MATH49111/MATH69111
Mini Projects Part 2
Contents

Guidelines v

Deadlines vii

1 Galerkin Methods 1
  1.1 Theory 1
    1.1.1 Approximating the unknown, \( u \) 1
    1.1.2 Formulating the problem 2
    1.1.3 Description of the algorithm 5
  1.2 Exercises 5
  1.3 Report 7

2 Elliptic PDEs 9
  2.1 Theory 9
    2.1.1 Poisson’s Equation 9
    2.1.2 The Linear Algebra Problem 10
    2.1.3 The SOR algorithm 11
    2.1.4 Line Relaxation 12
  2.2 Exercises 13
  2.3 Channel Flow 17
  2.4 Report 17

3 Circular Cylinders 19
  3.1 Background 19
    3.1.1 Thomas’s tridiagonal algorithm 19
    3.1.2 Solution of parabolic equations 20
  3.2 Impulsively started circular cylinder flow 21
  3.3 Report 22

4 Battery Storage 25
  4.1 Parabolic PDEs 25
    4.1.1 Solving Tridiagonal Systems 27
    4.1.2 Exercises 27
  4.2 Battery Storage Model 28
  4.3 Report 30
5 American Options 33
  5.1 Theory .................................................. 33
      5.1.1 Black-Scholes Pricing Framework ................. 33
      5.1.2 Formulation ....................................... 34
      5.1.3 Matrix Solver ................................. 36
  5.2 Exercises .............................................. 37
  5.3 Convertible Bond ................................. 38
  5.4 Report .................................................. 38

6 Revenue Management 41
  6.1 Theory – The Carpark Model ......................... 41
      6.1.1 Generating events from a Poisson Process ......... 42
      6.1.2 Generating a Booking ........................... 43
      6.1.3 Optimal Revenue Management ................. 43
  6.2 Exercises .............................................. 44
      6.2.1 The Customers class ........................... 44
      6.2.2 The Carpark Class ............................ 45
      6.2.3 Revenue Management .......................... 45
  6.3 Report .................................................. 46

7 Gillespie Algorithm 47
  7.1 Theory .................................................. 48
      7.1.1 An ODE model: chemical rate equations .......... 48
      7.1.2 Markov formulation ............................ 49
      7.1.3 The Gillespie algorithm ........................ 54
      7.1.4 Random numbers ................................ 54
  7.2 C++ Implementation .................................. 57
      7.2.1 Reactions ....................................... 57
      7.2.2 The GillespieSimulation class ................. 57
      7.2.3 Remarks ......................................... 57
  7.3 Exercises .............................................. 62
  7.4 Report .................................................. 68

8 Matrix–Vector Mult 71
  8.1 Sparse Matrix Storage Schemes ..................... 71
  8.2 Coding, Examples and Exercises .................... 73
      8.2.1 Creating the CSRMatrix class ................. 73
      8.2.2 Reading Large Matrices ....................... 74
      8.2.3 Creating the BCSRMatrix class ............... 75
      8.2.4 Evaluation ..................................... 75
  8.3 Report .................................................. 76
Guidelines

Choose only one project from the eight possible projects in this booklet to complete and hand in. If you are on the MSc in Applied you can choose from any of the projects. If you are on the MSc in QFFE then you must choose from projects 4-6. Projects are written to be self contained but you may contact the author of the project for clarification on the material. This is not a group work assignment and all codes and text must be written by you individually. Both the report and codes will be run through the TurnItIn system to automatically check for similarities between your work and that of others on the course. Please see the university guidelines on plagiarism.

All reports should be submitted in typed form using latex (preferred) or a word processing package and contain a title page with your student ID number. The submission should take place online at the appropriate section of the university Blackboard system. Any codes used to generate results in the report should be included in the appendix, you may be asked to supply electronic copies of the code if required. Anonymous marking is used for marking the reports so please do not include any names which may be used to identify the author of the report. The written part of the report should not exceed 25 pages in length and the font size should not be less than 11pt.

The report should be well structured containing

- an introduction;
- a section discussing the problem formulation;
- a description of the numerical techniques used;
- a results and conclusions section.

Any articles cited need to be included in a references section. Figures can be included anywhere in the text, but not before they are referenced, and should be clearly labelled. All figures in the report need to be cited in the text somewhere. Small portions of code listings may be included in the text for illustrative purposes but the full code should be left in the appendices. The report will be marked in accordance with the criteria as described in the attached marking scheme handout. Plagiarism is taken very seriously, so unless the project description mentions group working, you need to work independently. Do not copy/paste code or text from other peoples work. Reports containing results from codes which are not what one may expect from the code listing will be severely penalised.

As part of the assessment, comprising 10% of the mark, you will be asked to demonstrate your codes in a lab session. Details of what is to be expected will be released nearer the time.
Mark Scheme

Presentation (max 15)
The report should be well structured with an adequate introduction of the problem being solved, a problem formulation section, results and conclusions section. It should be largely free from typographical and grammatical errors. The figures and tables should be properly referenced and labelled with suitable captions. References should be given in detail.

Content (max 45)
The report should be factually accurate and the mathematical language precise and clear. Is there a clear description of the project with a clear outcome to be achieved? Any conclusions made need to be well supported and coherent. For example, if the solution obtained is claimed to be accurate, what grid size checks have been made, or what evidence is presented to validate the code. Is the original problem solved correctly, i.e., are the results credible. Does the report give evidence of the methods used to solve the problem and correct implementation. Is there evidence of an understanding and competent application of the range of techniques and methods used in the project, and evidence of technical skills. Does the student understand the results and interpret them in the correct manner. Overall, does the candidate understand the meaning, context and significance of the work being presented, or are there huge gaps.

Coding (max 30)
Is the code easy to understand with explanatory comments. Is the code easy to read, in separate files with short functions. Are there any obvious memory leaks. Is there evidence of OO programming, code reuse, use of standard/external libraries. Has the problem been approached in a novel manner with original ideas.

Lab tests (max 10)
The student should be able to run up to 3 different tests with the code to generate expected results. Marks will be awarded for the code compiling, running successfully and the ease with which the student can change between tests.
Deadlines

- Your code will be tested at the lab session 12-2pm on: Friday 14\textsuperscript{th} December 2012.
- The written report for the first mini project is due in by: 5pm Monday 17\textsuperscript{th} December 2012.
Mini Project 1

Discontinuous Galerkin Methods for Convection-dominated problems

Author: Dr. Andrew Hazel, andrew.hazel@ma.man.ac.uk

Consider the one-dimensional transport equation of a scalar field, \( u(x,t) \),

\[
\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0, \tag{0.1}
\]

where \( f(u) \) is a flux function, the time \( t \in [0, \infty) \) and \( x \in [a, b] \) is a bounded spatial domain.

We shall investigate its solution in two special cases:

- \( f(u) = Cu \), where \( C \) is a constant — the advection equation,
- \( f(u) = \frac{1}{2} u^2 \) — the inviscid Burgers’ equation.

In fact, it is very hard to solve these (hyperbolic) equations numerically. In this project, you will implement a method known as the discontinuous Galerkin method that can be used to solve such equations accurately. In particular, it can handle any discontinuities (shocks) that may develop. The use of C++ classes makes it easy to modify the code to handle different equations of the same form by overloading the flux function, \( f \), in different classes.

1.1 Theory

1.1.1 Approximating the unknown, \( u \)

The general approach is to split the spatial domain into \( N \) “elements” each consisting of two “nodes” \( x_0 \) (the node on the left) and \( x_1 \) (the node on the right).\(^1\) In order to distinguish nodes that belong to different elements we add an additional index so that \( x_0^e \) is the left-hand node of the \( e \)-th element. We label the elements from left to right and if the domain is to be connected, it follows that the right-hand node of one element must be equal to the left-hand node of the element to its right, \( x_1^{e-1} = x_0^e \), see Figure 1.

\(^1\)In order to aid the translation into C++ we shall start all indices from zero.
MINI PROJECT 1. GALERKIN METHODS

Figure 1.1: The region $x \in [0, 1]$ is divided into three (uniform) elements. The left-hand end of element $e$ is located at $x^e_0$ and the right-hand end is at $x^e_1$. Note that geometric continuity requires that $x^{e-1}_1 = x^e_0$.

In fact, it is most convenient to work in a local (elemental) coordinate $s$, scaled so that $s \in [-1, 1]$ in each element. Thus, in element $e$, we can write a linear approximation to the global coordinate:

$$x = \frac{1}{2} \left[ (x^e_0 + x^e_1) + s(x^e_1 - x^e_0) \right],$$

or, equivalently,

$$x = x^e_0 \left[ \frac{1}{2} (1 - s) \right] + x^e_1 \left[ \frac{1}{2} (1 + s) \right] = x^e_0 \psi_0(s) + x^e_1 \psi_1(s),$$

(1.2)

where

$$\psi_0(s) = \frac{1}{2} (1 - s) \quad \text{and} \quad \psi_1(s) = \frac{1}{2} (1 + s),$$

(1.3)

are known as the local interpolation (shape) functions.

In each element, we require an approximation to the function $u(x, t)$. We are free to use any suitable method, but the easiest and (most) obvious approach is to interpolate $u$ using the geometric shape functions, $\psi$. Thus, in element $e$

$$u(x, t) = u^e_0 \psi_0(s) + u^e_1 \psi_1(s),$$

(1.4)

where $u^e_i$ is the value of $u$ at the position $x^e_i$. In other words, we approximate the unknown $u$ using linear interpolation within each element.

1.1.2 Formulating the problem

Galerkin methods are based on the so-called “weak form” of the differential equation (0.1), obtained on multiplication by a test function, $v(x)$, and integrating over the problem domain:

$$\int \left[ \frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} \right] v(x) dx = 0.$$  

(1.5)

We note that if $u$ is a solution of the original equation (0.1) then it is also a solution of the “weak form”. On the other hand, if we insist that equation (1.5) must be satisfied for every (sensible)$^2$ test function, $v(x)$, then the solution of equation (1.5) will also be a solution of the “strong form” (0.1).

$^2$by sensible, of course, we mean satisfying suitable integrability and smoothness constraints
1.1. THEORY

In the discontinuous Galerkin method, we integrate the equation over each element separately. In element $e$, we have:

\[
\int_{x_0}^{x_1} \left[ \frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} \right] v \, dx = 0. 
\]  
(1.6)

We next integrate the second term by parts to obtain

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} v - f(u) \frac{\partial v}{\partial x} \right) \, dx + \left[ f(u) v \right]_{x_0}^{x_1} = 0,
\]

\[
\Rightarrow \int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} v - f(u) \frac{\partial v}{\partial x} \right) \, dx + f(u^e_1)v(x_1^e) - f(u^e_0)v(x_0^e) = 0; 
\]  
(1.7)

the final two terms represent the net flux out of the element.

In a continuous formulation the unknown $u$ must be the same in neighbouring elements, i.e. $u_{1-1}^e = u_0^e$ — the same constraint that we impose on the coordinate, $x$.

In a discontinuous formulation, this constraint is relaxed and the flux is replaced by a numerical flux, $h(u_{1-1}^e, u_0^e)$, that is used to approximate $f(u_0^e)$. The exact choice of numerical flux function depends on the equation being solved. The governing equation in each element is

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} v - f(u) \frac{\partial v}{\partial x} \right) \, dx + h(u_1^e, u_0^{e+1})v(x_1^e) - h(u_{1-1}^e, u_0^e)v(x_0^e) = 0. 
\]  
(1.8)

Our linear elements each contain two unknowns and so we require two equations to determine these unknown values. The two equations are obtained from equation (1.8) by using two different test functions. In Galerkin methods, the two test functions are precisely the two shape functions (1.3) used to interpolate the unknown. Thus, the two discrete equations that must be solved in each element are:

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} \psi_0 - f(u) \frac{\partial \psi_0}{\partial x} \right) \, dx + h(u_1^e, u_0^{e+1})\psi_0(x_1^e) - h(u_{1-1}^e, u_0^e)\psi_0(x_0^e) = 0, \tag{1.9a}
\]

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} \psi_1 - f(u) \frac{\partial \psi_1}{\partial x} \right) \, dx + h(u_1^e, u_0^{e+1})\psi_1(x_1^e) - h(u_{1-1}^e, u_0^e)\psi_1(x_0^e) = 0. \tag{1.9b}
\]

The definition of the shape functions (1.3) implies that $\psi_0(x_0^e) = \psi_0(s = -1) = 1$, $\psi_0(x_1^e) = \psi_0(s = 1) = 0$, $\psi_1(x_0^e) = \psi_1(s = -1) = 0$ and $\psi_1(x_1^e) = \psi_1(s = 1) = 1$. Thus, the governing equations are

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} \psi_0 - f(u) \frac{\partial \psi_0}{\partial x} \right) \, dx = h(u_{1-1}^e, u_0^e) = 0, \tag{1.10a}
\]

\[
\int_{x_0}^{x_1} \left( \frac{\partial u}{\partial t} \psi_1 - f(u) \frac{\partial \psi_1}{\partial x} \right) \, dx = h(u_1^e, u_0^{e+1}) = 0. \tag{1.10b}
\]

or

\[
\int_{x_0}^{x_1} \frac{\partial u}{\partial t} \psi_0 \, dx = \int_{x_0}^{x_1} f(u) \frac{\partial \psi_0}{\partial x} \, dx + h(u_{1-1}^e, u_0^e), \tag{1.11a}
\]

\[
\int_{x_0}^{x_1} \frac{\partial u}{\partial t} \psi_1 \, dx = \int_{x_0}^{x_1} f(u) \frac{\partial \psi_1}{\partial x} \, dx - h(u_1^e, u_0^{e+1}). \tag{1.11b}
\]
We now use the discrete representation of the continuous unknown \( u = u_0^e \psi_0 + u_1^e \psi_1 \) in the time derivative terms

\[
\int_{x_0^e}^{x_1^e} \frac{\partial}{\partial t} \left( u_0^e \psi_0 + u_1^e \psi_1 \right) \psi_0 \, dx = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_0}{\partial x} \, dx + h(u_1^{e-1}, u_0^e), \tag{1.12a}
\]

\[
\int_{x_0^e}^{x_1^e} \frac{\partial}{\partial t} \left( u_0^e \psi_0 + u_1^e \psi_1 \right) \psi_1 \, dx = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_1}{\partial x} \, dx - h(u_1^e, u_0^{e+1}). \tag{1.12b}
\]

The shape functions are independent of time and hence

\[
\int_{x_0^e}^{x_1^e} \dot{u}_0^e \psi_0 \psi_0 \, dx = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_0}{\partial x} \, dx + h(u_1^{e-1}, u_0^e), \tag{1.13a}
\]

\[
\int_{x_0^e}^{x_1^e} \dot{u}_1^e \psi_0 \psi_1 \, dx = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_1}{\partial x} \, dx - h(u_1^e, u_0^{e+1}). \tag{1.13b}
\]

where \( \dot{u}^e \) denotes the time-derivative of the discrete value \( u^e \).

Representing these equations in a matrix form, we obtain

\[ M^e \dot{U}^e = F^e(U^e) + H(U^{e-1}, U^e, U^{e+1}), \]

where the vectors of unknowns are

\[ U^e = \begin{pmatrix} u_0^e \\ u_1^e \end{pmatrix}, \quad \text{and} \quad \dot{U}^e = \begin{pmatrix} \dot{u}_0^e \\ \dot{u}_1^e \end{pmatrix}. \]

\( M^e \) is a matrix, conventionally known as the mass matrix, and \( F^e(U^e) \) is a vector-valued flux function:

\[ M^e_{ij} = \int_{x_0^e}^{x_1^e} \psi_i \psi_j \, dx \quad \text{and} \quad F(U^e)_i = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_i}{\partial x} \, dx. \]

Remember that according to our C-style index convention \( i, j \in \{0, 1\} \). The remaining vector-valued function is

\[ H(U^{e-1}, U^e, U^{e+1}) = \begin{pmatrix} h(u_1^{e-1}, u_0^e) \\ -h(u_1^e, u_0^{e+1}) \end{pmatrix}. \]

Multiplying through by the inverse of the mass matrix gives

\[ \dot{U}^e = (M^e)^{-1} \left[ F^e(U^e) + H(U^{e-1}, U^e, U^{e+1}) \right], \tag{1.14} \]

and we can use an explicit time-stepper to solve this coupled \((2 \times 2)\) system of ordinary differential equations. You may use any of the timestep classes that you may have created in previous projects. Alternatively, if we use simple first-order finite differences to approximate the time derivative, we obtain the iterative scheme:

\[ U^{e(n+1)} = U^{e(n)} + \Delta t (M^e)^{-1} \left[ F^e(U^{e(n)}) + H(U^{e-1(n)}, U^{e(n)}, U^{e+1(n)}) \right], \tag{1.15} \]

where \( U^{e(n)} \) is the solution at the \( n \)-th discrete time level and \( \Delta t \) is the fixed time increment. The system (1.15) must be assembled and solved for each element, but the computation of the function \( H \) requires information from neighbouring elements.
1.2. EXERCISES

1.1.3 Description of the algorithm

Initialisation

- Create \(N\) elements and for each element \(e\) assign spatial coordinates \(x^e_0\) and \(x^e_1\) and an initial guess for the unknowns \(u^e_0\) and \(u^e_1\).
- Setup the connectivity information for each element (assign its left and right neighbours).

Timestepping loop

- Loop over all elements and for each element \(e\):
  - Calculate the mass matrix, \(M^e_{ij}\), and flux vectors, \(F^e(U^e)\) and \(H(U^{e-1}, U^e, U^{e+1})\).
  - Assemble the system of ODEs (1.14).
  - Perform on timestep of the linear system (1.14) and store the results for the advanced time in temporary storage.
- Loop over all elements again and update the current unknown values to the previously stored advanced-time values.

Thus, the algorithm requires a method for numerical integration over each element, e.g. Gauss rule, Trapezium rule, etc, and matrix inversion and multiplication routines. You may (should) reuse classes from previous projects to perform these tasks.

