Technisch Wetenschappelijke Software
Scientific Software
Homework Assignment 4 – C++

Modelling pandemics - Parameter estimation

Pieter Appeltans, Thijs Steel, Emil Løvbak & Wouter Baert December 2nd, 2021

Introduction

This homework assignment deals with the theory and concepts encountered in Lectures 5 till 7, the introductory videos on C++ and Exercise sessions 6 and 7. More specifically, we will focus on the following concepts:

- source and header files;
- basic C++ syntax;
- debugging in C++;
- timings in C++;
- the C++ Standard Template Library (STL);
- functors;
- generic programming and templates;
- and lambda expressions.

As assignments 1 and 2, this homework deals with the SIQRD-model:

$$\begin{cases} \dot{S}(t) &= -\beta \frac{I(t)}{S(t)+I(t)+R(t)} S(t) + \mu R(t) \\ \dot{I}(t) &= \left(\beta \frac{S(t)}{S(t)+I(t)+R(t)} - \gamma - \delta - \alpha\right) I(t) \\ \dot{Q}(t) &= \delta I(t) - \left(\gamma + \alpha\right) Q(t) \\ \dot{R}(t) &= \gamma \left(I(t) + Q(t)\right) - \mu R(t) \\ \dot{D}(t) &= \alpha \left(I(t) + Q(t)\right). \end{cases} \tag{1}$$

More specifically, we want to estimate the parameters β, μ, γ, δ and α from daily observations of the number of individuals in each compartment. In contrast to the previous assignments, we will use C++. Therefore we first need to implement the three IVP solvers (Euler's forward method, Euler's backward method and Heun's method) from assignment 1 in C++ (Part I). Next, we can use these solvers to estimate the parameters underlying a given set of observations by minimizing the prediction error in the model parameters (Part II). Finally, in this assignment we will use the `ublas`-library. A short demo illustrating some useful functionality of the `ublas`-library is provided in appendix.

Important: Before starting with your implementation, carefully read the **complete** assignment (including the appendices).
Important: Note that the focus of this assignment lies on writing **performant** code. Keep this in mind in each part of the assignment.

Part I: Simulating the pandemic

Implement Euler's forward method, Euler's backward method and Heun's method in `C++`. In contrast to the Fortran assignments, make sure that your code can handle general systems of ordinary differential equations.

Remark: Your code should be general in the sense that it can deal with arbitrary ordinary differential equations (i.e., not only the SIQRD model). The other arguments of the interface of your IVP solvers may impose a specific type. For example, it is sufficient that your IVP solvers only accept `ublas::vector`s as initial values. However, write your code in such a way that the IVP solvers can work both in single and double precision.

Important: While writing your IVP solvers, take into account that these functions will be called numerous times with the same differential equation and the same simulation horizon, but with different parameters, in the next part of the assignment.

Verify your implementation

1. Write a file `simulation1.cpp`, in which you test your IVP solvers for the SIQRD model. Use the parameters $\beta = 0.5$, $\mu = 0$, $\gamma = 0.2$, $\alpha = 0.005$, $S_0 = 100$, $I_0 = 5$, $N = 100$, $T = 100$ days (i.e., a step size of 1 day) and

   (a) $\delta = 0$ for Euler's forward method (write the result to the file `fwe_no_measures.txt`);

   (b) $\delta = 0.2$ for Euler's backward method (write the result to the file `bwe_quarantine.txt`); and

   (c) $\delta = 0.9$ for Heun's method (write the result to the file `heun_lockdown.txt`).

