\mathbf{A}_1$ from eq. (5.22). With the transformation matrix $\mathbf{H}_2$ defined as

$$\mathbf{H}_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & 1 - \tau_2\mathbf{u}_2\mathbf{u}_2^\mathsf{T} & \\ 0 & & & \end{array}\right] . \tag{5.23}$$

   the two transformations together zero the sub-diagonal elements of the two first columns of matrix $\mathbf{A}$,

$$\mathbf{H}_2\mathbf{H}_1\mathbf{A} = \left[\begin{array}{cc|ccc} \star & \star & \star & \cdots & \star \\ 0 & \star & \star & \cdots & \star \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & \mathbf{A}_3 & \\ 0 & 0 & & & \end{array}\right] , \tag{5.24}$$

3. Repeating the process zero the sub-diagonal elements of the remaining columns. For column k the corresponding Householder matrix is

$$\mathbf{H}_k = \left[\begin{array}{c|c} \mathbf{I}_{k-1} & 0 \\ \hline 0 & 1 - \tau_k \mathbf{u}_k \mathbf{u}_k^\intercal \end{array}\right] , \tag{5.25}$$

where $\mathbf{I}_{k-1}$ is a unity matrix of size $k-1$, $\mathbf{u}_k$ is the size-(n-k+1) Householder vector that zeroes the sub-diagonal elements of matrix $\mathbf{A}_{k-1}$ from the previous step. The corresponding transformation step is

$$\mathbf{H}_k \dots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} = \left[\begin{array}{c|c} \mathbf{R}_k & \star \\ \hline 0 & \mathbf{A}_k \end{array}\right] , \tag{5.26}$$

where $\mathbf{R}_k$ is a size-k right-triangular matrix.

After $n-1$ steps the matrix $\mathbf{A}$ will be transformed into a right triangular matrix,

$$\mathbf{H}_{n-1} \cdots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} = \mathbf{R} . \tag{5.27}$$

4. Finally, introducing an orthogonal matrix $\mathbf{Q} = \mathbf{H}_1^\intercal \mathbf{H}_2^\intercal \dots \mathbf{H}_{n-1}^\intercal$ and multiplying eq. (5.27) by $\mathbf{Q}$ from the left, we get the sought QR-decomposition,

$$\mathbf{A} = \mathbf{Q}\mathbf{R} . \tag{5.28}$$

In practice one does not explicitly builds the $\mathbf{Q}$ matrix but rather applies the successive Householder reflections stored during the decomposition.

Givens rotations

A Givens rotation is a transformation in the form

$$\mathbf{A} \to \mathbf{G}(p, q, \theta)\mathbf{A} , \tag{5.29}$$

where $\mathbf{A}$ is the transformed object—matrix of vector—and $\mathbf{G}(p, q, \theta)$ is the Givens rotation matrix (also known as Jacobi rotation matrix): an orthogonal matrix in the form

$$G(p,q,\theta) = \begin{array}{c} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \end{array} \begin{array}{l} \\ \\ \leftarrow \text{row } p \\ \\ \leftarrow \text{row } q \\ \\ \\ \end{array} . \tag{5.30}$$

When a Givens rotation matrix $\mathbf{G}(p,q,\theta)$ multiplies a vector, only elements with indices p and q are affected. Restricting our attention to these two elements, say a and b, the Givens rotation is given explicitely as

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a\cos\theta + b\sin\theta \\ -a\sin\theta + b\cos\theta \end{bmatrix} . \tag{5.31}$$

Apparently, the rotation can zero the lowest element, if the angle θ is chosen as

$$\tan\theta = \frac{b}{a} \;\;\Rightarrow\;\; \theta = \mathrm{atan2}(b,a) . \tag{5.32}$$

A sequence of Givens rotations can zero all elements of a matrix below the main diagonal. Which obviously amounts to QR-decomposition of the matrix with $\mathbf{Q}$ being the product of all the applied Givens matrices.

Since each Givens rotation only affects two rows of the matrix it is possible to apply a set of rotations in parallel. Givens rotations are also more efficient on sparse matrices.

5.3.2 LU-decomposition

LU-decomposition is a factorization of a square matrix $\mathbf{A}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$,

$$\mathbf{A} = \mathbf{LU} . \tag{5.33}$$

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

If $\mathbf{A}$ is an $n \times n$ matrix, the condition (5.33) is a set of n^2 equations,

$$\sum_{k=1}^{n} L_{ik} U_{kj} = A_{ij} \;\Big|_{i,j=1\dots n} , \tag{5.34}$$

for $n^2 + n$ unknown elements of the triangular matrices $\mathbf{L}$ and $\mathbf{U}$. 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 $\mathbf{L}$ are equal one,

$$L_{ii} = 1 \, , \; i = 1 \dots n . \tag{5.35}$$

The system (5.34) with the extra conditions (5.35) can then be easily solved row after row using the *Doolittle's algorithm*,

for $i = 1 \dots n$:
  $L_{ii} = 1$
  for $j = i \dots n$: $U_{ij} = A_{ij} - \sum_{k<i} L_{ik} U_{kj}$
  for $j = i+1 \dots n$: $L_{ji} = \frac{1}{U_{ii}} \left(A_{ji} - \sum_{k<j} L_{jk} U_{ki} \right)$

In a slightly different *Crout's algorithm* it is the matrix **U** that has unit diagonal elements,

```
for i = 1...n :
   U_ii = 1
   for j = i...n :   L_ji = A_ji − Σ_{k<i} L_jk U_ki
   for j = i+1...n :   U_ij = (1/L_ii) (A_ji − Σ_{k<j} L_jk U_ki)
```

Without a proper ordering (permutations) in the matrix, the factorization may fail. For example, it is easy to verify that $A_{11} = L_{11}U_{11}$. If $A_{11} = 0$, then at least one of L_{11} and U_{11} has to be zero, which implies either **L** or **U** is singular, which is impossible if **A** is non-singular. This is however only a procedural problem. It can be removed by simply reordering the rows of **A** so that the first element of the permuted matrix is nonzero (or, even better, the largest in absolute value among all elements of the column below the diagonal). The same problem in subsequent factorization steps can be removed in a similar way. Such algorithm is refered to as *partial pivoting*. It requires an extra integer array to keep track of row permutations.

5.4 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,

$$\det \mathbf{A} = \det \mathbf{LU} = \det \mathbf{L} \det \mathbf{U} = \det \mathbf{U} = \prod_{i=1}^{n} U_{ii} \; . \tag{5.36}$$

For the QR-decomposition

$$\det \mathbf{A} = \det \mathbf{QR} = \det \mathbf{Q} \det \mathbf{R} \; . \tag{5.37}$$

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

$$|\det \mathbf{A}| = |\det \mathbf{R}| = \left| \prod_{i=1}^{n} R_{ii} \right| \; . \tag{5.38}$$

5.5 Matrix inverse

The inverse $\mathbf{A}^{-1}$ of a square $n \times n$ matrix **A** can be calculated by solving n linear equations

$$\mathbf{A}\mathbf{x}_i = \mathbf{e}_i \Big|_{i=1\dots n} \; , \tag{5.39}$$

where $\mathbf{e}_i$ is a column where all elements are equal zero except for the element number i, which is equal one; that is, columns $\mathbf{e}_i$ form a unity matrix. The matrix made of columns $\mathbf{x}_i$ is apparently the inverse of **A**.