1.2 Exercises

You are advised to go through the following exercises to aid in your understanding of the method

Initialisation

1. Write a C++ class called \texttt{AdvectionElement} that contains storage for two positions and two unknowns, all of which should be double precision variables. Your class should include two functions \texttt{interpolated\_x(s)} and \texttt{interpolated\_u(s)} that return the values \(x(s)\) and \(u(s)\) by implementing the equations (1.2) and (1.4) respectively; for example

```cpp
class AdvectionElement
{
    public:
        // Pointer to the left neighbour
        AdvectionElement *Left_neighbour_pt;

        // Pointer to the right neighbour
        AdvectionElement *Right_neighbour_pt;

        // Storage for the coordinates
        std::vector<double> X;

        // Storage for the unknowns
        std::vector<double> U;

        // Constructor: initialise the vectors to hold two entries.
        AdvectionElement()
        {
        // Resize the vectors to hold two entries each
        // FILL THIS IN
```
2. Test your class by writing a \texttt{main()} function that creates $N$ uniformly-spaced elements in the domain $x \in [0, 2\pi]$. Make sure that you set the neighbour pointers correctly. Set periodic boundary conditions by connecting the first and last elements. In addition, set the value of the unknown to the function:

$$u = 1.5 + \sin(x) \quad (1.16)$$

Don’t forget that you will need to set the two values of the coordinates $x$ as well as the two unknowns $u$ in each element.

3. Produce graphs of the approximation of the function defined in equation (1.16) by plotting the coordinate and the unknown at the centre of each element for $N = 10, 100 & 200$ elements. What do you notice as you increase $N$? Is this what you expect

\textbf{Timestepping loop}

1. Show that the components of the mass matrix are

$$M_{ij}^e = \frac{(x_i^e - x_0^e)}{2} \int_{-1}^{1} \psi_i \psi_j \, ds,$$

where $s$ is the local coordinate in the element. Hence, show that

$$M^e = \frac{x_i^e - x_0^e}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix},$$

and find $(M^e)^{-1}$.

2. Show that the components of the flux vector are

$$F_i^e = \int_{-1}^{1} f(u) \frac{\partial \psi_i}{\partial s} \, ds$$

and hence that

$$F^e = \frac{1}{2} \int_{-1}^{1} f(u) \, ds \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

3. Add a function to the \texttt{AdvectionElement} class that calculates the flux for the simple advection equation at a value of the scalar field, $u$. In the first instance let $C = 1$. The function is \texttt{virtual} so that it can be overloaded later to solve Burgers’ equation.

// Calculate the flux
\texttt{virtual double flux(double u)}
\{
    \texttt{return //FILL THIS IN}
\}
4. Write a function that returns the integral of the flux function over the element using, for example, a two-point Gauss rule, which states that

\[ \int_{-1}^{1} g(s) \, ds \approx g\left(-\frac{1}{\sqrt{3}}\right) + g\left(\frac{1}{\sqrt{3}}\right), \]

for any function \( g(s) \).

```c
// Calculate the integral of the flux function over the element
// using the two-point Gauss rule
double integrate_flux() {
    // FILL THIS IN
}
```

5. Write a function \( h(a, b) \) that returns the numerical flux. A good general choice is the local Lax–Friedrichs flux:

\[
h(a, b) = \frac{1}{2}(f(a) + f(b)) - \frac{1}{2} \max_{a \leq s \leq b} |f'(s)|(b - a). \tag{1.17}
\]

Again this function is virtual so that it can be overloaded later.

```c
virtual double h(double a, double b) {
    return // FILL THIS IN
}
```

6. Finally, write a `timestep(dt)` function that calculates the updated values of the unknowns \( U \) using the scheme (1.15).

```c
void timestep(double dt) {
    // FILL THIS IN
}
```

Note that for the scheme to be explicit you must not update the values of the unknowns until the `timestep(dt)` function has been called for every element, because the numerical flux function must always use values at the present time from the neighbouring elements, see equation (1.15). Thus, you will need to provide additional storage for the values of the unknowns at the advanced time level and a mechanism to update all the values once the `timestep(dt)` function has been called for every element.

### 1.3 Report

You are required to use the Discontinuous Galerkin method with linear interpolation to find numerical solutions to the two equations:

\[
\begin{align*}
\frac{\partial u}{\partial t} + C \frac{\partial u}{\partial x} &= 0, \quad \text{(Advection equation)} \\
\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} &= 0, \quad \text{(inviscid Burgers’ equation).} \tag{1.18a-1.18b}
\end{align*}
\]

on the domain \( x \in [0, 2\pi] \) with periodic boundary conditions.

You should prepare a report in the form of a continuous piece of prose that describes the problem and your solution to it. In particular, you should include the following:
• Produce a graph of the solution of the advection equation (1.18a) with the sine-wave initial condition (1.16) at times $t = 0, 0.25, 0.5, 1$. Are your results consistent with the known exact solution of the advection equation? You should now perform grid and timestep resolution studies. What is an acceptable resolution?

• Produce graphs of the solution of the advection equation (1.18a) using a square-wave initial condition

$$u = \begin{cases} 
1 & 0 \leq x \leq 1, \\
0 & \text{otherwise.}
\end{cases} \quad (1.19)$$

What do you notice in this case? How do you think that this problem could be resolved?

• Solve Burgers’ equation (1.18b) for the sine-wave and square-wave initial conditions and produce graphs of the solutions for times in the range $t \in [0, 2]$. What happens to the initial profile as time increases? Is this what you expect?

[You should use inheritance to create a new BurgersElement that overloads the appropriate flux and numerical flux functions.]

Bibliography
Mini Project 2

Elliptic PDEs - Channel Flow

Author: Dr. Paul Johnson, Paul.Johnson-2@manchester.ac.uk

In this project we shall obtain a numerical solution to the problem of two-dimensional incompressible inviscid fluid flow in a channel. The aim is to solve Laplace’s equation, an elliptic PDE. We shall use iterative methods to solve a large sparse matrix problem. More information on the numerical methods associated with PDEs can be found in Smith (1985).

2.1 Theory

2.1.1 Poisson’s Equation

A prototype elliptic pde is Poisson’s equation given by

\[ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f(x, y), \]

where \( f(x, y) \) is a known/given function.

The equation has to be solved in a domain \( D \), say, with boundary conditions given on the boundary \( \partial D \) of \( D \) (see right). These can be of three types:

- Dirichlet \( \phi = g(x, y) \) on \( \partial D \).
- Neumann \( \frac{\partial \phi}{\partial n} = g(x, y) \) on \( \partial R \).
- Robin/Mixed \( B \left( \phi, \frac{\partial \phi}{\partial n} \right) = 0 \) on \( \partial D \).

Robin boundary conditions involve a linear combination of \( \phi \) and its normal derivative on the boundary. Mixed boundary conditions involve different conditions for one part of the boundary, and another type for other parts of the boundary.
Solving a Model Problem

Consider the problem
\[ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f(x, y), \quad 0 \leq x, y \leq 1 \] (1.1)
with boundary conditions
\[ \phi = 0 \quad \text{on} \quad \partial D. \]
Here the domain \( D \) is the square region \( 0 < x < 1 \) and \( 0 < y < 1 \).

Then we must construct a finite difference mesh with points \((x_i, y_j)\), where \( x_i = i \Delta x \), for \( i = 0, 1, \ldots, N \), and \( y_j = j \Delta y \), for \( j = 0, 1, \ldots, M \). Let \( w_{i,j} = \phi(i \Delta x, j \Delta y) \) be the numerical approximation at a grid node. Here \( \Delta x = 1/N \), and \( \Delta y = 1/M \) are the grid sizes in the \( x \) and \( y \) directions.

Next, replace the derivatives in Poisson equation by the discrete approximations to get:
\[ \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{\Delta x^2} + \frac{w_{i,j+1} - 2w_{i,j} + w_{i,j-1}}{\Delta y^2} = f_{i,j}, \] (1.2)
for \( 1 \leq i \leq N - 1 \), and \( 1 \leq j \leq M - 1 \).

At the boundary we have the equations
\[ w_{i,j} = 0, \quad \text{if} \quad i = 0, N \quad \text{and} \quad 0 \leq j \leq M \] (1.3)
\[ w_{i,j} = 0, \quad \text{if} \quad j = 0, M \quad \text{and} \quad 0 \leq i \leq N \] (1.4)
The result is a system of \((N - 1) \times (M - 1)\) equations with \((N - 1) \times (M - 1)\) unknowns, and the unknowns are \( w_{i,j} \) in the region \( D \).

2.1.2 The Linear Algebra Problem

Expressing the problem as a matrix equation

Let us write the solution \( w_{i,j} \) as
\[ w_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,M-1})^T \]
and also the right hand side of the equation as
\[ f_i = (f_{i,1}, f_{i,2}, \ldots, f_{i,M-1})^T \]

Then we can use this notation to write the problem in matrix form.
\[
\begin{bmatrix}
B & I & B \\
I & B & I \\
I & B & I
\end{bmatrix}
\begin{bmatrix}
w_1 \\
w_2 \\
w_3 \\
w_{N-1}
\end{bmatrix} = \Delta x^2
\begin{bmatrix}
w_1 \\
w_2 \\
w_{N-1}
\end{bmatrix}
\begin{bmatrix}
f_1 \\
f_2 \\
f_{N-1}
\end{bmatrix}
\]

In the above \( I \) is the \((M - 1) \times (M - 1)\) identity matrix The matrix \( B \) is given by
\[
B = \begin{bmatrix}
b & c & \\
a & b & c \\
a & b & \ddots \\
& \ddots & \ddots \\
& & a & b
\end{bmatrix}
\]
2.1. THEORY

where \( a = c = \beta^2 \), \( b = -2(1 + \beta^2) \) and \( \beta = \frac{\Delta x}{\Delta y} \).

Then let us write the linear system as

\[
Aw = f
\]

Observe that the matrix \( A \) is both very large and very sparse.

2.1.3 The SOR algorithm

Gauss-Seidel iteration

Rewrite the discrete equations

\[
\frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{\Delta x^2} + \frac{w_{i,j+1} - 2w_{i,j} + w_{i,j-1}}{\Delta y^2} = f_{i,j},
\]

for \( 1 \leq i \leq N - 1 \), and \( 1 \leq j \leq M - 1 \), as

\[
w_{i,j} = \frac{1}{2 + 2\beta^2} \left[ w_{i+1,j} + w_{i-1,j} + \beta^2(2(w_{i,j+1} + w_{i,j-1}) - \Delta x^2 f_{i,j}) \right]
\]

with \( \beta = \Delta x/\Delta y \).

This suggests the iterative scheme

\[
w_{i,j}^{q+1} = \frac{1}{2 + 2\beta^2} \left[ w_{i+1,j}^{q+1} + w_{i-1,j}^{q+1} + \beta^2(2(w_{i,j+1}^{q} + w_{i,j-1}^{q}) - \Delta x^2 f_{i,j}) \right]
\]

here \( w_{i,j}^{q} \) is the \( q \)th guess at the solution, and we have already calculated up-to-date values for \( w \) at \((i-1,j)\) and \((i,j-1)\).

Convergence Criteria

Suppose we write the linear system as

\[
Av = f
\]

where \( v \) is the exact solution of the linear system. If \( w \) is an approximate solution, the error \( e \) is defined by

\[
e = v - w.
\]

Thus

\[
Ae = A(v - w) = f - Aw.
\]

Then the residual is defined by

\[
r = f - Aw.
\]

Since we don’t know the exact solution but we can calculate the residual, and the residual is related to the error by \( e = 0 \) if and only if \( r = 0 \) we can choose the magnitude of the residual as the convergence criteria.

For the Gauss-Seidel scheme the residual is given by

\[
r_{i,j} = w_{i+1,j} + w_{i-1,j} - (2 + 2\beta^2)w_{i,j} + \beta^2(2w_{i,j+1} + w_{i,j-1}) - \Delta x^2 f_{i,j}
\]

Suitable stopping conditions might be

\[
\max_{i,j} |r_{i,j}| < \epsilon, \quad \text{or} \quad \sqrt{\sum_{i,j} r_{i,j}^2} < \epsilon
\]
Relaxation and SOR

Point iteration can be slow to converge, so a faster way to find the solution is to overestimate the correction at each point. We call this relaxation. We define

\[ w_{i,j}^{q+1} = (1 - \omega)w_{i,j}^q + \omega w_{i,j}^* \]

where \( w_{i,j}^* \) is the solution from the iteration scheme. \( \omega \) is the relaxation factor.

If \( \omega = 1 \) we just have the Gauss-Seidel scheme. If \( \omega > 1 \) with Gauss-Seidel then it is called successive overrelaxation of the SOR scheme. Theorems can show that \( 0 < \omega < 2 \) is a condition for a scheme to converge, see Burton and Faires (2004) for more details on finding the optimal choice of \( \omega \) for a scheme. If we choose \( \omega < 1 \) then we under-relax, and the convergence will be slower.

2.1.4 Line Relaxation

The SOR scheme is a point relaxation method. The Thomas tridiagonal solver gives an extremely efficient way to solve a whole line of new values using a direct method, and we can take advantage of this with line-relaxation. The idea is to take a set of nodes in a line, and assume that we have a guess to the solution either side, then solve the equations for those nodes simultaneously.

So then, keeping \( i \) constant, we may write the system of equations as

\[
\beta_2 w_{i,j+1}^{q+1} - (2 + 2\beta_2)w_{i,j}^{q+1} + \beta_2 w_{i,j-1}^{q+1} = \Delta x^2 f_{i,j} - (w_{i+1,j}^q + w_{i-1,j}^q) \quad (1.5)
\]

for \( 1 \leq j \leq M - 1 \). The resulting tri-diagonal system may be solved using a direct-solver such as the Thomas algorithm. We can also over-relax the solution

\[ w_{i,j}^{q+1} = (1 - \omega)w_{i,j}^q + \omega w_{i,j}^* \]

Thomas’s tridiagonal algorithm

The LSOR method relies on the use of the Thomas algorithm for the solution of tridiagonal equations, i.e., consider the system of equations:

\[
\alpha_j u_{j+1} + \beta_j u_j + \gamma_j u_{j+1} = \delta_j, \quad j = 0, 1, 2, \ldots, N, \quad (1.6)
\]

with \( \alpha_0 = 0, \gamma_n = 0 \). The coefficients \( \alpha_j, \beta_j, \gamma_j \) are known together with the right hand side \( \delta_j \) and we need to solve for \( u_j \) for \( j = 0, 1, \ldots, N \). We can rewrite these in matrix form as

\[
\begin{bmatrix}
\beta_0 & \gamma_0 \\
\alpha_1 & \beta_1 & \gamma_1 \\
0 & \alpha_2 & \beta_2 & \gamma_2 \\
& \ddots & \ddots & \ddots \\
& & \alpha_N & \beta_N \\
\end{bmatrix}
\begin{bmatrix}
u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \\
\end{bmatrix}
= 
\begin{bmatrix}
\delta_0 \\ \delta_1 \\ \delta_2 \\ \vdots \\ \delta_N \\
\end{bmatrix} \quad (1.7)
\]

Note that we can write (1.5) in this form, by letting \( u_j = w_{i,j} \).

The equations (1.6) can be solved using Thomas’s algorithm. This version of a tridiagonal solver is based on Gaussian elimination. First we create zeros below the diagonal and then once
we have a triangular matrix, we solve for the \( u_j \) using back substitution. Thus the algorithm takes the form

\[
\beta_j = \beta_j - \frac{\gamma_j u_j}{\beta_j}, \quad j = 1, 2, ..., N, \\
\delta_j = \delta_j - \frac{\delta_j - 1}{\beta_j}, \quad j = 1, 2, ..., N, \\
u_N = \frac{\delta_N}{\beta_N}, \quad u_j = \frac{(\delta_j - \gamma_j u_j + 1)}{\beta_j}, \quad j = N - 1, ..., 1, 0.
\]

(1.8)

2.2 Exercises

Here we shall try to abstract the problem so that we can define it in terms of a finite difference stencil at each point in the grid. In order for the SOR/LSOR algorithms to work, you need to define

- number and position of nodes in the grid;
- the position of the boundary, and the value of the solution at (or within) the boundary;
- the discretised form of the PDE equation we wish to solve at each node.

The idea is that the Stencil defined at each point in the grid will either be a boundary stencil (assigning the value of the solution) or a PDE stencil (solving the discretised form of the PDE). Therefore the minimum requirement to solve the problem in section 2.3 is to generate a domain class containing a stencil for each point in the grid that solves the particular problem. In this section we shall step through how to solve the model problem using the Stencil and Domain class.

1. First we must generate an array to store the solution. You may use the standard libraries for this. If we use vectors, then the declaration is as follows:

   \[
   \text{vector<vector<double> > v;} \\
   \text{for an empty matrix or} \\
   \text{vector<vector<double> > v(n,vector<double>(m));} \\
   \text{for a } n \text{ by } m \text{ matrix.}
   \]

2. Declare vectors to store the values \( x_i \) and \( y_i \) of a uniform grid in \( x \) and \( y \).

3. Now make a 101 by 101 grid on \( x \in [0, 1] \), and \( y \in [0, 1] \), and then set \( v(x, y) = \sin(\pi x)\cos(\pi y) \).

   Create a function to output \( x \), \( y \) and \( v \) to file, check that you can create 3D plots of \( v \).

