Important Matrix and vector handling packages

The Numerical Recipes codes have been rewritten in Fortran 90/95 and C/C++ by us. The original source codes are taken from the widely used software package LAPACK, which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK.

- LINPACK: package for linear equations and least square problems.
- LAPACK: package for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK’s website http://netlib.org it is possible to download for free all source codes from this library. Both C/C++ and Fortran versions are available.
- BLAS (I, II and III): (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Blas I is vector operations, II vector-matrix operations and III matrix-matrix operations. Highly parallelized and efficient codes, all available for download from http://netlib.org.

Basic Matrix Features

Matrix properties reminder

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\quad I = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The inverse of a matrix is defined by

\[
A^{-1} \cdot A = I
\]

Basic Matrix Features

Matrix Properties Reminder

<table>
<thead>
<tr>
<th>Matrix Type</th>
<th>Name</th>
<th>Matrix Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A = A^T)</td>
<td>symmetric</td>
<td>(a_{ij} = a_{ji})</td>
</tr>
<tr>
<td>(A = A^T)</td>
<td>real orthogonal</td>
<td>(\sum_i a_{ik}a_{kj} = \sum_i a_{ki}a_{kj} = I_j)</td>
</tr>
<tr>
<td>(A = A^T)</td>
<td>real matrix</td>
<td>(a_{ij} = a_{ji})</td>
</tr>
<tr>
<td>(A = A^T)</td>
<td>hermitian</td>
<td>(a_{ij} = a_{ji})</td>
</tr>
<tr>
<td>(A = A^T)</td>
<td>unitary</td>
<td>(\sum_i a_{ik}a_{jk} = \sum_i a^*<em>k a</em>{kj} = I_j)</td>
</tr>
</tbody>
</table>

Some famous Matrices

- Diagonal if \(a_{ij} = 0\) for \(i \neq j\)
- Upper triangular if \(a_{ij} = 0\) for \(i > j\)
- Lower triangular if \(a_{ij} = 0\) for \(i < j\)
- Upper Hessenberg if \(a_{ij} = 0\) for \(i > j + 1\)
- Lower Hessenberg if \(a_{ij} = 0\) for \(i < j + 1\)
- Tridiagonal if \(a_{ij} = 0\) for \(|i - j| > 1\)
- Lower banded with bandwidth \(p\): \(a_{ij} = 0\) for \(i > j + p\)
- Upper banded with bandwidth \(p\): \(a_{ij} = 0\) for \(i < j + p\)
- Banded, block upper triangular, block lower triangular...
Basic Matrix Features

Some Equivalent Statements
For an \( N \times N \) matrix \( A \) the following properties are all equivalent:
- The inverse of \( A \) exists, \( A \) is nonsingular.
- The equation \( Ax = 0 \) implies \( x = 0 \).
- The rows of \( A \) form a basis of \( \mathbb{R}^N \).
- The columns of \( A \) form a basis of \( \mathbb{R}^N \).
- \( A \) is a product of elementary matrices.
- 0 is not eigenvalue of \( A \).

Important Mathematical Operations

The basic matrix operations that we will deal with are addition and subtraction:
\[ A - B \pm C =⇒ a_{ij} = b_{ij} \pm c_{ij}, \] (1)
scalar-matrix multiplication
\[ A = \gamma B =⇒ a_{ij} = \gamma b_{ij}, \] (2)
vector-matrix multiplication
\[ y = AX =⇒ y_i = \sum_{j=1}^{n} a_{ij} x_j, \] (3)
matrix-matrix multiplication
\[ A = BC =⇒ a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \] (4)
and transposition
\[ A = BT =⇒ a_{ij} = b_{ji}, \] (5)

Important Mathematical Operations

Similarly, important vector operations that we will deal with are addition and subtraction:
\[ x = y \pm z =⇒ x_i = y_i \pm z_i, \] (6)
scalar-vector multiplication
\[ x = \gamma y =⇒ x_i = \gamma y_i, \] (7)
vector-vector multiplication (called Hadamard multiplication)
\[ x = yz =⇒ x_i = y_i z_i, \] (8)
the inner or so-called dot product resulting in a constant
\[ x = y^T z =⇒ x = \sum_{j=1}^{n} y_j z_j, \] (9)
and the outer product, which yields a matrix,
\[ A = yz^T =⇒ a_{ij} = y_i z_j, \] (10)

Matrix Handling in C/C++, Static and Dynamical allocation

Static
We have an \( N \times N \) matrix \( A \) with \( N = 100 \) in C/C++:
\[ \text{int N = 100;} \]
\[ \text{double A[100][100];} \]
// initialize all elements to zero
\[ \text{for}\(i=0; i < N; i++) {\} \]
\[ \text{for}\(j=0; j < N; j++) {\} \]
\[ \text{A[i][j] = 0.0;} \]
Note the way the matrix is organized, row-major order.
Matrix Handling in C/C++

Row Major Order, Addition

We have $N \times N$ matrices $A$, $B$ and $C$ and we wish to evaluate $A = B + C$.

