MATH49111/69111 Scientific Computing
Semester 1, 2017

Projects 2

Chris Johnson
chris.johnson@manchester.ac.uk
http://www.maths.manchester.ac.uk/~cjohnson/

List of Projects

Guidelines

Choose only **one** project from those in this booklet to complete and hand in. Students on the MSc in QFFE are encouraged to choose either project 3 or project 4. The projects are written to be self contained, but you may (indeed, are encouraged to) reference relevant books, papers and other resources. You can of course ask me for any clarifications.

Project 2 is worth 50% of the marks for the course, and is be assessed by a written report, marked out of 40. The report will be marked in accordance with the criteria as described below under "Grading criteria".

This is **not** a group work assignment and your report and all code must be written by you individually. Do not read or copy the work of other students, and do not discuss the projects/-coursework or share your own work with others. Do not copy (or closely paraphrase) material from other sources. Please see the university guidelines on plagiarism (`http://documents.manchester.ac.uk/display.aspx?DocID=2870`). The reports and associated code should be submitted online on Blackboard by the deadline below.

- Write your report as though for a fellow student on your course: you will need to explain carefully the topic of your project, but there is no need to describe more basic mathematical concepts.
- Aim for precision and clarity of writing.
- The report should be well structured, containing:
  - an introduction and description of the problem
  - discussion of the problem formulation and numerical techniques used
  - your results, supported using figures and tables as required, and analysis of these results
  - a conclusion
- Ideally, reports should be typeset in LaTeX and submitted in PDF format[1]. A LaTeX coursework and project template is provided on the course website, which you may use if you wish.
- The report should **not exceed** 15 pages in length (excluding long tables, code etc. in appendices). This should be regarded as a maximum: a concisely written project may score very high marks using somewhat fewer pages than this.

[1] though you may use Microsoft Word or other word-processing software

- Any books or articles referred to must be included in a references section.
- Figures and tables should be clearly labelled, and all figures and tables in the report must be referenced in the text.
- Any code used to generate results used in the report must be included in an appendix. I may ask you to provide the source code files in addition to this. Small portions of code may be included in the text, or references to the code in the appendix can be used.
- Any numerical results presented should be accompanied by a reference to the code which generated them (e.g. the name of a source code file included in the appendix). It is very important that any numerical results presented in the report are generated from the numbers produced by your own code.
- You should attempt every task set in the project, but you do **not** need to write up every task for the report. The tasks that should be written up for the report are indicated clearly in the 'Report' section at the end of the project. However, you are welcome to to write up some of the other tasks in the report as well, and you may find this useful to explain what you have done.

Presentation

The report should be well structured with an introduction of the problem being solved, a problem formulation, results, and a conclusions section. It should be clearly written and free from spelling and grammatical errors. Mathematics should be typeset carefully. The figures and tables should be clear, references in the text, and labelled with suitable captions.

Mathematical content

The report should be factually accurate and the mathematical language precise and clear. There should be clear descriptions of the project and the outcome to be achieved, with evidence of understanding the wider context of the problem. Any results and conclusions made need to be well supported and coherent. Numerical results should be interpreted in the context of the mathematical problem, and the accuracy of the computations should be discussed in some depth. For example, numerical results should supported by validation/test cases (to demonstrate that the code has been implemented correctly), by grid or timestep convergence tests (to check that the result is accurate). Check that the results are a credible solution to the original problem! The report should provide evidence of an understanding, and the competent application, of the range of techniques and methods used in the project, and evidence of technical skills.

C++ code

Most importantly, the code must be correct, producing the results that are claimed. The code should be robust, incorporating tests and error checking (e.g. errors from coding mistakes or invalid user input, as well as numerical error due to discretisation and roundoff) where appropriate. The code should be well structured (e.g. using object oriented programming, where suitable or

where specified in the project), and suitably commented. Credit will be given for additional effort put into the code beyond the minimum requirements of the project (for example, where code has been optimised for speed, or has been structured in a novel and suitable way, or where extensive testing has been built in).

Resources

- See the project template file on the course website for an example of LaTeX syntax and for links to web resources for more information about LaTeX.
- For a guide to scientific writing in general, see
  - *Scientists Must Write* (R. Barrass, Routledge, 2002)
  - *The Elements of Style* (W. Strunk & E. B. White, 1999)
  - *What's wrong with these equations?* (N. D. Mermin, in *Physics Today*, October 1989), available at `http://www.pamitc.org/documents/mermin.pdf`
- For technical guidance on plotting figures in MATLAB/Octave and Gnuplot, see the online notes on the course website. For guidance on figure design (much of it beyond the scope of what is required for this course) see, for example, *The Visual Display of Quantitative Information* (E. R. Tufte, Graphics Press, 1983) or *Semiology of Graphics* (J. Bertin, ESRI Press, 2010).

Grading criteria

- 25 marks (50% of the marks for the project) are for obtaining the correct numerical answers, and writing the code that does so. These marks are gained from questions in the report that ask for numerical results to be tabulated or plotted, and from the questions that ask for code to be written. These numerical answers should be incorporated into the written report (not presented as standalone answers to each question).
- The remaining 50% of the marks for each project are for the understanding and analysis demonstrated in the report. These marks assess how you have chosen appropriate algorithms and program architectures in your code, how you have validated your numerical results, and presented these in the context of the mathematical problem. The levels of performance required to gain these marks are described in table 1 below.