Finite difference schemes work by relating the value at each point on the grid to surrounding points. Sometimes we call this a stencil. For Poisson type elliptic equations we need a five point stencil, to relate the point \((i, j)\) to the points above/below left/right. We store this relation in a class, and reference each point by \((s**s)\) as follows:

\[
\begin{array}{ccc}
  i - 1 & j - 1 & j \\
  \hline
  i & s_{21} & s_{11} \\
  i + 1 & s_{10} & s_{12} \\
  i - 1 & s_{01}
\end{array}
\]

Then for a stencil such as this the lhs of the general equation can be written

\[
s_{01}w_{i-1,j} + s_{10}w_{i,j-1} + s_{11}w_{i,j} + s_{12}w_{i,j+1} + s_{21}w_{i+1,j}
\]
MINI PROJECT 2. ELLIPTIC PDES

```c
struct Stencil
{
    double s01, s10, s11, s12, s21;
    Stencil(double a=0, double b=0, double c=1, double d=0, double e=0)
        : s01(a), s10(b), s11(c), s12(d), s21(e) {}
};
```

Listing 2.1: A stencil class to define the finite differencing scheme

1. Copy the definition for a Stencil class from listing 2.1 into your code.
2. Create a stencil and initialise its value in the constructor as follows:
   ```c
   Stencil s(1.,1.,-4.,1.,1.);
   ```
3. Output your stencil values to the screen. What are the values if you declare it as follows:
   ```c
   Stencil s;
   ```

Now we shall think about how to represent the domain of the problem using stencils. See figure 2.1 for a representation of a domain on a 6 by 5 grid, where the filled nodes represent the nodes closest to the boundary of the domain $dD$. If the grid is square and the boundary condition is to simply assign a value (such as $v = 0$) then we have the following stencils:

- On white nodes the stencil $s = (1, 1, -4, 1, 1)$
- On black nodes the stencil $s = (0, 0, 1, 0, 0)$

Now, assuming that we have a Domain variable $A$ that stores the stencil for each point $(i,j)$ in the grid, then in order to solve the equations we simply need to write

```c
for(int i=1;i<5;i++)
    for(int j=1;j<4;j++)
        v[i][j] = (d[i][j] - A(i,j).s01*v[i-1][j] ... // and so on for other terms
```

where care has been taken to not step on the outer boundary. When the solver gets to points (3,3) and (4,3) the stencil assignment makes this a boundary condition and sets $v_{i,j} = d_{i,j}$. In this way we can create any complex geometries in the domain we like, and the solution method can be independent of the domain.

A base class Domain is given in listing 2.2. In order to use this class we shall have to inherit from it and supply it with a function to specify the stencil at each point.

1. Using inheritance, create a UnitSquare class that inherits from Domain and sets up the stencils for the model problem defined in section 2.1.1. The constructor for the class should look like
   ```c
   UnitSquare(int n,int m):Domain(n,m){}
   ```
   Define an instance of UnitSquare in your main program and assign it the same size as your solution vector. Try to run your stencil setup function and check them for a small grid.

2. Now in your main program, write an iteration loop to solve the model problem with
   ```c
   f(x,y) = \frac{\sin(\pi x)\cos(\pi y)}{xy}
   ```
   using SOR.
2.2. EXERCISES

Figure 2.1: The Domain $D$ is contained within the dashed line. Filled nodes represent those nodes closest to the boundary.

```plaintext
class Domain{

protected:
    // store size of matrix
    int N,M;
    // store stencils
    std::vector<std::vector<Stencil>> A;

public:
    // create a n x M stencil matrix
    explicit Domain(int n,int m):N(n),M(m),A(n,std::vector<Stencil>(m)){};
    // virtual function to allow setup to be abstracted
    virtual void setupStencils(double dx,double dy)=0;
    // access stencils
    const Stencil& operator()(int i,int j) const {return A[i][j];}
    Stencil& operator()(int i,int j){return A[i][j];}
    // return size of matrix
    int iSize() const {return N;}
    int jSize() const {return M;}
};
```

Listing 2.2: A class define the domain and boundary conditions of a problem
void xLinetridag (const Domain& A, std::vector<double>& d, int i)
{
    // solve for fixed i and varying j
    int n = A.jSize();
    double b[n]; // temporary storage for b so Domain values
                // are not changed
    b[0]=A(i,j).s11;
    for(int j=1; j<n; j++)
    {
        b[j]=A(i,j).s11-A(i,j-1).s12*A(i,j).s10/b[j-1]; // elim a[j] a[n-1]
        d[j]=d[j]-d[j-1]*A(i,j).s10/b[j-1];
    }
    // back substitute
    d[n-1]=d[n-1]/b[n-1]; // overwrite values of d[j] to return solution
    for(int j=n-2;j>=0;j--)
    {
        d[j]=(d[j]-A(i,j).s12*d[j+1])/b[j];
    }
    // on return d is the solution to the problem
}

Listing 2.3: A function to carry out Thomas’s tridiagonal solver

3. Now write a new class Triangle, that also inherits from Domain, so that the domain is now defined by a triangle with the vertices (0, 0), (0, 1), and (1, 0). Re-solve the model problem on this domain using SOR.

4. Try to write a function

```cpp
int sorSolver (const Domain& A, std::vector<std::vector<double>> & v,
               const std::vector<std::vector<double>> & d);
```

that takes a domain and corresponding rhs and solve the matrix equation returning the solution v.

Further Extension to your program

- Use Thomas’s tridiagonal algorithm along with your Domain class to solve the problem using LSOR. A specialised tridiagonal solver using Domain class to solve for fixed i and varying j is given in listing 2.3. You can use this function or create your own to solve the problem.

- Try to create a Elliptic class which contains your grid vectors, solution matrix and the Domain class, and a function that solves the PDE using SOR. Try to think about any other functions you might need or data variables that you may wish to store.

- Add another member function to solve with LSOR.

- Add a further abstraction layer so that the PDE equation that we solve may be changed using inheritance. Try to think about what functions need to be created, and whether they would need to be virtual.
2.3 Channel Flow

You are required to obtain a numerical solution to the following problem. The streamfunction $\psi$ for a two-dimensional incompressible inviscid fluid flow satisfies Laplace’s equation at all points inside the field of flow. Calculate a numerical solution for flow through the channel as shown in figure 2.2, given that ABCD is the streamline $\psi = 0$, EFG is the streamline $\psi = 1$, and $\psi$ varies linearly across AE and GD.

![Figure 2.2: The channel.](image)

You should be able to use inheritance to create a `Channel` class that inherits from `Domain` and defines the domain and boundary conditions for this problem. Try to abstract your method in some of the ways discussed in the previous section.

2.4 Report

You should prepare a report in the form of a continuous piece of prose that describes the problem and your solution to it. In particular, you should include the following:

- You should start by using point SOR to solve the discrete equations.
- Compare the rate of convergence of the point SOR method to the Line SOR method.
- Produce contour plots of $\psi$.
- Produce plots of the slip velocity along ABCD and EFG.
- Comment on the accuracy of your results and any checks for grid independence that you may have carried out.
- A discussion of how your code may be used to solve other similar problems, and any ways in which you have used object orientation in your code.
Bibliography


Mini Project 3

Impulsively Started Circular Cylinder Flow

Author: Prof. Jitesh Gajjar, Jitesh.Gajjar@manchester.ac.uk

3.1 Background

In this project we are going to study the solution of parabolic partial differential equations and use the techniques to solve for the impulsively started flow past a circular cylinder governed by the unsteady boundary layer equations.

The methods rely on the use of the Thomas algorithm for the solution of tridiagonal equations, i.e. consider the system of equations:

\[ \alpha_j u_{j-1} + \beta_j u_j + \gamma_j u_{j+1} = \delta_j, \quad j = 1, 2, \ldots, N, \quad (1.1) \]

with \( \alpha_1 = 0, \gamma_n = 0 \). The coefficients \( \alpha_j, \beta_j, \gamma_j \) are known together with the right hand side \( \delta_j \) and we need to solve for \( u_j \) for \( j = 1, \ldots, N \). We can rewrite these in matrix form as

\[
\begin{bmatrix}
\beta_1 & \gamma_1 \\
\alpha_2 & \beta_2 & \gamma_2 \\
0 & \alpha_3 & \beta_3 & \gamma_3 \\
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
u_3 \\
\end{bmatrix} =
\begin{bmatrix}
\delta_1 \\
\delta_2 \\
\delta_3 \\
\end{bmatrix}
\]

The equations (1.1) can be solved using Thomas's algorithm given below.

3.1.1 Thomas's tridiagonal algorithm

This version of a tridiagonal solver is based on Gaussian elimination. First we create zeros below the diagonal and then once we have a triangular matrix, we solve for the \( u_j \) using back
substitution. Thus the algorithm takes the form
\[ \beta_j = \beta_j - \frac{\gamma_j - 1}{\beta_j} \delta_j - 1 \alpha_j, \quad j = 2, 3, ..., N, \]
\[ \beta_j = \beta_j - \frac{\delta_j - 1}{\beta_j} \alpha_j, \quad j = 2, 3, ..., N, \]
\[ u_N = \frac{\delta_N}{\beta_N}, \quad u_j = \frac{(\delta_j - \gamma_j) u_{j+1}}{\beta_j}, \quad j = N - 1, ..., 1. \] (1.3)

### 3.1.2 Solution of parabolic equations

Consider the diffusion equation for the function \( u(y,t) \) given by
\[ \frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial y^2}, \quad 0 < t, \quad 0 < y < 1, \] (1.4)

together with boundary and initial conditions given by
\[ u(y = 0, t) = 0, \quad u(y = 1, t) = 1, \quad u(y, t = 0) = 1. \]

We need to find the solution of (1.4) for \( t > 0 \).

There are a number of different schemes which could be used to solve (1.4).

**Explicit scheme**

We may approximate (1.4) by
\[ \frac{u_j^{k+1} - u_j^k}{\Delta t} = \kappa \left[ \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{\Delta y^2} \right]. \] (1.5)

Here \( u_j^k \) denotes an approximation to the exact solution \( u(y,t) \) of the pde at \( y = y_j = (j-1)\Delta y = (j - 1)/(N - 1) \), and \( t = t_k = k\Delta t \). The scheme given in (1.5) is first order in time \( O(\Delta t) \) and second order in space \( O(\Delta y^2) \). The scheme (1.5) is *explicit* because the unknowns at level \( k + 1 \) can be computed directly.

**Fully implicit, first order**

Another approximation is one which makes use of an *implicit* scheme. Then instead of (1.5) we have
\[ \frac{u_j^{k+1} - u_j^k}{\Delta t} = \kappa \left[ \frac{u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}}{\Delta y^2} \right]. \] (1.6)

The unknowns at level \( k + 1 \) are coupled together and we have a set of implicit equations to solve. If we rearrange (1.6) then with \( \beta = \kappa \Delta t/\Delta y^2 \) we have
\[ \beta u_j^{k+1} - (1 + 2\beta) u_j^{k+1} + \beta u_{j-1}^{k+1} = -u_j^k, \quad 2 \leq j \leq N - 1, \] (1.7)

In addition approximation of the boundary conditions gives
\[ u_1^{k+1} = 0, \quad u_N^{k+1} = 1, \] (1.8)

and initial conditions
\[ u_j^0 = 1, \quad j = 1, \ldots, N. \]

The discrete equations (1.7),(1.8) are of tridiagonal form and thus easily solved using Thomas’s algorithm.
3.2. IMPULSIVELY STARTED CIRCULAR CYLINDER FLOW

Crank-Nicolson

A popular scheme is the Crank-Nicolson scheme given by

\[
\frac{u_j^{k+1} - u_j^k}{\Delta t} = \frac{\kappa}{2} \left[ \frac{u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}}{\Delta y^2} + \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{\Delta y^2} \right].
\]

(1.9)

This is second order accurate \(O(\Delta t^2, \Delta y^2)\)

Exercises You are advised to go through the following exercises as this will help you to understand the methods for the main project described below.

1. Write a routine to implement Thomas’s algorithm (1.3) given the coefficient matrices and the right-hand side. Test it with sample data and you may find it useful to cross-check your results with Matlab.

2. Code up the fully explicit scheme. Obtain the solution for time \(t = 1\) using different values of \(\Delta t\) and \(\Delta y\). In particular check by plotting the values of \(u(y, t = 1)\) for fixed \(\Delta t = 0.1\) changing \(\Delta y = .1, .05, .01\) and another graph for \(u(y, t = 1)\) for fixed \(\Delta y = .1\) changing \(\Delta t = .1, .05, .01\). Are your results grid independent?

3. Repeat the exercise 2 but for the fully implicit scheme.

4. Repeat the exercise 2 but for the Crank-Nicolson scheme.

5. What do you conclude from your results. Can you correlate your results with theory? You may find it helpful to read the supplementary notes and look at the book by Smith. Read the sections regarding stability.

3.2 Impulsively started circular cylinder flow

Consider impulsively started unsteady boundary layer flow past a circular cylinder. The unsteady boundary layer equations may be written as

\[
u = \frac{\partial \psi}{\partial y}, \quad \frac{\partial u}{\partial y} + u \frac{\partial u}{\partial x} - \frac{\partial \psi}{\partial x} \frac{\partial u}{\partial y} = \sin x \cos x + \frac{\partial^2 u}{\partial y^2},
\]

(2.10)

where \(0 \leq x \leq \pi\) and \(0 \leq y < \infty\). The boundary conditions are

\[
u(0, y, t) = 0, \quad \nu(\pi, y, t) = 0
\]

\[
u(x, 0, t) = 0 = \psi(x, 0, t), \quad u(x, y, t) \rightarrow \sin(x) \quad \text{as} \quad y \rightarrow \infty.
\]

The initial condition may be taken as

\[
u(x, y, 0) = \sin(x), \quad \psi(x, y, t = 0) = y \sin(x), \quad 0 < y.
\]

You are required to solve the above system of equations. The following scheme is one possibility (and known to give sensible results). Let \(x_i = i\Delta x, y_j = j\Delta y\) where \(\Delta x = \pi/M, \Delta y = y_{\infty}/N\), and \(t = t_k = k\Delta t\). Denote approximations to \(u(x, y, t), \psi(x, y, t)\) at \(x = x_i, y = y_j, t = t_k\) by

\[
u(x_i, y_j, t_k) = u_{i,j}^k, \quad \psi(x_i, y_j, t_k) = \psi_{i,j}^k.
\]
Then we may approximate (2.10) by

\[
\frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t} + u_{i,j}^k \left( \frac{u_{i+1,j}^k - u_{i-1,j}^k}{2\Delta x} \right) - \frac{\psi_{i+1,j}^k - \psi_{i-1,j}^k}{2\Delta x} \left( \frac{u_{i,j+1}^{k+1} - u_{i,j-1}^{k+1}}{2\Delta y} \right) = \sin(x_i) \cos(y_i)
\]

(2.11)

The difference equations (2.11) lead to a tridiagonal system for each \( x = x_i \) station, and are easily solved with a tridiagonal solver. The \( \psi^{k+1} \) values can be obtained from (2.12) once the \( u^{k+1} \) values are calculated, by integrating from the wall outwards.

It is suggested that you modify any earlier codes to introduce two dimensional arrays \( u(i,j) \) to store \( u_{i,j}^{k+1} \) and \( u_{i,j}^k \) to store \( u_{i,j}^k \). Starting from \( x = 0 \) solve the discrete equations and continue for each \( x \) until \( x = M \). Obtain both \( u \) and \( \psi \) values. Add a loop to increase in time and make sure you only store the current and previous values as otherwise you may encounter storage problems when taking very small time steps.

### 3.3 Report

In your report you are asked to address the following points.

- In your problem formulation it would be helpful to write out the equations for \( u_{i,j}^{k+1} \) using the scheme suggested above in the form

  \[
  \alpha_j u_{i,j}^{k+1} + \beta_j u_{i,j}^k + \gamma_j u_{i,j+1}^{k+1} = r_j,
  \]

  (3.13)

  where the coefficients \( \alpha_j, \beta_j, \gamma_j \) and the right-hand side \( r_j \) should be clearly identified for each value of \( x_i \). [For clarity the dependence of these coefficients on \( i,k \) has been suppressed].

- Provide a graph of

  \[
  \tau(x,t) = \frac{\partial u}{\partial y}(y = 0)
  \]

  by taking a second-order backward difference, for the times \( 0 \leq t \leq 3 \) in steps of \( t = 0.5 \).

- Also compute and provide a graph of

  \[
  \delta(x,t) = y_{\infty} - \frac{\psi(x,y_{\infty},t)}{\sin(x)}, \quad 0 < x < \pi,
  \]

  for \( 0 < t < 3 \) in steps of \( t = 0.5 \).

- Also show contour plots of \( u(x,y,t), \psi(x,y,t) \) for the times \( t = 0, 1, 2, 2.5 \) and 3.

- In your report you should discuss the accuracy of your results, and provide evidence of any grid independence studies you have carried out.

- Discuss the validity of the solutions you have obtained have for large times? What do the results point to?

- For additional marks try a different scheme to that suggested above.
Bibliography

Mini Project 4

Valuing a Battery on the Electricity Market

Author: Dr. Paul Johnson, Paul.Johnson-2@manchester.ac.uk

In this project we study the solution of parabolic partial differential equations used to model electricity storage on a deregulated energy market. As governments around the world commit to increase the level of renewable energy they produce, wind power has come to the forefront as one of the most viable options. However, wind power does not come without its problems. Due to the intermittent nature of the wind, we may have a situation where the wind blows at the wrong time causing the electricity generated to be discarded. An obvious solution would seem to be the installation of batteries alongside the generators to store energy when there is a surplus so that it may be released when there is a deficit. There are many reasons (beyond the scope of this project) as to why this is not currently economically viable on a large scale in the UK, but non the less it will become important in the future as more and more energy is generated using wind. The simplified problem in this project investigates the value of installing a battery alongside a wind power generator in order to take advantage of a surplus in production to meet commitments if there is a deficit in production.

The structure of the project is as follows: in the first section you will learn how to solve the heat diffusion equation using some objects to simplify storage, then in the second section you will apply it to the full problem and investigate the resulting solutions. You are then offered a choice of extensions to the problem to investigate further.

4.1 Parabolic PDEs

One of the simplest parabolic PDEs is the heat diffusion equation which in one space dimension is

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}. \quad (1.1)$$

valid in the region $a \leq x \leq b$ where $\kappa$ is some given constant.
In the figure to the right we show the region $R$ in which the PDE (1.1) must be satisfied. The function $u$ must be specified both at the boundaries $x = a$ and $x = b$, and also at $t = 0$. This is shown by the dashed line $S$.

Consider the diffusion equation for the function $u(x, t)$ given by

$$u(x = 0, t) = 0, \quad u(x = 1, t) = 1, \quad u(x, t = 0) = 1.$$  

We need to find the solution of (1.1) for $t > 0$.

There are a number of different schemes which could be used to solve (1.1).