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

In C/C++ this would be coded like

```c
for(i=0 ; i < N ; i++) {
    for(j=0 ; j < N ; j++) {
        a[i][j] = b[i][j] + c[i][j];
    }
}
```

Matrix Handling in C/C++

Row Major Order, Multiplication

We have $N \times N$ matrices $A$, $B$ and $C$ and we wish to evaluate $A = BC$.

$$A = BC \implies a_{ij} = \sum_{k=1}^{N} b_{ik} c_{kj},$$

In C/C++ this would be coded like

```c
for(i=0 ; i < N ; i++) {
    for(j=0 ; j < N ; j++) {
        for(k=0 ; k < N ; k++) {
            a[i][j]+=b[i][k]*c[k][j];
        }
    }
}
```

Matrix Handling in Fortran 90/95

Column Major Order

```fortran
ALLOCATE (a(N,N), b(N,N), c(N,N))
DO j=1, N
    DO i=1, N
        a(i,j)=b(i,j)+c(i,j)
    ENDDO
ENDDO
...
DEALLOCATE(a,b,c)
```

Fortran 90 writes the above statements in a much simpler way

```fortran
a=b+c
```

Multiplication

```fortran
a=MATMUL(b,c)
```

Fortran contains also the intrinsic functions TRANSPOSE and CONJUGATE.

Matrix Handling in C/C++, Dynamic Allocation

```
Do it yourself
```

```c
int N;
double ** A;
A = new double*[N]
for ( i = 0; i < N; i++)
    A[i] = new double[N];
```

Always free space when you don’t need an array anymore.

```c
for ( i = 0; i < N; i++)
    delete[] A[i];
delete[] A;
```