Deadlines

- The written report for the project is due in by:
  3pm, Friday 15th December

Grade	0–24%	25–49%	50–74%	75–100%
Introduction	Cursory or no explanation of mathematical problem, no references	Some description of mathematical problem and/or what is done in this report	Descriptions of problem and the work in this report placed within broader context, with appropriate references	Interesting, cogent account of the work in this report, in the context of the mathematical problem considered and scientific computing more widely. Very well referenced.
Validation	No validation, or flawed validation, of numerical results	Simple validation and/or code error checking performed, with some explanation	Careful validation of numerical results and code tests, with explanation of how these tests relate to one another	Convincing and detailed reasoning of the validity of all the numerical results, drawing together error checks in code, convergence testing supported by theory, appreciation of likely sources of error, suitable and novel choice of test cases
Analysis and context	Results presented with no context, and without references.	Some linking of the results to the problem in question, little or no referencing.	Results drawn together and compared in the context of the mathematical problem described in the project, with references were appropriate to methods etc.	Careful analysis of the results and their meaning, in the context of the mathematical problem and more broadly. Methods well referenced, including broader discussion/references than that presented in the project booklet.
Conclusion	No (or mostly irrelevant) conclusion	Conclusion describes result(s) in simple terms	Conclusion describes work done throughout the report and how it links together	Conclusion draws together strands of work throughout the report, demonstrating how the results obtained relate to each other and the broader context of the mathematical problem
Presentation	Writing and equations unclear and/or marred by typographical errors. Figures poorly chosen and unclear and/or ambiguous (e.g. missing or unreadable scales or axis labels)	Writing is broadly clear though, with some areas of ambiguity and/or language errors. Arguments are not very clearly structured. Limited use of figures or figures are not designed well to serve the arguments in the text.	Accurate writing with few spelling or grammar errors. The argument structure is usually clear. Figures are clear and generally well chosen, though may not be ideally designed to illustrate the arguments in the text.	Precise, analytical writing free from spelling and grammatical errors. The structure of the argument is clear and appropriate for the intended audience (a fellow student on the course). Equations carefully chosen and explained, and well typeset. Figures are clear and attractive, with results presented so as to best illustrate the points being made in the text.

Table 1: Grading criteria for the understanding/analysis, worth 50% of the total marks for the project. The other 50% of marks are for the correctness of the numerical answers and C++ programs, as indicated in the specific questions in the project.

Project 1

Compressed Sensing

Author: Dr. Martin Lotz

Modern technology depends increasingly on the acquisition and processing of vast amounts of data. For many applications the data has an underlying simplicity that allows it to be greatly compressed; however, this is normally only possible after the full high-resolution data set has been acquired. This is a highly inefficient process, artificially slowing down many devices (one example being an MRI scanner). A recent breakthrough has been the discovery that compressible data can be efficiently acquired directly in a compressed form, and a high quality approximation of the full data then recovered by optimisation methods.

Compressible data is defined by there being a representation that approximates the data accurately, but has substantially fewer nonzero entries. A vector $x \in \mathbb{R}^n$ is called *k-sparse*, if at most k coordinates are non-zero. The prototype problem in Compressed Sensing consists of finding a k-sparse solution x to an under-determined system of linear equations $Ax = b$, where A is an $m \times n$ measurement matrix and $m < n$ (each row is considered a *measurement* of the signal x).

- If the number of measurements m satisfies $m \geq n$, then x is the solution of a least squares problem and can be found by standard methods. However, when x is sparse the number of measurements does not reflect the sparsity of the solution.
- If x is k-sparse and $m < n$, but $m > 2k$, then for *most* matrices A, the solution for x is uniquely determined – but this may require an exhaustive search over submatrices that is impossible for data sizes of practical interest.

An important question is thus whether it is possible to recover a k-sparse vector *efficiently*, while keeping the number of measurements m proportional to the *information content* k rather than the large *ambient dimension* n. Classical results from Compressed Sensing show that this is indeed the case. Namely, for *most* A we can recover a k-sparse solution efficiently using optimization methods, provided

$$m \geq C \cdot k \log(k/n). \tag{1.1}$$

The goal of this project is to explore a particular algorithm that is able to accomplish sparse recovery under certain conditions. The algorithm is a gradient projection method called *Normalised Iterative Hard Thresholding* (NIHT). By experimenting with the algorithm, you will find out that there is a *phase transition phenomenon* that governs the performance of the algorithm: there is a threshold $m_0 < n$, such that as the number of equations increases past m_0, the probability of NIHT successfully recovering a k-sparse vector from $Ax = b$ changes abruptly from almost 0 to almost 1.

1.1 Iterative Hard Thresholding

Before presenting the sparse recovery algorithm, we begin with a discussion of the classical steepest descent method for solving linear least squares problems. While this algorithm is not the best method for linear systems, it serves as a basis

for the NIHT algorithm presented in the next section.