**Explicit scheme**

We may approximate (1.1) by

$$\frac{u^{k+1}_i - u^k_i}{\Delta t} = \kappa \left[ \frac{u^{k+1}_{i+1} - 2u^{k+1}_i + u^{k+1}_{i-1}}{\Delta x^2} \right].$$  \hspace{1cm} (1.2)

Here $u^k_i$ denotes an approximation to the exact solution $u(x, t)$ of the PDE at $x = x_i = i\Delta x = i/N$, and $t = t_k = k\Delta t$. Here then $i$ indexes our position in space and $k$ our position in time. The scheme given in (1.2) is first order accurate in time $O(\Delta t)$ and second order accurate in space $O(\Delta x^2)$. The scheme (1.2) is explicit because the unknowns at level $k+1$ can be computed directly from known values at level $k$.

**Fully implicit, first order**

Another approximation is one which makes use of an implicit scheme. Then instead of (1.2) we have

$$\frac{u^{k+1}_i - u^k_i}{\Delta t} = \kappa \left[ \frac{u^{k+1}_{i+1} - 2u^{k+1}_i + u^{k+1}_{i-1}}{\Delta x^2} \right].$$  \hspace{1cm} (1.3)

The unknowns at level $k+1$ are coupled together and we have a set of implicit equations to solve. If we rearrange (1.3) then with $\beta = \kappa \Delta t / \Delta x^2$ we have

$$\beta u^{k+1}_{i+1} - (1 + 2\beta)u^{k+1}_i + \beta u^{k+1}_{i-1} = -u^k_i, \quad 0 < i < N,$$

(1.4)

In addition the boundary conditions gives

$$u^{k+1}_0 = 0, \quad u^{k+1}_N = 1,$$  \hspace{1cm} (1.5)

and initial conditions

$$u^0_i = 1, \quad i = 0, \ldots, N.$$  

The discrete equations (1.4), (1.5) form a tridiagonal system of equations.
4.1. PARABOLIC PDES

4.1.1 Solving Tridiagonal Systems

We require a direct method for the solution of tridiagonal equations, i.e., consider the system of equations:

\[ a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, 2, \ldots, N, \quad (1.6) \]

with \( a_0 = 0, c_N = 0 \). The coefficients \( a_i, b_i, c_i \) are known together with the right-hand side \( d_i \) and we need to solve for \( u_i \) for \( i = 0, \ldots, N \). We can rewrite these in matrix form as

\[
\begin{bmatrix}
  a_1 & 0 & 0 & \cdots & 0 \\
  b_1 & c_1 & 0 & \cdots & 0 \\
  a_2 & b_2 & c_2 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \ddots & \vdots \\
  a_N & b_N \\
\end{bmatrix}
\begin{bmatrix}
  u_0 \\
  u_1 \\
  u_2 \\
  \vdots \\
  u_N \\
\end{bmatrix}
= \begin{bmatrix}
  d_0 \\
  d_1 \\
  d_2 \\
  \vdots \\
  d_N \\
\end{bmatrix}.
\]

The equations (1.4) and (1.5) can be solved using Thomas’s algorithm given below.

**Thomas’s tridiagonal algorithm**

This version of a tridiagonal solver is based on Gaussian elimination. First we create zeros below the diagonal and then once we have a triangular matrix, we solve for the \( u_j \) using back substitution. Thus the algorithm takes the form

\[
b_i = b_i - \frac{c_{i-1} a_i}{b_{i-1}}, \quad i = 1, 2, 3, \ldots, N, \\
d_i = d_i - \frac{d_{i-1} a_i}{b_{i-1}}, \quad i = 1, 2, 3, \ldots, N, \\
u_N = \frac{d_N}{b_N}, \quad u_i = \frac{(d_i - c_i u_{i+1})}{b_i}, \quad i = N-1, \ldots, 1, 0.
\]

(1.8)

4.1.2 Exercises

1. Write a routine to implement Thomas’s algorithm (1.8) given the coefficient matrices and the right-hand side. Test it with sample data and you may find it useful to cross-check your results with Matlab.

2. Code up the fully explicit scheme (1.2). Obtain the solution for time \( t = 1 \) using \( \kappa = 1 \) and different values of \( \Delta t \) and \( \Delta x \). In particular check by plotting the values of \( u(x, t = 1) \) for fixed \( \Delta t = 0.1 \) changing \( \Delta x = .1, .05, .01 \) and another graph for \( u(x, t = 1) \) for fixed \( \Delta x = .1 \) changing \( \Delta t = .1, .05, .01 \). Are your results grid independent?

3. Repeat the exercise 2 but for the fully implicit scheme (1.3).

4. What do you conclude from your results. Can you correlate your results with theory? You may find it helpful to read the supplementary notes and look at the book by Smith (1985). Read the sections regarding stability.
4.2 Battery Storage Model

Let the variable $Q \in [0, Q_{\text{max}}]$ denote the energy stored within the battery, and $X \in (-\infty, \infty)$ denote the surplus of electricity (energy generated minus energy demanded), then the value of a battery $V(X, Q, t)$ is a function of the current state of both of these variables at time $t$. Now assume that $X$ follows a random walk through time and satisfies the following stochastic differential equation (SDE):

$$dX = \alpha(X, t)dt + \beta(X, t)dW$$

(2.9)

where $\alpha$ and $\beta$ are functions chosen to best fit data, and $dW$ is a standard Wiener process. We choose for simplicity that $\alpha(X, t) = 0$ and $\beta(X, t) = \sigma$ where $\sigma$ is a given constant. This will ensure that $V(X, Q, t) = V(X, Q)$ or that the solution is time independent. Then we may also look at how energy flows into and out of the battery, let the change in $Q$ over an instant be defined as

$$dQ = \begin{cases} 
0 & \text{if } X < 0 \text{ and } Q = 0 \\
0 & \text{if } X > 0 \text{ and } Q = Q_{\text{max}} \\
X dt & \text{otherwise} 
\end{cases} .$$

(2.10)

This says that the battery cannot discharge if already empty, or charge if already full. Further, if $X < 0$ (and $Q > 0$), then we have a deficit of energy and we may sell the energy discharged $|dQ|$ onto the market at the price $p$. Then the change in value of our option over an instant is given by

$$dV = -p \min(dQ, 0).$$

(2.11)

Then by using Itô’s lemma and combining equations (2.9), (2.10) and (2.11) it can be shown (see Howell et al. (2011)) that the net present value of the battery is described by the following PDE:

$$X \frac{\partial V}{\partial Q} + \frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - rV = p \min(X, 0) = 0$$

(2.12)

subject to the conditions

$$V = 0 \quad \text{as } X \to \infty$$
$$V = pQ \quad \text{as } X \to -\infty$$
$$\frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - rV = 0 \quad \text{if } X < 0 \text{ and } Q = 0$$
$$\frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - rV = 0 \quad \text{if } X > 0 \text{ and } Q = Q_{\text{max}}$$

Now in order to solve the PDE we must divide the domain into two regions (see figure 4.1), one ($X \leq 0$ or $A$) in which we can solve upwards, and the other ($X \geq 0$ or $B$) in which we can solve downwards. Along the boundary $X = 0$ the solutions must be matched.

**Fully implicit scheme**

Let us discretise the problem so that

$$X_i = (i - N) \Delta X \quad \Delta X = \frac{X_\infty}{N} \quad Q_j = j \Delta Q \quad \Delta Q = \frac{Q_{\text{max}}}{M}$$

where

$$0 \leq i \leq 2N \quad \text{and} \quad 0 \leq j \leq M$$

and the value function $v_{i,j} = V(X_i, Q_j)$. 
4.2. BATTERY STORAGE MODEL

The fully implicit method we propose to solve this problem is an iterative scheme where we solve in the region $A$ using a guess for the solution in $B$, and then solve in the region $B$ using a guess for the solution in $A$. So to complete one iteration involves sweeping across $A$ and then $B$. We use $q$ to denote the value at the $q$th iteration, or $v_{i,j}^q$, and $q + 1/2$ indicates we are halfway through one iteration.

**Region $A$**

Then the scheme for $v$ in the region $A$ is as follows. Firstly, when $j = 0$ and $0 < i < N$ we have

$$\frac{1}{2}\sigma^2 \frac{v_{i-1,0}^{q+1/2} - 2v_{i,0}^{q+1/2} + v_{i+1,0}^{q+1/2}}{\Delta x_i^2} - rv_{i,0}^{q+1/2} = 0,$$

and then on the boundary for $j = 0$ we get

$$v_{0,0}^{q+1/2} = 0, \quad v_{N,0}^{q+1/2} - \left(2 + \frac{r\Delta x_i^2}{\sigma^2}\right)v_{N,0}^{q+1/2} = -v_{N+1,0}^q.$$

Here we must guess the value of the system where $i > N$, hence the $q$ and this guess should simply be the most up-to-date value. The unknown values $v_{i,j}^{q+1/2}$ must be found.

Next if we are in the discharging region $A$ then we have $0 < j \leq M$ and $0 < i < N$ and the scheme is

$$X_i \frac{v_{i+1,j}^{q+1/2} - v_{i,j}^{q+1/2}}{\Delta Q} + \frac{1}{2}\sigma^2 \frac{v_{i-1,j}^{q+1/2} - 2v_{i,j}^{q+1/2} + v_{i+1,j}^{q+1/2}}{\Delta x_i^2} - rv_{i,j}^{q+1/2} = pX_i.$$  

(2.15)

If we rearrange (2.15) with $\kappa_i = \sigma^2\Delta Q/(2X_i\Delta x_i^2)$ and $\rho_i = r\Delta Q/X_i$ we have

$$\kappa_i v_{i-1,j}^{q+1/2} + (1 - 2\kappa_i - \rho_i)v_{i,j}^{q+1/2} + \kappa_i v_{i+1,j}^{q+1/2} = v_{i,j}^{q+1/2} + p\Delta Q, \quad 0 < i < N,$$

(2.16)

In addition approximation of the boundary conditions with $\kappa_N = \sigma^2/(2\Delta x_N^2)$ gives

$$v_{0,j}^{q+1/2} = Q_j, \quad \kappa_N v_{N-1,j}^{q+1/2} - (2\kappa_N + r)v_{N,j}^{q+1/2} = -\kappa_N v_{N+1,j}^q.$$

(2.17)

The discrete equations (2.13), (2.14), (2.16), (2.17) form tridiagonal systems of equations for each $0 \leq j \leq M$. So long as we solve first for $j = 0$ then we may move upwards from $j = 1$ up to $j = M$ so that all the values on the right hand side of the equations may be treated as knowns.
Region B

Then the scheme for $v$ in the region $B$ is as follows. Firstly, when $j = M$ and $N < i < 2N$ the battery is full and can no longer be charged so we have

$$ \frac{1}{2} \sigma^2 \frac{v_{i-1,M}^{q+1} - 2v_{i,M}^{q+1} + v_{i+1,M}^{q+1}}{\Delta X} - r v_{i,M}^{q+1} = 0. \quad (2.18) $$

and then on the boundary for $j = M$ we get

$$ - \left( 2 + \frac{r \Delta X^2}{2 \sigma^2} \right) v_{N,M}^{q+1} + v_{N+1,M}^{q+1} = - v_{N-1,M}^{q+1/2}, \quad v_{2N,M}^{q+1} = 0. \quad (2.19) $$

Next if we are in the charging region of $B$ then we have $M > j \geq 0$ and $N < i < 2N$ and the scheme is

$$ X_i \frac{v_{i,j+1}^{q+1} - v_{i,j}^{q+1}}{\Delta Q} + \frac{1}{2} \sigma^2 \frac{v_{i-1,j}^{q+1} - 2v_{i,j}^{q+1} + v_{i+1,j}^{q+1}}{\Delta X} - r v_{i,j}^{q+1} = 0. \quad (2.20) $$

If we rearrange (2.20) with $\kappa_i = \sigma^2 \Delta Q/(2X_i \Delta X^2)$ and $\rho_i = r \Delta Q/X_i$ we have

$$ \kappa_i v_{i-1,j}^{q+1} - (1 + 2\kappa_i + \rho_i) v_{i,j}^{q+1} + \kappa_i v_{i+1,j}^{q+1} = - v_{i,j+1}^{q+1}, \quad N < i < 2N, \quad (2.21) $$

In addition approximation of the boundary conditions with $\kappa_N = \sigma^2/(2\Delta X^2)$ gives

$$ -(2\kappa_N + r) v_{i,N}^{q+1} + \kappa_N v_{N+1,j}^{q+1} = -\kappa_N v_{N-1,j}^{q+1/2}, \quad v_{2N,j}^{q+1} = 0. \quad (2.22) $$

The discrete equations (2.18), (2.19), (2.21), (2.22) form tridiagonal systems of equations for each $M > j \geq 0$. So long as we solve for $j = M$ first we can then move downwards from $j = M$ down to $j = 0$ treating the right hand side of the equations as knowns ($v_{i,j+1}^{q+1}$).

In order to fully solve the problem you must iterate until your solution has converged, or $v_{i,j}^{q+1} = v_{i,j}^q$ for all $i$ and $j$.

4.3 Report

You should prepare a report in the form of a continuous piece of prose that describes the problem and your solution to it. In particular, you should include the following:

- A brief analysis of the heat diffusion equation, demonstrating both the solution and grid dependence of each scheme given the problem defined in section 4.1.2.

- Choosing the following parameters, $X_\infty = 25$, $Q_{\text{max}} = 1$, $\sigma = 0.5$, $r = 0.01$, and $p = 1$, solve the battery storage problem. Try using different grids by varying $N$ and $M$. Provide an accurate graph of $V(X,0)$ and $V(X,Q_{\text{max}})$ for these chosen parameters.

- Describe how you evaluate when your solution has converged.

- Try to vary both $\sigma$ and $r$. How does it affect your boundary conditions? Plot out different different types of solutions that you see.

Depending on how well the project goes, you might also try the following:

- Create a “Storage” class to solve the problem and store all of the parameters and the solution. You may try to include functions to set boundary conditions, and the form of the PDE we solve. You may also look to write a function to setup the problem, solve the problem and output the results to file.
• Adding mean reversion to the SDE (2.9) the PDE becomes

$$X \frac{\partial V}{\partial Q} + \frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - \alpha X \frac{\partial V}{\partial X} - rV - p \min(X, 0) = 0$$  \hspace{1cm} (3.23)

subject to the conditions

$$\frac{\partial V}{\partial Q} - \alpha \frac{\partial V}{\partial X} = p \quad \text{as} \quad X \to -\infty$$

$$\frac{\partial V}{\partial Q} - \alpha \frac{\partial V}{\partial X} = 0 \quad \text{as} \quad X \to +\infty$$

$$\frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - \alpha X \frac{\partial V}{\partial X} - rV = 0 \quad \text{if} \quad X < 0 \quad \text{and} \quad Q = 0$$

$$\frac{1}{2} \sigma^2 \frac{\partial^2 V}{\partial X^2} - \alpha X \frac{\partial V}{\partial X} - rV = 0 \quad \text{if} \quad X > 0 \quad \text{and} \quad Q = Q_{\max}$$

Try to solve this problem with $\alpha = 0.1$. Investigate how the solution changes, with respect to the value of the system and also the position of the boundary conditions.

**Bibliography**


Mini Project 5

American Options

Author: Dr. Paul Johnson, Paul.Johnson-2@manchester.ac.uk

The pricing of the American option is a long standing problem in mathematical finance. It can be formulated in terms of a free boundary, resulting in a problem similar to those seen in melting ice or dam problems.

In this project you will use advanced techniques to value an American call option using the Black-Scholes (BS) model, see Wilmott et al. (1995). The BS model generates a linear parabolic PDE, which along with nonlinear boundary conditions must be solved numerically. After a coordinate transformation the problem becomes a nonlinear PDE, which can be reduced to a linear problem using Newton linearisation. This problem will involve generating an adapted tridiagonal matrix solver, and using variable timestepping to contain instabilities. Object orientated coding will be used to allow the specification of generic boundary conditions and initial conditions (payoff of the option).

5.1 Theory

In this project we aim to price an option under the Black-Scholes framework. An option gives the holder the write to buy, in a call option, or sell, in a put option the underlying asset at a particular price, also known as the strike price on the expiry date. If the holder of the option buys or sells the option, they are said to have exercised the option. The European option is the simplest type of option and can be priced with an analytical formula. The American option is a simple extension to the European option, allowing the holder to exercise the option at any time, resulting in a nonlinear problem.

5.1.1 Black-Scholes Pricing Framework

Under the Black-Scholes framework a derivative \( V \) depending on the underlying asset \( S \) and time to expiry \( \tau \) may be valued according to the equation

\[
\frac{\partial V}{\partial \tau} = \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - d)S \frac{\partial V}{\partial S} - rV \tag{1.1}
\]

where \( r \) is the risk free interest rate, \( d \) is the dividend rate and \( \sigma \) the volatility of the underlying asset, subject to the appropriate boundary and initial conditions.
The initial condition for an American call option is as follows
\[ V(S, \tau = 0) = \max(S - X, 0) \tag{1.2} \]
where \( X \) is the strike price, or the price at which the holder of the option can buy the asset.

In order to specify boundary conditions we need to define the free boundary \( S_f(\tau) \) also known as the optimal exercise boundary. We assume that there exists a region in \( S \) where the holder will exercise the option, and a region where the holder will hold the option. The nonlinearity appears because we do not know the position of \( S_f(\tau) \), so it is therefore an unknown and must be calculated as part of any solution. The result is two conditions at \( S = S_f(\tau) \)
\[ V(S = S_f(\tau), \tau) = S - X \tag{1.3} \]
and
\[ \frac{\partial V}{\partial S}(S = S_f(\tau), \tau) = 1. \tag{1.4} \]
We also have a boundary condition for small \( S \) given by
\[ V(S, t) \to 0 \quad \text{as} \quad S \to 0. \tag{1.5} \]
Since \( S_f(\tau) \) is now an unknown it must also have an initial condition
\[ S_f(\tau = 0) = \max\left(\frac{r}{d}X, X\right) \tag{1.6} \]
The derivation of the conditions is outside the scope of the project, so take them as given.