Armadillo, recommended!!

```
Armadillo is a C++ linear algebra library (matrix maths)
aiming towards a good balance between speed and ease of use.
The syntax is deliberately similar to Matlab.
Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions.
Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries).
A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries. This is accomplished through recursive templates and template meta-programming.
Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.
The library is open-source software, and is distributed under a license that is useful in both open-source and...
Armadillo, simple examples

#include <iostream>
#include <armadillo>
using namespace std;
using namespace arma;

int main(int argc, char** argv)
{
    mat A = randu<mat>(5,5);
    mat B = randu<mat>(5,5);
    cout << A*B << endl;
    return 0;
}

Armadillo, simple examples

#include <iostream>
#include <armadillo>
using namespace arma;
using namespace std;

int main(int argc, char** argv)
{
    // directly specify the matrix size (elements are uninitialised)
    mat A(2,3);
    // .n_rows = number of rows (read only)
    // .n_cols = number of columns (read only)
    cout << "A.n_rows = " << A.n_rows << endl;
    cout << "A.n_cols = " << A.n_cols << endl;
    // directly access an element (indexing starts at 0)
    A(1,2) = 456.0;
    A.print("A: ");
    // scalars are treated as a 1x1 matrix,
    // hence the code below will set A to have a size of 1x1
    A = 5.0;
    A.print("A: ");
    // the .fill() member function can be used
    A.set_size(3,3);
    A.fill(5.0); A.print("A: ");

    mat B;
    // endr indicates "end of row"
    B << 0.555950 << 0.274690 << 0.540605 << 0.798938 << endr
    << 0.108929 << 0.830123 << 0.891726 << 0.895283 << endr
    << 0.948014 << 0.973234 << 0.216504 << 0.883152 << endr
    << 0.023787 << 0.675382 << 0.231751 << 0.450332 << endr;
    // print to the cout stream
    // with an optional string before the contents of the matrix
    B.print("B: ");
    // the << operator can also be used to print the matrix
    // to an arbitrary stream (cout in this case)
    cout << "B: " << endl << B << endl;
    // save to disk
    B.save("B.txt", raw_ascii);
    // load from disk
    mat C;
    C.load("B.txt");
    C += 2.0 * B;
    C.print("C: ");
    // submatrix types:
    //
    // .submat(first_row, first_column, last_row, last_column)
    // .row(row_number)
    // .col(column_number)
    // .cols(first_column, last_column)
    // .rows(first_row, last_row)
    cout << "C.submat(0,0,3,1) = " << endl;
    cout << C.submat(0,0,3,1) << endl;
    // generate the identity matrix
    mat D = eye<mat>(4,4);
    D.submat(0,0,3,1) = C.cols(1,2);
    D.print("D: ");
    // transpose
    cout << "trans(B) = " << endl;
    cout << trans(B) << endl;
    // maximum from each row (traverse along columns)
    cout << "max(B,1) = " << endl;
    cout << max(B,1) << endl;
    // maximum value in B
    cout << "max(max(B)) = " << max(max(B)) << endl;
    // sum of each column (traverse along rows)
    cout << "sum(B) = " << endl;
    cout << sum(B) << endl;
    // sum of each row (traverse along columns)
    cout << "sum(B,1) = " << endl;
    cout << sum(B,1) << endl;
    // sum of all elements
    cout << "accu(B) = " << accu(B) << endl;
    // trace = sum along diagonal
    cout << "trace(B) = " << trace(B) << endl;
    // random matrix -- values are uniformly distributed in the [0,1] interval
    mat E = randu<mat>(4,4);
    E.print("E: ");

Armadillo, how to compile and install

For people using Ubuntu, Debian, Linux Mint, simply go to the
synaptic package manager and install armadillo from there. You
may have to install Lapack as well. For Mac and Windows users,
follow the instructions from the webpage
http://arma.sourceforge.net. To compile, use for example
(linux/ubuntu)
c++ -O2 -o program.x program.cpp -larmadillo -llapack -lblas
where the -l option indicates the library you wish to link to.
For OS X users you may have to declare the paths to the include
files and the libraries as
c++ -O2 -o program.x program.cpp -L/usr/local/lib -I/usr/local/include

Armadillo, simple examples

#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
{
    // maximum from each row (traverse along columns)
    cost << "max(B,1) = " << endl;
    cost << max(B,1) << endl;
    // maximum value in B
    cost << "max(max(B)) = " << max(max(B)) << endl;
    // sum of each column (traverse along rows)
    cost << "sum(B) = " << endl;
    cost << sum(B) << endl;
    // sum of each row (traverse along columns)
    cost << "sum(B,1) = " << endl;
    cost << sum(B,1) << endl;
    // sum of all elements
    cost << "accu(B) = " << accu(B) << endl;
    // trace = sum along diagonal
    cost << "trace(B) = " << trace(B) << endl;
    // random matrix -- values are uniformly distributed in the [0,1] interval
    mat E = randu<mat>(4,4);
    E.print("E: ");

Armadillo, simple examples

#include <iostream>
#include <armadillo>
using namespace std;
using namespace arma;

int main(int argc, char** argv)
{
    mat A = randu<mat>(5,5);
    mat B = randu<mat>(5,5);
    cout << A*B << endl;
    return 0;
}
We can solve this system of equations recursively starting from the matrix elements along the diagonal. We assume also that the matrix is non-singular and that the equations of the form \( a_{ij} x_j = w_i \). With \( n = 4 \) such eliminations we obtain a so-called upper triangular set of equations:

