Technisch Wetenschappelijke Software
Scientific Software
C++, Homework Assignment 4
Modelling pandemics - Parameter estimation

Pieter Appeltans, Emil Loevbak & Wouter Baert Dorne, December 7, 2020

Introduction

This home work 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 3, 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. To these end, we will first implement the three ODE solvers (Euler's forward method, Euler's backward method and Heun's method) in C++. In the second and final part of this assignment, we will estimate the parameters underlying a given set of observations by minimizing the prediction error in the model parameters.

Carefully read the practical submission information given below the two parts of the assignment. In appendix, some material on using the C++ IO-functionality and on the uBlas library is provided.

Part 1: Simulating the pandemic

Implement the forward Euler method, the backward Euler 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. **What C++-functionality did you need to use to achieve this extra flexibility?** 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.

- Write a file `simulation1.cpp`, in which you test your ODE 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

1. $\delta = 0$ with the Forward Euler method (store the result in `fwe_no_measures.out`)
2. $\delta = 0.2$ with the Backward Euler method (store the result in `bwe_quarantine.out`)
3. $\delta = 0.9$ with Heun's method (store the result in `heun_lockdown.out`).

As in previous assignment, 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

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

To be able to use `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
```

- Write a program `simulation2.cpp` in which you test your ODE 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 all possibilities. Use $\begin{bmatrix}0.01 & 0.02 & 0.03 & \dots & 0.5\end{bmatrix}^T$ as initial value (avoid using an explicit `for`-loop to fill this vector), $T = 500$ and $N = 50000$.

Part 2: Parameter estimation from observations

Now we will estimate the parameters of the SIQRD model, $p = (\beta, \mu, \gamma, \delta, \alpha)$, 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

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

with N the number of people in the population and $\hat{x}_i(p)$ a vector with our prediction for the number of individuals in each compartment on day i, based on our current estimate for the parameters (p). 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.

To find the parameters that minimize (2), we will compare two approaches.

1. As first method we will consider the conjugate gradients method [1]. This method improves the convergence of the naive steepest descent method for cost functions that are approximately quadratic near the minimum. Algorithm 1 gives an high-level description of this method in which $\nabla \mathrm{LSE}(p_k)$ 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 subsection. An important parameter in the conjugate gradient method is ν_k, which tries to assure that the new search direction r_{k+1} is conjugated to old search direction r_k. Two popular choices for ν_k are

   - the Fletcher-Reeves formula:

$$\nu_k = \begin{cases} 0 & \text{if } \operatorname{mod}(k,n) = 0 \\ \dfrac{\nabla \mathrm{LSE}(p^k)^T \nabla \mathrm{LSE}(p^k)}{\nabla \mathrm{LSE}(p^{k-1})^T \nabla \mathrm{LSE}(p^{k-1})} & \text{otherwise} \end{cases} \tag{3}$$

     with n the number of parameters and mod the modulo operator,

   - and the Polak-Ribière formula

$$\nu_k = \begin{cases} 0 & \text{if } \operatorname{mod}(k,n) = 0 \\ \dfrac{\nabla \mathrm{LSE}(p^k)^T \left(\nabla \mathrm{LSE}(p^k) - \nabla \mathrm{LSE}(p^{k-1})\right)}{\nabla \mathrm{LSE}(p^{k-1})^T \nabla \mathrm{LSE}(p^{k-1})} & \text{otherwise} \end{cases} \tag{4}$$

     in which n is again the number of parameters.

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