   As in previous assignment, N and T should be passed as command line arguments. The other parameters may be hard coded. Recall that in order to use the file `plot.tex` from the previous assignment, your output should look as follows.

```
t_0 S_0 I_0 Q_0 R_0 D_0
t_1 S_1 I_1 Q_1 R_1 D_1
...
t_N S_N I_N Q_N R_N D_N
```

2. Write a program `simulation2.cpp` in which you test your IVP solver implementations for the following system of decoupled non-linear ordinary differential equations

$$\begin{aligned}
\dot{x}_1(t) &= -10\, x_1^3 \\
\dot{x}_2(t) &= -10\, (x_2 - 0.1)^3 \\
\dot{x}_3(t) &= -10\, (x_3 - 0.2)^3 \\
&\vdots \\
\dot{x}_{50}(t) &= -10\, (x_{50} - 4.9)^3.
\end{aligned}$$

   Question: How can you implement the right-hand side function without using explicit `for` or `while` loops?
   Question: How can you pass this differential equation to your IVP solver? Discuss all possibilities. Use three different approaches for Euler's forward method, Euler's backward method and Heun's method.

   Use the vector $\begin{bmatrix} 0.01 & 0.02 & 0.03 & \dots & 0.5 \end{bmatrix}^T$ as initial value.
   Question: How can you avoid using an explicit `for` or `while` loops to fill this vector?

Use $N = 50000$ and $T = 500$ (i.e., a step size of 1/100), which must again be passed as command line arguments. Write your results for x_1, x_{25} and x_{50} and $t = 0, 1, 2, ..., 499, 500$ to `fwe_sim2.txt`, `bwe_sim2.txt` and `heun_sim2.txt`.

Hint: What is the analytic solution of $\dot{x}(t) = -10x(t)^3$?[1] How can you use this result to compute the analytic solution for $\dot{x}(t) = -10(x(t) - 4.9)^3$

Part II: Parameter estimation from observations

In this section, we will estimate the parameters of the SIQRD model based on observations. For simplicity, we will assume that the number of individuals in each compartment is sampled once a day at a fixed hour for T days and we will denote the vector containing the observation on day i by

$$x_i = \begin{bmatrix} S_i & I_i & Q_i & R_i & D_i \end{bmatrix}^T.$$

The goal is now to find the parameters that best explains these observations. To this end, let us introduce the parameter vector p with

$$p := \begin{bmatrix} \beta & \mu & \gamma & \delta & \alpha \end{bmatrix}^T.$$

For a given set of parameters we can use the IVP solvers from the previous section to simulate the SIQRD model starting from the initial observation x_0 and a sufficiently small time step. We can then compare this simulation with the actual observations. We will quantify the difference between observation and prediction using the following scaled square error:

$$\text{LSE}(p) = \sum_{i=1}^{T} \frac{1}{T \cdot P_i^2} \|x_i - x_i^{\text{sim}}(p)\|_2^2, \tag{2}$$

with $P_i := S_i + I_i + Q_i + R_i + D_i$, the population size and $x_i^{\text{sim}}(p)$ a vector with the predicted for the number of individuals in each compartment on day i, based on our current estimate for the parameters, obtained using one of the IVP solvers.

To estimate the parameters underlying the observations, we will minimize (2) in the parameter vector p. To this end we will use the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method[2]. Algorithm 1 gives a high-level description of this method. Notice that BFGS is a quasi-Newton optimization method, which uses an approximation for the Hessian of the cost function ($\nabla^2 LSE(p)$), denoted by B. This approximation for the Hessian is iteratively updated based on the local change in the gradient. Note that the matrix B is symmetric (as the Hessian itself is symmetric).

An iteration of the BFGS method consists of the following steps. On line 3, a new search direction is computed by solving $-B_k^{-1} \nabla\text{LSE}(p_k)$ in which $\nabla\text{LSE}(p_k)$ denotes the gradient of cost function (2) with respect to p evaluated in p_k, and B_k denotes the current approximation for the Hessian. To approximate this gradient, we will use a finite difference approach, as explained in the next subsection. On line 4, we then preform a line search in this direction to find a novel estimate for the parameter vector. This line search procedure will be explained in more detail after the discussion of the finite difference method. If the relative change of the parameter vector after the line search is sufficient small, the algorithm is stopped and the last estimate for the parameters is returned. Otherwise, the approximate Hessian B is updated using the change in the gradient between the old and new parameters.

