Submission Date: 8th of January 2018 **Assessed Coursework 3**

The following questions are to be used for the coursework assessment in the module G14SCC. A single zip or tar file containing your solution should be submitted through the module webpage on Moodle as per the instructions on the Coursework Submission form. The style and efficiency of your programs is important. A barely-passing solution will include attempts to write programs which include some of the correct elements of a solution. A borderline distinction solution will include working code that neatly and efficiently implements the relevant algorithms, and that shows evidence of testing. An indication is given of the approximate weighting of each question by means of a figure enclosed by square brackets, eg [12]. All calculations should be done in double precision.

Systems of Ordinary Differential Equations

Cauchy problems, also known as initial-value problems, consist of finding solutions to a system of Ordinary Differential Equations (ODEs), given suitable initial conditions. We will be concerned with the numerical approximation of the solution to the problem

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(t, \boldsymbol{u}(t)), \qquad t \in [t_0, T], \tag{1}$$

$$\boldsymbol{u}(t_0) = \boldsymbol{u}_0, \tag{2}$$

where the dot denotes differentiation with respect to t, $\boldsymbol{f}$ is a sufficiently well-behaved function that maps $[t_0, T) \times \mathbb{R}^d$ to $\mathbb{R}^d$, the initial condition $\boldsymbol{u}_0 \in \mathbb{R}^d$ is a given vector, and the integer d is the dimension of the problem. We assume that $\boldsymbol{f}$ satisfies the Lipschitz condition

$$\|\boldsymbol{f}(t, \boldsymbol{v}) - \boldsymbol{f}(t, \boldsymbol{u})\| \leq \lambda \|\boldsymbol{v} - \boldsymbol{u}\| \qquad \text{for all } \boldsymbol{v}, \boldsymbol{u} \in \mathbb{R}^d,$$

where $\lambda > 0$ is a real constant independent of $\boldsymbol{v}$ and $\boldsymbol{u}$. This condition guarantees that the problem (1)–(2) possesses a unique solution. We seek an approximation to the solution $\boldsymbol{u}(t)$ to (1)–(2) at $N + 1$ evenly-spaced time points in the interval $[t_0, T]$, so we set

$$t_n = t_0 + nh, \qquad h = (T - t_0)/N, \quad 0 < n \leq N. \tag{3}$$

The scalar h is referred to as the *time step*. We use a subscript h to denote our approximation to $\boldsymbol{u}(t)$ at the time points $\{t_n\}$,

$$\boldsymbol{u}_h(t_n) \approx \boldsymbol{u}(t_n), \quad 0 \leq n \leq N,$$

and we are interested in the behaviour of the error $\boldsymbol{e}_{h,n} = \boldsymbol{u}_h(t_n) - \boldsymbol{u}(t_n)$. We expect this error to decrease as the step size h tends to 0: the sequence of approximations $\{\boldsymbol{u}_h(t_n)\}$ will be generated by a numerical method, which will be said to be convergent if

$$\lim_{h \to 0^+} \max_{n=0}^{N} \|\boldsymbol{e}_{h,n}\| = 0,$$

where $\|\cdot\|$ is a generic norm on $\mathbb{R}^d$.

Forward Euler Method

The simplest numerical scheme for the solution of ODEs is the Forward Euler Method.

$$\boldsymbol{u}_h(t_{n+1}) = \boldsymbol{u}_h(t_n) + h\boldsymbol{f}(t_n, \boldsymbol{u}_h(t_n)), \qquad 0 \leq n < N \tag{4}$$

with initial condition $\boldsymbol{u}_h(0) = \boldsymbol{u}_0$. If $\boldsymbol{f}$ is analytic, it can be shown that the Forward Euler Method is convergent and

$$E(h) = \max_{n=0}^{N} \|\boldsymbol{e}_{h,n}\| = O(h).$$

For this reason, the Forward Euler method is said to be *an order* 1 *method*. This method is didactically interesting but has poor convergence properties. We recall that this method may suffer from numerical instabilities, hence the step size h must be set to a sufficiently small value during computations.