1.1.1 Gradient descent

Suppose we would like to solve a linear least squares problem, i.e.

$$\text{minimise } \|Ax - b\|^2, \tag{1.2}$$

with $A \in \mathbb{R}^{m\times n}$ and $m \geq n$ (the matrix A is "tall and skinny"). Differentiating with respect to the (vector) x, we find that the solution to this minimisation problem is characterised by the *normal equations*

$$A^\top Ax = A^\top b, \tag{1.3}$$

where $A^\top$ is the transpose of A. This symmetric system of equations can be solved by direct methods, or by iterative methods such as Conjugate Gradient. Perhaps the simplest iterative method is Gradient Descent. Given a function $f(x)$ and a starting point x_0, one tries to minimise the function by iteratively taking steps in the direction of the negative gradient, the *steepest descent*:

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i). \tag{1.4}$$

(Here x_i denotes the vector x at the i^{th} stage of the algorithm, not the i^{th} component of the vector x). The step length α may be taken to minimise the function $\alpha \mapsto f(x_i - \alpha\nabla f(x_i))$. Consider the function $f(x) = \frac{1}{2}\|Ax - b\|^2$. The *residual* at x is defined as $r = A^\top(b - Ax)$. It is not difficult to show (try it!) that the gradient and the minimising step length α satisfy

$$\nabla f(x) = A^\top(Ax - b) = -r, \quad \alpha = \frac{r^\top r}{r^\top A^\top Ar}. \tag{1.5}$$

The gradient descent algorithm then takes the following form. Start with an initial guess x_0 (usually, $x_0 = 0$ should be fine) and $r_0 = A^\top(b - Ax_0)$. Then successively compute

$$\alpha_i = \frac{r_i^\top r_i}{r_i^\top A^\top Ar_i}, \tag{1.6}$$

$$x_{i+1} = x_i + \alpha_i r_i, \tag{1.7}$$

$$r_{i+1} = A^\top(b - Ax_{i+1}) \tag{1.8}$$

$$= r_i - \alpha A^\top Ar_i \tag{1.9}$$

until $\|r_{i+1}\|$ is smaller than some tolerance, or a maximum iteration bound is reached.

Exercises

- Recall/familiarise yourself with the `MVector` class from the first set of projects (a basic implementation of this class can be found on the course website). Implement a `Matrix` class (or use the `MMatrix` class from the first set of projects):

```
class Matrix
{
public:
  // Constructors and destructors (see also vector class)
  explicit Matrix() : N(0),M(0) {}
  Matrix(int n, int m) : N(n), M(m), A(n,vector<double>(m)) {}

  // Include other methods here
protected:
  unsigned N, M;  // Matrix dimensions
```

```
  vector <vector<double> > A;  // Store as a vector of vectors
};
```

Overload the operator * to implement matrix/vector and matrix/matrix multiplication.

- Add a method `transpose()` to the `Matrix` class that returns the transpose of a matrix:

```
Matrix transpose() const
{
  // Code to return the transpose of the matrix
}
```

- Implement the Steepest Descent Algorithm as a function

```
int SDLS(const Matrix& A, const MVector& b, MVector& x,
         int maxIterations, double tol)
{
    // Code here
}
```

The input consists of a matrix A, a vector b, an initial guess x, the maximum number of iterations `maxIterations`, and a tolerance (a suitable value might be 10^{-6}). The solution is returned by reference in x, and the return value of the function should be the number of iterations taken to reach a solution. Choose a suitable return value for the case where a converged solution is not reached within the specified maximum number of iterations.

- Run your implementation of the steepest descent algorithm to solve the 3×2 least-squares problem with

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix} \tag{1.10}$$

Record the trajectory of points x_i in $\mathbb{R}^2$ (how might we modify the signature of the SDLS function to optionally output these values?) and write these data to a file. Plot the trajectory.

- What happens when the entry lower right entry is changed from 0 to -2?
- Can you find ways to optimise the implementation of the algorithm?

1.1.2 Iterative Thresholding

We now turn the matrix around and look at solving for x in the system

$$Ax = b, \tag{1.11}$$

where the matrix $m \times n$ matrix A is 'fat', that is, the number of rows m is smaller than the number of columns n. We will specify that x that is *k-sparse* for some $k < m$, that is, x has at most k non-zero entries. (Without this constraint, the problem is under-determined, since the number of equations m is smaller than the number of unknowns n.)

- In (1.11), A is an $m \times n$ matrix, x is k-sparse (where $2k \leq m$) and all $m \times m$ submatrices of A are invertible. Show that x is uniquely determined.

Let $I \subset [n] = \{0, \ldots, n-1\}$ denote a subset of the column indices, and write A_I and x_I for the columns of a matrix and entries of a vector indexed by I. For example, if

$$x = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \tag{1.12}$$

and $I = \{0, 2\}$, then

$$x_I = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad A_I = \begin{pmatrix} 2 & 0 \\ -1 & -1 \\ 0 & 2 \end{pmatrix}. \tag{1.13}$$

Note that we use the C++ indexing convention, starting with 0 rather than 1.

If we knew exactly which entries of a solution of (1.11) were non-zero (the *support* of x) then, setting I to the indices of these entries, we have

$$A_I x_I = b. \tag{1.14}$$

This equation is over-determined (the $m \times k$ matrix A is 'tall' since $m > k$), and we can find x_I by solving a least squares problem with any method of choice (for example, the gradient descent method). We can then recover the solution x by filling the entries of x indexed by I with x_I, and zero otherwise. However, in general we don't know which entries of x are nonzero, and searching for the set of nonzero entries may involve a very large number of tries (up to n choose k), making this approach impractical.