5.1.2 Formulation

Body-fitted coordinates

The main problem when applying numerical schemes to problems involving early exercise is that the position of the early exercise boundary is not known a priori. This often means that when constructing a grid this boundary does not often fall on a grid point, or node, causing a degree of so-called non-linearity error. The usual numerical schemes: lattice methods and basic finite difference schemes are unable to fix the grid correctly. The method described here involves keeping the time axis fixed but transforming the \( S \) to an \( \hat{S} \) axis that moves with the free boundary.

To incorporate the body-fitted co-ordinate system simply introduce the transformation
\[ \hat{S} = \frac{S}{S_f(\tau)} \]
so that the range of \( \hat{S} \) is from 0 to 1. This will also ensure that, at every timestep, the \( S \)-dimension of the grid will range from zero to the exact position of the free-boundary.

It is, thus, necessary to perform a technique analogous to a multivalued Newton-Raphson iteration. This is undertaken in combination with the usual Crank Nicolson scheme. The first point to note is that the above transformation leads to a slight change in the governing equation, since
\[ \frac{\partial V}{\partial \tau} \to \frac{\partial V}{\partial \tau} + \frac{\partial V}{\partial \hat{S}} \frac{\partial \hat{S}}{\partial \tau} = \frac{\partial V}{\partial \tau} - \frac{\partial V}{\partial S} \frac{dS_f}{d\tau} \frac{\hat{S}}{S_f} \]
and
\[ \frac{\partial V}{\partial S} \to \frac{1}{S_f} \frac{\partial V}{\partial \hat{S}} \]
5.1. THEORY

so the Black-Scholes equation becomes:

\[
\frac{\partial V}{\partial \tau} - \frac{\partial V}{\partial \hat{S}} \frac{d}{d\tau} \hat{S} + \frac{1}{2} \sigma^2 \hat{S}^2 \frac{\partial^2 V}{\partial S^2} + (r - d) \hat{S} \frac{\partial V}{\partial \hat{S}} - rV = 0
\]

Fortunately the option value, \( V \) and the position of the free boundary \( S_f(\tau = 0) \) are both known at expiry (i.e. \( S_f = \max(rX/d, X) \) and \( V = \max(SS_f(0) - X, 0) \)) and so to calculate the values at \( \tau = \Delta \tau \) we simply perform an iterative scheme using these values as a starting point.

At a given timestep \( j\Delta \tau \), after \( m \) iterations write the values at the \( n \) distinct \( V \) values in the finite difference scheme and the value of \( S_f \) as:

\[
S_f^{(m+1)}(j\Delta \tau) = S_f^{(m)}(j\Delta \tau) + \delta S_f
\]

The value of \( V_i^{(0)}(i\Delta \hat{S}, j\Delta \tau) \) and \( S_f^{(0)}(j\Delta \tau) \) are taken to be the converged values of \( V_i(i\Delta \hat{S}, (j+1)\Delta \tau) \) and \( S_f(j+1\Delta \tau) \) respectively. All that remains is to calculate the successive values of the \( \delta V_i \)’s and \( \delta S_f \). In order to do this a Crank-Nicolson scheme is used.

Crank-Nicolson

The principal differences from the basic scheme are, first, that the unknown values are the \( \delta V_i \)’s and \( \delta S_f \). Secondly, there is also be an extra column in the matrix to cope with the new extra unknown, \( \delta S_f \). There are, now, \( N+1 \) equations in \( N+2 \) unknowns, this is easily overcome by using both the boundary conditions at \( \hat{S} = S_f \) including the condition from before and the smooth pasting condition

\[
\frac{\partial V}{\partial \hat{S}} = 1
\]

The next step requires arranging the difference equations into a system with the unknown values are on one side and the known ones on the other. The approximation for the derivatives is as usual. In general \( V(i\Delta \hat{S}, j\Delta \tau) \) is denoted \( V_{i,j} \), and the resulting set of equations, for \( 0 \le i \le N \), is:

\[
a_i \delta V_{i-1} + b_i \delta V_i + c_i \delta V_{i+1} + d_i \delta S_f = e_i
\]

where, as before, the values of \( a_i, b_i, c_i, d_i \) and \( e_i \) need to be calculated.

Boundary conditions

There is one boundary condition at \( \hat{S} = 0 \) and two at \( \hat{S} = 1 \). For these the values of \( a, b, c, d \) and \( e \) are known and are as follows: At \( \hat{S} = 0 \):

\[
V_{0,j}^{(m)} + \delta V_0 = 0
\]

Now at \( \hat{S} = 1 \) we have:

\[
V_{N,j}^{(m)} + \delta V_N = S_f^{(m)} + \delta S_f - X
\]

and using a one-sided difference scheme we can generate an equation for the derivative condition

\[
\frac{3V_{N,j}^{(m)} - 4V_{N-1,j}^{(m)} + V_{N-2,j}^{(m)}}{2\Delta \hat{S}(S_f^{(m)} + \delta S_f)} = 1
\]
which rearranges to give
\[ \delta V_0 = -V_{0,j}^{(m)}, \]
\[ \delta V_N - \delta S_f = -V_{N,j}^{(m)} - S_{f,j}^{(m)} - X, \]
\[ 3\delta V_N - 4\delta V_{N-1} + \delta V_{N-2} - 2\Delta \hat{S}\delta S_f = 2\Delta \hat{S}S_{f,j}^{(m)} - (3V_{N,j}^{(m)} - 4V_{N-1,j}^{(m)} + V_{N-2,j}^{(m)}) \]

### 5.1.3 Matrix Solver

There are now a series of \( N + 2 \) equations in \( N + 2 \) unknowns which can be solved using linear algebra techniques. In matrix form the problem can be displayed as
\[
\begin{pmatrix}
  b_0 & c_0 & 0 & 0 & . & . & . & . & 0 \\
  a_1 & b_1 & c_1 & 0 & . & . & . & d_1 & . \\
  0 & a_2 & b_2 & c_2 & . & . & . & d_2 & . \\
  . & . & . & a_i & b_i & c_i & . & d_i & . \\
  0 & . & . & . & a_{N-1} & b_{N-1} & c_{N-1} & d_N & . \\
  0 & . & . & . & . & a_N & b_N & d_N & . \\
  0 & . & . & . & 1 & . & . & -4 & 3 & d_{N+1} \\
\end{pmatrix}
\begin{pmatrix}
  \delta V_0 \\
  \delta V_1 \\
  \delta V_2 \\
  \delta V_{N-1} \\
  \delta V_N \\
  \delta S_f \\
\end{pmatrix}
= \begin{pmatrix}
  -V_{0,j}^{(m)} \\
  e_1 \\
  e_2 \\
  . \\
  . \\
  . \\
  . \\
\end{pmatrix}
\]

A reduction technique is then applied to solve the system. First, remove all the \( a_i \) terms, starting from \( a_1 \) by using
\[ a'_1 = a_1 - b_{i-1} \frac{a_i}{b_{i-1}} \]
and as a result the other terms are affected, namely
\[ b'_i = b_i - c_{i-1} \frac{a_i}{b_{i-1}} \]
\[ d'_i = d_i - d_{i-1} \frac{a_i}{b_{i-1}} \]
\[ c'_i = e_i - e_{i-1} \frac{a_i}{b_{i-1}} \]

There is a slight anomaly in the reduction technique in that the last row in the matrix has extra columns. This is easily overcome as the three extra terms can be removed using \( b_{N-2} \), \( b_{N-1} \) and \( b_N \). All other terms must be adjusted accordingly. The matrix is now in the form:
\[
\begin{pmatrix}
  b_0 & c_0 & 0 & 0 & . & . & . & . & 0 \\
  0 & b'_1 & c'_1 & 0 & . & . & . & d'_1 & . \\
  0 & 0 & b'_2 & c'_2 & . & . & . & d'_2 & . \\
  . & . & . & b'_i & c'_i & 0 & . & d'_i & . \\
  . & . & . & . & . & . & . & . & . \\
  . & . & . & . & 0 & b'_N & d'_N & . & . \\
  0 & . & . & . & 0 & 0 & d'_{N+1} & . & . \\
\end{pmatrix}
\begin{pmatrix}
  \delta V_0 \\
  \delta V_1 \\
  \delta V_2 \\
  \delta V_{N-1} \\
  \delta V_N \\
  \delta S_f \\
\end{pmatrix}
= \begin{pmatrix}
  -V_{0,j}^{(m)} \\
  e'_1 \\
  e'_2 \\
  . \\
  . \\
  . \\
  . \\
\end{pmatrix}
\]
5.2. EXERCISES

We now have an upper triangular matrix which can be solved using back substitution. We can find the position of the free boundary using the last equation:

$$\delta S_f = \frac{e_{N+1}'}{d_{N+1}}.$$ 

and the value of the $\delta V_N$ using

$$\delta S_f = \frac{e_N' - d_N' \delta S_f}{b_N}.$$ 

Then continue to calculate the values of $\delta V_i$ for $i = N - 1, \ldots, 0$ using

$$\delta V_i = \frac{e_i' - d_i' \delta S_f - c_i' \delta V_{i+1}}{b_i'}.$$ 

This scheme is equivalent to a Newton-Raphson iteration and, as such, must have converged before moving forward to the next timestep.

Depending on the accuracy required, we select a value for which all the $\delta V_i$'s or $\delta S_f$ need to be less than to have assumed convergence - a typical value is $1 \times 10^{-8}$.

5.2 Exercises

1. Write a function `gaussianSolver` to implement the Gaussian elimination on the vectors $a$, $b$, $c$, $d$, and $e$ all of length $N + 2$. You may return the solution as a vector, pass a solution vector as an argument, or set $e$ to be the solution on return.

2. Check your algorithm with a test matrix, and compare the solution from another source.

3. Now using the finite difference formulas below, calculate $a_i$, $b_i$ etc. Find them by plugging the finite difference formulas into the governing PDE, and $O(1)$ terms on the RHS, putting all of the $\delta$ terms on the LHS, and setting any $\delta^2$ term equal to zero.

$$\frac{\partial V}{\partial \tau} = \frac{1}{\Delta \tau} (V_{i,j}^{(m)} + \delta V_i - V_{i,j-1})$$

$$\frac{\partial S_f}{\partial \tau} = \frac{1}{\Delta \tau} (S_{f,j}^{(m)} + \delta S_f - S_{f,j-1})$$

$$\frac{\partial V}{\partial S} = \frac{1}{4 \Delta S} (V_{i+1,j-1} - V_{i-1,j-1} + V_{i+1,j} + V_{i+1} - V_{i-1,j} - \delta V_{i-1})$$

Other formulas are left as an exercise.

4. Setup storage for the option value, $v_{\text{new}}$ and $v_{\text{old}}$, and coordinate vector $s_{\text{hat}}$. Also declare the free boundary $S_{f_{\text{new}}}$ and $S_{f_{\text{old}}}$ as doubles. Then implement the initial conditions for $\tau = 0$ on the option value and free boundary with $X = 1$, $r = 0.05$, and $d = 0.025$. Print the values out to screen/file. How would you transform the results back into the original variables?

5. Using $N = 10$, a timestep $\Delta \tau = 0.1$, and $\sigma = 0.4$, implement the Newton algorithm to find $V_{i,1}$ where $V_{i,0}$ is $v_{\text{old}}$ and $V_{i,1} = v_{\text{new}}$. You will need to set up the vectors $a$ through to $e$ with the appropriate values that you have calculated, and use the solution to the matrix problem as the correction to update $v_{\text{new}}$. The algorithm is given by

- Loop until converged:
setup the linear algebra problem to find $\delta V_i$ and $\delta S_f$, setting appropriate values for $a$, $b$ etc.

solve the linear algebra problem using your $gaussianSolver$ function to find the correction.

if correction is small enough exit.

implement the corrections on $V_{i,j}$ and $S_{f,j}$

\[
V_{i,j} = V_{i,j} + \delta S_f
\]

\[
S_{f,j} = S_{f,j} + \delta S_f
\]

Check that your solution seems viable (i.e boundary conditions are met and the option value is positive).

6. Now iterate the scheme through $j_{max}$ timesteps, and put the entire algorithm into a $solve$ function.

7. Create an $Option$ class that contains member functions to specify the initial conditions and boundary conditions for a particular option, such as the call option in this case. You will need to store the option value and the position of the free boundary inside the class. Rewrite your $solve$ function to accept the $Option$ class as an argument.

5.3 Convertible Bond

The convertible bond is an option which can be defined by the initial conditions:

\[
V(S, \tau = 0) = \max(S, B), \quad \text{and} \quad S_f(\tau = 0) = B,
\]

where $B$ is the principal value of the bond. The boundary conditions for the bond are given by:

\[
V(S = 0, \tau) = Be^{-r\tau},
\]

\[
V(S = S_f, \tau) = S_f,
\]

\[
\frac{\partial V}{\partial S} \bigg|_{S = S_f} = 1.
\]

Using inheritance or otherwise, and given that the convertible bond value $V$ satisfies the same governing PDE as the American call option, write a program to solve the convertible bond.

5.4 Report

You should prepare a report in the form of a continuous piece of prose that describes the problem and your solution to it. In particular, you should include the following:

- In part 1 of your report, investigate the American call option where $d < r$.
  - Plot of the value of the option and the payoff for some arbitrary values
  - Produce a table to show the convergence of the option value to the exact solution (and computation times) as the number of steps in $\hat{S}$ and $\tau$ increase
  - Add an extra column to show convergence if we use Richardson extrapolation
Plots of the free boundary $S_f(\tau)$ for different values of $r$ and $d$

What happens when $d = 0$?

- In part 2, investigate the American call option with $d > r$.
  - Explain why the solution does not always converge when $d > r$?
  - You will need to alter the timesteps in this case so that the following scaling is implemented. Simply divide time into $j_{max}$ steps as usual, however when using $\Delta \tau$ in the numerical calculation instead use
    \[
    \Delta \tau = \frac{(2j - 1)T}{j_{max}^2}
    \]
  - Plot the value of the option and the payoff for some arbitrary values
  - Produce a table to show the convergence of the option value to the ‘exact’ solution (and computation times) as the number of steps in $S$ and $\tau$ increase
  - Add an extra column to show convergence if we use Richardson extrapolation
  - Plot the free boundary $S_f(\tau)$ for different values of $r$ and $d$

- In the final part, investigate the convertible bond by plotting the free boundary $S_f(\tau)$ for different values of $r$ and $d$.

Bibliography


Mini Project 6

Revenue Management of Carparks

Author: Dr. Paul Johnson, Paul.Johnson-2@manchester.ac.uk

Revenue Management is a relatively new area of economic pricing theory and is one that touches our lives on an almost daily basis. Whenever we use online booking systems to purchase perishable goods (such as travel tickets, concert tickets, or a hotel room) we shall often see products (such as different price classes) appear or disappear in real time. This is a dynamic form of revenue management in which the booking system uses some procedure to evaluate whether a product should be available or not.

Under the setting of a carpark booking system at an airport we shall use an object orientated approach to test different revenue management systems on a randomly generated set of customers. Further, you will be asked to develop a revenue management rejection algorithm.

6.1 Theory – The Carpark Model

In this project we try to develop algorithms to optimally manage the booking system of an airport carpark. We must try to manage the carpark as best we can to generate the most revenue possible. The carpark will have a finite number of spaces available which we can sell to customers. Bookings for the carpark may be made in advance on the internet or on arrival at the carpark. We assume that bookings, arrivals and departures from the carpark all happen in real time. A booking will consist of three separate times, the time the booking was made, the time of arrival and the time of departure. Given the capacity constraints we will need to make sure that space is available (at the time of booking) for the duration of the stay. Then the ith booking made on the system $B^i$ may be written as

$$B^i = \begin{pmatrix} t_b \\ t_a \\ t_d \end{pmatrix}$$

where $t_b$ is the booking time, $t_a$ the arrival time and $t_d$ is the departure time.

In order to keep track of all bookings made over time and their effect on the number of cars in the car park, and also the revenue generated by the bookings, we split the problem into discrete
timesteps. Take the interval in time \( t \in [a,b] \) and split it into \( K \) equally sized steps of length \( \Delta t \) so that

\[ t^k = a + k\Delta t. \quad \text{for} \quad k = 0, 1, \ldots, K \]

Then let \( C^k \) denote the number of cars present in the car park at any time during the period \( t \in [t^k, t^{k+1}) \). We may write

\[ C^k = \sum_i f(B^i) \quad \text{where} \quad f(B) = \begin{cases} 
1 & \text{if} \quad t^k \leq t_a < t^{k+1} \\
1 & \text{if} \quad t^k < t_d \leq t^{k+1} \\
1 & \text{if} \quad t_a < t^k \quad \text{and} \quad t_d > t^{k+1} \\
0 & \text{otherwise} 
\end{cases} \quad (1.1) \]

Note that we discount cars departing at \( t = t^k \) as being present during the period. We can also calculate the duration of stay for a booking as the number of periods at which a single booking is present in the car park

\[ D^i = \sum_k f(B^i) \quad (1.2) \]

This is useful in our model as the price paid will be linked to the duration of stay.

Now let the price per period for a booking \( B^i \) be given by a pricing function \( p(D^i) \) where \( D \) is the duration of stay. It follows that the revenue generated in the \( k \)th period is

\[ R^k = \sum_i f(B^i)p(D^i). \quad (1.3) \]

and a typical pricing function may look like

\[ p(D) = \frac{\alpha + \beta D}{D} \quad (1.4) \]

where \( D \) is the duration in days and \( \alpha \) and \( \beta \) are positive constants.

Therefore we can calculate the total revenue for the carpark as

\[ V^k = \sum_k \sum_i f(B^i)p(D^i) \quad (1.5) \]

### 6.1.1 Generating events from a Poisson Process

To generate the set of bookings we use the Poisson distribution. The Poisson distribution gives the probability that a number of events occur during a fixed time interval. The average number of events that occur in the interval is usually denoted by \( \lambda \). If we let the average number of bookings that are made within the period of a day (our standard unit of analysis in this project) be denoted by \( \lambda_b \), sometimes called the intensity of the process, then the probability that \( k \) bookings are made is given by

\[ P(k; \lambda_b) = \frac{(\lambda_b)^k e^{-\lambda_b}}{k!} \]

More important to us is how we can generate random samples from the Poisson distribution using a random sample from the uniform distribution (available on all C++ compilers).