A C implementation of this algorithm using QR-decomposition is shown in Table (5.3).

Table 5.3: Matrix inverse with QR-decomposition in C

```
#include <matrix.h>
void qrback_inplace(matrix*,matrix*,vector*);
void qrinverse(matrix*Q, matrix*R, matrix*Ainv){
  matrix_set_identity(Ainv);
  for(int i=0;i<Q->size1;i++) qrback_inplace(Q,R,matrix_get_column(Ainv,i));
}
```

Chapter 6

Ordinary least squares

6.1 Introduction

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 linear system. It often arises in applications where a theoretical model is fitted to experimental data.

6.2 Linear least-squares problem

Consider a linear system

$$\mathbf{Ac} = \mathbf{b} \,, \tag{6.1}$$

where $\mathbf{A}$ 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 $\mathbf{Ac}$ is only approximately equal $\mathbf{b}$—in the sence that the Euclidean norm of the difference between $\mathbf{Ac}$ and $\mathbf{b}$ is minimized,

$$\mathbf{c} : \min_{\mathbf{c}} \|\mathbf{Ac} - \mathbf{b}\|^2 \,. \tag{6.2}$$

The problem (6.2) is called the ordinary least-squares problem and the vector $\mathbf{c}$ that minimizes $\|\mathbf{Ac} - \mathbf{b}\|^2$ is called the *least-squares solution.*

6.3 Solution via QR-decomposition

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

$$\|\mathbf{Ac} - \mathbf{b}\|^2 = \|\mathbf{QRc} - \mathbf{b}\|^2$$
$$= \|\mathbf{Rc} - \mathbf{Q}^T\mathbf{b}\|^2 + \|(1 - \mathbf{QQ}^T)\mathbf{b}\|^2 \geq \|(1 - \mathbf{QQ}^T)\mathbf{b}\|^2 \; . \tag{6.3}$$

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

$$\mathbf{Rc} = \mathbf{Q}^T\mathbf{b} \; . \tag{6.4}$$

The system is right-triangular and can be readily solved by back-substitution. Thus the solution to the ordinary least-squares problem (6.2) is given by the solution of the triangular system (6.4).

6.4 Ordinary least-squares curve fitting

Ordinary 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, F, of m functions $\{f_k(x) \mid k = 1, \dots, m\}$,

$$F_{\mathbf{c}}(x) = \sum_{k=1}^{m} c_k f_k(x) \; . \tag{6.5}$$

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

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{F(x_i) - y_i}{\Delta y_i} \right)^2 \; . \tag{6.6}$$

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 (6.5) is apparently equivalent to the least-squares problem (6.2) where

$$A_{ik} = \frac{f_k(x_i)}{\Delta y_i} \; , \; b_i = \frac{y_i}{\Delta y_i} \; . \tag{6.7}$$

If $\mathbf{QR} = \mathbf{A}$ is the QR-decomposition of the matrix $\mathbf{A}$, the formal least-squares solution to the fitting problem is

$$\mathbf{c} = R^{-1}Q^T\mathbf{b} \; . \tag{6.8}$$

However, in practice one has to back-substitute the system $\mathbf{Rc} = \mathbf{Q}^T\mathbf{b}$.

6.4.1 Variances and correlations of fitting parameters

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

$$\delta c_k = \sum_i \frac{\partial c_k}{\partial y_i} \delta y_i \,. \tag{6.9}$$

In a good experiment the deviations δy_i are statistically independent and distributed normally with the standard deviations Δy_i. The deviations (6.9) are then also distributed normally with *variances*

$$\langle \delta c_k \delta c_k \rangle = \sum_i \left(\frac{\partial c_k}{\partial y_i} \Delta y_i \right)^2 = \sum_i \left(\frac{\partial c_k}{\partial b_i} \right)^2 \,. \tag{6.10}$$

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

$$\Delta c_k = \sqrt{\langle \delta c_k \delta c_k \rangle} = \sqrt{\sum_i \left(\frac{\partial c_k}{\partial b_i} \right)^2} \,. \tag{6.11}$$

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

$$\Sigma_{kq} \equiv \langle \delta c_k \delta c_q \rangle = \sum_i \frac{\partial c_k}{\partial b_i} \frac{\partial c_q}{\partial b_i} \,. \tag{6.12}$$

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,

$$\frac{\langle \delta c_k \delta c_q \rangle}{\sqrt{\langle \delta c_k \delta c_k \rangle \langle \delta c_q \delta c_q \rangle}} \tag{6.13}$$

are called *correlations*.

Using (6.12) and (6.8) the covariance matrix can be calculated as

$$\Sigma = \left(\frac{\partial \mathbf{c}}{\partial \mathbf{b}} \right) \left(\frac{\partial \mathbf{c}}{\partial \mathbf{b}} \right)^T = R^{-1} (R^{-1})^T = (R^T R)^{-1} = (A^T A)^{-1} \,. \tag{6.14}$$

The square roots of the diagonal elements of this matrix provide the estimates of the errors $\Delta \mathbf{c}$ of the fitting coefficients,

$$\Delta c_k = \sqrt{\Sigma_{kk}} \,\Big|_{k=1\ldots m} \,, \tag{6.15}$$

and the (normalized) off-diagonal elements provide the estimates of their correlations.

A C implementation of the ordinary least squares fit with QR-decomposition is shown in Table (6.1).

An illustration of a fit is shown on Figure (6.1) where a polynomial is fitted to a set of data.

Table 6.1: C implementation of ordinary least squares fit with QR-decomposition.

```
#include <matrix.h>
#include <gc.h> // garbage collector
#include <qr.h> // qr-decomposition
typedef struct {vector*c; matrix*S;} lsfit;
lsfit* lsfit_alloc(vector*x,vector*y,vector*dy,
int nf,double f(int k,double x))
{ int n=x->size, m=nf;
  matrix *A = matrix_alloc(n,m), *R = matrix_alloc(m,m);
  vector *b = vector_copy(y); vector_div(b,dy);
  for(int i=0;i<n;i++)
  { double xi=vector_get(x,i), dyi=vector_get(dy,i);
    for(int k=0;k<m;k++) matrix_set(A,i,k,f(k,xi)/dyi);
  }
  qrdec(A,R);
  lsfit* fit = (lsfit*)GC_MALLOC(sizeof(lsfit));
  fit->c = qrback(A,R,b);
  fit->S = qrinverse(matrixT_times_matrix(R,R));
  return fit;
}
```

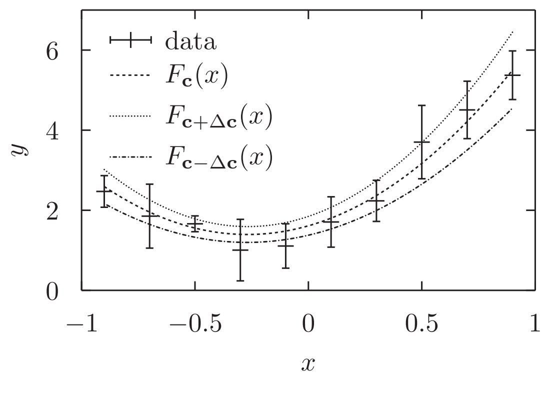

Figure 6.1: Ordinary least squares fit of $F_{\mathbf{c}}(x) = c_1 + c_2 x + c_3 x^2$ to a set of data. Shown are fits with optimal coefficiens $\mathbf{c}$ as well as with $\mathbf{c} + \Delta\mathbf{c}$ and $\mathbf{c} - \Delta\mathbf{c}$.