An alternative method of solving (1.11) is the Normalised Iterative Thresholding Algorithm (NIHT) by Blumensath and Davies (2010). This algorithm resembles the normal Gradient Descent, except that at each step a *thresholding operation* is performed: only the k greatest elements of x (in absolute value) are retained:

$$x_{i+1} = \mathcal{H}_k(x_i + \alpha A^\top (b - A x_i)), \tag{1.15}$$

where the operator $\mathcal{H}_k(v)$, applied to a vector v, returns v but with all except the k largest (in modulus) elements of v to zero. For example, choosing $k = 3$, $\mathcal{H}_3((-9, 6, -7, 8, -5)^T) = (-9, 0, -7, 8, 0)^T$. This algorithm is an example of a *projected gradient method*, as it alternates a gradient step with a projection step (to the set of k-dimensional coordinate arrangements). In this project we study a simplified, but slightly less effective, version of the NIHT algorithm proposed by Blumensath and Davies (2010).

If the algorithm converges, it returns a k-sparse solution. But is it a solution to our problem? The algorithm can't always succeed in finding the right sparse solution.[1] However, it is known that when the number of equations m is big enough, then for *most* matrices A the algorithm solves the problem satisfactorily. By *most* we mean that if we choose a matrix A at random, according to some distribution, the matrix will be such that NIHT finds a sparse solution of $Ax = b$.

Exercises

These exercises work towards an implementation of NIHT.

- Add a method `threshold` to the `MVector` class, which sets all but the largest k elements (by modulus) to zero.

```
void threshold(int k)
{
    // Set all but k largest elements to zero
}
```

  - Find the k-th largest element by using a sorting algorithm. You may either use one of the algorithms from the first set of projects, or `std::sort` from the standard library (in the `<algorithm>` header). Be careful, as the sorting algorithms tend to change the vector under consideration! You can also think about modifying the quicksort algorithm from the first set of projects into a *quickselect* algorithm that finds the k-th largest element without having to sort the whole array (or use `std::nth_element`). Then set all the entries smaller than the k-th largest one to zero.

- Implement the NIHT algorithm. The input should be like the one for Gradient Descent, but in addition contain a sparsity parameter k.

[1] if it could, we would be able to solve an NP-hard problem in polynomial time, one of the big open problems in mathematics and computer science!

```
int NIHT(const Matrix& A, const MVector& b, MVector&x, int k,
        int maxIterations, double tol) {
    // Initialise starting vector
    x = A.transpose()*b;
    x.threshold(k);   // Get s largest values

    // Code here
}
```

Note that in NIHT we must use (1.8) not (1.9) to calculate the residual, since the latter formula assumes that no thresholding has occurred.

The algorithm should return the number of iterations it took to obtain find a solution. The value of `maxIterations` may have to be large here.

- In order to test the algorithm, it is useful to be able to be able to set b and A to a random vector / matrix. Implement a method `initialize_normal` in the `Matrix` class that sets each entry to a random value, normally distributed, with zero mean and variance $1/m$. Write a similar method for the `MVector` class that initialises the vector to a random k-sparse vector with zero mean and unit variance (this method should have an integer parameter k.)

  You may find the following function useful; it generates a normally distributed value with variance 1 and zero mean by applying the Box-Müller transform to the uniformly distributed values produced by the `std::rand` function.

```
#include <cstdlib>
double rand_normal()
{
  static const double pi = 3.141592653589793238;
  double u = 0, v;
  while (u == 0) // loop to ensure u nonzero, for log
  {
    u = rand() / static_cast<double>(RAND_MAX);
  }
  double v = rand() / static_cast<double>(RAND_MAX);
  return sqrt(-2.0*log(u))*cos(2.0*pi*v);
}
```

  Create a random 1000×1 matrix and record the entries in a file. Create a histogram of the data to confirm that the data looks like a normal distribution (you can use a plotting package of your choice, or calculate the histogram yourself in C++).

- To test the algorithm first generate a random k-sparse vector x, a random $m \times n$ matrix A and the vector $b = Ax$ for various values of n, m and k (for example, $n = 100$, $m = 50$, $k = 10$). Test whether the algorithm is able to recover x when given A and b as input. Can you find a combination of parameters for which the algorithm always works? If so, visualise the performance by plotting the residuals.

- Can you find ways to detect signs of non-convergence other than letting the main loop run `maxIterations` times? (For example, if the residuals do not change sufficiently between successive iterations).

1.1.3 The Phase Transition phenomenon

In numerical mathematics, a *phase transition* is a sharp change in the character of a computational problem as its parameters vary. In the context of compressed sensing, an algorithm for recovering a sparse vector x may succeed with high probability when the number of samples exceeds a threshold that depends on the sparsity level; otherwise, it fails

with high probability. Recent work indicates that phase transitions emerge in a variety of random optimisation problems from mathematical signal processing and computational statistics. Figure 1.1 illustrates this phenomenon in the context of compressed sensing, and in particular NIHT. Clearly, if $m = 1$ (one equation), then no signal can be recovered. On the other extreme, if $m = n$ (full system), then recovery is possible in most cases (if A is not too badly conditioned). In between there is an m_0, depending on the sparsity k, at which the probability of successful recovery jumps quite rapidly from 0 to 1.