[1] You can use the internet, Matlab or any other tool (a fellow student is not a tool) to derive this solution.

[2] For more information, see `https://en.wikipedia.org/wiki/BFGS`.

Algorithm 1 An high-level description of the BFGS-method for minimizing (2) in the model parameters.

Require: an initial estimate for the parameter vector (p_0), an initial estimate for the Hessian (B_0), and a tolerance (`tol`)

1: $k \leftarrow 0$
2: **loop**
3: $d_k \leftarrow -B_k^{-1} \nabla \text{LSE}(p_k)$
4: Perform a line search in the direction d_k to find a suitable p_{k+1}
5: $s \leftarrow p_{k+1} - p_k$
6: **if** $\|s\|_2 / \|p_k\|_2 <$ `tol` **then**
7: **return** the current estimate for the parameters, p_{k+1}
8: **end if**
9: $y \leftarrow \nabla \text{LSE}(p_{k+1}) - \nabla \text{LSE}(p_k)$
10: $B_{k+1} \leftarrow B_k - \dfrac{B_k s s^T B_k^T}{s^T B_k s^T} + \dfrac{y y^T}{s^T y}$
11: $k \leftarrow k + 1$
12: **end loop**

Gradient estimation As in the previous assignment, we will use finite differences to approximate the gradient of a vector-valued function [3]. In Algorithm 1, we require the gradient of cost function (2) with respect to the parameter vector p

$$\nabla LSE(p) = \left[\frac{\partial LSE(p)}{\partial \beta} \quad \frac{\partial LSE(p)}{\partial \mu} \quad \frac{\partial LSE(p)}{\partial \gamma} \quad \frac{\partial LSE(p)}{\partial \delta} \quad \frac{\partial LSE(p)}{\partial \alpha} \right]^T.$$

This gradient can be approximated component by component using finite differences. For example, the derivative of LSE with respect to β can be approximated by

$$\frac{\partial LSE(p)}{\partial \beta} \approx \frac{LSE(\beta + \epsilon, \mu, \gamma, \delta, \alpha) - LSE(\beta, \mu, \gamma, \delta, \alpha)}{\epsilon}$$

with ϵ the step size. Computing a finite difference approximation for the complete vector $\nabla LSE(p)$ thus requires six function evaluations.
Question: Choose a good value for ϵ based on the experiments you preformed during the second assignment.

Line search, backtracking and Wolfe conditions On line 4 of Algorithm 1, we have to preform a line search to find a new estimate for the parameter vector. As the name suggests, this means that we have to search for suitable parameters on a line, more specifically,

$$p_{k+1} \leftarrow p_k + \eta_k d_k$$

with η_k the step size and d_k the search direction. In exact line search one looks for the step size that minimizes the cost function along this line

$$\eta_k = \arg\min_{\eta} LSE(p_k + \eta d_k).$$

This exact search is however typically too computationally costly. We will therefore opt for an inexact line search based on the Wolfe conditions[4]. Instead of finding the exact minimum, we use backtracking to find a suitable step size for which the cost function is sufficiently reduced, which is assessed using the Armijo condition. This backtracking procedure is summarized in Algorithm 2.

[3] Note that the exact gradient can be obtained by extending the original system of differential equations with additional differential equations describing the time-evolution of the sensitivities (derivatives) of the quantities of interest with respect to the model parameters. This is however out of scope for the current assignment.

[4] For more information see `https://en.wikipedia.org/wiki/Wolfe_conditions`.

Algorithm 2 Backtracking procedure for inexact line search.