Explicit Runge–Kutta 4 method

A popular numerical scheme is the so-called *Explicit Runge–Kutta 4* method, whose order is 4. Starting from an approximate solution $\boldsymbol{u}_h(t_n)$, the Runge–Kutta 4 method generates a new approximation by computing 4 intermediate vectors $\{\boldsymbol{k}_s\}_{s=1}^4$, which are then combined to form $\boldsymbol{u}_h(t_{n+1})$.
For each $0 \leq n < N$, the method computes:

$$\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{f}\big(t_n, \boldsymbol{u}_h(t_n)\big) \\
\boldsymbol{k}_2 &= \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{u}_h(t_n) + \frac{h}{2}\boldsymbol{k}_1\right) \\
\boldsymbol{k}_3 &= \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{u}_h(t_n) + \frac{h}{2}\boldsymbol{k}_2\right) \\
\boldsymbol{k}_4 &= \boldsymbol{f}\big(t_n + h, \boldsymbol{u}_h(t_n) + h\boldsymbol{k}_3\big) \\
\boldsymbol{u}_h(t_{n+1}) &= \boldsymbol{u}_h(t_n) + \frac{h}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)
\end{aligned} \tag{5}$$

with initial condition $\boldsymbol{u}_h(0) = \boldsymbol{u}_0$. As in the Euler method, the step size h must be set to a sufficiently small value during computations.

Neural field equation

In this coursework we will time-step a neural field equation, which is a popular model in neuroscience. For this coursework you need no prior knowledge of neuroscience or mathematical biology. I write below some background material for Question 3 which is, however, self-contained.

In the coursework you will discretise and solve numerically the following initial-value problem

$$\partial_t u(x,t) = -u(x,t) + \int_{-L}^{L} w(x,x')g(u(x',t))\,dx', \qquad (x,t) \in [-L,L] \times (0,T], \tag{6}$$

$$u(x,0) = q(x). \tag{7}$$

where L and T are positive real numbers, u is a real function of the spatial coordinate x and of the time coordinate t, and w, g and q are given functions. The first equation is a neural field equation, modelling the activity u of neurons at time t, and spatial location $x \in [-L, L] \in \mathbb{R}$.

The integral term sums all the inputs that neurons at positions $x' \in [-L, L]$ provide to neurons at position x. The function $w(x, x')$ models how neurons at position x' are wired to neurons at position x. In this coursework we will choose a specific function w, which will be given below (see (17)).

The integral contains also the term $g(u(x', t))$ which converts the activity of neurons at position x' into an input which is transmitted to neurons at position x, via the wiring function w. The function that operates this conversion is g, which will be specified below (see (18)).

How do we solve problem (6)–(7)? A possible strategy is to approximate (6)–(7) with a set of ODEs, and then use a time stepper (for instance the Runge–Kutta 4 scheme) to perform the time integration. We can obtain a set of ODEs approximating (6) in 2 steps, as follows

1. Select a quadrature rule to approximate the integral in x'. In this coursework we use the composite Simpson's rule with m subintervals, leading to

$$\partial_t \tilde{u}(x,t) = -\tilde{u}(x,t) + \sum_{j=0}^{m} w(x,x_j) g(\tilde{u}(x_j,t))\rho_j, \tag{8}$$

where m is assumed to be an **even integer**, x_j are the nodes defined by

$$x_j = -L + j\delta, \qquad \delta = 2L/m, \qquad j = 0,\dots,m. \tag{9}$$

and ρ_j the corresponding weights

$$\rho_j = \begin{cases} \delta/3 & \text{if } j = 0 \text{ or } j = m, \\ 4\delta/3 & \text{if } j = 1,3,\dots,m-1, \\ 2\delta/3 & \text{if } j = 2,4,\dots,m-2. \end{cases} \tag{10}$$

In passing we note that the tilde in (8) denotes the fact that this equation is an approximation to (6), induced by Simpson's quadrature rule.