Chapter 7

Nonlinear equations

7.1 Introduction

Non-linear equations (or *root-finding*) is a problem of finding a set of n variables $\mathbf{x} = \{x_1, \dots, x_n\}$ which satisfy a system of n (generally) non-linear equations

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

In matrix notation the system is written as

$$\mathbf{f}(\mathbf{x}) = 0 , \tag{7.2}$$

where $\mathbf{f} \doteq \{f_1, \dots, f_n\}$.

In one-dimensional cases it is generally possible to plot the function in the region of interest and see whether the graph crosses the x-axis. One can then be sure the root exists and even figure out its approximate position to start one's root-finding algorithm with. In multi-dimensions one generally does not know if the root exists at all, until it is found.

The root-finding algorithms generally proceed by iteration, starting from some approximate solution and making consecutive steps (hopefully) in the direction of the suspected root until some convergence criterion is satisfied. The procedure is generally not even guaranteed to converge unless starting from a point close enough to the sought root.

We shall only consider the multi-dimensional case here since i) the multi-dimensional root-finding is more difficult, and ii) the multi-dimensional routines can also be used in a one-dimensional case.

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 $\mathbf{x} = \{x_1, \dots, x_n\}$ 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

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = 0 \Big|_{i=1,\dots,n} . \tag{7.3}$$

The first order Taylor expansion of (7.3) gives a system of linear equations,

$$f_i(\mathbf{x}) + \sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k} \Delta x_k = 0 \Big|_{i=1,\dots,n} , \tag{7.4}$$

or, in the matrix form,

$$\mathbf{J}\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}), \tag{7.5}$$

where $\mathbf{f}(\mathbf{x}) \doteq \{f_0(\mathbf{x}), \dots, f_n(\mathbf{x})\}$ and $\mathbf{J}$ is the matrix of partial derivatives[1],

$$J_{ik} \doteq \frac{\partial f_i}{\partial x_k} , \tag{7.6}$$

called the *Jacobian matrix.*

The solution $\Delta\mathbf{x}$ to the linear system (7.5)—called the Newton's step—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 the condition

$$\|\mathbf{f}(\mathbf{x} + \lambda\Delta\mathbf{x})\| < \left(1 - \frac{\lambda}{2}\right) \|\mathbf{f}(\mathbf{x})\| \tag{7.7}$$

is satisfied. If the condition is not satisfied for sufficiently small λ_{min} the step is taken with λ_{min} simply to step away from this diffictul place and try again. A typical algrorithm of the Newton's method with backtracking line search and condition (7.7) is shown in Table 7.1.

[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.

Table 7.1: Newton's root-finding algorithm with backtracking.

```
repeat
   solve JΔx = −f(x) for Δx
   λ ← 1
   while ( ‖f(x + λΔx)‖ > (1 − λ/2) ‖f(x)‖ and λ > 1/64 ) do λ ← λ/2
   x ← x + λΔx
until converged (e.g. ‖f(x)‖ < tolerance)
```

A somewhat more refined backtracking linesearch is based on an approximate minimization of the function

$$g(\lambda) \doteq \frac{1}{2}\|\mathbf{f}(\mathbf{x}+\lambda\Delta\mathbf{x})\|^2 \tag{7.8}$$

using interpolation. The values $g(0) = \frac{1}{2}\|\mathbf{f}(\mathbf{x})\|^2$ and $g'(0) = -\|\mathbf{f}(\mathbf{x})\|^2$ are already known (check this). If the previous step with certain λ_{trial} was rejected, we also have $g(\lambda_{\text{trial}})$. These three quantities allow to build a quadratic approximation,

$$g(\lambda) \approx g(0) + g'(0)\lambda + c\lambda^2 \,, \tag{7.9}$$

where

$$c = \frac{g(\lambda_{\text{trial}}) - g(0) - g'(0)\lambda_{\text{trial}}}{\lambda_{\text{trial}}^2} \,. \tag{7.10}$$

The minimum of this approximation (determined by the condition $g'(\lambda) = 0$),

$$\lambda_{\text{next}} = -\frac{g'(0)}{2c} \,, \tag{7.11}$$

becomes the next trial step-size.

The procedure is repeated recursively until either condition (7.7) is satisfied or the step becomes too small (in which case it is taken unconditionally in order to simply get away from the difficult place).

7.3 Broyden's quasi-Newton method

The Newton's method requires calculation of the Jacobian matrix 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's algorithm estimates the Jacobian $\mathbf{J}+\Delta\mathbf{J}$ at the point $\mathbf{x}+\Delta\mathbf{x}$ using the finite-difference approximation,

$$(\mathbf{J}+\Delta\mathbf{J})\Delta\mathbf{x} = \Delta\mathbf{f} \,, \tag{7.12}$$

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

Table 7.2: C implementation of Newton's algorithm with backtracking.

```
#include<matrix.h> // garbage collected matrices and vectors
#include<qr.h>
vector* newton
  (vector* f(vector*x), vector* startx, vector* dx, double eps)
{
  vector *x=vector_copy(startx), *fx, *Dx;
  int n=x->size;
  matrix *A=matrix_alloc(n,n), *R=matrix_alloc(n,n);
  do
  {
    fx=f(x); // : gets garbage collected
    for(int j=0;j<n;j++) // calculating the Jacobian matrix :
    {
      vector_set(x,j,vector_get(x,j)+vector_get(dx,j));
      vector* df=vector_sum(1,f(x),-1,fx);
      for(int i=0;i<n;i++)
        matrix_set(A,i,j,vector_get(df,i)/vector_get(dx,j));
      vector_set(x,j,vector_get(x,j)-vector_get(dx,j));
    }
    qrdec(A,R);
    vector* minusfx=vector_copy(fx); vector_scale(minusfx,-1);
    Dx=qrback(A,R,minusfx); // : Newton's step
    double lambda=2, normfx=vector_norm(fx);
    vector *z, *fz;
    do // back-tracking line-search:
    {
      lambda/=2; z=vector_sum(1,x,lambda,Dx); fz=f(z);
    }while(vector_norm(fz)>(1-lambda/2)*normfx && lambda>0.02);
    x=z; fx=fz;
  }while(vector_norm(Dx)>vector_norm(dx) && vector_norm(fx)>eps);
  return x;
}
```

The matrix equation (7.12) is under-determined in more than one dimension as it contains only n equations to determine n^2 matrix elements of $\Delta\mathbf{J}$. Broyden suggested to choose $\Delta\mathbf{J}$ as a rank-1 update, linear in $\Delta\mathbf{x}$,

$$\Delta\mathbf{J} = \mathbf{c}\,\Delta\mathbf{x}^T \,, \tag{7.13}$$

where the unknown vector $\mathbf{c}$ can be found by substituting (7.13) into (7.12), which gives

$$\Delta\mathbf{J} = \frac{\Delta\mathbf{f} - \mathbf{J}\Delta\mathbf{x}}{\|\Delta\mathbf{x}\|^2}\Delta\mathbf{x}^T \,. \tag{7.14}$$

In practice if one wanders too far from the point where $\mathbf{J}$ was first calculated the accuracy of the updates may decrease significantly. In such case one might need to recalculate $\mathbf{J}$ anew. For example, two successive steps with λ_{min} might be interpreted as a sign of accuracy loss in $\mathbf{J}$ and subsequently trigger its recalculation.