\[
\begin{bmatrix}
a_{11} x_1 + a_{12} x_2 + a_{13} x_3 + a_{14} x_4 = w_1 \\
a_{21} x_1 + a_{22} x_2 + a_{23} x_3 + a_{24} x_4 = w_2 \\
a_{31} x_1 + a_{32} x_2 + a_{33} x_3 + a_{34} x_4 = w_3 \\
a_{41} x_1 + a_{42} x_2 + a_{43} x_3 + a_{44} x_4 = w_4
\end{bmatrix}
\]

The basic idea of Gaussian elimination is to use the first equation to eliminate the first unknown \( x_1 \) from the remaining \( n - 1 \) equations. Then we use the new second equation to eliminate the second unknown \( x_2 \) from the remaining \( n - 2 \) equations. With \( n - 1 \) such eliminations we obtain a so-called upper triangular set of equations of the form

\[
\begin{bmatrix}
a_{11} x_1 + a_{12} x_2 + a_{13} x_3 + a_{14} x_4 = w_1 \\
a_{21} x_1 + a_{22} x_2 + a_{23} x_3 + a_{24} x_4 = w_2 \\
a_{31} x_1 + a_{32} x_2 + a_{33} x_3 + a_{34} x_4 = w_3 \\
a_{41} x_1 + a_{42} x_2 + a_{43} x_3 + a_{44} x_4 = w_4
\end{bmatrix}
\]

We can solve this system of equations recursively starting from \( x_4 \) (in our case \( x_4 \)) and proceed with what is called a backward substitution.
Gaussian Elimination

This process can be expressed mathematically as

\[ x_m = \frac{1}{a_{mm}} \left( y_m - \sum_{k=m+1}^{n} a_{mk} x_k \right) \quad m = n-1, n-2, \ldots, 1. \]  

(11)

To arrive at such an upper triangular system of equations, we start by eliminating the unknown \( x_j \) for \( j = 2, n \). We achieve this by multiplying the first equation by \( a_{j1}/a_{11} \) and then subtract the result from the \( j \)th equation. We assume obviously that \( a_{11} \neq 0 \) and that \( A \) is not singular.

Gaussian Elimination

The new coefficients are

\[ b_{2k} = a_{1k}^{(1)} \quad k = 1, \ldots, n, \]  

(13)

where each \( a_{1k}^{(1)} \) is equal to the original \( a_{1k} \) element. The other coefficients are

\[ d_{jk}^{(2)} = d_{jk}^{(1)} - \frac{1}{a_{11}^{(1)}} a_{1k}^{(1)} a_{jk} \quad j, k = 2, \ldots, n, \]  

(14)

with a new right-hand side given by

\[ y_{1}^{(2)} = y_1^{(1)} \quad y_{j}^{(2)} = y_j^{(1)} - \frac{1}{a_{11}^{(1)}} a_{1k}^{(1)} y_k \quad j = 2, \ldots, n. \]  

(15)

We have also set \( y_1^{(2)} = y_1 \), the original vector element. We see that the system of unknowns \( x_1, \ldots, x_n \) is transformed into an \((n-1) \times (n-1)\) problem.

Gaussian Elimination

This step is called forward substitution. Proceeding with these substitutions, we obtain the general expressions for the new coefficients

\[ y_{m+1}^{(m+1)} = y_{m+1}^{(m)} - \frac{a_{mm}^{(m)}}{a_{mm}^{(m-1)}} y_{m}^{(m)} \quad j = m, m+1, \ldots, n. \]  

(16)

with \( m = 1, \ldots, n-1 \) and a right-hand side given by

\[ y_{m+1}^{(m+1)} = y_{m+1}^{(m)} - \frac{a_{mm}^{(m)}}{a_{mm}^{(m-1)}} y_{m}^{(m)}. \]  

(17)

This set of \( n-1 \) eliminations leads us to an equations which is solved by back substitution. If the arithmetic is exact and the matrix \( A \) is not singular, then the computed answer will be exact.

Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may

Gaussian Elimination and Tridiagonal matrices, project 1

Suppose we want to solve the following boundary value equation

\[ -u''(x) = f(x, u(x)) \]

with \( x \in (a, b) \) and with boundary conditions \( u(a) = u(b) = 0 \). We assume that \( f \) is a continuous function in the domain \( x \in (a, b) \). Since, except the few cases where it is possible to find analytic solutions, we will seek after approximate solutions, we choose to represent the approximation to the second derivative from the previous chapter

\[ f'' = \frac{6u'' - 2u'' + u''}{h^2} + O(h^2). \]

We subdivide our interval \( x \in (a, b) \) into \( n \) subintervals by setting \( x_i = a + ih \), with \( i = 0, 1, \ldots, n+1 \). The step size is then given by \( h = (b-a)/(n+1) \) with \( n \in \mathbb{N} \). For the internal grid points \( i = 1, 2, \ldots, n \) we replace the differential operator with the above formula resulting in

\[ u_i = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & \cdots & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & \cdots & \cdots & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \cdots \\ \vdots & \vdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} + \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ \vdots \\ \vdots \\ f(x_{n+1}) \end{bmatrix}. \]

(12)

We can rewrite our original differential equation in terms of a discretized equation with approximations to the derivatives as

\[ -u_{i+1} - 2u_i + u_{i-1} = f(x_i, u(x_i)) \]

with \( i = 1, 2, \ldots, n \). We need to add to this system the two boundary conditions \( u(a) = u_0 \) and \( u(b) = u_{n+1} \). If we define a matrix

\[ A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & \cdots & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & \cdots & \cdots & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \cdots \\ \vdots & \vdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & \cdots & \cdots & 1 \end{bmatrix} \]

and the corresponding vectors \( u = \begin{bmatrix} u_1, u_2, \ldots, u_n \end{bmatrix}^T \) and \( f(u) = \begin{bmatrix} f(x_1), f(x_2), \ldots, f(x_{n+1}) \end{bmatrix}^T \) we can rewrite the differential equation including the boundary conditions as a system
We start with the linear set of equations

\[ Au = f, \]

where \( A \) is a tridiagonal matrix which we rewrite as

\[
