Slides PHY 480 and PHY 905 Lectures: Linear Algebra methods

Phil Duxbury[1] Morten Hjorth-Jensen[2,3]

Department of Physics and Astronomy, Michigan State University[1]

Department of Physics, University of Oslo[2]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[3]

Spring 2016

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://www.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://www.netlib.org`.

Basic Matrix Features

Matrix properties reminder

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

Basic Matrix Features

The inverse of a matrix is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I$$

Basic Matrix Features

Matrix Properties Reminder

Relations	Name	matrix elements
$A = A^T$	symmetric	$a_{ij} = a_{ji}$
$A = \left(A^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik} a_{jk} = \sum_k a_{ki} a_{kj} = \delta_{ij}$
$A = A^*$	real matrix	$a_{ij} = a_{ij}^*$
$A = A^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$A = \left(A^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik} a_{jk}^* = \sum_k a_{ki}^* a_{kj} = \delta_{ij}$

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 $\mathbf{A}$ the following properties are all equivalent

- If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.
- The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.
- The rows of $\mathbf{A}$ form a basis of R^N.
- The columns of $\mathbf{A}$ form a basis of R^N.
- $\mathbf{A}$ is a product of elementary matrices.
- 0 is not eigenvalue of $\mathbf{A}$.

Important Mathematical Operations

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{1}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma \mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij}, \tag{2}$$

vector-matrix multiplication

Important Mathematical Operations

$$\mathbf{y} = \mathbf{Ax} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij} x_j, \tag{3}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{BC} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \tag{4}$$

and transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji} \tag{5}$$

Important Mathematical Operations

Similarly, important vector operations that we will deal with are addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i, \tag{6}$$

scalar-vector multiplication

$$\mathbf{x} = \gamma \mathbf{y} \Longrightarrow x_i = \gamma y_i, \tag{7}$$

vector-vector multiplication (called Hadamard multiplication)

Important Mathematical Operations

$$\mathbf{x} = \mathbf{yz} \Longrightarrow x_i = y_i z_i, \tag{8}$$

the inner or so-called dot product resulting in a constant

$$x = \mathbf{y}^T \mathbf{z} \Longrightarrow x = \sum_{j=1}^{n} y_j z_j, \tag{9}$$

and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{yz}^T \Longrightarrow a_{ij} = y_i z_j, \tag{10}$$

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

Static

We have an $N \times N$ matrix A with $N = 100$ In C/C++ this would be defined as

```
int N = 100;
double A[100][100];
//   initialize all elements to zero
for(i=0 ; i < N ; i++) {
   for(j=0 ; j < N ; j++) {
      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 $A = B + C$.

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij},$$

In C/C++ this would be coded like

```
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 $A = BC$.

$$\mathbf{A} = \mathbf{BC} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik}c_{kj},$$

In C/C++ this would be coded like

```
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

```
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

```
a=b+c
```

Multiplication

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

Fortran contains also the intrinsic functions TRANSPOSE and CONJUGATE.

Dynamic memory allocation in C/C++

At least three possibilities in this course

- Do it yourself
- Use the functions provided in the library package lib.cpp
- Use Armadillo http://arma.sourceforgenet (a C++ linear algebra library, discussion next two weeks, both here and at lab). !split

Matrix Handling in C/C++, Dynamic Allocation

Do it yourself

```
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.

```
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

```cpp
#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, 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

```
c++ -O2 -o program.x program.cpp  -larmadillo -llapack -lblas
```

where the `-l` option indicates the library you wish to link to.

Armadillo, simple examples

```cpp
#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:");
  // if you want a matrix with all elements set to a particular value
  // the .fill() member function can be used
  A.set_size(3,3);
  A.fill(5.0);  A.print("A:");
```

Armadillo, simple examples

```cpp
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:");
```

Armadillo, simple examples

```cpp
// 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 column (traverse along rows)
cout << "max(B) =" << endl;
cout << max(B) << endl;
```

Armadillo, simple examples

