Technisch Wetenschappelijke Software
Scientific Software

C++ - Homework Assignment 4

Modelling pandemics - Parameter estimation

1 Introduction

This homework assignment deals with the theory and concepts encountered in lectures 5 till 8, the introductory videos on C++ and exercise sessions 7 and 8. More specifically, we will focus on the following concepts:

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

As assignments 1 and 2, this assignment is centred around 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, in this assignment we want to estimate the parameters β, μ, γ, δ and α from observations of the number individuals in each compartment at given time instances. Therefore, we will first 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 estimate the parameters underlying a given set of observations by minimizing the prediction error in the model parameters using these solvers (part II).

Before starting on the assignment, carefully read the practical submission information given below the two parts of the assignment. Finally, in appendix, some material on using the C++ IO-functionality and on the `uBlas` library is provided.

2 Part I: Simulating the pandemic

Complete the following tasks:

1. Implement the forward Euler method, the backward Euler method and Heun's method in C++. It suffices to use double-precision, but as an extra task later you can also make your code more generic so you can easily switch in between different floating-point types (see section 3.3). In contrast to the Fortran assignments, make sure that your code can easily handle general systems of ordinary differential equations. While writing this code, take into account that these functions will be called numerous times with the same differential equations, but with different parameters, in the next part of the assignment.

2. Verify your implementation:

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

i. $\delta = 0$ with the Forward Euler method (store the result in `fwe_no_measures.out`)

ii. $\delta = 0.2$ with the Backward Euler method (store the result in `bwe_quarantine.out`)

iii. $\delta = 0.9$ with Heun's method (store the result in `heun_lockdown.out`).

As in assignment 2, N and T should be passed as command line arguments. The other parameters must be read from a file named `parameters.in`. This file has the following structure (δ is part of the file format, but your program should ignore it and set the parameter following the tasks described above):

```
beta mu gamma alpha delta S0 I0
```