A = \begin{bmatrix}
  b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\
  a_2 & b_2 & c_2 & & \cdots & \\
  a_3 & b_3 & c_3 & \cdots & \cdots & \\
  \vdots & \vdots & \vdots & \ddots & \cdots & \\
  a_{n-1} & b_{n-1} & c_{n-1} & & a_n & \\
  a_n & b_n & \end{bmatrix}
\]

where \( a, b, c \) are one-dimensional arrays of length \( 1 \ldots n \). In project 1
the arrays \( a \) and \( c \) are equal, namely \( a_i = c_i = -1/h^2 \). The matrix
is also positive definite.

A tridiagonal matrix is a special form of banded matrix where all
the elements are zero except for those on and immediately above
and below the leading diagonal. The above tridiagonal system can
be written as

\[
a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i
\]

for \( i = 1, 2, \ldots, n \). We see that \( u_{i-1} \) and \( u_{i+1} \) are not required and
we can set \( a_i - c_i = 0 \). In many applications the matrix is
symmetric and we have \( a_i - c_i \). The algorithm for solving this set
of equations is rather simple and requires two steps only, a forward
substitution and a backward substitution. These steps are also
common to the algorithms based on Gaussian elimination that we
discussed previously. However, due to its simplicity, the number of
floating point operations is in this case proportional with \( O(n) \)
while Gaussian elimination requires \( 2n^3/3 + O(n^2) \) floating point
operations.

The backward substitution gives then the final solution

\[
u_{i-1} = \frac{f_i - c_{i-1} u_i}{b_{i-1}}
\]

with \( u_n = \tilde{f}_n / \tilde{b}_n \) when \( i = n \), the last point.

In case your system of equations leads to a tridiagonal matrix, it is
clearly an overkill to employ Gaussian elimination or the standard
LU decomposition.

Our algorithm starts with forward substitution with a loop over of
the elements \( i \) and gives an update of the diagonal elements \( \tilde{b}_i \)
given by the new diagonals \( \tilde{b}_i \)

\[
\tilde{b}_i = b_i - \frac{a_{i-1} f_{i-1}}{b_{i-1}}
\]

and the new righthand side \( \tilde{f}_i \) given by

\[
\tilde{f}_i = f_i - a_{i-1} \tilde{f}_{i-1}
\]

Recall that \( \tilde{b}_1 = b_1 \) and \( \tilde{f}_1 = f_1 \) always.

The matrix \( A \) which rephrases a second derivative in a discretized
form is much simpler than the general matrix

\[
A = \begin{bmatrix}
2 & -1 & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & 0 \\
0 & -1 & 2 & -1 & 0 \\
0 & 0 & 0 & 2 & -1 \\
0 & 0 & 0 & 1 & 2 \\
\end{bmatrix}
\]

This matrix fulfills the condition of a weak dominance of the
diagonal, with \( |b_1|>|c_1| \), \( |b_2|>|a_1| \) and \( |b_3|>|a_2|+|c_2| \) for
\( k = 2, 3, \ldots, n-1 \). This is a relevant but not sufficient condition
to guarantee that the matrix \( A \) yields a solution to a linear equation
problem. The matrix needs also to be irreducible. A tridiagonal
irreducible matrix means that all the elements \( a_i \) and \( c_i \) are
non-zero. If these two conditions are present, then \( A \) is nonsingular
and has a unique LU decomposition.
Project 1, hints

When setting up the algo it is useful to note that the different operations on the matrix (here as a $4 \times 4$ case with diagonals $d_i$ and off-diagonals $e_i$) give is an extremely simple algorithm, namely

$$
\begin{pmatrix}
d_1 & e_1 & 0 & 0 \\
e_1 & d_2 & e_2 & 0 \\
0 & e_2 & d_3 & e_3 \\
0 & 0 & e_3 & d_4
\end{pmatrix}
\rightarrow
\begin{pmatrix}
d_1 & e_1 & 0 & 0 \\
e_1 & d_2 & e_2 & 0 \\
0 & e_2 & d_3 & e_3 \\
0 & 0 & e_3 & d_4
\end{pmatrix}
\rightarrow
\begin{pmatrix}
d_1 & e_1 & 0 & 0 \\
e_1 & d_2 & e_2 & 0 \\
0 & e_2 & d_3 & e_3 \\
0 & 0 & e_3 & d_4
\end{pmatrix}
$$

and finally

$$
\begin{pmatrix}
d_1 & e_1 & 0 & 0 \\
e_1 & d_2 & e_2 & 0 \\
0 & e_2 & d_3 & e_3 \\
0 & 0 & e_3 & d_4
\end{pmatrix}
$$

Simple expressions for project 1

For the special matrix we can actually precalculate the updated matrix elements $\tilde{d}_i$. The non-diagonal elements $e_i$ are unchanged.

For our particular matrix in project 1 we have

$$\tilde{d}_i = 2 \cdot \frac{1}{d_{i-1}} = \frac{i + 1}{i},$$

and the new righthand side $\tilde{f}_i$ given by

$$\tilde{f}_i = f_i + \frac{1}{i} \tilde{f}_{i-1}.$$ 

Recall that $\tilde{d}_2 = 2$ and $\tilde{f}_2 = f_2$. These arrays can be set up before computing $u$. The backward substitution gives then the final solution

$$u_{i-1} = \frac{i - 1}{i} \left( \tilde{f}_{i-1} + u_i \right),$$

with $u_n = \tilde{f}_n/\tilde{d}_n$.

Linear Algebra Methods

- Gaussian elimination, $O(2/3n^3)$ flops, general matrix
- LU decomposition, upper triangular and lower tridiagonal matrices, $O(2/3n^3)$ flops, general matrix. Get easily the inverse, determinant and can solve linear equations with back-substitution only, $O(n^3)$ flops
- Cholesky decomposition. Real symmetric or hermitian positive definite matrix, $O(1/3n^3)$ flops.
- Tridiagonal linear systems, important for differential equations. Normally positive definite and non-singular. $O(n)$ flops for symmetric. Special case of banded matrices.
- Singular value decomposition
- the QR method will be discussed in chapter 7 in connection with eigenvalue systems. $O(4/3n^3)$ flops.

LU Decomposition

The LU decomposition method means that we can rewrite this matrix as the product of two matrices $L$ and $U$ where

$$
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
= 
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
$$

Program example