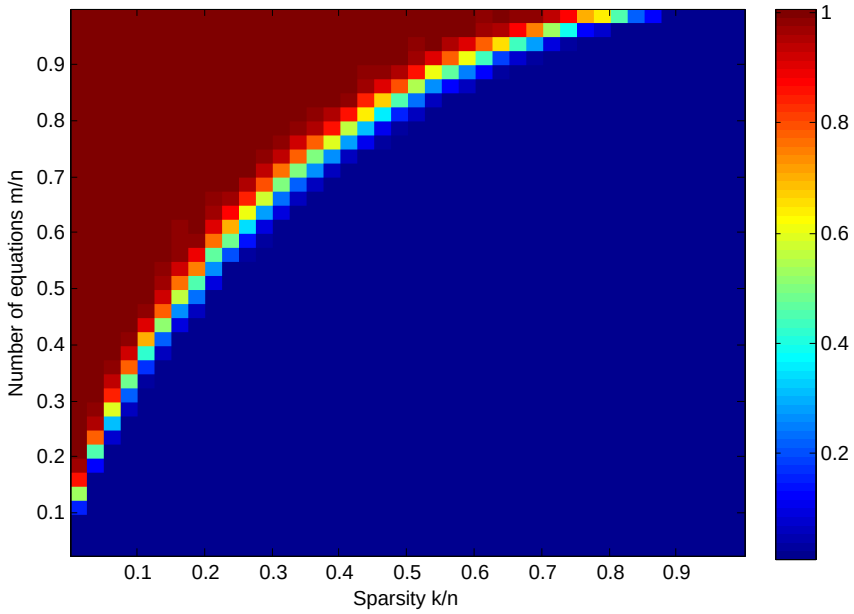

Figure 1.1: For parameters $(k/n, m/n)$ in the red region, sparse recovery succeeds with overwhelming probability, while in the blue region it fails with overwhelming probability.

Exercises

The goal of these exercises is to find the phase transition location for NIHT for a selection of parameters.

- Generate a random vector of length $n = 200$ with at most $k = 10$ non-zero entries.
- Repeat the above for values of m ranging from 4 to 199 in steps of 5. For each such value m, run NIHT T times (where, for example, $T = 50$ or 100) on random data, and record the success ratio $p(m)$ as number of successful recoveries divided by T. Plot $p(m)$ as a function of m and try to determine the range in which p changes from 0 to 1. **Note:** This experiment may take some time, so it can be helpful to output some values to the screen during the calculation to keep track of the progress.
- Repeat the above steps for $k = 20$. How does the successful recovery region change?
- If you feel ambitious, you can try to recreate a figure such as Figure 1.1 (beware of the computational cost involved, this may involve more sophisticated experimental planning!). You may find that the simplified version of NIHT presented in this project is slightly less effective at recovery than the example in figure 1.1.

1.1.4 Additional information

A general introduction to compressed sensing can be found in Foucart and Rauhut (2013). A comprehensive experimental study of NIHT and related algorithms on GPUs (graphics processor units) can be found in Blanchard and Tanner

(2013), while a comprehensive theoretical analysis is found in Cartis and Thompson (2013). Phase transition phenomena for greedy algorithms are not yet fully understood. In contrast, a conclusive explanation phase transitions for *convex optimisation* approaches to finding sparse (and other simple) solutions of under-determined systems has been derived in Amelunxen et al. (2013).

1.2 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 25 marks available for correct numerical results and code. The other 25 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

1. Give a description of the problem of compressed sensing, with possible applications and major achievements. You may consult the literature for this.
2. Describe the Gradient Descent method and show, using examples, how the performance depends on the conditioning of the matrix considered. Produce a plot illustrating the convergence of points to a solution for a 3×2 system. **[10]**
3. Discuss the Normalised Iterative Hard Thresholding algorithm. Give details about the implementation of the thresholding step, and discuss the computational overhead that this step may cause per iteration. **[9]**
4. Describe the phase transition phenomenon for the NIHT algorithm. Illustrate this by determining, for vectors of length $n = 200$ and sparsities $k = 20$ and $k = 50$ (that is, sparsity ratios 0.1 and 0.25), the point m_0 such that for $m > m_0$, NIHT recovers the sparse vector with overwhelming probability. Plot a graph of the empirical recovery probabilities against m and explain the experimental setup. **[6]**

Marks will be awarded for clarity and correctness of code as well as answers to the questions and discussions.

Bibliography

D. Amelunxen, M. Lotz, M. B. McCoy, and J. A. Tropp. Living on the edge: A geometric theory of phase transitions in convex optimization. arXiv:1303.6672v1 [cs.IT], 2013.

J. D. Blanchard and J. Tanner. Performance comparisons of greedy algorithms in compressed sensing, 2013. URL http://www.gaga4cs.org.

T. Blumensath and M. E. Davies. Normalized iterative hard thresholding: Guaranteed stability and performance. *Selected Topics in Signal Processing, IEEE Journal of*, 4(2):298–309, 2010.

C. Cartis and A. Thompson. A new and improved quantitative recovery analysis for iterative hard thresholding algorithms in compressed sensing. *ArXiv e-prints*, September 2013.

S. Foucart and H. Rauhut. *A mathematical introduction to compressive sensing*, volume 336 of *Applied and Numerical Harmonic Analysis*. Birkhäuser, Basel, 2013. ISBN 978-0-8176-4947-0.

Project 2

Discontinuous Galerkin Methods for Convection-dominated problems

Author: Dr. Andrew Hazel

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

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0, \tag{2.1}$$

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.

2.1 Theory

2.1.1 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.

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:

$$x = \frac{1}{2}\left[(x_0^e + x_1^e) + s(x_1^e - x_0^e)\right],$$

[1] In order to aid the translation into C++ we shall start all indices from zero.