```cpp
// 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 << "sum(sum(B)) = " << sum(sum(B)) << endl;
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] in
mat E = randu<mat>(4,4);
E.print("E:");
```

Armadillo, simple examples

```
// row vectors are treated like a matrix with one row
rowvec r;
r << 0.59499 << 0.88807 << 0.88532 << 0.19968;
r.print("r:");

// column vectors are treated like a matrix with one column
colvec q;
q << 0.81114 << 0.06256 << 0.95989 << 0.73628;
q.print("q:");

// dot or inner product
cout << "as_scalar(r*q) = " << as_scalar(r*q) << endl;

  // outer product
cout << "q*r =" << endl;
cout << q*r << endl;


// sum of three matrices (no temporary matrices are created)
mat F = B + C + D;
F.print("F:");

  return 0;
```

Armadillo, simple examples

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

int main(int argc, char** argv)
  {
  cout << "Armadillo version: " << arma_version::as_string() << endl;

  mat A;

  A << 0.165300 << 0.454037 << 0.995795 << 0.124098 << 0.047084 << end
    << 0.688782 << 0.036549 << 0.552848 << 0.937664 << 0.866401 << end
    << 0.348740 << 0.479388 << 0.506228 << 0.145673 << 0.491547 << end
    << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << end
    << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << end

  A.print("A =");

  // determinant
  cout << "det(A) = " << det(A) << endl;
```

Armadillo, simple examples

```
// inverse
cout << "inv(A) = " << endl << inv(A) << endl;
double k = 1.23;

mat    B = randu<mat>(5,5);
mat    C = randu<mat>(5,5);

rowvec r = randu<rowvec>(5);
colvec q = randu<colvec>(5);


// examples of some expressions
// for which optimised implementations exist
// optimised implementation of a trinary expression
// that results in a scalar
cout << "as_scalar( r*inv(diagmat(B))*q ) = ";
cout << as_scalar( r*inv(diagmat(B))*q ) << endl;

// example of an expression which is optimised
// as a call to the dgemm() function in BLAS:
cout << "k*trans(B)*C = " << endl << k*trans(B)*C;

  return 0;
```

Gaussian Elimination

We start with the linear set of equations

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

We assume also that the matrix $\mathbf{A}$ is non-singular and that the matrix elements along the diagonal satisfy $a_{ii} \neq 0$. Simple 4×4 example

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

Gaussian Elimination

or

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

Gaussian Elimination

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

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\ b_{22}x_2 + b_{23}x_3 + b_{24}x_4 &= y_2 \\ b_{33}x_3 + b_{34}x_4 &= y_3 \\ b_{44}x_4 &= y_4. \end{aligned}$$

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

Gaussian Elimination

This process can be expressed mathematically as

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \quad m = n-1, n-2, \dots, 1. \quad (11)$$

To arrive at such an upper triangular system of equations, we start by eliminating the unknown x_1 for $j = 2, n$. We achieve this by multiplying the first equation by a_{j1}/a_{11} and then subtract the result from the jth equation. We assume obviously that $a_{11} \neq 0$ and that $\mathbf{A}$ is not singular.

Gaussian Elimination

Our actual 4×4 example reads after the first operation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y \\ w \\ w \\ w \end{pmatrix}$$

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\ a^{(2)}_{22}x_2 + a^{(2)}_{23}x_3 + a^{(2)}_{24}x_4 &= w^{(2)}_2 \\ a^{(2)}_{32}x_2 + a^{(2)}_{33}x_3 + a^{(2)}_{34}x_4 &= w^{(2)}_3 \\ a^{(2)}_{42}x_2 + a^{(2)}_{43}x_3 + a^{(2)}_{44}x_4 &= w^{(2)}_4, \end{aligned} \quad (12)$$

