Data analysis and Machine Learning
Lectures: Linear Algebra methods

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Jan 27, 2018

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`.

Add python material on linear algebra and array handling, text on numpy etc

Basic Matrix Features

Matrix properties reminder.

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

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 $\mathbf{A}$.

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];
```

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 both here and at lab).

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 commercial/proprietary contexts.

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, 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 -larmadillo -llapack -lbla
```

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:");
  // 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

```
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

```
// 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

```
// 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] interval
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 << endr
    << 0.688782 << 0.036549 << 0.552848 << 0.937664 << 0.866401 << endr
    << 0.348740 << 0.479388 << 0.506228 << 0.145673 << 0.491547 << endr
    << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << endr
    << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << endr;

  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

$$\mathbf{Ax} = \mathbf{w}.$$

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

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

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

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

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

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ w_2^{(2)} \\ w_3^{(2)} \\ w_4^{(2)} \end{bmatrix},$$

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= w_4^{(2)}, \end{aligned} \tag{2}$$

Gaussian Elimination

The new coefficients are

$$b_{1k} = a_{1k}^{(1)} \quad k = 1, \ldots, n, \tag{3}$$

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

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)} a_{1k}^{(1)}}{a_{11}^{(1)}} \quad j, k = 2, \ldots, n, \tag{4}$$

with a new right-hand side given by

$$y_1 = w_1^{(1)}, \quad w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)} w_1^{(1)}}{a_{11}^{(1)}} \quad j = 2, \ldots, n. \tag{5}$$

We have also set $w_1^{(1)} = w_1$, 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

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}} \quad j, k = m+1, \ldots, n, \tag{6}$$

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

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}} \quad j = m+1, \ldots, n. \tag{7}$$

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 yield losses of precision. Suppose for example that our first division in $(a_{22} - a_{21}a_{12}/a_{11})$ results in -10^{-7} and that a_{22} is one. one. We are then adding $10^7 + 1$. With single precision this results in 10^7.

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 $\mathbf{L}$ and $\mathbf{U}$ where

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

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

$$\mathbf{Ax} = \mathbf{w}.$$

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

$$\mathbf{Ax} \equiv \mathbf{LUx} = \mathbf{w}.$$

LU Decomposition, linear equations

The previous equation can be calculated in two steps

$$\mathbf{Ly} = \mathbf{w}; \qquad \mathbf{Ux} = \mathbf{y}.$$

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

$$\mathbf{LUx} = \mathbf{w},$$

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

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

which yields the intermediate step

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

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

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, 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

$$\mathbf{LUA}^{-1} = \mathbf{I}.$$

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

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

then we have a linear set of equations

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

LU Decomposition, the inverse

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

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

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

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

$$\mathbf{Ax} = \mathbf{b},$$

where $\mathbf{A}$ 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

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

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

If the matrix $\mathbf{A}$ 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

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

with $\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L}$ and $\mathbf{D}$ being a diagonal matrix, $\mathbf{U}$ an upper triangular matrix and $\mathbf{L}$ 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 $\mathbf{A}$. These methods converge when the matrix is symmetric positive-definite, or is strictly or irreducibly diagonally dominant. Both methods sometimes converge even if these conditions are not satisfied.

Iterative methods, Successive over-relaxation

Given a square system of n linear equations with unknown $\mathbf{x}$:

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

where

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

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:

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

where

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

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:

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

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:

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

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.