Chapter 8

Minimization and optimization

8.1 Introduction

Minimization is a problem of finding the minimum (or the maximum) of a given, generally non-linear, real valued function $f(\mathbf{x})$ of an n-dimensional argument $\mathbf{x} \doteq \{x_1, \dots, x_n\}$.

Minimization is a simple case of a more general poblem—*optimization*—which includes finding best available values of some objective function within a given domain and subject to given constrains.

Minimization is related to root-finding as at the minimum all partial derivatives of the objective function vanish,

$$\frac{\partial f}{\partial x_i} = 0 \,. \tag{8.1}$$

One can alternatively solve this system of non-linear equations.

8.2 Local minimization

8.2.1 Newton's methods

Newton's method is based on the quadratic approxiamtion of the objective function f in the vicinity of the suspected minimum,

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \mathbf{H}(\mathbf{x})\Delta\mathbf{x} \,, \tag{8.2}$$

where the vector $\nabla f(\mathbf{x})$ is the gradient of the objective function at the point $\mathbf{x}$,

$$\nabla f(\mathbf{x}) \doteq \left[\frac{\partial f(\mathbf{x})}{\partial x_i}\right]_{i=1,\dots,n} \,, \tag{8.3}$$

and $\mathbf{H}(\mathbf{x})$ is the *Hessian matrix* – a square matrix made of second-order partial derivatives of the objective function at the point $\mathbf{x}$,

$$\mathbf{H}(\mathbf{x}) \doteq \left[\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\right]_{i,j\in 1,\dots,n} . \tag{8.4}$$

The minimum of this quadratic form as function of $\Delta\mathbf{x}$ is found at the point where its gradient with respect to $\Delta\mathbf{x}$ vanishes,

$$\nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x})\Delta\mathbf{x} = 0 . \tag{8.5}$$

This gives an approximate step towards the minimum, called the *Newton's step*,

$$\Delta\mathbf{x} = -\mathbf{H}(\mathbf{x})^{-1}\nabla f(\mathbf{x}) . \tag{8.6}$$

The original Newton's method is simply the iteration,

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}(\mathbf{x}_k)^{-1}\nabla f(\mathbf{x}_k) , \tag{8.7}$$

where at each iteration the full Newton's step is taken and the Hessian matrix is recalculated. In practice, instead of calculating $\mathbf{H}^{-1}$ one rather solves the linear equation (8.5).

Usually Newton's method is modified to include a smaller step size, $\lambda\Delta\mathbf{x}$, with $0 < \lambda < 1$. The step-size λ can be found by a backtracking algoritm similar to that in the Newton's method for root-finding. One starts with $\lambda = 1$ and than backtracks, $\lambda \leftarrow \lambda/2$, until the *Armijo condition*,

$$f(\mathbf{x} + \lambda\Delta\mathbf{x}) < f(\mathbf{x}) + \alpha\nabla f(\mathbf{x})\lambda\Delta\mathbf{x} , \tag{8.8}$$

is satisfied. The parameter α can be chosen as small as 10^{-4}.

8.2.2 Quasi-Newton methods

Quasi-Newton methods are variations of the Newton's method which attempt to avoid recalculation of the Hessian matrix at each iteration, trying instead certain updates based on the analysis of the gradient vectors. The update $\delta\mathbf{H}$ is usually chosen to satisfy the condition

$$\nabla f(\mathbf{x} + \mathbf{s}) = \nabla f(\mathbf{x}) + (\mathbf{H} + \delta\mathbf{H})\mathbf{s} , \tag{8.9}$$

called *secant equation*, which is the Taylor expansion of the gradient. This secant condition is under-determined in more than one dimension as it consists of only n equations for the n^2 unknown elements of the update $\delta\mathbf{H}$. Various quasi-Newton methods use different choices for the form of the solution of the secant equation.

In quasi-Newton methods one often starts with a unity matrix as the zeroth approximation for the Hessian matrix and then applies the updates. In this case one can actually use the inverse Hessian matrix—thus avoiding the need to solve the linear equation at each iteration—and apply the updates directly to the inverse matrix.

Broydens's update

The Broyden's update is chosen in the form

$$\delta\mathbf{H} = \mathbf{c}\mathbf{s}^T \,. \tag{8.10}$$

where the vector $\mathbf{c}$ is found from the condition (8.9). The resulting update is

$$\mathbf{H} \to \mathbf{H} + \frac{\mathbf{y} - H\mathbf{s}}{\mathbf{s}^T\mathbf{s}}\mathbf{s}^T \,, \tag{8.11}$$

where $\mathbf{y} \doteq \nabla f(\mathbf{x}+\mathbf{s}) - \nabla f(\mathbf{x})$. The update for the inverse matrix is given as[1]

$$\mathbf{H}^{-1} \to \mathbf{H}^{-1} + \frac{(\mathbf{s} - \mathbf{H}^{-1}\mathbf{y})\mathbf{s}^T\mathbf{H}^{-1}}{\mathbf{y}^T\mathbf{H}^{-1}\mathbf{s}} \,. \tag{8.12}$$

SR1 update

The symmetric-rank-1 update (SR1) in chosen in the form

$$\delta\mathbf{H} = \mathbf{u}\mathbf{u}^T \,, \tag{8.13}$$

where the vector $\mathbf{u}$ is found from the condition (8.9). The resulting update is

$$\mathbf{H} \to \mathbf{H} + \frac{(\mathbf{y} - \mathbf{H}\mathbf{s})(\mathbf{y} - \mathbf{H}\mathbf{s})^T}{(\mathbf{y} - \mathbf{H}\mathbf{s})^T\mathbf{s}} \,. \tag{8.14}$$

And for the inverse matrix

$$\mathbf{H}^{-1} \to \mathbf{H}^{-1} + \frac{(\mathbf{s} - \mathbf{H}^{-1}\mathbf{y})(\mathbf{s} - \mathbf{H}^{-1}\mathbf{y})^T}{(\mathbf{s} - \mathbf{H}^{-1}\mathbf{y})^T\mathbf{y}} \,. \tag{8.15}$$

Davidon–Fletcher–Powell (DFP) update

$$\delta\mathbf{H}^{-1} = \frac{\mathbf{y}\mathbf{y}^T}{\mathbf{y}^T s} - \frac{\mathbf{H}^{-1}\mathbf{s}\mathbf{s}^T\mathbf{H}^{-1}}{\mathbf{s}^T\mathbf{H}^{-1}\mathbf{s}} \tag{8.16}$$

8.2.3 Downhill simplex method

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

[1] using the Sherman-Morrison formula,

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}.$$

The advantages of the downhill simplex method is its stability and the lack of use of derivatives. However, the convergence is realtively slow as compared to Newton's methods.