If \( u^n \) is an independent random draw from a uniform distribution then the function to generate the time of the next event \( t^n \) of a Poisson distribution with intensity \( \lambda \) given \( t^{n-1} \) the time of the last event is given by:

\[ t^n = t^{n-1} - \frac{1}{\lambda} \log(u^n) \quad (1.6) \]
6.1. THEORY – THE CARPARK MODEL

6.1.2 Generating a Booking

Given that we have generated the time of the next booking, we need to know when the customer will arrive in the car park and how long they will stay. Note that if the arrival intensity is given by $\lambda$, then the average time of the next event will be $\frac{1}{\lambda}$. So if we wish to have a process where the average time between booking and arrival is say, 28 days, and the average time between arrival and departure is 7 days, then we may use Poisson processes with intensity $\lambda_a = \frac{1}{28}$ and $\lambda_d = \frac{1}{7}$ for the arrival and departure times respectively.

Put simply, given three random draws from a uniform distribution and the time the last booking was made $t_i$ we may generate the next booking as:

$$B_{i+1} = \begin{pmatrix}
  t_b &=& t^i - (1/\lambda_b) \log(u^n) \\
  t_a &=& t_b - (1/\lambda_a) \log(u^{n+1}) \\
  t_d &=& t_a - (1/\lambda_d) \log(u^{n+2})
\end{pmatrix}$$

(1.7)

Note that this model is in no way realistic to the real world (no correlations for a start!) but it is very easy to generate, and could easily be extended to allow the intensity parameters to become functions of time to capture weekly or seasonal effects.

6.1.3 Optimal Revenue Management

Obviously the explanation of Revenue Management in this project will be brief to say the least but we wish to lay out the basics relevant to the simplified model in this project. This project focuses on the problem of capacity allocation. The problem arises when the same product (a space in a car park) is sold to different customers at different prices, so the question arises of how many bookings we should allow the low-price customers to make when there is a possibility that high-price customers may arrive later on.

In our carpark model, we assume that there exists a set of low price customers that take advantage of the large discount for staying in the car park for a long period. They tend to book into the car park online in advance, and are typically leisure passengers. In contrast, we assume that there exists a set of high-price customers who turn up and pay a high price to stay for only a day or two. They are typically business passengers, and do not book trips in advance, and therefore neither do they book their carparking in advance. In this model we do not assume that different customers are subject to different prices, just that they will receive a discount the longer they stay in the car park. Price is assumed to be given as a function of duration of stay as in equation (1.4) with positive constants.

You will try two different methods of rejecting bookings. The first will be simply to assign a constant proportion of the carpark to each of the customers. You can then find the optimal proportion to assign to each set of customers by comparing the revenue generated in each case.

The second more difficult method will be to generate a simple rejection rule based on the number of spaces left in the carpark, the price per period of the booking, and the time remaining the period in question. We should reject a booking only if the total revenue generated from the booking is less than the expected revenue from future bookings that the car will displace over all periods it is present. For example, imagine that it is Friday and we have one space left in the carpark on Monday and one space left for Tuesday. A booking is requested on the system that will stay both days, and if we generate our price from (1.4) with $\alpha = 10$ and $\beta = 5$ the the booking will pay a rate of £10 per day. Now if we know that the probability that a someone will turn up and stay for one day on Monday and Tuesday and pay £15 per period is 0.8, then we can say that the net contribution from the booking is $(10 - 15 \times 0.8) + (10 - 15 \times 0.8) = -£4$. This means we should reject the booking and take our chances that the high-price customers will arrive. Alternatively if the carpark had 100 spaces left, then the carpark is unlikely to fill, the
value of the spaces displaced is low (zero if there is no chance that the carpark will fill up) and we should accept the booking. How we should determine the notional value of the space that is displaced is left for you to discover.

6.2 Exercises

6.2.1 The Customers class

- Create a function that returns the time of the next event for a Poisson distribution with intensity $\lambda$. You may use the internal c++ random number generator `rand()` to generate samples from a uniform distribution. Alternatively you may use one on the Net, see the Mersenne Twister algorithm for example.

- Enter the following data structure for bookings into your code:

```cpp
struct Booking{
    double bookingTime;
    double arrivalTime;
    double departureTime;
};
// compare for sort algorithms
bool operator<(const Booking &x, const Booking &y);
```

Complete the implementation for the “less than” operator $<$ so that bookings may be ordered by `bookingTime`. You may also wish to overload stream outputs.

- Now create a class (similar to the MVector) to store all of the bookings for one class of customer.

```cpp
class Customers{
    std::vector<Booking> vectorBookings;

public:
    // generate a set of bookings
    void generateBookings(double bookingRate, double arrivalRate, double departureRate, double startTime, double finishTime);
};
```

Complete the implementation for the generating bookings function. The first three arguments should be the intensity parameters for the Poisson distributions. Using `startTime` as your initial time, keep generating bookings until you reach `finishTime`. See section 6.1.2 for information on how to generate bookings.

- Complete the class by writing member functions to allow access to `vectorBookings` and also to return the number of bookings (see MVector for how we did this with `double` as the data type).

- Generate bookings using $\lambda_b = 5$, $\lambda_a = 1/14$, and $\lambda_d = 1/7$ in the interval $t \in [0, 150]$ and write them to screen. These will be the low-price leisure customers.

- Generate bookings using $\lambda_b = 25$, $\lambda_a = 1$, and $\lambda_d = 2$ in the interval $t \in [0, 150]$ and compare them to the previous booking set. These will be the high-price business customers.

- Write a member function to add bookings from another `Customer` class, the definition (inside the class) should look like:

```cpp
void addBookings(Customers &C);
```
Complete the implementation. Try adding the business and leisure customers together to form one single set of ordered bookings.
You can sort a vector using the std::sort algorithm. Simply include the algorithm library, then writing `sort(vectorBookings.begin(), vectorBookings.end())` should sort the bookings so long as the less than operator has been overloaded correctly. Print your bookings and check that the sets have combined correctly.

## 6.2.2 The Carpark Class

- You are encouraged to develop classes for the rest of the project but may proceed with a more procedural approach if you wish.
- Writing a class (or otherwise) create storage for the number of cars present in the carpark in each period \( k \) and also the revenue they generate (you could combine both using your own data structure). Now create a function that takes as input:
  - an empty vector for cars present
  - a set of customers
  - the start time
  - the end time
  - the number of periods

and gives as output
  - a vector containing the number of cars present in the carpark for each period

You will need to use equation (1.1) to calculate the entries for each period.

- What is the expected number of cars in the carpark at \( t = 150 \) using either of the sets of customers.
- Write in a price function of the form shown in equation (1.4) with \( \alpha = 10 \) and \( \beta = 5 \) and use it to calculate the revenues generated at each period, and also the total revenue.
- Now include a capacity into the carpark as an argument to a function or as a data member of the class. Write a function to check if space is available for the duration of the booking, and use it to reject bookings if there is no space.
- Using a combined set of both customer types, what is the expected time that a carpark with 50 spaces will first reach capacity?

## 6.2.3 Revenue Management

- For the combined set of customers, calculate the total expected revenue for carparks with capacity 10, 20, 30, ..., 100.
- Now create two different carparks for each set of customers so that the sum of the capacity of each carpark is equal to capacity of the combined carpark. Investigate what is the optimal proportion of the carpark to reserve for each set of customers.
- Given the combined set of customers, can you write a new booking rejection function that takes into account the notional expected revenue of a carpark space?
6.3 Report

You should prepare a report in the form of a continuous piece of prose that describes the problem and your solution to it. In particular, you should include the following:

- Detail any classes that you have used in the problem.
- For the unlimited capacity case, analysis of the expected number of cars in the carpark, and also the expected revenue.
- After including capacity to the problem, analyse the expected time that a carpark with 50 spaces will first reach capacity.
- Analysis of the optimal proportion of the carpark that should be reserved for the business customers.
- Detail either your new rejection function or at least how you think it may be implemented. Does your new rejection ‘beat’ the optimal proportion?

Bibliography

Mini Project 7

Random Numbers, Stochastic Simulation and the Gillespie Algorithm

Author: Mark Muldoon, M.Muldoon@Manchester.ac.uk

The chemistry I learned in school involves reactions among huge numbers of molecules, so huge that chemists have a special unit, the mol\textsuperscript{1}, to refer to the vast quantities of molecules that are typically in play. But there are some scientifically important settings in which the number of molecules is necessarily very much smaller. In these cases the time courses of reactions show evidence of stochasticity arising from the essentially random, thermally-driven collisions among the reactants. One particularly important example is the network of reactions associated with the regulation of gene expression: most cells have only a few—often as few one or two—copies of any given gene.

In this project you’ll study the Gillespie algorithm Gillespie (1976, 1977), which is a standard approach to stochastic simulation of chemical reaction networks. Applications to biological systems have prompted a lot of recent research in this area and there is a sizeable literature. Des Higham has very nice pedagogical article Higham (2008) that compares the Gillespie approach to other commonly-used models of chemical kinetics while Darren Wilkinson’s recent book Wilkinson (2006) offers a more comprehensive survey with special attention to statistical issues. The most recent edition of Numerical Recipes Press et al. (2007) also treats this topic, in the last part of the chapter on ODE solvers.

In addition to implementing the algorithm itself, you’ll also learn how to generate random numbers drawn from various distributions and how to define and use a \textit{virtual base class}. Such a class provides a sort of generic outline for a whole family of a C++ classes, but leaves some details to be specified in derived classes: if you were interested in algebraic number theory you might start with a virtual base class called \texttt{Ring} and specify that all its derived classes had to implement multiplication, subtraction and addition operators. You could then go on to write such derived classes as \texttt{IntegerRing}, \texttt{GaussianIntegerRing} and \texttt{PolynomialRing}, each with its

\textsuperscript{1}\footnote{A mole is an Avogadro number, \( L \approx 6.02 \times 10^{23} \), of atoms or molecules a substance. It is currently defined as the number of atoms in a 12 gram sample of pure \textsuperscript{12}C, the carbon isotope whose nucleus has six neutrons and six protons.}
7.1 Theory

Although the simulation algorithms we’ll study apply to arbitrary networks of chemical reactions, for the sake of concreteness I’ll focus on a particular example, the Michaelis-Menten system. In this, as in many other matters, my treatment is guided by Des Higham’s excellent article Higham (2008). I’ll begin by working with an ODE-based model to obtain a quasi-steady-state approximation to the dynamics (1.3) that was first proposed by Leonor Michaelis and Maud Leonora Menten Michaelis and Menten (1913): my account owes a great deal to Chapter 6 of Jim Murray’s famous book Murray (2003). I’ll then explain how to replace the ODE model with a discrete-state, continuous-time Markov process and talk about how to simulate such a process on a computer. This leads naturally into a discussion about the generation of random numbers, which completes the theoretical background.

7.1.1 An ODE model: chemical rate equations

The Michaelis-Menten system is the standard model of enzyme catalysis, the process through which one kind of molecule, the enzyme, facilitates the conversion of a second type of molecule, the enzyme’s substrate, into a third type of molecule, the reaction’s product. Although the enzyme participates in this process, it is neither created nor destroyed—it acts as a catalyst.

\[
S + E \xrightleftharpoons[{k_{-1}}]{{k_1}} C \xrightarrow{k_2} P
\]

The system involves three basic reactions, the first of which is a reversible step in which the one molecule each of the enzyme \(E\) and substrate \(S\) combine to form a complex \(C\). This complex can either dissociate—fall apart back into its constituent \(E\) and \(C\) molecules—or the action of the enzyme can transform the substrate molecule into a molecule of product \(P\), liberating the enzyme to form a new complex. The product formation step is taken to be irreversible and so, eventually, one expects that all the substrate will be converted to product.

The standard ODE model of this system describes the time evolution of the concentrations of the various chemical species, which I’ll denote with lowercase symbols:

\[
\begin{align*}
\frac{dc}{dt} &= -k_1 es + (k_{-1} + k_2)c \\
\frac{ds}{dt} &= -k_1 es + k_{-1}c \\
\frac{dc}{dt} &= k_1 es - (k_{-1} + k_2)c \\
\frac{dp}{dt} &= k_2c
\end{align*}
\]  

Note that there are two sorts of terms on the right hand sides of the equations above, linear ones such as the \(k_2C\) appearing in the ODE for the product and the bilinear terms \(k_1 ES\) that involve the formation of the complex. These are manifestations of the law of mass action, which says that the rate at which a reaction proceeds is proportional to the product of the concentrations of the reactants. The connection between this law and microscopic models of what happens at the molecular level are incompletely understood, but models such as (1.1) are used very widely and with great success.
7.1. THEORY

One can simplify (1.1) by noting that product formation is such a uncomplicated business that, given \( c(t) \), the time course of the concentration of the complex, one can obtain \( p(t) \) via straightforward integration:

\[
p(T) = p(0) + \int_0^T k_2 c(t) \, dt.
\]

Also, as enzyme molecules can be either free or locked up in a complex with substrate, but are neither created nor destroyed, one has that

\[
e_0 \equiv e(0) + c(0) = e(t) + c(t) \quad \text{so} \quad e(t) = e_0 - c(t).
\]

These observations mean that its sufficient to study the simpler system

\[
s' = -k_1 e_0 s + (k_1 s + k_{-1})c
\]

\[
c' = k_1 e_0 s - (k_1 s + k_{-1} + k_2)c
\]

(1.2)

The standard account of (1.2) investigates the initial value problem in which, at the outset, there is much much more substrate than enzyme, \( s_0 \equiv s(0) \gg e(0) \), and all other concentrations are zero. In this limit \( dc/dt \approx 0 \) (this is sometimes called the quasi-steady state hypothesis) and so the dynamics are well described by

\[
c(t) = \frac{e_0 s(t)}{s(t) + K_m}
\]

(1.3)

where \( K_m = (k_{-1} + k_2)/k_1 \) is called the Michaelis constant. Putting this form into the expression for \( ds/dt \) yields

\[
\frac{ds}{dt} = -\frac{k_2 e_0 s}{s + K_m},
\]

which one can integrate to obtain an implicit solution

\[
s(t) + K_m \ln \left( \frac{s(t)}{s_0} \right) = s_0 + k_2 e_0 t.
\]

(1.4)

The solution (1.4) is valid for all save the very earliest times, but a careful asymptotic analysis shows that there is a short period whose duration is of order

\[
\tau = \frac{1}{k_1 (s_0 + K_m)}
\]

(1.5)

during which complex is formed rapidly and so one cannot assume \( dc/dt = 0 \). We will be interested in exploring the regime in which the number of molecules is so small that it becomes unhelpful to consider concentrations and so the ODEs (1.2) and even (1.1) are poor approximations.

7.1.2 Markov formulation

If we restrict our attention to a volume \( v \) so small that there are very few molecules indeed, the picture changes radically. Continuously-varying concentrations are replaced by actual numbers of molecules and so the state of the Michaelis-Menten system is described by a vector of whole numbers

\[
n(t) = (N_S(t), N_E(t), N_C(t), N_P(t))
\]
which I’ll refer to as the populations of the various chemical species. The number of possible states is now finite, though potentially very large: Table 1 lists all the states compatible with the initial conditions where there are three molecules of the substrate, two of the enzyme and none of the complex or product.

The temporal evolution of the system now consists of periods spent in one of the system’s finitely many states, punctuated by abrupt jumps from state to state. These transitions occur when one of the three basic reactions—complex formation, dissociation and product formation—occur: Figure 7.1 illustrates the nine states from table along with the transitions between them implied by the various reactions. Transitions are imagined to occur at random, with average rates that depend on the populations of the participating chemical species. That is, we’ll imagine that, at least in the limit of very short time intervals \( \Delta t \), the probability that, say, a complex will form is proportional to some rate \( r(n(t)) \):

\[
P(\text{reaction while } t_0 \leq t \leq t_0 + \Delta t) \approx r(n(t_0)) \times \Delta t \tag{1.6}
\]
7.1. THEORY

We’ll defer the problem of determining the rates $r(n(t))$ for a moment and concentrate on what (1.6) implies about the dynamics. All possible reactions are imagined to proceed independently and so the probability that no reactions occur in an interval of duration $\Delta t$ is

$$\prod_j [1 - r_j(n(t)) \times \Delta t] = 1 - \Delta t \sum_j r_j(n(t)) + O(\Delta t^2).$$

where the sum and product range over all the reactions that are possible when the system is in the state $n(t)$. In the limit $\Delta t \to 0$ the expression above implies that transitions away from a state with populations $n_0$ occur at a rate

$$R(n_0) = \sum_j r_j(n_0) \quad (1.7)$$
or, equivalently, the gaps between such transitions are exponentially distributed with mean $\tau = 1/R(n_0)$. Further, given that such a transition has occurred, the probability that it went via some particular reaction is

$$P(\text{reaction } k \mid \text{transition away from } n_0) = \frac{r_k(n_0)}{R(n_0)} = \frac{r_k(n_0)}{\sum_j r_j(n_0)}. \quad (1.8)$$

Equations (1.7) and (1.8) are the heart of the Gillespie algorithm, which I’ll explain in detail in Section 7.1.3 below. But before we can apply these equations we need to work out the relationship between the rates $r_j(n_0)$ and more familiar, experimentally accessible descriptions of the dynamics. There are two complimentary ways to do this: on the one hand, one can choose the rates in such a way that, in the limit of large populations $n(t)$, the dynamics implied by (1.7) and (1.8) reduce to (1.1). On the other hand, one can try to argue from first principles, by thinking about collisions among the reactants.