Require: an initial step size ($\eta_{initial}$), a value for the parameter c_1
  $\eta \leftarrow \eta_{initial}$
  while $\mathrm{LSE}\Big(p_k + \eta d_k\Big) > \mathrm{LSE}\big(p_k\big) + c_1 \eta {d_k}^T \nabla \mathrm{LSE}\big(p_k\big)$ **do**
    $\eta \leftarrow \eta/2$
  end while
  return $\eta_k = \eta$

Generality

Your implementation for this part of the assignment may be tailored to the problem at hand. Your code for the BFGS method thus does **not** need to accept general cost functions. Furthermore, for simplicity, you may assume that the observations are always given once a day. However, write your code in such a way that you can easily add a different optimization algorithm in the future. As in the previous part, make sure that your code can be used in both single and double precision. Finally, make sure that the parameters given in Table 2 can easily be changed.

Questions

Implement the functionality described above in `C++`. Further, make sure that you can easily switch between Euler's forward method, Euler's backward method and Heun's method for simulating the model. Complete the following steps to verify your code.

1. In a file called `estimation1.cpp`, test your implementation for the two provided data sets (`observations1.in` and `observations2.in`). These files have the following structure: on the first line the number of observations and the number of observables (i.e., the number of elements in the vector x_i - for the SIQRD-model, this is always 5-) are given; the following lines give the actual observations in the same format as required by `plot.tex`. Use Heun's method for simulating the SIQRD model and use double precision floating point arithmetic. An initial estimate for the parameters in the SIQRD model is given in Table 1. Values for the other meta-parameters defined in this section are given in Table 2. Print the resulting estimate for the parameters for both observation files. To verify your result you can use the provided `plot_predictions.tex`, which allows to compare the observations and the predictions. Use the same output format as for `plot.tex`.

2. For `observations1.in`, compare the different methods for solving the IVP. For each method measure the execution time and keep track of the number of iterations in the BFGS algorithm. It is not necessary to take the time average over multiple runs, but perform one run with the backward Euler method as warm-up experiment. Save your implementation in a file called `estimation2.cpp`. The output of this program should look as follows:

```
Forward Euler:
 - Number of BFGS iterations: 42
 - Execution time: 12.34 seconds
 - Obtained parameters: (1,2,3,4,5)
-------------------------------------
Backward Euler:
- Number of BFGS iterations: 42
- Execution time: 12.34 seconds
- Obtained parameters: (1,2,3,4,5)
-------------------------------------
Heun's method:
- Number of BFGS iterations: 42
- Execution time: 12.34 seconds
```

- Obtained parameters: (1,2,3,4,5)

Table 1: Initial estimate for the parameter in the SIQRD model

	observations_1.out	observations_2.out
β	0.32	0.5
μ	0.03	0.08
γ	0.151	0.04
α	0.004	0.004
δ	0.052	0.09

Table 2: Values for the meta-parameters introduced in this section

Step size IVP solver (T/N)	1/8 day
Initial estimate for the Hessian (B_0)	The identity matrix
`tol` for Algorithm 1	10^{-7}
Initial step size ($\eta_{initial}$) in the backtracking algorithm	1
c_1 in the backtracking algorithm	10^{-4}

Extra questions

If you have finished the above parts of the assignment and still have sufficient time (and motivation), you can also consider the following **optional** questions. These questions do not carry a significant weight in the final evaluation and can therefore be skipped. If you are struggling with the main part of the assignment, please address these issues first. If your workload is already very high (>30 hours), we also suggest that you do not spend any time on these extra questions.

1. Notice that for `simulation2.cpp` the Jacobian in Euler's backward method has a diagonal structure. Now extend your code such that you can take advantage of this special structure. Make sure that both variants are callable with the same function name and the same number of arguments. The modifications to your original Backward Euler code, should be minimal. Checking whether the Jacobian is diagonal may not incur a runtime overhead. Code to efficiently solve diagonal systems needs of course to be added, but beside this, you should minimize modifications (and duplications) to your code. Compare the original and the new implementation for the following system of decoupled ordinary differential equations

$$\begin{aligned}
\dot{x}_1(t) &= -10\, x_1(t)^3 \\
\dot{x}_2(t) &= -10\, (x_2(t) - 0.01)^3 \\
\dot{x}_3(t) &= -10\, (x_3(t) - 0.02)^3 \\
&\vdots \\
\dot{x}_M(t) &= -10 \left(x_M(t) - \tfrac{M-1}{100} \right)^3 .
\end{aligned}$$