In order 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, $\mathbf{p}_{\rm hi}$, with the highest value of the function: $f(\mathbf{p}_{\rm hi}) = \max_k f(\mathbf{p}_k)$.
- Lowest point: the vertex, $\mathbf{p}_{\rm lo}$, with the lowest value of the function: $f(\mathbf{p}_{\rm lo}) = \min_k f(\mathbf{p}_k)$.
- Centroid: the center of gravity of all points, except for the highest: $\mathbf{p}_{\rm ce} = \frac{1}{n}\sum_{(k\neq {\rm hi})} \mathbf{p}_k$

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

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

A C-implementation of the simplex operations is listed in Table 8.4.

Finally, Table 8.1 shows one possible algorithm for the downhill simplex method.

8.3 Global optimization

Global optimization is the problem of locating (a good approximation to) the global minimum of a given objective function in a search space large enough to prohibit exhaustive enumeration.

When only a small sub-space of the search space can be realistically sampled the stochastic methods usually come to the fore.

A good local minimizer converges to the nearest local miminum relatively fast, so one possible global minimizer can be constructed by simply starting the local miminizer from different random starting points.

In the following several popular global minimization algorithms are shortly described.

Table 8.1: Downhill simplex (Nelder-Mead) algorithm

```
REPEAT :
  find highest , lowest , and centroid points of the simplex
  try reflection
  IF f(reflected) < f(lowest) :
    try expansion
    IF f(expanded) < f(reflected) :
      accept expansion
    ELSE :
      accept reflection
  ELSE :
    IF f(reflected) < f(highest) :
      accept reflection
    ELSE :
      try contraction
      IF f(contracted) < f(highest) :
        accept contraction
      ELSE :
        do reduction
UNTIL converged (e.g. size(simplex)<tolerance)
```

8.3.1 Simulated annealing

Simulated annealing is a stochastic metaheuristic algorithm for global minimization. The name and inspiration come from annealing — heating up and cooling slowly — in material science. The slow cooling allows a piece of material to reach a state with "lowest energy".

The objective function in the space of states is interpreted as some sort of potential energy and the states — the points in the search space — are interpeted as physical states of some physical system. The system attempts to make transitions from its current state to some randomly sampled nearest states with the goal to eventually reach the state with minimal energy – the global minimum.

The system is attached to a thermal resevoir with certain temperature. Each time the energy of the system is measured the reservoir supplies it with a random amount of thermal energy sampled from the Boltzmann distribution,

$$P(E) = Te^{-E/T} . \tag{8.17}$$

If the temperature equals zero the system can only make transitions to the neighboring states with lower potential energy. In this case the algorithm turns merely into a local minimizer with random sampling.

If temperature is finite the system is able to climb up the ridges of the potential energy — about as high as the current temperature — and thus escape from local minima and hopefully eventually reach the global minimum.

Table 8.2: Simulated annealing algorithm

```
state ← start_state
T ← start_temperature
energy ← E(state) + random_thermal_energy(T)
REPEAT :
  new_state ← neighbour(state)
  new_energy ← E(new_state) + random_thermal_energy(T)
  IF new_energy < energy :
    state ← new_state
    energy ← new_energy
  reduce_temperature_according_to_schedule(T)
UNTIL terminated
```

One typically starts the simulation with some finite temperature on the order of the height of the typical hills on the potential energy surface, letting the system to wander almost unhindered around the landscape with a good chance to locale if not the best then at least a good enough minimum. The temperature is then slowly reduced following some annealing schedule which may be supplied by the user but must end with $T = 0$ towards the end of the alloted time budget.

Table 8.2 lists one possible variant of the algorithm.

The function `neigbour(state)` should return a randomly chosen neighbour of the given state.

Downhill simplex method can incorporate simulated annealing by adding the stochastic thermal energy to the values of the objective function at the verices.

8.3.2 Quantum annealing

Quantum annealing is a general global minimization algorithm which — like simulated annealing — also allows the search path to escape from local minima. However instead of the thermal jumps over the potential barriers quantum annealing allows the system to tunnel through the barriers.

In its simple incarnation the quantum annealing algorithm allows the system to attempt transitions not only to the nearest states but also to distant states within certain "tunneling distance" from the current state. The transition is accepted only if it reduces the potential energy of the system.

At the beginning of the minimization prcedure the tunnelling distance is large — on the order of the size of the region where the global minimum is suspected to be localed — allowing the system to explore the region. The tunneling distance is then slowly reduced according to a schedule such that by the end of the alloted time the tunnelling distance reduces to zero at which point the system hopefully is in the state with minimal energy.

Table 8.3: Quantum annealing algorithm

```
state ← start_state
energy ← E(state)
R ← start_radius
REPEAT :
  new_state ← neighbour_within_radius(state,R)
  new_energy ← E(new_state)
  IF new_energy < energy :
    state ← new_state
    energy ← new_energy
  reduce_radius_according_to_schedule(R)
UNTIL terminated
```

8.3.3 Evolutionary algorithms

Unlike annealing algorithms, which follow the motion of only one point in the search space, the evolutionary algorithms typically follow a set of points called a population of individuals. A bit like the downhill simplex method which follows the motion of a set of points – the simplex.

The population evolves towards more fit individuals where fitness is understood in the sence of minimizing the objective functions. The parameters of the individuals (for example, the coordinates of the points in multi-dimentional minimization of a continuous objective function) are called genes.

The algorithm proceeds iteratively with the population in each iteration called a generation. In each generation the fitness of each individual — typically, the value of the objective function — is evaluated and the new generation is generated stochastically from the gene pool of the current generation through crossovers and mutations such that the genes of more fit individuals have a better chance of propagating into the next generation.

Each new individual in the next generation is produced from a pair of "parent" individuals of the current generation. The use of two "parents" is biologically inspired, in practice more than two "parents" can be used as well. The parents for a new individual are selected from the individuals of the current generation through a fitness based stochastic process where fitter individuals are more likely to be selected.

The "child" individual shares many characteristics of its "parents". In the simplest case the "child" may get its genes by simply averaging the genes of its parents. Then a certain amount of mutations — random changes in the genes — are added to the "child's" genes.

Generation of "children" continues until the population of the new generation reaches the appropriate size after which the iteration repeats itself.

The algorithm is terminated when the fitness level of the population is deemed sufficient or when the allocated budget is exhausted.