```c
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    ofstream ofile;
    if (argc <= 1) {
        // Read also file name on same line and max power 10^n
        cout << fileout; exit(1);
    }
    cout << " read also file name on same line and max power 10^n" << endl;
    fileout = argv[1];
    ofile.open(fileout);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    for (int i = 1; i <= exponent; i++) {
        ofile << setw(15) << setprecision(8) << log10(RelativeError) << endl;
    }
    ofile.close();
    delete [] x; delete [] d; delete [] b; delete [] solution;
    return 0;
}
```

We notice the sub-blocks which get repeated

$$
\begin{pmatrix}
d_1 & e_1 & 0 & 0 \\
e_1 & d_2 & e_2 & 0 \\
0 & e_2 & d_3 & e_3 \\
0 & 0 & e_3 & d_4
\end{pmatrix}
$$

The matrices we often end up with in rewriting for for example partial differential equations, have the feature that all leading principal submatrices are non-singular.
LU Decomposition

LU Decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

\[
\begin{align*}
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_1 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_2 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_3 \\
41x_1 + 42x_2 + 43x_3 + 44x_4 &= w_4 \\
\end{align*}
\]

The above set of equations is conveniently solved by using LU decomposition as an intermediate step.

This example shows the basis for the algorithm needed to solve the set of linear equations.

LU Decomposition, why?

There are at least three main advantages with LU decomposition compared with standard Gaussian elimination:

- It is straightforward to compute the determinant of a matrix.
- If we have to solve sets of linear equations with the same matrix but with different vectors \( w \), the number of FLOPS is of the order \( n^3 \).
- The inverse is such an operation.

LU Decomposition, linear equations

With the LU decomposition it is rather simple to solve a system of linear equations

\[
\begin{align*}
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_1 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_2 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_3 \\
41x_1 + 42x_2 + 43x_3 + 44x_4 &= w_4 \\
\end{align*}
\]

This can be written in matrix form as

\[
Ax = w,
\]

where \( A \) and \( w \) are known and we have to solve for \( x \). Using the LU decomposition we write

\[
Ax \equiv LUx = w.
\]

LU Decomposition, why?

For our four-dimensional example this takes the form

\[
\begin{align*}
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_1 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_2 \\
21x_1 + 22x_2 + 23x_3 + 24x_4 &= w_3 \\
41x_1 + 42x_2 + 43x_3 + 44x_4 &= w_4 \\
\end{align*}
\]

and

\[
\begin{align*}
01x_1 + 02x_2 + 03x_3 + 04x_4 &= y_1 \\
02x_2 + 03x_3 + 04x_4 &= y_2 \\
23x_3 + 24x_4 &= y_3 \\
44x_4 &= y_4 \\
\end{align*}
\]

This example shows the basis for the algorithm needed to solve the set of linear equations.
LU Decomposition, the inverse of a matrix

If the inverse exists then

\[ A^{-1}A = I, \]

the identity matrix. With an LU decomposed matrix we can rewrite the last equation as

\[ LUA^{-1} = I. \]

LU Decomposition, the inverse of a matrix

If we assume that the first column (that is column 1) of the inverse matrix can be written as a vector with unknown entries

\[ A^{-1}_{1} = \begin{bmatrix} a^{-1}_{11} \\ a^{-1}_{21} \\ \vdots \\ a^{-1}_{n1} \end{bmatrix}, \]

then we have a linear set of equations

\[ LU \begin{bmatrix} a^{-1}_{11} \\ a^{-1}_{21} \\ \vdots \\ a^{-1}_{n1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \]

LU Decomposition, the inverse

In a similar way we can compute the unknown entries of the second column,

\[ LU \begin{bmatrix} a^{-1}_{12} \\ a^{-1}_{22} \\ \vdots \\ a^{-1}_{n2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \]

and continue till we have solved all \( n \) sets of linear equations.

How to use the Library functions