   with initial values $\begin{bmatrix} 0.001 & 0.002 & \dots & \frac{M}{1000} \end{bmatrix}^T$, $T = 500$ and $N = 50000$. Time the execution duration of both variants for $M = \{25, 50, 100\}$. What do you observe?
   Hint: The `ublas::diagonal_matrix` class is defined inside the `<boost/numeric/ublas/banded.hpp>` header file.

2. Do you see other improvements that you can make to your code?

Practical information

The deadline for submission on Toledo is **Sunday December 19th at 22h00**. This deadline is strict![5] Therefore, do not wait until the last minute to submit, as we will not accept technical issues as an excuse for late submissions. Your submission should be a zip archive named `hw4_lastname_firstname_studentnumber.zip` with `lastname` your last name, `firstname` your first name and `studentnumber` your student number. For example if your name is John Smith and your student number is r0123456, your file should be called `hw4_smith_john_r0123456.zip`. This zip archive should contain the following:

- All code you wrote to complete this assignment, including `simulation1.cpp`, `simulation2.cpp`, `estimation1.cpp` and `estimation2.cpp`. Make sure that this code is well documented and easy to read. Either provide a custom makefile[6], or add the instructions needed for compiling your code in the provided makefile. For sake of automation, the name of the executable for `simulation1.cpp` should be `simulation1` and so on.
- A brief document (named `hw4_smith_john_r0123456.pdf`, using the same naming format as before) containing your design decisions and some comments on which concepts from the lectures and exercise sessions you applied in this assignment. Also include an estimate of the total amount of time spent on this assignment. This number has no influence on your grade, but helps us in determining the load of the assignments for future years. A template is provided on Toledo.
- Please include any figures or terminal output relevant to your discussion in your zip archive.

You can post questions about the assignment on the discussion forum on Toledo. If you get stuck on a problem that you are unable to fix, you can contact one of the teaching assistants for help. It is best to do this well in advance of the deadline, as we might not be able to reply otherwise. Finally, as always, if you take part in the Dutch course, you can write your report, code and documentation in Dutch.

Appendix

Some useful tips

- Before you start with your implementation, make a scheme describing your plan of attack. Which functions will you need? What arguments will these functions take? How will you debug your code? It can also be useful to answer questions (G.1), (I.1), (I.2), (II.1) and (II.2) in the provided template before you start with your implementation.
- Split the work into smaller parts. For example, first implement Euler's forward method and test it. Then implement and test Heun's method and so on. Also for the second part of the assignment, the work can be split into smaller task: e.g., first implement and test your finite difference code, then your linesearch code and so on.
- If you can not connect to a computer in the departmental PC rooms using the ssh-extension in Visual Studio Code, try to connect to this computer via the terminal or powershell. If this also does not work, contact one of the teaching assistants. Otherwise, verify whether you are still below your storage quota using the command `quota`. If you have reached your quota use the command `du -hd2 | sort -h` to list which files and folders take the most storage space. Folders that can be removed (`rm -rf` [7]) safely are
  - `~/.cache/mozilla/firefox/`,

[5] This deadline can however be extended on an individual basis if sufficiently motivated.
[6] `make simulation1` should compile the code for `simulation1.hpp` and so on.
[7] **Be careful to specify the correct path. There is no way to recover files deleted with `rm -rf`.**

- `~/.local/share/Trash/*`,
- `~/.cache/google-chrome/Default/Cache/*` and
- `~/.vscode-server` [8].