2. Require that (8) holds at $x = x_i$, $i = 0,\dots,m$, leading to

$$\dot{\tilde{u}}(x_i,t) = -\tilde{u}(x_i,t) + \sum_{j=0}^{m} w(x_i,x_j) g(\tilde{u}(x_j,t))\rho_j, \qquad i = 0,\dots,m. \tag{11}$$

By introducing the vectors

$$\boldsymbol{u}(t) = \begin{bmatrix} \tilde{u}(x_0,t) \\ \vdots \\ \tilde{u}(x_m,t) \end{bmatrix}, \qquad \boldsymbol{q} = \begin{bmatrix} q(x_0) \\ \vdots \\ q(x_m) \end{bmatrix} \tag{12}$$

we arrive at the set of $m+1$ ODEs

$$\dot{u}_i(t) = -u_i(t) + \sum_{j=0}^{m} w(x_i,x_j) g(u_j(t))\rho_j, \qquad i = 0,\dots,m \tag{13}$$

$$u_i(0) = q_i \qquad i = 0,\dots,m \tag{14}$$

where the nodes $\{x_i\}$ are given in (9), the weights $\{\rho_i\}$ in (10), and w, g are given functions. System (13)–(14) can be written in the form (1)–(2), by defining the vector-valued function $\boldsymbol{f}(\boldsymbol{u})$ with components

$$f_i(\boldsymbol{u}) = -u_i(t) + \sum_{j=0}^{m} w(x_i,x_j) g(u_j(t))\rho_j, \qquad i = 0,\dots,m. \tag{15}$$

In Figure 1 you can find an example of a numerical solution of the set of ODEs above, showing an approximated spatio-temporal profile $u(x,t)$.

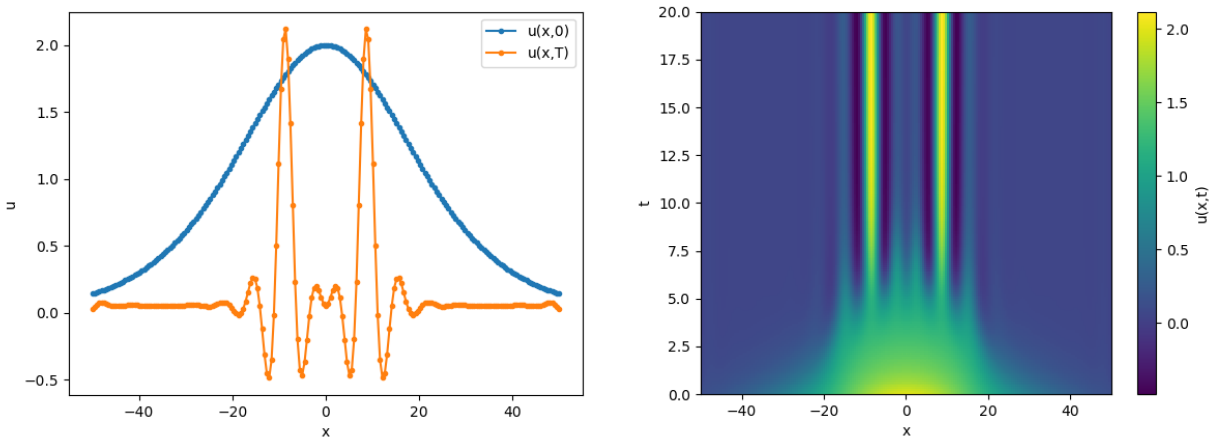

Figure 1: A simulation obtained using w, g as specified in (17) and (18), respectively, with $m = 200$, $A_1 = 0.1$, $A_2 = 1.0$, $b = 0.4$, $\mu = 3.5$, $\theta = 1$, $q(x) = 2/\cosh^2(0.04x)$, $L = 50$, $T = 20$. **You are not requested to reproduce these results**. To the left I plot the initial state $u(x, 0) = q(x)$ and final profile $u(x, T)$. To the right you can see a color map of the full solution $u(x, t)$ for $(x, t) \in [-L, L] \times [0, T]$: the activity is initially high around $x = 0$, and it "spreads" towards the periphery, Around $t = 7.5$, 2 strongly localised peaks of activity form and persist: the presence of these bumps is due to the nonlinearity of the problem, which makes it interesting and rich from a biological, numerical and analytical viewpoint.