If we imagine that the volume under discussion, $v$, is very small, then all the molecules will encounter each other frequently, undergoing repeated collisions. There will also be a great many other molecules present whose populations don’t appear in our model because these extra molecules (water, other enzymes, etc. . . ) don’t participate in the reactions that interest us. If we further imagine that most collisions are without consequence—that is, that the probability that any given collision leads to a reaction is low—then it is not unreasonable to imagine that the rate at which pairwise reactions occur is proportional to the number of possible pairs of molecules of reactants. So, for example, the rate of complex formation would be proportional to the number of enzyme-substrate pairs: $N_E \times N_S$. We thus arrive at a formula for the rate of complex formation that looks like

$$r(n) = \alpha_{SE} N_S N_E. \quad (1.9)$$

Now consider what this implies about the large-$n$ limit. The reaction whose transition rate is (1.9) produces one molecule of the complex and so this implies

$$\frac{dc}{dt} = \lim_{\Delta t \to 0} \frac{\Delta c}{\Delta t} = \frac{1}{\Delta t} \left( \frac{\Delta N_{CE}}{Lv} \right) = \frac{1}{\Delta t} \left( \frac{(\alpha_{SE} N_S N_E) \Delta t}{Lv} \right) = \alpha_{SE} \frac{N_S}{Lv} \left( \frac{N_E}{Lv} \right) = (\alpha_{SE} Lv) c. \quad (1.10)$$
Table 7.2: The reactions in a Gillespie simulation of the Michaelis-Menten system, along with the corresponding changes in the system’s state and the rates. Here \( k_1, k_{-1} \text{ and } k_2 \) are as in (1.1) while \( \mathcal{L} \) is Avogadro’s number and \( v \) is the volume of the system.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Reaction} & \delta N_S & \delta N_E & \delta N_C & \delta N_P & \text{Rate} \\
\hline
\text{Complex formation} & -1 & -1 & 1 & 0 & k_1(N_SN_E)/(\mathcal{L}v) \\
\text{Complex dissociation} & 1 & 1 & -1 & 0 & k_{-1}N_C \\
\text{Product formation} & 0 & 1 & -1 & 1 & k_2N_C \\
\hline
\end{array}
\]

where I’ve repeatedly used factors of \( \mathcal{L}v \)—the product of Avogadro’s number and the system’s volume—to convert between concentrations measured in moles per litre and whole-number populations. Also, in passing from the second line to the third I’ve replaced \( \Delta N_C \) with \((\alpha_SEN_SN_E)\Delta t\) which, given rate formula (1.9), is the expected number of complex formations: it tends to zero with \( \Delta t \).

Comparison of the last line above with (1.1) suggest that

\[
\alpha_{SE} = \frac{k_1}{\mathcal{L}v},
\]

Similar considerations suggest that the two remaining reactions, which both involve a single molecule of the complex, have the rates given in Table 7.2.
Figure 7.1: The diagram above shows the states and transitions for a Gillespie model of the Michaelis-Menten system. The states—which are numbered as in Table 7.1—are specified by pairs \((N_C, N_P)\). Transitions corresponding to complex formation are shown as curved, blue arrows while those arising from dissociation of the complex are curved and green. The straight, orange arrows represent state transitions due to formation of the product.
7.1.3 The Gillespie algorithm

Once we have the information in Table 7.2 the Gillespie algorithm is easy to describe. Assume the various chemical species have initial populations \( n_0 \). We’ll need keep track of the time \( t \), a vector of populations \( n \) and a vector of rates \( r = (r_1, r_2, \ldots) \). Also associated with the \( j \)-th reaction is a vector \( \delta n_j \) that gives the corresponding changes in the populations.

1. **Initialization:**
   \[
   t \leftarrow 0 \\
   n(0) \leftarrow n_0
   \]

2. **Compute reaction rates:**
   \[
   r \leftarrow r(n)
   \]

3. **Choose \( \delta t \):** Generate a random interval \( \delta t \) drawn from the exponential distribution with mean \( \tau = 1/R(n) \) where \( R(n) \) is as in (1.7).

4. **Update the time:**
   \[
   t \leftarrow t + \delta t
   \]

5. **Choose a reaction:** Use the probabilities (1.8) to decide at random which reaction occurs.

6. **Update the populations:** Say we chose reaction \( k \) in step (5)
   \[
   n \leftarrow n + \delta n_k
   \]

7. Either stop or return to step (2).

This sketch leaves out a few details that are important in implementation, but we’ll address these below, in Section 7.2. By concentrating on the Michaelis-Menten system we have ignored an important class of reactions, the so-called dimerizations. In these reactions two molecules of the same substance—referred to as the monomer—combine to form a single new molecule called a dimer. If we write the reaction as

\[
X + X \xrightarrow{k} D
\] (1.11)

then the corresponding rate in a Gillespie simulation is

\[
r = N_X(N_X - 1) \times \left( \frac{k}{L_v} \right)
\] (1.12)

The factor \( N_X(N_X - 1) \) arises from counting the number of pairs of the monomers \( X \), but it also makes sense in that it serves to make the rate zero when dimerization is impossible because \( N_X = 1 \).

7.1.4 Random numbers

*Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.*

John von Neumann (1951)

The Gillespie algorithm calls for lots of “random” time intervals and choices of reaction, so here I discuss briefly how to generate them. All commonly-used generators need to be initialized with some “seed” and subsequently produce a sequence \( x_j \) of so-called pseudorandom numbers. The “pseudo” here serves as a warning that these generators aren’t “really random”: rather,
7.1. THEORY

they are purely deterministic and, when given the same seed, yield exactly the same sequence of $x_j$ (this can be very convenient when debugging). On the other hand, the sequences these algorithms produce are random-like in the sense that short subsequences of the $x_j$ share many distributional and correlational properties with sequences of independent, identically-distributed random variables.

Random number generation in computers holds a certain paradoxical fascination and, especially in cryptographic applications, can be critically important for the success of algorithms. But the proofs behind the best generators involve deep connections to number theory and the literature can seem daunting. If you want to read some of it you might start with Chapter 3 of Donald Knuth’s celebrated *Art of Computer Programming* (1997). Chapter 7 of *Numerical Recipes* (Press et al. 2007) offers a practical discussion of random number generation aimed at the general mathematically literate reader.

Uniformly distributed numbers

The basic generator that we’ll use is the Mersenne Twister of Matsumoto and Nishimura (1998), which is currently the default generator in MATLAB, Maple and the statistical package R. The authors have proposed a range of initialization schemes, but the implementation we’ll use takes a single integer as a seed and then produces a sequence of unsigned, 32-bit integers uniformly distributed over the range

$$0 \leq n_j \leq 2^{32} - 1.$$  

It is easy to convert these into floating point numbers $u_j = n_j / 2^{32}$ that are approximately uniformly distributed over the interval $[0, 1)$ and so the rest of the algorithms discussed below will assume the availability of a function whose prototype is

```c
// Return a pseudorandom u drawn from the
// uniform distribution on [0, 1).
double uniform ( void );
```

Exponentially distributed numbers

We’ll need to generate sequences of numbers $t \geq 0$ drawn from the continuous distribution whose probability density function is $f(t) = Re^{-Rt}$. That is, we want it to be true that

$$P(a \leq t \leq b) = \int_a^b f(t) \, dt = \int_a^b Re^{-Rt} \, dt = e^{-Ra} - e^{-Rb}$$

This distribution has mean and variance $\tau = 1/R$ and one can generate samples using the method of transformation.

The idea is to start with a random a value $u \in [0, 1)$ and then interpret it as a value of the cumulative density function (CDF) of the distribution from which you’d like your $t$ to be drawn. Here the relevant CDF is defined by

$$F(T) = P(t \leq T) = \int_0^T Re^{-Rt} \, dt = 1 - e^{-RT}.$$
// Initialization: lambda is a parameter supplied by the user.
unsigned k = 0; // The result
double crnt_product = 1.0;
double target_product = exp(-lambda);

// Multiply uniformly distributed u’s
// until the product falls beneath the target.
crnt_product *= uniform();
while (crnt_product > target_product) {
    crnt_product *= uniform();
    ++k;
}
return(k);

Listing 7.1: Knuth’s algorithm for the Poisson distribution

It’s easy to see that $F : [0, \infty) \to [0, 1)$ and so its inverse maps the unit interval to the positive real line. The formula that transforms a uniformly-distributed $u \in [0, 1)$ to an exponentially distributed $t \in [0, \infty)$ is

$$t = F^{-1}(u) = -\ln(1-u) \quad (1.14)$$

The Poisson distribution

If we have a large volume of liquid in which the concentration of some chemical is $C$ then the number of molecules in a small subvolume $v$ is, on average, $Cv$. But we may be interested in such small volumes that the number of molecules is more properly treated as a random variable. In this case we should use the Poisson distribution with mean $\lambda = Cv$,

$$P(k \text{ molecules in volume } v) = \frac{\lambda^k}{k!} e^{-\lambda}, \quad (1.15)$$

which has mean and variance $\lambda = Cv$.

Fast algorithms to generate Poisson random variables rely on so-called acceptance-rejection methods, a circle of techniques I won’t discuss here (see Chapter 7 of Press et al. (2007) if you are interested), but we won’t need to produce very many such values and so we’ll use a slower, but very straightforward algorithm due to Knuth: see Listing 7.1.

Arbitrary (finite) discrete distributions

Finally, we will often need to generate a random number drawn from a discrete distribution with a finite range. Say that we want a number $n$ drawn from the set $\{0, 1, \ldots, N\}$ with distribution $P(n = j) = p_j$, where $p_j = 0$ unless $0 \leq j \leq N$ and

$$1 = \sum_{j=0}^N p_j.$$ 

The algorithm we’ll use is essentially a discrete version of the method of transformation from Section 7.1.4. Begin by defining the discrete analogue of the cumulative density function

$$s_k = \sum_{j=0}^k p_j$$
for $0 \leq k \leq N$ and note that $s_N = 1$. To generate a random $n$ drawn from the desired distribution, first generate a random $u$ drawn from the uniform distribution on $[0, 1)$, then say 

$$n = \min \{ k \mid 0 \leq k \leq N \text{ and } s_k \geq u \}. \quad (1.16)$$

### 7.2 C++ Implementation

It’s easy enough to implement the Gillespie algorithm to simulate the Michaelis-Menten system—Higham provides an example that runs to no more than a page MATLAB code—but you should write something of greater generality. To do this we’ll define a virtual base class to describe a general Reaction and then use it to write a class that provides the bulk of the machinery for a GillespieSimulation: the latter includes a vector that holds the populations of the various chemical species. Then to simulate a given reaction network you’ll need only write a derived classes for the relevant reactions and the network.

#### 7.2.1 Reactions

The things one needs to know about a reaction are (a) which changes it induces in the molecular populations and (b) how to compute its rate. The virtual base class defined in listing 7.2 contains enough information to update populations, but doesn’t know how to compute rates: that’s devolved to derived classes.

Listing 7.3 provides an example of a derived class for the dimerization reaction (1.11). There is a small subtlety in the implementation of the rate formula (1.12) because one will get nonsense if one computes $(n_{\text{monomer}} - 1)$ when $n_{\text{monomer}} == 0$ and $n_{\text{monomer}}$ is a size_t, which is an unsigned type.

#### 7.2.2 The GillespieSimulation class

Given those features of the reactions required by the virtual base class, it’s possible to write a class that does most of the hard work of performing a Gillespie simulation. Listing 7.4 declares a simple GillespieSimulation class, some of whose members I’ll ask you to implement. You can download the header, as well as a skeleton of the source file that implements it, from http://bit.ly/mrmNScCpp.

#### 7.2.3 Remarks

- My code makes frequent use of assert(). The idea is to lard your code with expressions like

```c
#include <cassert>

// etc ...

assert( /* something that should be true */ ) ;
```

for example,

```c
assert( volume != 0.0 ) ;
ratio *= 1.0 / volume ;
```
MINI PROJECT 7. GILLESPIE ALGORITHM

#include <vector>

// See http://physics.nist.gov/cgi-bin/cuu/Value?na
const double Avogadro_const = 6.02214179e23;

// Define a virtual base class for reactions

class Reaction
{
public:
    // All reactions have these
    double rate_const; // from the ODE
    std::vector<std::size_t> rateDependsOn;
    std::vector<std::size_t> speciesAffected;
    std::vector<int> deltaN;

    // A do-nothing constructor: derived classes
    // should define their own constructors that
    // set up the various vectors above.
    Reaction() { rate_const = 0.0; }

    // Armed with the data above, one can make the appropriate
    // changes in chemical species numbers, even if one doesn’t
    // know what the reaction is.
    void doReaction(std::vector<std::size_t>& particles) const
    {
        for (std::size_t j = 0; j < speciesAffected.size(); ++j)
            particles[crnt_species] += deltaN[j];
    }

    // The member defined below is the one that makes
    // this a virtual base class: given the volume and
    // the populations of all the chemical species,
    // derived classes must know how to compute their own rates.
    virtual double rate(const std::vector<std::size_t>& pop, double volume) const = 0;
};

Listing 7.2: The virtual base class for reactions.
class Dimerization : public Reaction // X + X -> D 
{
    public:
    // The creator requires the indices (in the population vector)
    // of the species X and D.
    Dimerization( double k, std::size_t x_idx, std::size_t d_idx )
    {
        rate_constr = k ; // Install the rate constant
        // Note which species the rate depends on
        rateDependsOn.resize( 1 ) ;
        rateDependsOn[0] = x_idx ;
        // Note which species the reaction affects
        speciesAffected.resize( 2 ) ;
        speciesAffected[0] = x_idx ; speciesAffected[1] = d_idx ;
        // Note the consequences of the reaction
        deltaN.resize(2) ;
        deltaN[0] = -2 ; deltaN[1] = 1 ;
    }

    // Compute the rate
    double rate( const std::vector<std::size_t> &pop ) const
    {
        std::size_t monomer_idx = rateDependsOn[0] ;
        std::size_t n_monomer = pop[monomer_idx] ;
        // Avoid sickness with (n_monomer - 1) when
        // (n_monomer == 0)
        double rate = 0.0 ; // the default
        if( n_monomer > 0 ) {
            rate = rate_constr * n_monomer * (n_monomer - 1) ;
        }
        return( rate ) ;
    }
};

Listing 7.3: A derived class for dimerizations
Listing 7.4: A class of objects to do Gillespie simulations
The point of this software idiom is that if the argument of `assert()` isn’t true, your code quits instantly, printing an error message to tell you where it died: this can be very handy when you are developing a program. It’s also easy to turn assertion-checking off once you’re finally ready to use your code for production work: see the documentation for `<cassert>` to learn how.
7.3 Exercises

The first few exercises have to do with random number generation.

1. Prove that the transformation (1.14) works. That is, show that if \( u \) is distributed uniformly over \([0, 1)\) and \( t \) and \( u \) are related by (1.14), then \( t \) is distributed according to (1.13).

2. Prove that Knuth’s algorithm in Section 7.1.4 really does draw random samples from a Poisson distribution. You may find the following lemma helpful: it’s easy to prove by induction on \( n \).

**Lemma.** The volume of the set \( \Delta^n \in \mathbb{R}^n \) defined by

\[
\Delta^n = \{ (t_1, \ldots, t_n) \in \mathbb{R}^n | \sum_j t_j \leq \lambda \text{ and } t_j \geq 0 \text{ for all } j \}
\]

is \( \lambda^n / n! \).

3. Explain briefly why the algorithm sketched in Section 7.1.4 works. In particular, is there a problem if some of the \( p_j \) are zero?

4. Write a C++ class called `PseudoRandomGenerator` that inherits from a base class called `MersenneTwister`. The latter, whose source you can download\(^2\), has a member called `uniform()` that generates random numbers distributed uniformly on \([0, 1)\). See Listing 7.5 for an outline of the class declaration for `PseudoRandomGenerator`.

5. Test your generators as follows

(a) Generate 10,000 exponentially distributed numbers using the parameter value \( r = 1.0 \) and compute the sample mean and variance: both should be close to 1.0.

(b) Generate 10,000 Poisson-distributed numbers with parameter value \( \lambda = 10 \) and again compute the mean and variance: both should be close to 10.

(c) The binomial distribution has two parameters, the number of trials \( N \) and the probability of success \( p \), and it assigns probabilities to non-negative integers according to

\[
p_k = \binom{N}{k} p^k (1-p)^{N-k}
\]

If \( 0 \leq k \leq N \)

otherwise

You can think of \( p_k \) as, say, the probability of getting \( k \) Heads in \( N \) tosses of a coin where the probability of getting Heads on a single toss is \( p \). Use your `discrete` generator to draw 10,000 values from the binomial distribution with \( N = 10 \) and \( p = 0.35 \) and keep track of the number of times each possible value occurs (That is: How many 0’s do you get? How many 1’s ... ). Print a small table listing \( p_k \) and the number of times the value \( k \) came up.

6. The most straightforward implementation of `discrete()` includes something like the following `linear search`

```cpp
// Get a u from [0, 1)
double u = uniform();

// Seek k such that s[k] equals or exceeds u
unsigned k = 0;
while(s[k] < u) { ++k; }
```

// Extend the MersenneTwister class to generate more types of random numbers.

// ifndef _PseudoRandomGenerator_  // Include these lines just once
# define _PseudoRandomGenerator_
// Include standard headers
# include <vector>

// The underlying uniform generator
# include "MersenneTwister.h"

class PseudoRandomGenerator : public MersenneTwister
{
public:
  // Use the inherited constructors - we don't need a destructor as default is okay.
  PseudoRandomGenerator() : MersenneTwister() {}
  PseudoRandomGenerator(unsigned seed) : MersenneTwister(seed) {}

  // Exponentially distributed numbers
double exponential(double r)
  {
    // your code here
  }

  // Poisson distributed numbers
  unsigned poisson(double lambda)
  {
    // Your code here
  }

  // Numbers drawn from a discrete distribution
  unsigned discrete(std::vector<double> &cdf)
  {
    // Your code here
  }
}; //End of the class definition

Listing 7.5: A skeleton for a random number generator class.
Figure 7.2: The distribution of birth dates for kids born in New York state during 1978. The data for weekdays are shown as green dots, those for Saturdays in blue and those for Sunday in red: the solid yellow curve is a two-week moving average intended to smooth out the weekly fluctuations and highlight the seasonal variation. By far the most striking feature of these data are that obstetricians didn’t like to work weekends. The few green dots that fall among the data for weekends are readily identified as holidays including Memorial Day (Monday 29 May), the 4th of July, Memorial Day (Monday 4 September), Thanksgiving (Thursday 23 November) and Christmas.

which has expected running time $O(N)$ for distributions over \{0, 1, \ldots, N\}. This is perfectly satisfactory for small $N$, but when $N$ is large—and it can be on the order of 1000’s for Gillespie simulations on full-scale metabolic networks—linear search can become a significant computational burden. One can make a more efficient implementation of discrete() by using the binary search routines provided as part of C++’s Standard Template Library (STL). The necessary declarations appear in <algorithm> and, for this application, the routine of choice is lower_bound(). Read the documentation for this routine and then write two versions of discrete(), one that uses linear search as above and another that does binary search.