For more information see `https://system.cs.kuleuven.be/cs/system/wegwijs/computerklas/homedir/`

- As in Fortran, be careful with integer division: dividing two integers will result in another integer.
- When computing (2), keep in mind that you have to take every 8^{th} (as the step size is $1/8$ day) sample of your simulation.
- Verify that you have included the necessary header files. Header files that are frequently necessary, are `<cassert>`, `<iostream>`, `<algorithm>`, `<cmath>`, `<numeric>`, `<cstdlib>` and `<chrono>`. For the header files required by the `ublas`-library, see the examples below.
- The following example demonstrates the correct way to indicate that an argument is passed by reference and that the argument remains unchanged (`const`) during the execution of the function

  ```
  f(std::vector<double> const & v){}
  ```

  Switching `const` and `&` is not allowed. The following code will give a compile error

  ```
  f(std::vector<double> & const v){}
  ```

- The order of initializer list should match the order in which the member variables are declared. For an example on what can go wrong if you switch the order, see `https://www.geeksforgeeks.org/order-of-execution-in-initializer-list-in-c/`.
- For functions or classes defined inside a namespace, do not forget to specify this namespace when calling them.
- The order in which functions, classes and types are declared matter. A function, class or type can only be used after it is declared. For example, if you want to use a function `linesearch` in your BFGS method, you have to declare it before your BFGS code. An exception are member functions or variables accessed from within the class itself. In the following example

```
class demo{
public:
  demo(int a)
  :a_(a)
  {}
  void do_something(){
    for (size_t i=0; i<this->size(); i++){
      // Do something
    }
  }

  size_t size() const{
    return a_;
  }
private:
  int a_;
};
```

[8] If you remove this folder, you will have to reinstall your remote VS Code plug-ins.

the constructor can set the member variable `a_` and the function `do_something` can call the function `size()` even though they are defined later on.

- Make sure that you use the correct compiler flags.
- The `g++`-compiler can give long and daunting looking compile errors, even for a small mistake. The best advice is to start reading from the top, then fix the first error and recompile your code. Repeat this cycle until all compiler errors are solved.
- Function arguments that declared `const`, must remain unchanged during the execution of the function. This also means that all functions that are applied to this argument must guarantee that the object remains `const`. For example, if you have the following code

```
void function(Object const& o){
       std::cout<<o.size()<<std::endl;
}
```

then the function `size()` (which is a member function of the class `Object`) may not change `o`:

```
class Object{
...
size_t size() const {...}
}
```

Note that inside a constant member function, only member variables that are declared `mutable` can change.

```
class Object{
private:
  mutable double d1;
  double d2;
public:
  void const_fun(double a) const{
   // Here we can change d1 but not d2.
  }
 }
```

Using the C++ IO-functionality

To open a file for reading or writing, you need to include the `<fstream>`-header file in `C++`. This header defines the `std::ifstream`-class which can be used to read a file:

```
std::ifstream file(filename);
```

with `filename` a string containing the filename. To read this file line by line you can use the following syntax

```
std::string line;
while (std::getline(file,line)){
       std::cout<<line<<std::endl;
}
```

To split this line on spaces use

```
std::istringstream split_line(line);
for(std::string s ; split_line >> s; ){
       std::cout<<s<<std::endl;
}
```

To write to a file you need an instance of the `std::ofstream`-class. Writing to a file is now as simple as writing to the standard output stream:

```
std::ofstream outputFile(filename);
outputFile<<"Scientific Software is fun!"<<std::endl;
```

in which `filename` is again a string containing the filename.

To convert a `std::string` to an `int` or `double`, you can use

```
int a = std::stoi(s); // and
double d = std::stod(s);
```

which are both defined in the `<string>`-header file.

The uBlas library

In this assignment you can use the uBlas library[9] to perform linear algebra operations. This uBlas library is part of Boost, a large collection of well-maintained `C++` libraries that support a wide range of systems and operating systems and that is installed in the departmental PC rooms. Below we give some examples of frequently used uBlas operations. To avoid having to type out the complete namespace for functions and classes in the uBlas library, we will frequently use the following code snippet to define a shorter alias:

```
namespace ublas = boost::numeric::ublas;
```

The codes for these demos are available on Toledo.