Gaussian Elimination

The new coefficients are

$$b_{1k} = a^{(1)}_{1k} \quad k = 1, \dots, n, \quad (13)$$

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

$$a^{(2)}_{jk} = a^{(1)}_{jk} - \frac{a^{(1)}_{j1}a^{(1)}_{1k}}{a^{(1)}_{11}} \quad j, k = 2, \dots, n, \quad (14)$$

with a new right-hand side given by

$$y_1 = w^{(1)}_1, \quad w^{(2)}_j = w^{(1)}_j - \frac{a^{(1)}_{j1}w^{(1)}_1}{a^{(1)}_{11}} \quad j = 2, \dots, n. \quad (15)$$

We have also set $w^{(1)}_1 = w_1$, the original vector element. We see that the system of unknowns $x_1, \dots, x_n$ 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

$$a^{(m+1)}_{jk} = a^{(m)}_{jk} - \frac{a^{(m)}_{jm}a^{(m)}_{mk}}{a^{(m)}_{mm}} \quad j, k = m+1, \dots, n, \quad (16)$$

with $m = 1, \dots, n-1$ and a right-hand side given by

$$w^{(m+1)}_j = w^{(m)}_j - \frac{a^{(m)}_{jm}w^{(m)}_m}{a^{(m)}_{mm}} \quad j = m+1, \dots, n. \quad (17)$$

This set of $n-1$ elimations leads us to an equations which is solved by back substitution. If the arithmetics is exact and the matrix $\mathbf{A}$ 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

$$-\frac{d^2u(x)}{dx^2} = f(x, u(x)),$$

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

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

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

Gaussian Elimination and Tridiagonal matrices, project 1

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

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} = f(x_i, u(x_i)),$$

with $i = 1, 2, \dots, n$. 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

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}$$

and the corresponding vectors $\mathbf{u} = (u_1, u_2, \dots, u_n)^T$ and $\mathbf{f}(\mathbf{u}) = f(x_1, x_2, \dots, x_n, u_1, u_2, \dots, u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system

Gaussian Elimination and Tridiagonal matrices, project 1

We start with the linear set of equations

$$\mathbf{A}\mathbf{u} = \mathbf{f},$$

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}$$

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

Gaussian Elimination and Tridiagonal matrices, project 1

We can rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.$$

Gaussian Elimination and Tridiagonal matrices, project 1

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 $i = 1, 2, \dots, n$. We see that u_{-1} and u_{n+1} are not required and we can set $a_1 = c_n = 0$. 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.

Gaussian Elimination and Tridiagonal matrices, project 1

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. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 10.

Our algorithm starts with forward substitution with a loop over of the elements i and can be expressed via the following piece of code

```
btemp = b[1];
u[1] = f[1]/btemp;
for(i=2 ; i <= n ; i++) {
   temp[i] = c[i-1]/btemp;
   btemp = b[i]-a[i]*temp[i];
   u[i] = (f[i] - a[i]*u[i-1])/btemp;
```

Gaussian Elimination and Tridiagonal matrices, project 1

Note that you should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. Finally we perform the backsubstitution leading to the following code

```
for(i=n-1 ; i >= 1 ; i--) {
   u[i] -= temp[i+1]*u[i+1];
```

Gaussian Elimination and Tridiagonal matrices, project 1

Note that our sums start with $i = 1$ and that one should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. However, a tridiagonal matrix problem is not a guarantee that we can find a solution. The matrix $\mathbf{A}$ which rephrases a second derivative in a discretized form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix},$$

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \ge |a_k| + |c_k|$ for $k = 2, 3, \dots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix $\mathbf{A}$ 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

Project 1, hints

When setting up the algo it is useful to note that the different operations on the matrix (here as a 4×4 case with diagonals d_i and off-diagonals e_i

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix} \rightarrow \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix} \rightarrow \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}$$

and finally

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & 0 & \tilde{d}_4 \end{pmatrix}$$

Project 1, hints

We notice the sub-blocks which get repeated

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & 0 & \tilde{d}_4 \end{pmatrix}$$

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. If the matrix is symmetric as well it can be rewritten as $A = LDL^T$ with D the diagonal and we have the following relations $a_{11} = d_1$, $a_{k,k-1} = e_{k-1}d_{k-1}$ for $k = 2, \dots, n$ and finally

$$a_{kk} = d_k + e_{k-1}^2 d_{k-1} = d_k + e_{k-1} a_{k,k-1}$$

for $k = 2, \dots, n$.