```
void inverse(double **a, int n)
{
int i,j, *indx;
double d, *col, **y;
// allocate space in memory
indx = new int[n];
col = new double[n];
y = (double **) matrix(n, n, sizeof(double));
ludcmp(a, n, indx, &d);
// LU decompose a[][]
printf("LU form of matrix of a[][]: \\
");
for(i = 0; i < n; i++)
{
printf(" \\
");
for(j = 0; j < n; j++)
{
printf(" a[%2d][%2d] = %12.4E",i, j, a[i][j]);
}
printf("\n");
}
// find inverse of a[][] by columns
for(j = 0; j < n; j++)
{
// initialize right-side of linear equations
for(i = 0; i < n; i++)
col[i] = 0.0;
col[j] = 1.0;
lubksb(a, n, indx, col);
// save result in y[][]
for(i = 0; i < n; i++)
y[i][j] = col[i];
}
// free_matrix(void **) p, /* release local memory*/
free_matrix();
delete [] col;
delete [] indx;
}
// End: function inverse()
```
Iterative methods, Chapter 6

- Direct solvers such as Gauss elimination and LU decomposition discussed in connection with project 1.
- Iterative solvers such as Basic iterative solvers, Jacobi, Gauss-Seidel, Successive over-relaxation. These methods are easy to parallelize, as we will see later. Much used in solutions of partial differential equations.
- Other iterative methods such as Krylov subspace methods with Generalized minimum residual (GMRES) and Conjugate gradient etc will not be discussed.

Iterative methods, Jacobi’s method

We can demonstrate Jacobi’s method by this 4 × 4 matrix problem. We assume a guess for the vector elements $x^{(0)}$, a guess which represents our first iteration. The new values are obtained by substitution

$$
\begin{align*}
    x^{(1)}_1 &= \left( b_1 - a_{12} x^{(0)}_2 - a_{13} x^{(0)}_3 - a_{14} x^{(0)}_4 \right) / a_{11} \\
    x^{(1)}_2 &= \left( b_2 - a_{21} x^{(0)}_1 - a_{23} x^{(0)}_3 - a_{24} x^{(0)}_4 \right) / a_{22} \\
    x^{(1)}_3 &= \left( b_3 - a_{31} x^{(0)}_1 - a_{32} x^{(0)}_2 - a_{34} x^{(0)}_4 \right) / a_{33} \\
    x^{(1)}_4 &= \left( b_4 - a_{41} x^{(0)}_1 - a_{42} x^{(0)}_2 - a_{43} x^{(0)}_3 \right) / a_{44},
\end{align*}
$$

which after $k + 1$ iterations reads

$$
\begin{align*}
    x^{(k+1)}_1 &= \left( b_1 - a_{12} x^{(k)}_2 - a_{13} x^{(k)}_3 - a_{14} x^{(k)}_4 \right) / a_{11} \\
    x^{(k+1)}_2 &= \left( b_2 - a_{21} x^{(k)}_1 - a_{23} x^{(k)}_3 - a_{24} x^{(k)}_4 \right) / a_{22} \\
    x^{(k+1)}_3 &= \left( b_3 - a_{31} x^{(k)}_1 - a_{32} x^{(k)}_2 - a_{34} x^{(k)}_4 \right) / a_{33} \\
    x^{(k+1)}_4 &= \left( b_4 - a_{41} x^{(k)}_1 - a_{42} x^{(k)}_2 - a_{43} x^{(k)}_3 \right) / a_{44},
\end{align*}
$$

It is a simple method for solving

$$
Ax = b,
$$

where $A$ is a matrix and $x$ and $b$ are vectors. The vector $x$ is the unknown. It is an iterative scheme where we start with a guess for the unknown, and after $k + 1$ iterations we have

$$
\begin{align*}
    x^{(k+1)} &= D^{-1} (b - (L + U)x^{(k)}),
\end{align*}
$$

with $A = D + U + L$ and $D$ being a diagonal matrix, $U$ an upper triangular matrix and $L$ a lower triangular matrix. If the matrix $A$ is positive definite or diagonally dominant, one can show that this method will always converge to the exact solution.
Iterative methods, Jacobi’s method

We can generalize the above equations to

\[ x^{(k+1)}_i = \left( b_i - \sum_{j \neq i} a_{ij} x^{(k+1)}_j \right) / a_{ii} \]

or in an even more compact form as

\[ x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}) \]

with \( A = D + U + L \) and \( D \) being a diagonal matrix, \( U \) an upper triangular matrix and \( L \) a lower triangular matrix.

Iterative methods, Gauss-Seidel’s method

We can generalize the above equations to

\[ x^{(k+1)}_i = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x^{(k+1)}_j - \sum_{j=i+1}^{n} a_{ij} x^{(k)}_j \right) / a_{ii} \]

for \( i = 1, 2, \ldots, n \).

The procedure is generally continued until the changes made by an iteration are below some tolerance. The convergence properties of the Jacobi method and the Gauss-Seidel method are dependent on the matrix \( A \). These methods converge when the matrix is symmetric positive definite.

Iterative methods, Successive over-relaxation

Given a square system of \( n \) linear equations with unknown \( x \):

\[ Ax = b \]

where

\[ A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \]

The method of successive over-relaxation is an iterative technique that solves the left hand side of this expression for \( x \), using previous value for \( x \) on the right hand side. Analytically, this may be written as:

\[ x^{(k+1)} = (D + \omega L)^{-1} \left( (D - \omega U) x^{(k)} + \omega b \right) \]

However, by taking advantage of the triangular form of \( (D + \omega L) \), the elements of \( x^{(k+1)} \) can be computed sequentially using forward substitution:

\[ x_i^{(k+1)} = \left( 1 - \omega \right) x_i^{(k)} + \omega \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k+1)} \right), \quad i = 1, 2, \ldots, n \]

The choice of relaxation factor is not necessarily easy, and depends upon the properties of the coefficient matrix. For symmetric, positive-definite matrices it can be proven that \( 0 < \omega < 2 \) will lead to convergence, but we are generally interested in faster convergence rather than just convergence.