8.4 Implementation in C

Table 8.4: C implementation of simplex operations

```
#include <math.h>
#define float double
void reflection
   (float* highest, float* centroid, int dim, float* reflected){
   for(int i=0;i<dim;i++) reflected[i]=2*centroid[i]-highest[i];
}
void expansion
   (float* highest, float* centroid, int dim, float* expanded) {
   for(int i=0;i<dim;i++) expanded[i]=3*centroid[i]-2*highest[i];
}
void contraction
   (float* highest, float* centroid, int dim, float* contracted){
   for(int i=0;i<dim;i++)
      contracted[i]=0.5*centroid[i]+0.5*highest[i];
}
void reduction( float** simplex, int dim, int lo){
   for(int k=0;k<dim+1;k++) if(k!=lo) for(int i=0;i<dim;i++)
         simplex[k][i]=0.5*(simplex[k][i]+simplex[lo][i]);
}
float distance(float* a, float* b, int dim){
   float s=0; for(int i=0;i<dim;i++) s+=pow(b[i]-a[i],2);
   return sqrt(s);
}
float size(float** simplex,int dim){
   float s=0; for(int k=1;k<dim+1;k++){
      float dist=distance(simplex[0],simplex[k],dim);
      if(dist>s) s=dist; }
   return s;
}
```

```
#define float double
void simplex_update(float** simplex, float* f_values, int d,
   int* hi, int* lo, float* centroid)
{
   *hi=0; float highest=f_values[0];
   *lo=0; float lowest =f_values[0];
   for(int k=1;k<d+1;k++) {
      float next=f_values[k];
      if(next>highest){highest=next;*hi=k;}
      if(next<lowest) {lowest=next; *lo=k;}
      }
   for(int i=0;i<d;i++) {
      float s=0;for(int k=0;k<d+1;k++)if(k!=*hi)s+=simplex[k][i];
      centroid[i]=s/d; }
}
void simplex_initiate(
   float fun(float*), float** simplex, float* f_values, int d,
   int* hi, int* lo, float* centroid)
{
   for(int k=0;k<d+1;k++) f_values[k]=fun(simplex[k]);
   simplex_update(simplex,f_values,d,hi,lo,centroid);
}
```

Table 8.5: C implementation of downhill simplex algorithm

```
#include <simplex.h>
#include <stdlib.h>
#include <stdio.h>
int downhill_simplex(
  double F(double*),double**simplex,int d,double simplex_size_goal)
{
int hi,lo,k=0; double centroid[d], F_value[d+1], p1[d], p2[d];
simplex_initiate(F,simplex,F_value,d,&hi,&lo,centroid);
while(size(simplex,d)>simplex_size_goal){
  simplex_update(simplex,F_value,d,&hi,&lo,centroid);
  reflection(simplex[hi],centroid,d,p1); double f_re=F(p1);
  if(f_re<F_value[lo]){ // reflection looks good: try expansion
    expansion(simplex[hi],centroid,d,p2); double f_ex=F(p2);
    if(f_ex<f_re){ // accept expansion
      for(int i=0;i<d;i++)simplex[hi][i]=p2[i]; F_value[hi]=f_ex;}
    else{ // reject expansion and accept reflection
      for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_re;}}
  else{ // reflection wasn't good
    if(f_re<F_value[hi]){ // ok, accept reflection
      for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_re;}
    else{ // try contraction
      contraction(simplex[hi],centroid,d,p1); double f_co=F(p1);
      if(f_co<F_value[hi]){ // accept contraction
        for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_co;}
      else{ // do reduction
        reduction(simplex,d,lo);
        simplex_initiate(F,simplex,F_value,d,&hi,&lo,centroid);}}}
  k++;} return k;
}
```

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 $\lambda_1, \ldots, \lambda_n$, and its (orthogonalized) eigenvectors $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ 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 multitude 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,

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

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 elements 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 $V = \mathbf{1} J_1 J_2 ...$, where J_i are the successive Jacobi matrices. At each stage the transformation is

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

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

QR/QL algorithm

An orthogonal transformation of a real symmetric matrix, $\mathbf{A} \rightarrow \mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{R}\mathbf{Q}$, where $\mathbf{Q}$ is from the QR-decomposition of $\mathbf{A}$, partly turns the matrix $\mathbf{A}$ into diagonal form. Successive iterations eventually make it diagonal. If there are degenerate eigenvalues there will be a corresponding block-diagonal sub-matrix. For convergence properties it is of advantage to use *shifts*:

instead of QR[$\mathbf{A}$] we do QR[$\mathbf{A} - s\mathbf{1}$] and then $\mathbf{A} \to \mathbf{RQ} + s\mathbf{1}$. The shift s can be chosen as $\mathbf{A}_{nn}$. As soon as an eigenvalue is found the matrix is deflated, that is the corresponding row and column are crossed out.

Accumulating the successive transformation matrices $\mathbf{Q}_i$ into the total matrix $\mathbf{Q} = \mathbf{Q}_1 \ldots \mathbf{Q}_N$, such that $Q^T \mathbf{AQ} = \Lambda$, gives the eigenvectors as columns of the $\mathbf{Q}$ matrix.

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

Tridiagonalization.

Each iteration of the QR/QL algorithm is an $O(n^3)$ operation. On a tridiagonal matrix it is only $O(n)$. Therefore the effective strategy is first to make the matrix tridiagonal and then apply the QR/QL algorithm. Tridiagonalization of a matrix is a non-iterative operation with a fixed number of steps.

9.3 Eigenvalues of updated matrix

In practice it happens quite often that the (symmetric) matrix A to be diagonalized is given in the form of a diagonal matrix, $\mathbf{D}$, plus an update matrix, $\mathbf{W}$,

$$\mathbf{A} = \mathbf{D} + \mathbf{W} , \tag{9.12}$$

where the update $\mathbf{W}$ is a simpler — in a certain sense — matrix which allows a more efficient calculation of the updated eigenvalues, as compared to general diagonalization algorithms.

The most common updates are

- symmetric rank-1 update,

$$\mathbf{W} = \mathbf{u}\mathbf{u}^T , \tag{9.13}$$

  where $\mathbf{u}$ is a columnt-vector;

- symmetric rank-2 update,

$$\mathbf{W} = \mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T ; \tag{9.14}$$

- symmetric row/column update – a special case of rank-2 update,

$$\mathbf{W} = \begin{bmatrix} 0 & \ldots & u_1 & \ldots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ u_1 & \ldots & u_p & \ldots & u_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \ldots & u_n & \ldots & 0 \end{bmatrix} \equiv \mathbf{e}(p)\mathbf{u}^T + \mathbf{u}\mathbf{e}(p)^T , \tag{9.15}$$

  where $\mathbf{e}(p)$ is the unit vector in the p-direction.

9.3.1 Rank-1 update

We assume that the size-n real symmetric matrix A to be diagonalized is given in the form of a diagonal matrix plus a rank-1 update,

$$\mathbf{A} = \mathbf{D} + \sigma \mathbf{u}\mathbf{u}^T , \tag{9.16}$$

where $\mathbf{D}$ is a diagonal matrix with diagonal elements $\{d_1, \dots, d_n\}$ and $\mathbf{u}$ is a given vector. The diagonalization of such matrix can be done in $O(m^2)$ operations, where $m \le n$ is the number of non-zero elements in the update vector $\mathbf{u}$, as compared to $O(n^3)$ operations for a general diagonalization [7].