To be able to use `plot.tex` from assignment 2, 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
```

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

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

How can you pass this differential equation to your ODE solver code? Discuss three possibilities. Use $\begin{bmatrix}0.01 & 0.02 & 0.03 & \dots & 0.5\end{bmatrix}^T$ as the initial value (avoid using an explicit `for`-loop to fill this vector), $T = 500$ and $N = 50000$. Store the results for each solver in the files `fwe_simulation2.out`, `bwe_simulation2.out` and `heun_simulation2.out`.

3 Part II: Parameter estimation from observations

3.1 Method description

Now we can estimate the parameters of the SIQRD model, $p := \begin{bmatrix}\beta & \mu & \gamma & \alpha & \delta\end{bmatrix}^T$, based on observations. For simplicity, we will assume that the provided data is sampled once a day at a fixed hour. We will denote the vector containing the observation (the number of individuals in the different compartments) on day i by

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

for $i = 0, \dots, T$. To find a good estimate for the underlying parameters, we will minimize a scaled least square prediction error in these parameters

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

with $P_i := S_i + I_i + Q_i + R_i + D_i$ being the number of people in the population and $\hat{x}_i(p)$ being a vector with our prediction for the number of individuals in each compartment on day i,

based on our current estimate for the parameters. This prediction is obtained by simulating the SIQRD-model starting from x_0 (the first observation) for our current estimate of the parameters and a sufficiently small time step (see Table 2) using an IVP solver implemented in part I.

To find the parameters that minimize (2), we will use the the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm[1]. This is a quasi-Newton optimization method and thus uses an approximation for the Hessian which we will denote by B_k. Algorithm 1 gives an high-level description of the BFGS method for computing the minimum of (2), in which $\nabla\text{LSE}(p_k)$ denotes the gradient of LSE with respect to p evaluated in p_k. To find an approximation for this gradient, we will use a finite difference approach as explained in the next paragraph.

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

Require: initial parameters p_0, 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 step-size η_k (see below)
5: **if** $\eta_k\|d_k\|_2/\|p_k\|_2 < tol$ **then**
6: **return** the current estimate for the parameters, p_k
7: **end if**
8: $s \leftarrow \eta_k\, d_k$
9: $p_{k+1} \leftarrow p_k + s$
10: $y \leftarrow \nabla\text{LSE}(p_{k+1}) - \nabla\text{LSE}(p_k)$
11: $B_{k+1} \leftarrow B_k - \dfrac{B_k s s^T B_k^T}{s^T B_k s} + \dfrac{y y^T}{s^T y}$
12: $k \leftarrow k + 1$
13: **end loop**

Gradient estimation As in assignment 2, we will use finite differences to approximate the gradient[2]. Recall that the gradient of LSE with respect to the model parameters p is given by

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

in which the derivative of LSE with respect to β can be approximated by

$$\frac{\partial\text{LSE}(p)}{\partial\beta} \approx \frac{\text{LSE}(\beta(1+\epsilon),\mu,\gamma,\delta,\alpha) - \text{LSE}(\beta,\mu,\gamma,\delta,\alpha)}{\beta\epsilon}$$

and so on. Choose a good value for ϵ based on your experience from assignment 2.

Line search, backtracking and Wolfe conditions In line 4 of the BFGS algorithm, we have to perform a line search. As the name suggests, this means that we have to search for suitable parameters on a line, more specifically, on the line

$$p_k + \eta d_k$$

with η the step size and d_k the search direction. In exact line search one needs to find the minimizer along this line

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

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

[2] The exact gradient can be obtained by extending the original system of differential equations with additional equations describing the time-evolution of the sensitivities (derivatives) of the quantities of interest with respect to the model parameters.

This exact search can however be computationally costly and we will therefore opt for an inexact line search based on the Wolfe conditions[3]. Instead of finding the exact minimum, we use backtracking to find a suitable step size for which the cost function is sufficiently reduced, using the first Wolfe condition:

$$\mathrm{LSE}\Big(p_k + \eta d_k\Big) \leq \mathrm{LSE}\big(p_k\big) + c_1 \eta {d_k}^T \nabla \mathrm{LSE}\big(p_k\big) \tag{3}$$

The backtracking procedure is summarized in Algorithm 2.

Algorithm 2 Backtracking procedure for inexact line search.

Require: initial step size η
  while condition (3) is not fulfiled **do**
    $\eta \leftarrow \eta/2$
  end while
  return $\eta_k = \eta$

3.2 Tasks

1. Implement the described functionality for parameter estimation in C++ (using double-precision). Make sure that you can easily switch between the forward Euler method, the backward Euler method and Heun's method for simulating the model.

2. Verify your code:

   (a) 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 observed quantities of interests are given (for the SIQRD-model, this is always 5); below this are the actual observations, having the same structure as the output described above to use `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. To verify your result you can use the provided `plot_predictions.tex`, which allows to compare the observations and the predictions (provided they are stored in `prediction1.txt`). Use the same output format as for `plot.tex`. Make sure the resulting executable `estimation1` prints, both for `observations1.in` and `observations2.in`, the estimated parameter values, the resulting LSE-value and the number of BFGS iterations used.

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

	`observations1.in`	`observations2.in`
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

   (b) For `observations1.in`, compare the different methods for solving the ODE, while using the BFGS method to compute the minimizers. Do the obtained parameters differ? What about execution time? Save your work in the program `estimation2.cpp`.

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

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

	BFGS
ϵ	see above
Step size ODE solver	1/8 day
c_1 in (3)	10^{-4}
tol	10^{-7}
Initial step size	1
Initial estimate Hessian B_0	Identity matrix

3.3 Extra tasks

If you have finished the above parts of this assignment and still have sufficient time (and motivation), you can also consider the following questions. These questions will only be considered if you have properly implemented the requested basic functionality and will not carry a significant weight in the final evaluation. If you are struggling with the main part of the assignment, please address these issues first. If your workload is already very high (> 40 hours), we also suggest that you do not spend too much time on these extra questions.

1. Notice that in `simulation2.cpp` when using Euler's backward method the Jacobian is a diagonal matrix. 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. 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

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

   with initial values $\begin{bmatrix}0.001 & 0.002 & \dots & \frac{M}{1000}\end{bmatrix}^T$, $T = 5000$ and $N = 50000$. Time the execution duration of both variants for $M = \{50, 100, 200, 400\}$. What do you observe?

2. How would you write your code to use different floating point types upon request with minimal code duplication? Demonstrate this by implementing this support for your IVP solvers (after implementing the functionality from the previous extra task).

4 Practical information

The deadline for submission on Toledo is **December 14 at 14h00**. This deadline is strict! 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 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 `estimation1.cpp`. Make sure that this code is well-documented and easy to read. The result files (`fwe_no_measures.out`, ...) should **not** be included, instead, they should be generated by the aforementioned C++ source files after compilation and execution.
- A filled-in version of `Makefile` provided on Toledo (preferred), or a separate text file containing the instructions needed for compiling your code.
- A filled-in version of the report template that can be found on Toledo. If you want to discuss figures or terminal output that are too large to reasonably be integrated in the report, you can include them as separate files in your zip file.
- A text file named `time_spent.txt`, only containing a single number, the amount of hours you spent on this assignment. This has no influence on your grade but helps us to determine the average workload of each assignment for future years, so please be honest.

You can post questions about the assignment on the discussion forum on Toledo. As always, if you take part in the Dutch course, you can write your report (and documentation) in Dutch. Finally, we want to remind you of the guidelines with respect to cooperation:

The solution and/or report and/or program code that are handed in, have to fully be the result of work you have performed yourself. You can of course discuss the assignment with other students, in the sense that you may talk about general solution methods or algorithms, but the discussion cannot be about specific code or report text that you are writing, nor about specific results that you wish to hand in. If you talk with others about your tasks, this can **never** lead to you being in possession of a whole or partial copy of the code or report of others, regardless of the code or the report being on paper or available in electronic form, and independent of who wrote the code or report (fellow students, possibly from other study years, complete outsiders, internet sources, etc.). This also encompasses that there is no valid reason at all to pass your code or report to fellow students, nor to make this available via publicly accessible directories or websites.

5 Appendix

5.1 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(file_name);
```