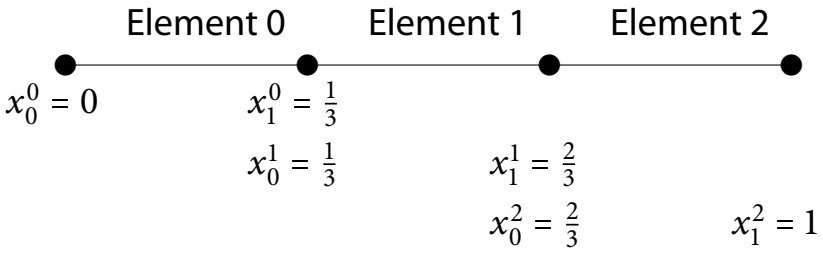

Figure 2.1: The region $x \in [0,1]$ is divided into three (uniform) elements. The left-hand end of element e is located at x_0^e and the right-hand end is at x_1^e. Note that geometric continuity requires that $x_1^{e-1} = x_0^e$.

or, equivalently,

$$x = x_0^e \left[\frac{1}{2}(1-s)\right] + x_1^e \left[\frac{1}{2}(1+s)\right] = x_0^e \psi_0(s) + x_1^e \psi_1(s), \tag{2.2}$$

where

$$\psi_0(s) = \frac{1}{2}(1-s) \quad \text{and} \quad \psi_1(s) = \frac{1}{2}(1+s), \tag{2.3}$$

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, ψ. Thus, in element e

$$u(x,t) = u_0^e \psi_0(s) + u_1^e \psi_1(s), \tag{2.4}$$

where u_i^e is the value of u at the position x_i^e. In other words, we approximate the unknown u using **linear interpolation** within each element.

2.1.2 Formulating the problem

Galerkin methods are based on the so-called "weak form" of the differential equation (2.1), obtained on multiplication by a test function, $v(x)$, and integrating over the problem domain:

$$\int \left[\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x}\right] v(x)\,\mathrm{d}x = 0. \tag{2.5}$$

We note that if u is a solution of the original equation (2.1) then it is also a solution of the "weak form". On the other hand, if we insist that equation (2.5) must be satisfied for **every** (sensible)[2] test function, $v(x)$, then the solution of equation (2.5) will also be a solution of the "strong form" (2.1).

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

$$\int_{x_0^e}^{x_1^e} \left[\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x}\right] v\,\mathrm{d}x = 0. \tag{2.6}$$

We next integrate the second term by parts to obtain

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

$$\Rightarrow \quad \int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} v - f(u)\frac{\partial v}{\partial x}\right)\mathrm{d}x + f(u_1^e)v(x_1^e) - f(u_0^e)v(x_0^e) = 0; \tag{2.7}$$

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.* $u_1^{e-1} = u_0^e$ — the same constraint that we impose on the coordinate, x.

[2] by sensible, of course, we mean satisfying suitable integrability and smoothness constraints

In a **discontinuous** formulation, this constraint is relaxed and the flux is replaced by a numerical flux, $h(u_1^{e-1}, u_0^e)$, 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

$$\int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} v - f(u) \frac{\partial v}{\partial x} \right) \mathrm{d}x + h(u_1^e, u_0^{e+1}) v(x_1^e) - h(u_1^{e-1}, u_0^e) v(x_0^e) = 0. \tag{2.8}$$

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 (2.8) by using two different test functions. In Galerkin methods, the two test functions are precisely the two shape functions (2.3) used to interpolate the unknown. Thus, the two discrete equations that must be solved in each element are:

$$\int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} \psi_0 - f(u) \frac{\partial \psi_0}{\partial x} \right) \mathrm{d}x \quad + \quad h(u_1^e, u_0^{e+1}) \psi_0(x_1^e) - h(u_1^{e-1}, u_0^e) \psi_0(x_0^e) = 0, \tag{2.9a}$$

$$\int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} \psi_1 - f(u) \frac{\partial \psi_1}{\partial x} \right) \mathrm{d}x \quad + \quad h(u_1^e, u_0^{e+1}) \psi_1(x_1^e) - h(u_1^{e-1}, u_0^e) \psi_1(x_0^e) = 0. \tag{2.9b}$$

The definition of the shape functions (2.3) 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

$$\int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} \psi_0 - f(u) \frac{\partial \psi_0}{\partial x} \right) \mathrm{d}x \quad - \quad h(u_1^{e-1}, u_0^e) = 0, \tag{2.10a}$$

$$\int_{x_0^e}^{x_1^e} \left(\frac{\partial u}{\partial t} \psi_1 - f(u) \frac{\partial \psi_1}{\partial x} \right) \mathrm{d}x \quad + \quad h(u_1^e, u_0^{e+1}) = 0. \tag{2.10b}$$

or

$$\int_{x_0^e}^{x_1^e} \frac{\partial u}{\partial t} \psi_0 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_0}{\partial x} \, \mathrm{d}x + h(u_1^{e-1}, u_0^e), \tag{2.11a}$$

$$\int_{x_0^e}^{x_1^e} \frac{\partial u}{\partial t} \psi_1 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_1}{\partial x} \, \mathrm{d}x - h(u_1^e, u_0^{e+1}). \tag{2.11b}$$

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

$$\int_{x_0^e}^{x_1^e} \frac{\partial}{\partial t} \left(u_0^e \psi_0 + u_1^e \psi_1 \right) \psi_0 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_0}{\partial x} \, \mathrm{d}x + h(u_1^{e-1}, u_0^e), \tag{2.12a}$$

$$\int_{x_0^e}^{x_1^e} \frac{\partial}{\partial t} \left(u_0^e \psi_0 + u_1^e \psi_1 \right) \psi_1 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_1}{\partial x} \, \mathrm{d}x - h(u_1^e, u_0^{e+1}). \tag{2.12b}$$