Program example

```
/*
**      Project1: a) and b)
**      The algorithm for solving the tridiagonal matrix
**      equation is implemented (requiering O(8n) FLOPS).
*/
# include <iostream>
# include <fstream>
# include <iomanip>
# include <cmath>

using namespace std;
ofstream ofile;

// Declaring two functions that will be used:
double Solution(double x) {return 1.0-(1-exp(-10))*x-exp(-10*x);}

double f(double x) {return 100*exp(-10*x);}

// Main program reads filename and n from command line:
int main(int argc, char* argv[]) {

    // Declaration of initial variables:
    char *outfilename;
    int n;

    // Read in output file and n,
    // abort if there are too few command-line arguments:
    if( argc <= 2 ){
```

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, $O(n^2)$ 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. $O(8n)$ 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

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

LU Decomposition

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

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

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

The matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and $\mathbf{A}$ is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = det\{\mathbf{LU}\} = det\{\mathbf{L}\}det\{\mathbf{U}\} = u_{11}u_{22}\dots u_{nn}.$$

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 $\mathbf{y}$, 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

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

This can be written in matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

where $\mathbf{A}$ and $\mathbf{w}$ are known and we have to solve for $\mathbf{x}$. Using the LU dcomposition we write

$$\mathbf{A}\mathbf{x} \equiv \mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{w}.$$

LU Decomposition, linear equations

The previous equation can be calculated in two steps

$$\mathbf{L}\mathbf{y} = \mathbf{w}; \qquad \mathbf{U}\mathbf{x} = \mathbf{y}.$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{w},$$

and since the determinat of $\mathbf{L}$ is equal to 1 (by construction since the diagonals of $\mathbf{L}$ equal 1) we can use the inverse of $\mathbf{L}$ to obtain

$$\mathbf{U}\mathbf{x} = \mathbf{L}^{-1}\mathbf{w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{L}^{-1}\mathbf{w} = \mathbf{y}$$

and as soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{U}\mathbf{x} = \mathbf{y}$.

LU Decomposition, why?

For our four-dimentional example this takes the form

$$
\begin{aligned}
y_1 &= w_1 \\
l_{21}y_1 + y_2 &= w_2 \\
l_{31}y_1 + l_{32}y_2 + y_3 &= w_3 \\
l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= w_4.
\end{aligned}
$$

and

$$
\begin{aligned}
u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= y_1 \\
u_{22}x_2 + u_{23}x_3 + u_{24}x_4 &= y_2 \\
u_{33}x_3 + u_{34}x_4 &= y_3 \\
u_{44}x_4 &= y_4
\end{aligned}
$$

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

LU Decomposition, linear equations

The algorithm goes as follows

- Set up the matrix $\mathbf{A}$ and the vector $\mathbf{w}$ with their correct dimensions. This determines the dimensionality of the unknown vector $\mathbf{x}$.
- Then LU decompose the matrix $\mathbf{A}$ through a call to the function `ludcmp(double a, int n, int indx, double &d)`. This functions returns the LU decomposed matrix $\mathbf{A}$, its determinant and the vector indx which keeps track of the number of interchanges of rows. If the determinant is zero, the solution is malconditioned.
- Thereafter you call the function `lubksb(double a, int n, int indx, double w)` which uses the LU decomposed matrix $\mathbf{A}$ and the vector $\mathbf{w}$ and returns $\mathbf{x}$ in the same place as $\mathbf{w}$. Upon exit the original content in $\mathbf{w}$ is destroyed. If you wish to keep this information, you should make a backup of it in your calling function.