To compare the speeds of these two implementations, we’ll do a numerical experiment inspired by the famous Birthday problem from elementary probability. The idea is to imagine people arriving at a party one at a time and ask: How big does the crowd have to get before it becomes likely that two guests share a birthday? If one assumes that all birth dates are equally likely (and ignores the 29th of February), then the probability of a shared birthday first exceeds 50% when there are around 23 guests. But real people’s birthdays aren’t uniformly distributed throughout the year, as Figure 7.2 shows.

Use the data\(^3\) behind Figure 7.2 to estimate $p_j$, the probability that a New Yorker born

\(^3\)The data, which are in a file called BdayDistrib_NY78.txt, consist of 365 lines like 1/1/78 7701, where the first item is a date and the second is the number of births on that date. This file is available for download from http://bit.ly/mrmMScCpp.
# include <ctime>  // for clock() and CLOCKS_PER_SEC

// Read the data, get the empirical distrib of birthdays,
// compute the corresponding cdf and store it in a vector
// of doubles called bday_cdf ... 

// Set up a random number generator
PseudoRandomGenerator prng;

// Time the linear search version
double linsearch_start = ((double) clock()) / CLOCKS_PER_SEC;
for( size_t j=0; j < N_REPS; ++j )
  // Get ready to keep track of the birthdays we've seen
  vector<bool> bday_seen_yet(365);
  for( size_t k=0; k < 365; ++k )
    bday_seen_yet[k] = false;

  // Simulate arrivals at the party and note how many
  // guests there are when the first shared birthday arises.
  size_t n_guests = 0;
  unsigned crnt_bday = prng.linsearch_discrete( bday_cdf );
  while( bday_seen_yet[crnt_bday] == false )
  {
    ++n_guests;
    bday_seen_yet[crnt_bday] = true;
    crnt_bday = prng.linsearch_discrete( bday_cdf );
  }

  double linsearch_stop = ((double) clock()) / CLOCKS_PER_SEC;

double linsearch_elapsed = linsearch_stop - linsearch_start;

Listing 7.6: A fragment to illustrate the use of clock() for timing.

in '78 was born on the j-th day of the year, then simulate 100,000 Birthday-problem type parties and accumulate data about when the first shared birthday arises. Do this with both versions discrete() and compare the running times. The necessary timing loop is sketched in Listing 7.6.
7. Download GillespieSimulation.h and GillespieSimulation.cpp and then implement the missing routines in the latter. The GillespieSimulation class includes a private member called rate and another called rate_sum: both are vectors of doubles. You should arrange things so that rate[j] holds the rate of the j-th reaction and

\[ \text{rate}_\text{sum}[j] = \sum_{k=0}^{j} \text{rate}[k]. \]

It’s then possible to choose the next reaction more efficiently than suggested in step 5 of the Gillespie algorithm, as one needn’t convert the vector of rate-sums into a discrete probability distribution. The idea is to first compute

```c++
double rx_u = R * prng.uniform();
```

where

\[ R = \sum_j \text{rate}[j] \]

is the sum of all the rates—it’s also the last entry in the rate_sum vector. Then choose reaction \( k \), where

\[ k = \min\{ j | \text{rate}_\text{sum}[j] \geq rx_u \}. \]

8. In addition to simulating the time course of chemical reactions one can use a Gillespie simulation to explore the distribution of chemical populations in systems at chemical (as opposed to dynamical) equilibrium. The idea is to simulate for a long time, taking samples at sufficiently widely spaced intervals that they are effectively probabilistically independent. The program Dimerization.cpp shows how to use a GillespieSimulation object in this way. Download it, study the code and then run it, using your implementation of GillespieSimulation.cpp.

The simulation concerns a small volume in which there are initially 11 monomers and no dimers. The chemistry includes a dimerization reaction like the one described in Eqn. (1.11) and Listing 7.3 and a dissociation reaction through which the dimer falls apart into two monomers. A separate analysis that I won’t discuss here shows that if one started off a great many identical copies of this system and allowed them to evolve stochastically then, after transients had died away, the probability that a randomly-selected copy of the system would have \( k \) dimers (and hence \( 11 - 2k \) monomers) would be given by the table below.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( p_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0094</td>
</tr>
<tr>
<td>1</td>
<td>0.0940</td>
</tr>
<tr>
<td>2</td>
<td>0.3076</td>
</tr>
<tr>
<td>3</td>
<td>0.3916</td>
</tr>
<tr>
<td>4</td>
<td>0.1780</td>
</tr>
<tr>
<td>5</td>
<td>0.0194</td>
</tr>
</tbody>
</table>
9. Write classes for the reactions needed to do a Gillespie simulation of the Michaelis-Menten system, then use them to run 100 simulations based on the parameters below, which come from an example in Wilkinson (2006).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_0$</td>
<td>Initial enzyme concentration</td>
<td>2.0e-7 moles per liter</td>
</tr>
<tr>
<td>$s_0$</td>
<td>Initial substrate concentration</td>
<td>5.0e-7 moles per liter</td>
</tr>
<tr>
<td>$c_0$</td>
<td>Initial enzyme complex</td>
<td>0.0 moles per liter</td>
</tr>
<tr>
<td>$p_0$</td>
<td>Initial enzyme product</td>
<td>0.0 moles per liter</td>
</tr>
<tr>
<td>$k_1$</td>
<td>Rate const. for complex formation</td>
<td>1.0e6 litres/mol×sec</td>
</tr>
<tr>
<td>$k_{-1}$</td>
<td>Rate const. for complex dissociation</td>
<td>1.0e-4 sec⁻¹</td>
</tr>
<tr>
<td>$k_2$</td>
<td>Rate const. for product formation</td>
<td>0.1 sec⁻¹</td>
</tr>
<tr>
<td>$v$</td>
<td>System’s volume</td>
<td>1.0e-15 liters</td>
</tr>
</tbody>
</table>

Before each of the 100 simulations, reset the initial populations, drawing the number of particles from a Poisson distribution whose mean is the product of the initial concentration and the volume. Run each simulation for 50 seconds (simulated time, not clock time) recording the populations every 0.5 seconds to produce a file whose rows look like:

```
1 0.0 291 113 0 0
1 0.5 268 90 23 0
1 1.0 258 81 32 1
. . .
1 48.5 2 99 14 275
1 49.0 2 100 13 276
1 49.5 1 99 14 276
1 50.0 1 100 13 277
2 0.0 315 120 0 0
2 0.5 287 92 28 0
2 1.0 270 75 45 0
. . .
```

Here the first column gives the number of the simulation (1–100), the second gives the time and the remaining columns are the populations of the chemical species.
7.4 Report

Prepare a report that describes your work on this project. You should write it in such a way that a fellow M.Sc. student who is not doing this project could read it with pleasure, or at least comprehension. It should include the following things:

- Brief answers to Exercise 1–3.
- Output from the test program or programs that you wrote to do Exercise 5.
- The results of your timing experiment from Exercise 6 and a discussion of why they turned out as they did. If linear search has expected running time $O(N)$, what is the expected running time of the binary search?
- A pair of plots to summarize your Michaelis-Menten simulations. Use your favourite plotting package to take the output from Exercise 9 and prepare a plot that shows all 100 simulated time courses of the ratio $N_S(t)/N_S(0)$, plotted as 100 separate curves, each in a light shade of grey. On top of these, plot the time courses of the median, 25-th percentile and 75-th percentile of the ratios. Then make a similar plot for the ratios $N_E(t)/N_E(0)$.
- A brief discussion of the efficiency of our implementation. If we were simulating a network with a great many reactants and reactions the computations done in `update_rates_n_sums()` could become very inefficient. Most reactions would change the populations of only a few species and those changes would lead to only a few changes in the reaction rates, but our straightforward implementation recomputes all the rates at every time step. Think about, and discuss, ways of speeding this step up.
- All your code—replete with comments and sensible variable names—in an appendix.

Bibliography


Mini Project 8

Efficient Sparse Matrix–Vector Multiplication

Author: Nick Dingle  nicholas.dingle@manchester.ac.uk

Multiplying a matrix $A$ by a vector $x$ forms the kernel of many scientific computing codes; see, for example, the accompanying project on the Conjugate Gradient method. When $A$ is sparse, which means that the majority of entries are zeros, the way in which the matrix is laid-out in memory can have a great effect on how quickly the matrix–vector multiplication $y = Ax$ can be performed.

This project will investigate two storage formats for sparse matrices: Compressed Sparse Row (CSR), which can be regarded as the “default” storage scheme, and Blocked Compressed Sparse Row (BCSR), which is designed to give faster run-times.

8.1 Sparse Matrix Storage Schemes

At the heart of the CSR format are three arrays:

- **value** contains the non-zero matrix entries. Its length is $nz$, which is the number of non-zero entries in the matrix.
- **col_ind** contains the column indices of the entries. Its length is $nz$.
- **row_ptr** contains the starting indices of the rows. Its length is the number of matrix rows + 1.

For example$^1$:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{bmatrix}$$

$^1$Thanks to Ramaseshan Kannan for the use of these examples
The following algorithm shows how the matrix–vector product \( y = Ax \) is computed for an \( n \times n \) matrix of doubles \( A \) stored in CSR format:

```cpp
void CSRMatrix::multiply(double* x, double* y) {
    for (i=0; i < n; i++) {
        y[i] = 0;
        for (j=row_ptr[i]; j < row_ptr[i+1]; j++) {
            y[i] += val[j] * x[col_ind[j]];
        }
    }
    return;
}
```

The CSR format is relatively straightforward to implement and saves on memory compared with storing all matrix entries (including zeros) explicitly. Its biggest drawback, however, is that the vector \( x \) is essentially accessed randomly during the multiplication, and this means that there are few opportunities to cache and reuse its values; the random accesses to \( x \) mean that a value from \( x \) is likely to be evicted from the cache by other values before it is reused, and thus it must be fetched from main memory again.

Blocked CSR is designed to overcome this problem. Instead of storing just the non-zero entries, as in CSR, it instead represents the matrix as an array of blocks that contain zero and non-zero entries (we say that these blocks are dense). Each block has \( r \) rows and \( c \) columns. If we take \( r = c = 2 \), the previous CSR matrix becomes:

\[
A = \begin{bmatrix}
a_{00} & a_{01} & a_{02} & a_{03} \\
a_{10} & a_{11} & 0 & 0 \\
0 & 0 & a_{22} & a_{23} \\
0 & 0 & a_{32} & 0
\end{bmatrix}
\]

\[
r = 2 \\
c = 2 \\
val = [a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, a_{22}, a_{23}, a_{32}, 0] \\
\text{col_ind} = [0, 1, 1] \\
\text{row_ptr} = [0, 2, 3]
\]

\( A \) is therefore stored as 3 dense blocks and \( \text{col_ind} \) and \( \text{row_ptr} \) now refer to the array of blocks rather than the columns and rows of the individual entries of the matrix. The information about the entries can be recovered by knowing which block they are in and the size of the blocks. Note that the bottom-left block is not stored as it contains only zeros, but that in the other three blocks we have to store the zeros as well as the non-zeros.

The matrix–vector multiplication for a BCSR matrix with \( r = c = 2 \) becomes:
8.2. CODING, EXAMPLES AND EXERCISES

```cpp
void BCSRMatrix::multiply(double* x, double* y) {
    for (i=0; i < n/r; i++) {
        y0 = 0;
        y1 = 0;
        ir = i*r;
        for (j=row_ptr[i]; j<row_ptr[i+1]; j++, val+=4) {
            jc = col_ind[j]*c;
            y0 += val[0] * x[jc];
            y1 += val[2] * x[jc];
            y0 += val[1] * x[jc+1];
            y1 += val[3] * x[jc+1];
        }
        y[ir] = y0;
        y[ir+1] = y1;
    }
    return;
}
```

Note how values from x are reused in close proximity: this allows BCSR to make better use of the processor’s cache than CSR and hence increase its performance. We can increase r and c to further improve the amount of reuse of x, but this will also have the effect of increasing the number of zeros we need to store and thus increasing the amount of wasted work and storage. In this project you will investigate this trade-off.

Also note that because the blocks are fixed in size we do not need a third loop to iterate over the values they contain. Instead, we can explicitly specify all the required. This is known as loop unrolling and can result in further performance improvements.

8.2 Coding, Examples and Exercises

8.2.1 Creating the CSRMatrix class

To begin with you will need to implement a class for storing sparse matrices in CSR format. The class definition is given below, but you will need to provide the function bodies.

```cpp
class CSRMatrix{
    protected:
        long rows;
        long cols;
        long nz;

        double* val;
        long* col_ind;
        long* row_ptr;

    public:
        CSRMatrix(long inrows, long incols, long innz);
        ~CSRMatrix();
        void multiply(double* x, double* y);
};
```
Tasks

1. Implement the constructor. Based on the input parameters, this needs to assign the three
class members `rows`, `cols` and `nz` and dynamically allocate the three data arrays.

2. Implement the destructor. It should free the memory allocated dynamically in the con-
structor.

3. In the constructor, populate the matrix using the non-zero structure in the first example
in Section 8.1. You can pick your own values for the non-zero entries.

4. Implement `CSRMatrix::multiply()` based on the definition in Section 8.1.

5. Write a main program which creates the `CSRMatrix` A from Section 8.1 and an array of
doubles x, and then uses `CSRMatrix::multiply()` to compute \( y = Ax \). Check that it gives
the correct result for your input values.

6. Modify the main program with a loop that takes the output of a CSR matrix–vector mul-
tiplication and uses it as input to the next iteration. Check that it gives the correct result
for your input values.

8.2.2 Reading Large Matrices

In the previous section we considered only small example matrices, but these storage formats are
obviously intended to operate with very large matrices. The University of Florida Sparse Matrix
Collection\(^2\) is a fantastic source of large sparse matrices. For the purposes of this project you
will be evaluating the performance of CSR and BCSR using three specific matrices:

- **atmosmodm**: 1 489 752 × 1 489 752 matrix with 10 319 760 non-zeros.
- **ecology1**: 1 000 000 × 1 000 000 matrix with 4 996 000 non-zeros.
- **nlpkkt80**: 1 062 400 × 1 062 400 matrix with 28 192 672 non-zeros.

Reading the matrices directly from the versions on the UFSMC download site is beyond the
scope of this project, and instead I will provide you with copies of them in an easier to handle
format. The matrix is stored as a text file containing the three arrays `val`, `col.ind` and `row.ptr`
concatenated together. To read in the contents using the `cin` stream you will need to implement
a function based on the following:

```cpp
void CSRMatrix::read(char* filename) {
    ifstream instr(filename, ios::in);
    while (!instr.eof()) {
        instr >> value;
        cout << value << endl;
    }
    instr.close();
    return;
}
```

\(^2\)http://www.cise.ufl.edu/research/sparse/matrices/
8.2. CODING, EXAMPLES AND EXERCISES

Tasks

1. Obtain the archive containing the large matrices from Nick Dingle.

2. Implement a method to read the matrix in from a file and populate a CSRMatrix.

3. Test that you can successfully read in each matrix and multiply it repeatedly with a vector.

8.2.3 Creating the BCSRMatrix class

```cpp
class BCSRMatrix: public CSRMatrix{
    protected:
        // store the number of rows and columns in a block
        long b_rows;
        long b_cols;
    public:
        // give the block size as input parameters
        BCSRMatrix(CSRMatrix* input, long block_r, long block_c);
        ~BCSRMatrix();
        void multiply(double* x, double* y);
};
```

Tasks

1. Modify the CSRMatrix class to provide accessor functions to its private members.

2. Implement the BCSRMatrix constructor. Based on the input parameters, this needs to assign the class members and dynamically allocate the three data arrays. It should then populate the BCSR format based on the contents of the CSRMatrix passed into it.

3. Implement the BCSRMatrix destructor. It should free the memory allocated dynamically in the constructor.

4. Implement BCSRMatrix::multiply() based on the definition in Section 8.1.

5. Modify your main program to create the BCSRMatrix A from Section 8.1 with $2 \times 2$ blocks, and then use BCSRMatrix::multiply() to compute $y = Ax$. Check that it gives the correct result for your input values.

6. Modify the main program with a second loop that takes the output of a BCSR matrix-vector multiplication and uses it as input to the next iteration. Check that it gives the correct result for your input values.

8.2.4 Evaluation

Tasks

1. Modify CSRMatrix with a member function that outputs the amount of memory taken up by the val, col_ind and row_ptr arrays. You will need to use the sizeof() function.
2. Modify your `BCSRMatrix` constructor so that it times how long it takes to convert a matrix into BSCR format.

3. Modify your main program so that it times how long it takes to perform each of the two matrix multiplication loops.

4. Evaluate the relative performance of the two matrix storage schemes for the three matrices described in Section 8.2.2. For BCSR, in each case try $2 \times 2$, $4 \times 4$ and $8 \times 8$ blocks. Measure the time taken to perform the iterations, and additionally in the BCSR cases also measure how long the conversion process takes. How many iterations need to be performed for BCSR to outperform CSR? What effect does changing the block size have?

5. Evaluate the memory consumed by each of the test matrices for the two formats.

6. Summarise your results with graphs.

8.3 Report

Write a project report as a connected piece of prose. You may use any suitable reference sources, but these must be clearly identified in the report at the points of use.

1. Summarise the two matrix storage schemes (CSR and BCSR) and explain the advantages offered by BCSR.

2. Describe how you converted a CSR matrix to a BCSR matrix.

3. Report the run-times and memory usage of the two schemes from your experiments in Section 8.2.4.

4. Compare and contrast your results. Did you observe what you expected? If not, can you explain why not?

Marks will be awarded for clarity and correctness of code as well as answers to the questions and discussion. 60% of the marks for this project will be awarded for the successful implementation of the `CSRMatrix` and `BCSRMatrix` classes, with the remaining 40% for your evaluation in Section 8.2.4 as written-up in your report.

Bibliography