The shape functions are independent of time and hence

$$\int_{x_0^e}^{x_1^e} \dot{u}_0^e \psi_0 \psi_0 + \dot{u}_1^e \psi_1 \psi_0 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_0}{\partial x} \, \mathrm{d}x + h(u_1^{e-1}, u_0^e), \tag{2.13a}$$

$$\int_{x_0^e}^{x_1^e} \dot{u}_0^e \psi_0 \psi_1 + \dot{u}_1^e \psi_1 \psi_1 \, \mathrm{d}x \quad = \quad \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_1}{\partial x} \, \mathrm{d}x - h(u_1^e, u_0^{e+1}), \tag{2.13b}$$

where $\dot{u}_i^e$ denotes the time-derivative of the discrete value u_i^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:

$$M_{ij}^e = \int_{x_0^e}^{x_1^e} \psi_i \psi_j \, \mathrm{d}x \quad \text{and} \quad F^e(U^e)_i = \int_{x_0^e}^{x_1^e} f(u) \frac{\partial \psi_i}{\partial x} \, \mathrm{d}x.$$

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

$$\dot{U}^e = (M^e)^{-1}\left[F^e(U^e) + H(U^{e-1}, U^e, U^{e+1})\right], \tag{2.14}$$

and we can use an explicit time-stepper to solve this coupled (2×2) system of ordinary differential equations. If we use simple first-order finite differences to approximate the time derivative, we obtain the iterative scheme:

$$U^{e(n+1)} = U^{e(n)} + \Delta t(M^e)^{-1}\left[F^e(U^{e(n)}) + H(U^{e-1(n)}, U^{e(n)}, U^{e+1(n)})\right], \tag{2.15}$$

where $U^{e(n)}$ is the solution at the n-th discrete time level and Δt is the fixed time increment. The system (2.15) must be assembled and solved for each element, but the computation of the function H requires information from neighbouring elements.

2.1.3 Description of the algorithm

Initialisation

- Create N elements and for each element e assign spatial coordinates x_0^e and x_1^e and an initial guess for the unknowns u_0^e and u_1^e.
- 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_{ij}^e, and flux vectors, $F^e(U^e)$ and $H(U^{e-1}, U^e, U^{e+1})$.
  - Assemble the system of ODEs (2.14).
  - Perform on timestep of the linear system (2.14) 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.

2.2 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 `AdvectionElement` that contains storage for two positions and two unknowns, all of which should be double precision variables. Your class should include two functions `interpolated_x(s)` and `interpolated_u(s)` that return the values $x(s)$ and $u(s)$ by implementing the equations (2.2) and (2.4) respectively; for example

```
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
    }

    // Return the value of the coordinate at local coordinate s using
    // equation (1.2)
    double interpolated_x(double s) {//FILL THIS IN}

    // Return the value of the unknown at local coordinate s using
    // equation (1.4)
    double interpolated_u(double s) {//FILL THIS IN}

}; //End of the class definition
```

2. Test your class by writing a `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:

$$u = 1.5 + \sin(x) \tag{2.16}$$

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 (2.16) 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?

Timestepping loop

1. Show that the components of the mass matrix are

$$M^e_{ij} = \frac{(x^e_1 - x^e_0)}{2} \int_{-1}^{1} \psi_i \psi_j \, \mathrm{d}s,$$

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

$$M^e = \frac{x^e_1 - x^e_0}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix},$$

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

2. Show that the components of the flux vector are

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

and hence that

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

3. Add a function to the `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 `virtual` so that it can be overloaded later to solve Burgers' equation.

```
// Calculate the flux
virtual double flux(double u)
{
    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

$$\int_{-1}^{1} g(s) \, \mathrm{d}s \approx g(-1/\sqrt{3}) + g(1/\sqrt{3}),$$

for any function $g(s)$.

```
// 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:

$$h(a, b) = \frac{1}{2}(f(a) + f(b)) - \frac{1}{2} \max_{a \le \zeta \le b} |f'(\zeta)|(b - a). \tag{2.17}$$

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

```
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 (2.15).

```
 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 (2.15). 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.

2.3 Report

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

$$\frac{\partial u}{\partial t} + C\frac{\partial u}{\partial x} = 0, \quad \text{(Advection equation)} \tag{2.18a}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, \quad \text{(inviscid Burgers' equation).} \tag{2.18b}$$

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

Write your report as a connected piece of prose that describes the problem and your solution to it. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 25 marks available for correct numerical results and code. The other 25 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

- Produce a graph of the solution of the advection equation (2.18a) with the sine-wave initial condition (2.16) 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? **[12]**
- Produce graphs of the solution of the advection equation (2.18a) using a square-wave initial condition

$$u = \begin{cases} 1 & 0 \le x \le 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

  What do you notice in this case? How do you think that this problem could be resolved? **[6]**
- Solve Burgers' equation (2.18b) 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? **[7]**

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

Project 3

American Options

Original author: Dr. Paul Johnson

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 value an American call option using the Black-Scholes (BS) model (Wilmott et al., 1995). The BS model results in a linear parabolic PDE with nonlinear boundary conditions, which must be solved numerically. After a coordinate transformation the problem becomes a nonlinear PDE, which through the Crank-Nicolson algorithm is converted to a set of nonlinear algebraic equations. This is then solved using the Newton-Raphson iteration. This problem will involve generating an adapted tri-diagonal 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).