with `file_name` 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 the `std::ofstream`-class. Writing to a file is now as simple as writing to the standard output stream:

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

where `file_name` is again a string containing the filename.

5.2 The `mutable` keyword

When writing classes in C++, it is generally a good idea to mark any method that does not change the observable state of an object as `const` by putting the `const` keyword at the end of the function signature (after the arguments and their matching parentheses), which means the method does not modify any member variables of the object. Then, objects of this class may be passed around as `const` to other functions as long as they only call `const` methods.

However, in some cases, it may still be beneficial for these methods to modify some internal state, for example for caching, despite the fact that this does not change the observable state of the object and simply improves performance. In such cases, you can mark certain member variables as `mutable`, meaning that they can be modified by `const` methods, unlike other member variables without this keyword. For example (using the C++ 20 standard):

```
#include <type_traits>
#include <unordered_map>

template<typename F, typename InputType>
class CachedFunctionEvaluator {
public:
	typedef std::invoke_result_t<F, InputType> OutputType;
	CachedFunctionEvaluator(const F& _f) : f(_f) { }
	OutputType operator()(const InputType& input) const {
		if (cache.contains(input)) {
			return cache[input];
		} else {
			OutputType output = f(input);
			// We can modify the variable cache in this const method
			// because it's marked mutable
			cache[input] = output;
			return output;
		}
	}
private:
	const F& f;
	mutable std::unordered_map<InputType, OutputType> cache;
};
```

5.3 The uBlas library

In this assignment you can use the uBlas library[4] to perform linear algebra operations. The 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 in the uBlas library, you can use the following code to define a shorter alias:

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

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

5.3.1 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>

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 row major, but you can also use column major matrices:

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

5.3.2 Inner 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);
```

5.3.3 Matrix vector product

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

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

5.3.4 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;
}
```

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

5.3.5 Reading and writing to a row/column

We can access a row or column of a matrix A using the `<boost/numeric/ublas/matrix_proxy.hpp>` and the `<boost/numeric/ublas/vector_proxy.hpp>` headers, as follows:

```
#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;
}
```

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;
}
```

Finally, to extract rows or columns from a `const` matrix, you should of course use an adapted type, like `const ublas::matrix_row<const ublas::matrix<double>>`.