LU Decomposition, the inverse of a matrix

If the inverse exists then

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I},$$

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

$$\mathbf{L}\mathbf{U}\mathbf{A}^{-1} = \mathbf{I}.$$

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

$$\mathbf{A}_1^{-1} = \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix},$$

then we have a linear set of equations

$$\mathbf{LU} \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \end{pmatrix}.$$

LU Decomposition, the inverse

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

$$\mathbf{LU} \begin{pmatrix} a_{12}^{-1} \\ a_{22}^{-1} \\ \dots \\ a_{n2}^{-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \dots \\ 0 \end{pmatrix},$$

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

How to use the Library functions

Standard C/C++: fetch the files `lib.cpp` and `lib.h`. You can make a directory where you store these files, and eventually its compiled version lib.o. The example here is program1.cpp from chapter 6 and performs the matrix inversion.

```
//  Simple matrix inversion example
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include "lib.h"

using namespace std;

/* function declarations */

void inverse(double **, int);
```

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("\n\nLU form of matrix of a[][]:\n");
  for(i = 0; i < n; i++) {
    printf("\n");
    for(j = 0; j < n; j++) {
      printf(" a[%2d][%2d] = %12.4E",i, j, a[i][j]);
```

How to use the Library functions

```
  // 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];
  }   //j-loop over columns
  // return the inverse matrix in a[][]
  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) a[i][j] = y[i][j];

  free_matrix((void **) y);     // release local memory
  delete [] col;
  delete []indx;
}  // End: function inverse()
```

How to use the Library functions

For Fortran users:

```
PROGRAM matrix
  USE constants
  USE F90library
  IMPLICIT NONE
  !      The definition of the matrix, using dynamic allocation
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: a, ainv, unity
  !      the determinant
  REAL(DP) :: d
  !      The size of the matrix
  INTEGER :: n
  ....
  !      Allocate now place in heap for a
  ALLOCATE ( a(n,n), ainv(n,n), unity(n,n) )
```

How to use the Library functions

For Fortran users:

```
    WRITE(6,*) ' The matrix before inversion'
    WRITE(6,'(3F12.6)') a
    ainv=a
    CALL matinv (ainv, n, d)
    ....
    !      get the unity matrix
    unity=MATMUL(ainv,a)
    WRITE(6,*) ' The unity matrix'
    WRITE(6,'(3F12.6)') unity
    !      deallocate all arrays
    DEALLOCATE (a, ainv, unity)
  END PROGRAM matrix
```

Using Armadillo to perform an LU decomposition

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

int main()
  {
   mat A = randu<mat>(5,5);
   vec b = randu<vec>(5);

  A.print("A =");
  b.print("b=");
  // solve Ax = b
  vec x = solve(A,b);
  // print x
  x.print("x=");
  // find LU decomp of A, if needed, P is the permutation matrix
  mat L, U;
  lu(L,U,A);
  // print l
  L.print(" L= ");
  // print U
  U.print(" U= ");
  //Check that A = LU
  (A-L*U).print("Test of LU decomposition");
    return 0;
  }
```

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 se 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

It is a simple method for solving

$$\hat{A}\mathbf{x} = \mathbf{b},$$

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

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L}+\hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D}+\hat{U}+\hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.
If the matrix $\hat{A}$ 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 demonstrate Jacobi's method by this 4×4 matrix problem. We assume a guess for the vector elements $x_i^{(0)}$, a guess which represents our first iteration. The new values are obtained by substitution

$$\begin{aligned}
x_1^{(1)} &= (b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)})/a_{11}\\
x_2^{(1)} &= (b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)})/a_{22}\\
x_3^{(1)} &= (b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)})/a_{33}\\
x_4^{(1)} &= (b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)})/a_{44},
\end{aligned}$$

which after $k+1$ iterations reads

$$\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}\\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}\\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33}\\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}$$

Iterative methods, Jacobi's method

We can generalize the above equations to

$$x_i^{(k+1)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)})/a_{ii}$$

or in an even more compact form as

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L}+\hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D}+\hat{U}+\hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

Iterative methods, Gauss-Seidel's method

Our 4×4 matrix problem

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}
$$

can be rewritten as

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}
$$

which allows us to utilize the preceding solution (forward substitution). This improves normally the convergence behavior and leads to the Gauss-Seidel method!