```
Require: initial parameters $p_0$ and a tolerance $tol$
 1: $k \leftarrow 0$
 2: $d_{-1} \leftarrow 0$
 3: loop
 4:   $d_k \leftarrow -\nabla \mathrm{LSE}(p_k) + \nu_k d_{k-1}$ with $\nu_k$ as defined in either (3) or (4)
 5:   Preform a line search in the direction $d_k$ to find a suitable step-size $\eta_k$ (see below)
 6:   if $\eta_k \|d_k\|_2 / \|p_k\|_2 < tol$ and $\nu_k \neq 0$ then
 7:     $d_k \leftarrow -\nabla \mathrm{LSE}(p_k)$
 8:     Preform a line search in the direction $d_k$ to find a suitable step-size $\eta_k$ (see below)
 9:   end if
10:   if $\eta_k \|d_k\|_2 / \|p_k\|_2 < tol$ then
11:     return the current estimate for the parameters $p_k$
12:   end if
13:   $p_{k+1} \leftarrow p_k + \eta_k\, d_k$
14:   $k \leftarrow k + 1$
15: end loop
```

2. The second algorithm that we will consider is the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [2]. This is a quasi-Newton method and thus uses an approximation for the Hessian which we will denote by B_k. Algorithm 2 gives an high-level description of the BFGS method for computing the minimum of (2).

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

Algorithm 2 An 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:      $d_k \leftarrow -B_k^{-1} \nabla \mathrm{LSE}(p_k)$
4:      Preform a line search in the direction $d_k$ to find a suitable step-size $\eta_k$ (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:      $y \leftarrow \nabla \mathrm{LSE}(p_{k+1}) - \nabla \mathrm{LSE}(p_k)$
10:     $p_{k+1} \leftarrow p_k + s_k$
11:     $B_{k+1} \leftarrow B_k - \frac{B_k s s^T B_k}{s^T B_k s} + \frac{y y^T}{s^T y}$
12:     $k \leftarrow k + 1$
13: end loop
```

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

$$\nabla LSE(p) = \begin{bmatrix} \frac{\partial LSE(p)}{\partial \beta} & \frac{\partial LSE(p)}{\partial \mu} & \frac{\partial LSE(p)}{\partial \gamma} & \frac{\partial LSE(p)}{\partial \delta} & \frac{\partial LSE(p)}{\partial \alpha} \end{bmatrix}^T$$

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

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

and so on. Computing a finite difference approximation for $\nabla LSE(p)$ thus requires six function evaluations. Choose a good value for ϵ based on the experiments you preformed during the last assignment.

Line search, backtracking and Wolfe conditions In both the conjugate gradients and the BFGS method, we have to preform a line search. As the name suggests, this means that we have to search for suitable parameters on a 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

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

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 Wolfe conditions:

$$\mathrm{LSE}\Big(p_k + \eta d_k\Big) \leq \mathrm{LSE}(p_k) + c_1 \eta {d_k}^T \nabla \mathrm{LSE}(p_k) \tag{5}$$

$$-d_k^T \nabla \mathrm{LSE}(p_k + \eta d_k) \leq -c_2 d_k^T \nabla \mathrm{LSE}(p_k). \tag{6}$$

The backtracking procedure is summarized in Algorithm 3.

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

Algorithm 3 Backtracking procedure for inexact line search.

```
Require: initial step size $\eta$
  while conditions (5) and (6) are not fulfiled do
    $\eta \leftarrow \eta/2$
  end while
```

Questions

1. Implement functionality for parameter estimation based on both the conjugate gradients and the BFGS algorithm in C++. Make sure that you can easily switch between the forward Euler method, the backward Euler method and Heun's method for simulating the model. For the conjugate gradients method, you can use the Fletcher-Reeves formula for ν_k, but write your code in such a way that you can easily switch to the Polak-Ribière formula.

2. In a file called `estimation1.cpp`, test both algorithms for the two provided data sets (`observations1.out` and `observations2.out`). These files have the following structure: on the first line the number of observations and the number of observed variables are given; 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. Use the same output format as for `plot.tex`.

3. 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? Save your experimentation in the file `estimation2.cpp`

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

	CG	BFGS
ϵ	see above	see above
Step size ODE solver	1/8 day	1/8 day
c_1 in (5)	10^{-4}	10^{-4}
c_2 in (6)	0.9	0.9
tol	10^{-7}	10^{-7}
initial step size	0.01	1
Initial estimate Hessian B_0	/	Identity matrix

Extra questions

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 (>30 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
$$\begin{aligned}\dot{x}_1(t) &= -\,10\,x_1^3\\ \dot{x}_2(t) &= -\,10\,(x_2-0.01)^3\\ \dot{x}_3(t) &= -\,10\,(x_3-0.02)^3\\ &\ \vdots\\ \dot{x}_M(t) &= -\,10\left(x_{50}-\tfrac{M-1}{100}\right)^3.\end{aligned}$$
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. Do you see other improvements you can make to your code?

Practical information

The deadline for submission on Toledo is **December 15 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 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 file 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. Either provide a makefile, or add the instructions needed for compiling your code.
- A brief document 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 has no influence on your grade, but helps us in determining the load of the assignments for future years.
- Please include any figures or terminal output relevant to your discussion in your zip.

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 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.

Appendix

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.

The uBlas library

In this exercise session 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

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);
```

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);
```

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));
```

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.

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.