The eigenvalue equation for the updated matrix reads

$$\left(\mathbf{D} + \sigma \mathbf{u}\mathbf{u}^T\right) \mathbf{q} = \lambda \mathbf{q} , \tag{9.17}$$

where λ is an eigenvalue and $\mathbf{q}$ is the corresponding eigenvector. The equation can be rewritten as

$$(\mathbf{D} - \lambda \mathbf{1})\, \mathbf{q} + \sigma \mathbf{u}\mathbf{u}^T \mathbf{q} = 0 . \tag{9.18}$$

Multiplying from the left with $\mathbf{u}^T (\mathbf{D} - \lambda \mathbf{1})^{-1}$ gives

$$\mathbf{u}^T \mathbf{q} + \mathbf{u}^T (\mathbf{D} - \lambda \mathbf{1})^{-1} \sigma \mathbf{u}\mathbf{u}^T \mathbf{q} = 0 . \tag{9.19}$$

Finally, dividing by $\mathbf{u}^T \mathbf{q}$ leads to the (scalar) *secular equation* (or *characteristic equation*) in λ,

$$1 + \sum_{i=1}^{m} \frac{\sigma u_i^2}{d_i - \lambda} = 0 , \tag{9.20}$$

where the summation index counts the m non-zero components of the update vector $\mathbf{u}$. The m roots of this equation determine the (updated) eigenvalues[1].

Finding a root of a rational function requires an iterative technique, such as the Newton-Raphson method. Therefore diagonalization of an updated matrix is still an iterative procedure. However, each root can be found in $O(1)$ iterations, each iteration requiring $O(m)$ operations. Therefore the iterative part of this algorithm needs $O(m^2)$ operations.

Finding roots of this particular secular equation can be simplified by utilizing the fact that its roots are bounded by the eigenvalues d_i of the matrix $\mathbf{D}$. Indeed if we denote the roots as $\lambda_1, \lambda_2, \dots, \lambda_n$ and assume that $\lambda_i \le \lambda_{i+1}$ and $d_i \le d_{i+1}$, it can be shown that

1. if $\sigma \ge 0$,

$$d_i \le \lambda_i \le d_{i+1} , \qquad i = 1, \dots, n-1 , \tag{9.21}$$

$$d_n \le \lambda_n \le d_n + \sigma \mathbf{u}^T \mathbf{u} ; \tag{9.22}$$

[1] Multiplying this equation by $\prod_{i=1}^{m} (d_i - \lambda)$ leads to an equivalent polynomial equation of the order m, which has exactly m roots.

2. if $\sigma \le 0$,

$$d_{i-1} \le \lambda_i \le d_i \,, \qquad i = 2, \dots, n \,, \tag{9.23}$$
$$d_1 + \sigma \mathbf{u}^T \mathbf{u} \le \lambda_1 \le d_1 \,. \tag{9.24}$$

9.3.2 Symmetric row/column update

The matrix $\mathbf{A}$ to be diagonalized is given in the form

$$\mathbf{A} = \mathbf{D} + \mathbf{e}(p)\mathbf{u}^T + \mathbf{u}\mathbf{e}(p)^T = \begin{bmatrix} d_1 & \dots & u_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ u_1 & \dots & d_p & \dots & u_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & u_n & \dots & d_n \end{bmatrix} , \tag{9.25}$$

where $\mathbf{D}$ is a diagonal matrix with diagonal elements $\{d_i | i = 1, \dots, n\}$, $\mathbf{e}(p)$ is the unit vector in the p-direction, and $\mathbf{u}$ is a given update vector where the p-th element can be assumed to equal zero, $u_p = 0$, without loss of generality. Indeed, if the element is not zero, one can simply redefine $d_p \to d_p + 2u_p$, $u_p \to 0$.

The eigenvalue equation for matrix $\mathbf{A}$ is given as

$$(D - \lambda)\mathbf{x} + \mathbf{e}(p)\mathbf{u}^T\mathbf{x} + \mathbf{u}\mathbf{e}(p)^T\mathbf{x} = 0 \,, \tag{9.26}$$

where $\mathbf{x}$ is an eigenvector and λ is the corresponding eigenvalue. The component number p of this vector-equation reads

$$(d_p - \lambda)x_p + \mathbf{u}^T\mathbf{x} = 0 \,, \tag{9.27}$$

while the component number $k \ne p$ reads

$$(d_k - \lambda)x_k + u_k x_p = 0 \,, \tag{9.28}$$

Dividing the last equation by $(d_k - \lambda)$, multiplying from the left with $\sum_{k=1}^{n} u_k$, substituting $\mathbf{u}^T\mathbf{x}$ using equation (9.27) and dividing by x_p gives the secular equation,

$$-(d_p - \lambda) + \sum_{k \ne p}^{n} \frac{u_k^2}{d_k - \lambda} = 0 \,, \tag{9.29}$$

which determines the updated eigenvalues.

$$\tag{9.30}$$

9.3.3 Symmetric rank-2 update

A symmetric rank-2 update can be represented as two consecutive rank-1 updates,

$$\mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T = \mathbf{a}\mathbf{a}^T - \mathbf{b}\mathbf{b}^T , \tag{9.31}$$

where

$$\mathbf{a} = \frac{1}{\sqrt{2}}(\mathbf{u} + \mathbf{v}) , \; \mathbf{b} = \frac{1}{\sqrt{2}}(\mathbf{u} - \mathbf{v}) . \tag{9.32}$$

The eigenvalues can then be found by applying the rank-1 update method twice.

Table 9.1: Jacobi diagonalization in C. The header `matrix.h` is supposed to define the self-explanatory functions `vector_get`, `vector_set`, `matrix_get`, `matrix_set`, and `matrix_set_identity`.