Iterative methods, Gauss-Seidel's method

We can generalize

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}
$$

to the following form

$$
x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1, 2, \ldots, n.
$$

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 $\hat{A}$. 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 $\mathbf{x}$:

$$
\hat{A}\mathbf{x} = \mathbf{b}
$$

where

$$
\hat{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.
$$

Iterative methods, Successive over-relaxation

Then A can be decomposed into a diagonal component D, and strictly lower and upper triangular components L and U:

$$
\hat{A} = \hat{D} + \hat{L} + \hat{U},
$$

where

$$
D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix}
$$

The system of linear equations may be rewritten as:

$$
(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}
$$

for a constant $\omega > 1$.

Iterative methods, Successive over-relaxation

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:

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

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:

$$
x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1, 2, \ldots
$$

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 Splines, Chapter 6

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 $n + 1$ points $x_0, x_1, \ldots x_n$ arranged so that $x_0 < x_1 < x_2 < \ldots x_{n-1} < x_n$ (such points are called knots). A spline function s of degree k with $n + 1$ knots is defined as follows

- On every subinterval $[x_{i-1}, x_i)$ s is a polynomial of degree $\leq k$.
- s has $k - 1$ continuous derivatives in the whole interval $[x_0, x_n]$.

Splines

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

$$s(x) = \begin{cases} s_0(x) = a_0 x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1 x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1} x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases}$$

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 $x_0, x_1, \dots x_n$ and their corresponding function values.

Splines

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

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

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

$$s_i(x) = a_{i0} + a_{i1} x + a_{i2} x^2 + a_{i2} x^3,$$

yielding $4n$ coefficients to determine.

Splines

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

$$s'_{i-1}(x_i) = s'_i(x_i),$$

yields $n - 1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i),$$

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.

Splines

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

$$s''_i(x_i) = f_i,$$

and

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

and setting up a straight line between f_i and f_{i+1} we have

$$s''_i(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i),$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1} - x_i)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1} - x_i)}(x - x_i)^3 + c(x - x_i) + d(x_{i+1} -$$

Splines

Using the conditions $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ we can in turn determine the constants c and d resulting in

$$\begin{aligned} s_i(x) = & \tfrac{f_i}{6(x_{i+1}-x_i)}(x_{i+1} - x)^3 + \tfrac{f_{i+1}}{6(x_{i+1}-x_i)}(x - x_i)^3 \\ + & \left(\tfrac{y_{i+1}}{x_{i+1}-x_i} - \tfrac{f_{i+1}(x_{i+1}-x_i)}{6}\right)(x - x_i) + \left(\tfrac{y_i}{x_{i+1}-x_i} - \tfrac{f_i(x_{i+1}-x_i)}{6}\right)(x_{i+1} - x(\end{aligned}$$

Splines

How to determine the values of the second derivatives f_i and f_{i+1}? We use the continuity assumption of the first derivatives

$$s'_{i-1}(x_i) = s'_i(x_i),$$

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

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

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

Splines

Gaussian elimination

$$
\begin{bmatrix}
u_1 & h_1 & 0 & \dots & & & & \\
h_1 & u_2 & h_2 & 0 & \dots & & & \\
0 & h_2 & u_3 & h_3 & 0 & \dots & & \\
\dots & & \dots & \dots & \dots & \dots & \dots & \\
 & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\
 & & & & & 0 & h_{n-2} & u_{n-1}
\end{bmatrix}
\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{bmatrix}
=
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}.
$$

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 *spline* and *splint*. In order to use cubic spline interpolation you need first to call

```
spline(double x[], double y[], int n, double yp1,  double yp2, double
```

This function takes as input $x[0, .., n-1]$ and $y[0, .., n-1]$ containing a tabulation $y_i = f(x_i)$ with $x_0 < x_1 < .. < x_{n-1}$ 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 $f(x_i)$ 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

```
splint(double x[], double y[], double y2a[], int n, double x, double *
```

which takes as input the tabulated values $x[0, .., n-1]$ and $y[0, .., n-1]$ and the output y2a[0,..,n - 1] from *spline*. It returns the value y corresponding to the point x.