Creating a matrix and a vector

To create a vector and a matrix, you need to include the `<boost/numeric/ublas/vector.hpp>` and `<boost/numeric/ublas/matrix.hpp>` header files, respectively. The following program creates a 3×3 matrix A and a three dimensional vector x.

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <iostream>
namespace ublas = boost::numeric::ublas;

int main(int argc, char *argv[]){
        ublas::matrix<double> A(3,3); // matrix
        A(0,0) = 1; A(0,1) = 1; A(0,2) = 1;
        A(1,0) = 1; A(1,1) = -1; A(1,2) = 0;
        A(2,0) = 1; A(2,1) = 0; A(2,2) = -1;
        ublas::vector<double> x(3); // vector
        x(0) = 1; x(1) = 2; x(2) = 3;
        std::cout<<A<<std::endl; // io
        std::cout<<x<<std::endl;
        return 0;
}
```

Note that matrices are by default stored in row major order, but you can also use column major order:

```
ublas::matrix<double,ublas::column_major> A(3,3);
```

[9] `https://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas/doc/index.html`

Initializing the elements of a matrix or vector

To set the values of the elements of a matrix or a vector you can also use:

```
ublas::matrix<double> A1(3,3); // matrix
A1<<=1,1,1,1,-1,0,1,0,-1;
ublas::vector<double> x1(3); // vector
x1<<=1,2,3;
```

Note that this requires the `<boost/numeric/ublas/assignment.hpp>`-header file.

Inspecting the dimensions of vectors and matrices

To inspect the dimensions of a vector or a matrix you can use:

```
x.size(); // Length of the vector x
A.size1(); // Number of rows of the matrix A
A.size2(); // Number of columns of the matrix A
```

Inner and outer product

The inner product of two vectors, $\alpha = x^T y$, can be computed using

```
ublas::vector<double> x(3),y(3);
double alpha = ublas::inner_prod(x,y);
```

The outer product of these two vectors can be computed using

```
ublas::matrix<double> P(ublas::outer_prod(x,y));
```

Matrix vector product

The matrix vector product $y = Ax$ can be computed using

```
ublas::matrix<double> A(3,3);
ublas::vector<double> x(3),y(3);
y.assign(ublas::prod(A, x));
```

The `assign` function

It is often beneficial to use the `assign`-function instead of the overloaded `=`-operator. Consider for example the following two programs.

Listing 1: prog1.cpp

```
#include <boost/numeric/ublas/vector.hpp>
namespace ublas = boost::numeric::ublas;

int main(){
  ublas::vector<double> x(10,0.1), y(10), z(10);
  for (unsigned int i=0; i<1000000; i++){
    y = x+z;
    z = y+x;
  }
}
```

and

Listing 2: prog2.cpp

```
#include <boost/numeric/ublas/vector.hpp>
namespace ublas = boost::numeric::ublas;

int main(){
ublas::vector<double> x(10,0.1), y(10), z(10);
  for (unsigned int i=0; i<1000000; i++){
    y.assign(x+z);
    z.assign(y+x);
  }
}
```

Although these programs look quite similar, their performance is not. If we compile both programs with `g++ -O3 -DNDEBUG -std=c++17` and run them with valgrind we see that the former requires 2,000,004 allocations, while the latter only requires 4. This of course leads to a huge performance gain (0.061 seconds vs 0.018 seconds).

Solving linear system

The system $Ax = b$ can be solved by including the `<boost/numeric/ublas/lu.hpp>`-header file:

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/lu.hpp>

namespace ublas = boost::numeric::ublas;

int main(int argc, char *argv[]){
        ublas::matrix<double> A(2,2);
        A(0,0) = 1; A(0,1) = 1;
        A(1,0) = 1; A(1,1) = -1;
        ublas::vector<double> x(2);
        x(0) = 2; x(1) = 0;
        ublas::permutation_matrix<size_t> pm(A.size1());
        ublas::lu_factorize(A,pm);
        ublas::lu_substitute(A, pm, x);
        std::cout<<x<<std::endl;
        return 0;
}
```

Important: Note that A is overwritten with its LU factorisation at the end of this operation.