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

int jacobi(matrix* A, vector* e, matrix* V){
// Jacobi diagonalization.
// Upper triangle of A is destroyed.
// e and V accumulate eigenvalues and eigenvectors
int changed, rotations=0, n=A->size1;
for(int i=0;i<n;i++)vector_set(e,i,matrix_get(A,i,i));
matrix_set_identity(V);
do{
  changed=0; int p,q;
  for(p=0;p<n;p++)for(q=p+1;q<n;q++){
    double app=vector_get(e,p);
    double aqq=vector_get(e,q);
    double apq=matrix_get(A,p,q);
    double phi=0.5*atan2(2*apq,aqq-app);
    double c = cos(phi);
    double s = sin(phi);
    double app1=c*c*app-2*s*c*apq+s*s*aqq;
    double aqq1=s*s*app+2*s*c*apq+c*c*aqq;
    if(app1!=app || aqq1!=aqq){
      changed=1; rotations++;
      vector_set(e,p,app1);
      vector_set(e,q,aqq1);
      matrix_set(A,p,q,0.0);
      for(int i=0;i<p;i++){
        double aip=matrix_get(A,i,p);
        double aiq=matrix_get(A,i,q);
        matrix_set(A,i,p,c*aip-s*aiq);
        matrix_set(A,i,q,c*aiq+s*aip);
        }
      for(int i=p+1;i<q;i++){
        double api=matrix_get(A,p,i);
        double aiq=matrix_get(A,i,q);
        matrix_set(A,p,i,c*api-s*aiq);
        matrix_set(A,i,q,c*aiq+s*api);
        }
      for(int i=q+1;i<n;i++){
        double api=matrix_get(A,p,i);
        double aqi=matrix_get(A,q,i);
        matrix_set(A,p,i,c*api-s*aqi);
        matrix_set(A,q,i,c*aqi+s*api);
        }
      for(int i=0;i<n;i++){
        double vip=matrix_get(V,i,p);
        double viq=matrix_get(V,i,q);
        matrix_set(V,i,p,c*vip-s*viq);
        matrix_set(V,i,q,c*viq+s*vip);
        }
      }
    }
  }while(changed!=0);
return rotations;
}
```

Chapter 10

Power iteration methods and Krylov subspaces

10.1 Introduction

Power method is an iterative method to calculate an eigenvalue and the corresponding eigenvector of a matrix A using the *power iteration*

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i \; . \tag{10.1}$$

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

$$\lambda[\mathbf{x}_i] = \frac{\mathbf{x}_i^T \mathbf{A}\mathbf{x}_i}{\mathbf{x}_i^T \mathbf{x}_i} = \frac{\mathbf{x}_{i+1}^T \mathbf{x}_i}{\mathbf{x}_i^T \mathbf{x}_i}. \tag{10.2}$$

The *inverse power iteration* with the inverse matrix,

$$\mathbf{x}_{i+1} = \mathbf{A}^{-1}\mathbf{x}_i \; , \tag{10.3}$$

converges to the smallest eigenvalue of matrix A.

Alternatively, the *shifted inverse iteration*,

$$\mathbf{x}_{i+1} = (\mathbf{A} - s\mathbf{1})^{-1}\mathbf{x}_i \; , \tag{10.4}$$

where $\mathbf{1}$ signifies a unity matrix of the same size as $\mathbf{A}$, converges to the eigenvalue closest to the given number s.

The *inverse iteration method* is a refinement of the inverse power method where the trick is not to invert the matrix in (10.4) but rather solve the linear system

$$(\mathbf{A} - s\mathbf{1})\mathbf{x}_{i+1} = \mathbf{x}_i \tag{10.5}$$

using e.g. QR-decomposition.

The better approximation s to the sought eigenvalue is chosen, the faster convergence one gets. However, incorrect choice of s can lead to slow convergence or to the convergence of a different eigenvector. Usually in practice the method is used when good approximation for the eigenvalue is known, and hence one needs only few (quite often just one) iteration.

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 $(\mathbf{A}-s\mathbf{1})$. The optimal strategy is probably an update after several iterations.

10.2 Krylov subspaces

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

The set of vectors

$$\mathcal{K}_n = \left\{\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \dots, \mathbf{A}^{n-1}\mathbf{b}\right\} , \tag{10.6}$$

where $n < \mathrm{rank}(\mathbf{A})$, 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 $\mathbf{A}^{n-1}\mathbf{b}$ 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.

10.3 Arnoldi iteration

Arnoldi iteration is an algorithm where the order-n orthogonalised Krylov matrix $\mathbf{Q}_n$ for a given matrix $\mathbf{A}$ is built using stabilised Gram-Schmidt process:

```
start with a set Q = {q^1} consisting of one random normalised vector q^1;
repeat for k = 2 to n :
   make a new vector q^k = A q^{k-1}
   orthogonalise q^k to all vectors q^i ∈ Q storing q^i† q^k → h_{i,k-1}
   normalise q^k storing ||q^k|| → h_{k,k-1}
   add q^k to the set Q
```

By construction the matrix $\mathbf{H}_n$ made of the elements h_{jk} is an upper Hessenberg matrix,

$$\mathbf{H}_n = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,n} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,n} \\ 0 & h_{3,2} & h_{3,3} & \cdots & h_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n,n-1} & h_{n,n} \end{bmatrix}, \tag{10.7}$$

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

$$\mathbf{H}_n = \mathbf{Q}_n^\dagger \mathbf{A} \mathbf{Q}_n \,. \tag{10.8}$$

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

Since $\mathbf{H}_n$ 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.

10.4 Lanczos iteration

Lanczos iteration is Arnoldi iteration for Hermitian matrices, in which case the Hessenberg matrix $\mathbf{H}_n$ of Arnoldi method becomes a tridiagonal matrix $\mathbf{T}_n$.

The Lanczos algorithm thus reduces the original hermitian $N \times N$ matrix $\mathbf{A}$ into a smaller $n \times n$ tridiagonal matrix $\mathbf{T}_n$ 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.

10.5 Generalised minimum residual (GMRES)

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

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,, \tag{10.9}$$

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

$$\mathbf{x} \approx \mathbf{x}_n \leftarrow \min_{\mathbf{x}\in\mathcal{K}_n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|\,. \tag{10.10}$$

The vector $\mathbf{x}_n \in \mathcal{K}_n$ can be represented as $\mathbf{x}_n = \mathbf{Q}_n\mathbf{y_n}$ where $\mathbf{Q}_n$ is the projector on the space $\mathcal{K}_n$ and $\mathbf{y_n}$ is an n-dimensional vector. Substituting this into (10.9) gives

$$\mathbf{A}\mathbf{Q}_n\mathbf{y_n} = \mathbf{b}\,, \tag{10.11}$$

which can be solved by least-squares method.

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

FFT 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 subsets 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 $x_{n_1 n_2}$, $n_1 = 1 \dots N_1$, $n_2 = 1 \dots N_2$ has the discrete Fourier transform

$$c_{k_1 k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1 n_2} e^{-2\pi i \frac{n_1 k_1}{N_1}} e^{-2\pi i \frac{n_2 k_2}{N_2}} . \tag{11.5}$$

11.4 C implementation

Bibliography

[1] Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of Assoc. for Comp. Mach.*, 17(4):589–602, 1970.

[2] Przemyslaw Bogacki and Lawrence F. Shampine. A 3(2) pair of Runge–Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989.

[3] C.W. Clenshaw and A.R. Curtis. A method for numerical integration on an automatic computer. *Numerische Mathematik*, 2:197–205, 1960.

[4] M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 3rd edition, 2009.

[5] Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems*. NASA Technical Report, 1969.

[6] Wallace Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *J. SIAM*, 6(1):26–50, 1958.

[7] Gene H. Golub. Some modified matrix eigenvalue problems. *SIAM Rev.*, 15(2):318–334, 1973.

[8] Gene H. Golub and John H. Welsch. Calculation of Gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230, 1969.

[9] Pedro Gonnet. Increasing the reliability of adaptive quadrature using explici interpolants. *ACM Trans. Math. Soft.*, 37(3):26:2–26:32, 2010.

[10] A.S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM*, 5(4):339–342, 1958.

[11] Aleksandr Semenovich Kronrod. *Nodes and weights of quadrature formulas. Sixteen-place tables*. Consultants Bureau, 1965.

[12] Masatake Mori. Quadrature formulas obtained by variable transformation and the DE–rule. *Journal of Computational and Applied Mathematics*, 12&13:119–130, 1985.

[13] T. N. L. Patterson. The optimum addition of points to quadrature formulae. *Mathematics of Computation*, 22(104):847–856, 1968.

[14] Robert Piessens, Elise de Doncker-Kapenga, Christoph W. Überhuber, and David Kahaner. *QUADPACK: A subroutine package for automatic integration*. Springer-Verlag, 1983.

[15] B. Riemann. Über die darstellbarkeit einer function durch eine trigonometrische reihe. *Abhandlungen der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, 13:87–132, 1868.

Index