Cubic spline interpolation is among one of the most used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called cubic spline method to be described below.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations. Assume we have at our disposal some continuity relations.

Splines

The most commonly used spline function is the one with $k = 3$, the so-called cubic spline function. Assume that we have in addition to the $n + 1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n)$. By definition, the polynomials $s_{i-1}$ and $s_i$ are those supposed to interpolate the same point $i$, that is

$$s_{i-1}(x) - y_i = s_i(x),$$

with $1 \leq i \leq n - 1$. In total we have $n$ polynomials of the type

$$s_i(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3,$$

yielding $4n$ coefficients to determine.

Using the last equation we define two values for the second derivative, namely

$$s''(x)_{i-1} = \ell_i,$$

and

$$s''(x)_i = \ell_{i+1},$$

and setting up a straight line between $\ell_i$ and $\ell_{i+1}$ we have

$$s''(x)_{i-1} = \frac{\ell_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{\ell_{i+1}}{x_{i+2} - x_i}(x - x_i),$$

and integrating twice one obtains

$$s_i(x) = \frac{\ell_i}{6(x_{i+1} - x_i)}(x_{i+1} - x)^3 + \frac{\ell_{i+1}}{6(x_{i+2} - x_i)}(x - x_i)^3 + c(x - x_i) + d(x_{i+1} - x).$$

Splines

As an example, consider a spline function of degree $k = 1$ defined as follows

$$s(x) = \begin{cases} 
  y_0 + a_0(x - x_0) + b_0(x - x_0)^2, & x \in [x_0, x_1) \\
  y_1 + a_1(x - x_1) + b_1(x - x_1)^2, & x \in [x_1, x_2) \\
  \vdots \\
  y_{n-1} + a_{n-1}(x - x_{n-1}) + b_{n-1}(x - x_{n-1})^2, & x \in [x_{n-1}, x_n] \\
  y_n + a_n(x - x_n) + b_n(x - x_n)^2, & x \in [x_n, x_{n+1}] 
\end{cases}$$

In this case the polynomial consists of series of straight lines connected to each other at every endpoint. The number of continuous derivatives is then $k - 1 = 0$, as expected when we deal with straight lines. Such a polynomial is quite easy to construct given $n + 1$ points $x_0, x_1, \ldots, x_n$ and their corresponding function values.

Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i),$$

and

$$s(x_{i+1}) = y_{i+1},$$

to be fulfilled. If we also assume that $s'$ and $s''$ are continuous, then

$$s''(x_{i+1}) = s''(x_i),$$

yields $n - 1$ conditions. Similarly,

$$s''(x_i) = s''(x_{i+1}),$$

results in additional $n - 1$ conditions. In total we have $4n$ coefficients and $4n - 2$ equations to determine them, leaving us with 2 degrees of freedom to be determined.

Using the conditions $s(x_0) = y_0$ and $s(x_{n+1}) = y_{n+1}$ we can in turn determine the constants $c$ and $d$ resulting in

$$s(x) = \frac{c(x_{i+1} - x)}{x_i - x_0} \left(x_{i+1} - x\right)^3 + \frac{d(x_{i+1} - x)}{x_i - x_0} \left(x - x_i\right)^3 + \frac{y_i}{x_{i+1} - x_0} \left(x - x_i\right) + \frac{y_{i+1}}{x_{i+1} - x} \left(x - x_{i+1}\right),$$

(18)
Splines

How to determine the values of the second derivatives \( f' \) and \( f'' \)? We use the continuity assumption of the first derivatives

\[
 f_i' = f'_i(x) = \frac{y_i - y_{i-1}}{h_i} - \frac{y_{i+1} - y_i}{h_{i+1}},
\]

and set \( x = x_i \). Defining \( h_i = x_{i+1} - x_i \) we obtain finally the following expression

\[
 h_i f_{i-1} + 2 (h_i + h_{i-1} f_i + h_{i+1} f_{i+1} = \frac{6}{h_i} (y_{i+1} - y_i) - \frac{6}{h_{i-1}} (y_i - y_{i-1}),
\]

and introducing the shorthand \( u_i = 2 (h_i + h_{i-1}) \), \( v_i = \frac{6}{h_i} (y_{i+1} - y_i) - \frac{6}{h_{i-1}} (y_i - y_{i-1}), \) we can reformulate the problem as a set of linear equations to be solved through e.g.,

Gaussian elimination

\[
\begin{pmatrix}
 u_1 & h_1 & 0 & \cdots \\
 h_1 & u_2 & h_2 & 0 & \cdots \\
 0 & h_2 & u_3 & h_3 & 0 & \cdots \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & \cdots & 0 & h_{n-3} & h_{n-2} & h_{n-1} \\
 & 0 & \cdots & 0 & h_{n-2} & h_{n-1} & h_{n-1}
\end{pmatrix}
\begin{pmatrix}
 f_0 \\
 f_1 \\
 f_2 \\
 \vdots \\
 f_{n-3} \\
 f_{n-2} \\
 f_{n-1}
\end{pmatrix}
= \begin{pmatrix}
 v_0 \\
 y_1 \\
 0 \\
 \vdots \\
 v_{n-2} \\
 v_{n-1}
\end{pmatrix}
\]

Note that this is a set of tridiagonal equations and can be solved through only \( O(n) \) operations.

Splines

The functions supplied in the program library are \texttt{spline} and \texttt{splint}.

In order to use cubic spline interpolation you need first to call

\[
\texttt{spline(double x[], double y[], int n, double yp1, double yp2, double y2[])}
\]

This function takes as input \( y[0,.., n - 1] \) and \( y[0,.., n - 1] \) containing a tabulation \( y_i = f(x_i) \) with \( x_0 < x_1 < \ldots < x_{n-1} \) together with the first derivatives of \( f(x) \) at \( x_0 \) and \( x_{n-1} \), respectively. Then the function returns \( y2[0,.., n - 1] \) which contains the second derivatives of \( f(x) \) at each point \( x_i \). \( n \) is the number of points. This function provides the cubic spline interpolation for all subintervals and is called only once.

Splines

Thereafter, if you wish to make various interpolations, you need to call the function

\[
\texttt{splint(double x[], double y[], int n, double x, double y2[])}
\]

which takes as input the tabulated values \( x[0,.., n - 1] \) and \( y[0,.., n - 1] \) and the output \( y2[0,.., n - 1] \) from \texttt{spline}. It returns the value \( y \) corresponding to the point \( x \).