Important: Note that if you want to reuse your permutation vector to solve a new system, you have to reset it with its initial values:

```
size_t n = 20;
  ublas::vector<double> x(n), b(n), r(n);
  b(0) = 1;
  ublas::matrix<double> A(n,n), A2(n,n);
  ublas::permutation_matrix<size_t> pm1(A.size1());
  ublas::permutation_matrix<size_t> pm2(A.size1());
  for (size_t i = 0;i<100;i++){
    random_numbers(A);
    A2.assign(A);
    ublas::lu_factorize(A,pm1);
    x.assign(b);
    ublas::lu_substitute(A, pm1, x);
    r.assign(ublas::prod(A2,x));
```

```
    r-=b;
    std::cout<<ublas::norm_2(r)<<std::endl;
    pm1.assign(pm2); // Removing this line will lead to a large residue
}
```

Reading and writing to a row/column

To access a row or column of a matrix A you can use the following syntax:

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
namespace ublas = boost::numeric::ublas;
int main(int argc, char *argv[]){
        ublas::matrix<double> A(3,3);
        ublas::vector<double> v(3);
        v(0) = 1; v(1) = 2; v(2) = 4;
        ublas::matrix_row<ublas::matrix<double>>a0(A,0);
        std::cout<<a0<<std::endl;
        a0.assign(v);
        std::cout<<a0<<std::endl;
        std::cout<<A<<std::endl;
        ublas::matrix_column<ublas::matrix<double>>ac0(A,0);
        std::cout<<ac0<<std::endl;
        ac0.assign(v);
        std::cout<<A<<std::endl;
        return 0;
}
```

Note that this requires the <boost/numeric/ublas/matrix_proxy.hpp>-header file. To access a slice of a row or column you can use:

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
namespace ublas = boost::numeric::ublas;
int main(int argc, char *argv[]){
        ublas::matrix<double> A(500,500,27.);
        ublas::vector<double> v(200,1.);
        auto mr= ublas::row (A,0);
        auto mr1 = ublas::subrange(mr,1,5);
        std::cout<<mr1<<std::endl;
        for (unsigned int i = 0; i < 500; i++)
        {
                auto mr= ublas::row (A,i);
                auto mr1 = ublas::subrange(mr,1,201);
                mr1.assign(v*(i+1));
        }
        std::cout<<A(0,3)<<" "<<A(1,3)<<" "<<A(200,3)<<std::endl;
        return 0;
}
```

Important: This requires both the <boost/numeric/ublas/matrix_proxy.hpp>-header file and the <boost/numeric/ublas/vector_proxy.hpp>-header file.

Other useful (ublas-)functionality

The code snippets below give some other useful (ublas-)functionality.
The identity matrix I can be created using

```
ublas::identity_matrix<double> I(3);
std::cout<<I<<std::endl;
```

This identity matrix can be assigned to another matrix.

```
ublas::matrix<double> M(3,3);
M.assign(I);
std::cout<<M<<std::endl;
```

The Euclidean norm of a vector can be computed as follows

```
ublas::vector<double> v(2);
v(0) = 3.; v(1) = 4.;
std::cout<<ublas::norm_2(v)<<std::endl;
```

To multiply all elements of a vector of matrix with a common scalar you can use

```
v1*= 0.1;
v2.assign(v1*0.1);

M1*=0.1;
M2.assign(M2*0.1);
```

To set all values of a vector or matrix to zero, you can use

```
M.clear();
std::cout<<M<<std::endl;
```

The following mathematical operations are defined in the <cmath>-header file

```
// Power of a number
std::cout<<std::pow(3.,2)<<std::endl; // prints 9.
// Square root of a number
std::cout<<std::sqrt(9.)<<std::endl; // prints 3.
```