Coursework questions

In `Templates/` you will find a set of directories, one for each question. The directories contain fully working codes (`.hpp`, `.cpp` `.py` files) as well as emtpy files, which you must edit for the coursework. You can use any software you want for plotting, so you don't have to use the python plotters (`.py` files) provided in the templates. **You must keep the directory structure and all file names as they are in the templates:** the directory `Q1` in your submission, for instance, should be self contained, and should include all the code necessary to compile and reproduce your result for Question 1. The template directories may also serve as a checklist for your submission. As part of your submission, you may also add files to the directories (for instance new classes, output files, plotting routines, etc.): if you do so, then write a brief `README.txt` file, with a short explanation of each new file. When you attack Question 2, make a new directory, and put all the files necessary to reproduce your results in the new folder; if needed, copy some files from `Q1` to `Q2`. **In other words, use the same workflow that we have used in Units 7–12.**

1. In this question you will use the Forward Euler method to solve the scalar initial value problem

$$\begin{aligned} \dot{u}(t) &= \frac{1}{a + t^2} \qquad t \in [0, T], \quad a, T > 0 \\ u(0) &= 0, \end{aligned} \tag{16}$$

whose solution is

$$u(t) = \frac{1}{\sqrt{a}} \tan^{-1} \frac{t}{\sqrt{a}} \qquad t \in [0, T].$$

Familiarise yourself with the classes `Vector` and `ODEInterface`. The class `Vector` is a modification of the class that has previously been used in Unit 12 on Iterative Linear Solvers. The class `ODEInterface` is an abstract class that encapsulates an interface to problems of type (1).

(a) Write an abstract class `AbstractODESolver` which contains the following members:

- Protected variables for initial and final times

```
double mFinalTime;
double mInitialTime;
```

- A protected pointer for the ODE system under consideration

```
ODEInterface* mpODESystem;
```

- A protected variable for the stepsize h

```
double mStepSize;
```

- A pure virtual public method

```
virtual void Solve() = 0;
```

- Any other member that you choose to implement. Your choices and your ability to design this class according to object-orientation design principles will be assessed.

[10]

(b) Write a class `LinearODE` derived from `ODEInterface` which

- Overrides the virtual method `ComputeF` in order to evaluate the right-hand side of (16).
- Overrides the virtual method `ComputeAnalyticSolution`.

[3]

(c) Write a class `ForwardEulerSolver`, derived from `AbstractODESolver`, with the following members

- A public constructor

```
ForwardEulerSolver(ODEInterface& anODESystem,
    const Vector& initialState,
    const double initialTime,
    const double finalTime,
    const double stepSize,
    const std::string outputFileName="output.dat",
    const int saveGap = 1,
    const int printGap = 1);
```

- A public solution method

```
void Solve();
```

which computes $\{\boldsymbol{u}_h(t_n)\}$ using the Forward Euler method of a **generic** system of ODEs of the form (1)–(2), saves **selected elements** of the sequences $\{t_n\}$, $\{\boldsymbol{u}_h(t_n)\}$ on a file, prints on screen an initial header and **selected elements** of the sequences $\{t_n\}$, $\{\boldsymbol{u}_h(t_n)\}$: the method should save to file every `saveGap` iterations and it should print on screen every `printGap` iterations; all components of the solution vector $\boldsymbol{u}_h \in \mathbb{R}^d$ should be saved on file, but only $\min(3, d)$ components should be printed on screen, to avoid screen clutter.

- Any other member that you choose to implement. Your choices and your ability to design this class according to object-orientation design principles will be assessed.

[6]

(d) Write and execute a main `Driver.cpp` file which

- Solves the ODE (16) for $a = 1$, $T = 10$, using the Forward Euler Method with $h = 0.01$, and outputs the solution to a file.
- Solves the ODE (16) with $a = 1$, $T = 2$ using the Forward Euler Method with various values of h of your choice, computes the corresponding errors $E(h)$ and saves the sequences $\{h_k\}$, $\{E(h_k)\}$ on a file.

Plot the approximate solution $u_h(t)$ for $t \in [0, 10]$. Plot $E(h)$ as a function of h, producing numerical evidence that $E(h) = O(h)$.

[5]

2. In this question, you will re-use some of the classes developed in Question 1 to solve the initial value problem (16) with the Runge–Kutta 4 method.

(a) Write a class `RungeKutta4Solver` derived from `AbstractODESolver` with the following members:

- A public constructor

```
RungeKutta4Solver( ODEInterface& anODESystem,
    const Vector& initialState,
    const double initialTime,
    const double finalTime,
    const double stepSize,
    const std::string outputFileName="output.dat",
    const int saveGap=1,
    const int printGap=1);
```

- A public solution method

```
void Solve();
```

which computes $\{\boldsymbol{u}_h(t_n)\}$ using the Runge–Kutta method of a **generic** system of ODEs, saves **selected elements** of the sequences $\{t_n\}$, $\{\boldsymbol{u}_h(t_n)\}$ to a file, prints on screen an initial header and **selected elements** of the sequences $\{t_n\}$, $\{\boldsymbol{u}_h(t_n)\}$: the method should save to file every `saveGap` iterations and it should print on screen every `printGap` iterations; all components of the solution vector $\boldsymbol{u}_h \in \mathbb{R}^d$ should be saved on file, but only $\min(3, d)$ components should be printed on screen, to avoid screen clutter.

- Any other member that you choose to implement. Your choices and your ability to design this class according to object-orientation design principles will be assessed.

[14]

(b) Write and execute a main `Driver.cpp` file which

- Solves the ODE (16) for $a = 1$, $T = 10$, using the Runge–Kutta 4 Method with $h = 0.01$, and outputs the solution to a file.
- Solves the ODE (16) with $a = 1$, $T = 2$ using the Runge–Kutta 4 Method with various values of h, computes the corresponding errors $E(h)$ and saves the sequences $\{h_k\}$, $\{E(h_k)\}$ on a file.

Plot the approximate solution $u_h(t)$ for $t \in [0, 10]$. Plot $E(h)$ as a function of h, producing numerical evidence that $E(h) = O(h^4)$. Your choices to present this evidence will be assessed.

[6]

3. Let $A_1, A_2, b, \mu, \theta, \alpha, L, T \in \mathbb{R}$, and let $m \in \mathbb{N}$ be even. Further, let

$$w(x, x') = A_1 e^{-|x-x'|^2} + A_2 e^{-b|x-x'|}(b \sin|x - x'| + \cos|x - x'|), \tag{17}$$

$$g(u) = \frac{1}{1 + \exp[-\mu(u - \theta)]}, \tag{18}$$

$$q(x) = \frac{4}{\cosh^2(\alpha x)}. \tag{19}$$

$$u_*(x) = \theta - \frac{1}{\mu} \ln\left(\frac{e^{x^2}}{0.8} - 1\right) \tag{20}$$

$$v_*(x) = -u_*(x) + 0.8\sqrt{\frac{\pi}{8}} \exp\left(-\frac{x^2}{2}\right)\left[\operatorname{erf}\left(\frac{2L - x}{\sqrt{2}}\right) + \operatorname{erf}\left(\frac{2L + x}{\sqrt{2}}\right)\right] \tag{21}$$

where erf denotes the error function (https://en.wikipedia.org/wiki/Error_function). Recall the C++ functions `erf`, `log`, `cosh` and `exp`, which can be loaded via the headers `math.h` or `cmath` and will be useful in this question.

This question concerns the solution of the following initial-value problem

$$\partial_t u(x, t) = -u(x, t) + \int_{-L}^{L} w(x, x') g(u(x', t))\, dx', \qquad (x, t) \in [-L, L] \times (0, T], \tag{22}$$

$$u(x, 0) = q(x). \tag{23}$$

and its approximation by the set of $m + 1$ ODEs (see discussion leading to (12)–(15)).

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(\boldsymbol{u}(t)), \qquad \boldsymbol{u}(0) = \boldsymbol{q} \tag{24}$$

where, for all $i = 0, \ldots, m$

$$u_i(t) \approx u(x_i, t), \qquad q_i = q(x_i), \qquad f_i(\boldsymbol{u}) = -u_i(t) + \sum_{j=0}^{m} w(x_i, x_j) g(u_j(t)) \rho_j,$$

and

$$x_i = -L + i\delta, \qquad \delta = 2L/m, \qquad \rho_i = \begin{cases} \delta/3 & \text{if } i = 0 \text{ or } i = m, \\ 4\delta/3 & \text{if } i = 1, 3, \ldots, m - 1, \\ 2\delta/3 & \text{if } i = 2, 4, \ldots, m - 2. \end{cases}$$

In order to test our codes, we will use the following result: if $A_1 = 1$ and $A_2 = 0$, then

$$v_*(x) = -u_*(x) + \int_{-L}^{L} w(x, x') g(u_*(x'))\, dx'. \tag{25}$$

(a) Write a class `NeuralField1D` derived from `ODEInterface` with the following members:

- A private member

```
Vector mParameters;
```

containing the vector of parameters $(A_1, A_2, b, \mu, \theta)$ and, if your implementation requires it, other problem parameters.

- A public method overriding the virtual method `ComputeF` of `ODEInterface`

```
void ComputeF( const double t, const Vector& u,
   Vector& f ) const;
```

This method computes and stores in `f` the right-hand side $\boldsymbol{f}$ of the $m+1$ ODEs (24). Note that the argument `u` is `const`, yet its components are accessible via one of the methods of the class `Vector`, with which you should be familiar before attempting this question.

- Any other method that you choose to implement. Your choices and your ability to design this class according to objection-orientation design principles will be assessed.

[22]

(b) Write and execute a main `Driver.cpp` file which:

i. Tests the method `ComputeF` of the class `NeuralField1D` using the identity (25), in order to ensure that the neural field has been discretised correctly. More specifically, set $A_1 = 1$, $A_2 = 0$, b equal to an arbitrary real number, $\mu = 3.5$, $\theta = 1.0$, $m = 2000$, $L = 50$, construct the vectors $\boldsymbol{u}_*, \boldsymbol{v}_* \in \mathbb{R}^{m+1}$ with components $\{u_*(x_i)\}_{i=0}^m$ and $\{v_*(x_i)\}_{i=0}^m$, respectively, calculate $\boldsymbol{f}(\boldsymbol{u}_*)$ using the method `ComputeF` of the class `NeuralField1D`, and verify that

$$\|\boldsymbol{v}_* - \boldsymbol{f}(\boldsymbol{u}_*)\|_\infty < 10^{-13}, \quad \text{where} \quad \|\boldsymbol{u}\|_\infty = \max_{i=0}^{m} |u_i| \qquad \boldsymbol{u} \in \mathbb{R}^{m+1}.$$

`Driver.cpp` must print on screen a message containing $\|\boldsymbol{v}_* - \boldsymbol{f}(\boldsymbol{u}_*)\|_\infty$.

[8]

ii. Solves the the initial-value problem (22)–(23), discretised with $m = 300$ for $A_1 = 0$, $A_2 = 1$, $b = 0.4$ $\mu = 3.5$, $\theta = 1$, $\alpha = 0.1$, $L = 50$, $T = 10$, using the Runge–Kutta method with $h = 0.01$. Your code should print and and save the solution at $t = 0$ and $t = T$.

[10]

iii. Repeats the compuation of question 3.(b).ii with $\alpha = 0.4$, and all other parameters unchanged. Your code should print and and save the solution at $t = 0$ and $t = T$.

[6]

(c) Using the data obtained in question 3.(b).ii, plot the initial and final solution profiles $u_h(x, 0)$ and $u_h(x, T)$ as functions of x (similarly to what is shown in the left panel of Figure 1).

[5]

(d) Using the data obtained in question 3.(b).iii, plot the initial and final solution profiles $u_h(x, 0)$ and $u_h(x, T)$ as functions of x (similarly to what is shown in the left panel of Figure 1).

[5]