The mathematics behind pricing an American option is significantly more involved than the pricing of a European option, and as such this project is recommended only for students with a strong mathematics background.

3.1 Theory

In this project we aim to price an option under the Black-Scholes framework. An option gives the holder the right 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, in which the holder must exercise the option on the expiry date; these can be priced with an analytical formula. The American option allows the holder to exercise at any time before the option expires, resulting in a nonlinear problem for the price.

3.1.1 Black-Scholes Pricing Framework

Under the Black-Scholes framework, a derivative V, depending on the underlying asset S and time to expiry τ, may be valued according to the equation

$$\frac{\partial V}{\partial \tau} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - d) S \frac{\partial V}{\partial S} - rV \tag{3.1}$$

where r is the risk free interest rate, d is the dividend rate and σ the volatility of the underlying asset, subject to the appropriate boundary and initial conditions.

The initial condition for an American call option, applied at the expiry $\tau = 0$ is

$$V(S, \tau = 0) = \max(S - X, 0), \tag{3.2}$$

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 $S = S_f(\tau)$

$$V(S = S_f(\tau), \tau) = S - X \tag{3.3}$$

and

$$\frac{\partial V}{\partial S}(S = S_f(\tau), \tau) = 1. \tag{3.4}$$

We also have a boundary condition for small S given by

$$V(S = 0, \tau) = 0. \tag{3.5}$$

Note that three boundary conditions are provided for V, rather than the two that would usually be expected for a second-order parabolic equation such as (3.1).

The additional boundary condition is required because of the additional unknown in the system, the position $S_f(\tau)$ of the boundary.

The unknown $S_f(\tau)$ has an initial condition,

$$S_f(\tau = 0) = \max\left(\frac{r}{d}X, X\right). \tag{3.6}$$

The derivation of these equations is outside the scope of the project; see Wilmott et al. (1995) for a very readable explanation.

3.1.2 Formulation

Body-fitted coordinates

A difficulty with finding numerical solutions to problems involving early exercise is that the position of the early exercise boundary $S_f(\tau)$ is not known *a priori*. This means that, in numerical methods involving fixed grids (where values are discretised at pre-specified values of (S, τ)), the boundary does not usually lie exactly on a grid point or node. This complicates the process of applying the boundary conditions at this boundary. A common solution to this problem is to change coordinate variables to map the problem into a rectangular domain, which can then be discretised easily. In the present problem, we transform the coordinate S to a coordinate $\hat{S}$, defined by

$$\hat{S} = \frac{S}{S_f(\tau)},$$

so that $\hat{S} = 1$ at the free boundary $S = S_f(\tau)$ and $\hat{S} = 0$ at $S = 0$.

Under this transform, the derivatives are rewritten in terms of the new variable as

$$\left.\frac{\partial V}{\partial \tau}\right|_S = \left.\frac{\partial V}{\partial \tau}\right|_{\hat{S}} + \frac{\partial V}{\partial \hat{S}}\frac{\partial \hat{S}}{\partial \tau} = \left.\frac{\partial V}{\partial \tau}\right|_{\hat{S}} - \frac{\partial V}{\partial \hat{S}}\frac{dS_f}{d\tau}\frac{\hat{S}}{S_f},$$

and

$$\frac{\partial V}{\partial S} = \frac{1}{S_f}\frac{\partial V}{\partial \hat{S}},$$

In the new variables, the Black-Scholes equation becomes

$$\frac{\partial V}{\partial \tau} - \frac{\partial V}{\partial \hat{S}}\frac{dS_f}{d\tau}\frac{\hat{S}}{S_f} = \tfrac{1}{2}\sigma^2\hat{S}^2\frac{\partial^2 V}{\partial \hat{S}^2} + (r - d)\hat{S}\frac{\partial V}{\partial \hat{S}} - rV, \tag{3.7}$$

the boundary conditions become

$$V(\hat{S} = 1, \tau) = S_f \hat{S} - X, \tag{3.8}$$

$$\frac{\partial V}{\partial S}(\hat{S} = 1, \tau) = 1, \tag{3.9}$$

$$V(\hat{S} = 0, \tau) = 0, \tag{3.10}$$

and the initial conditions are

$$S_f = \max(rX/d, X), \tag{3.11}$$

$$V(\hat{S}, 0) = \max(\hat{S} S_f(0) - X, 0). \tag{3.12}$$

We note that the transform to the coordinate $\hat{S}$, while simplifying the boundary condition at $S = S_f(\tau)$, results in the problem becoming nonlinear, due to the term involving S_f in (3.7).

Numerical discretisation

We use a finite-difference method to discretise the equation, evaluating V and $\hat{S}$ at discrete values of time $\tau = i\,\Delta\tau$ (for i a non-negative integer) and discrete values of $\hat{S} = j\,\Delta\hat{S}$, where $\Delta\hat{S} = 1/N$, and j and N are integers with $0 \le j \le N$. We use the notation

$$V_{i,j} := V(i\Delta\hat{S}, j\Delta\tau) \tag{3.13}$$

$$S_{f\,j} := S_f(j\Delta\tau) \tag{3.14}$$

to denote the values of V and at these discrete grid points (i, j) and S_f at the timesteps j.

The initial conditions provide values for $V_{i,0}$ and S_{f0}. To integrate the system numerically in τ, we must specify a rule to obtain the values at the next time step, $V_{i,j+1}$ and $S_{f\,j+1}$, from the known values at the current time step, $V_{i,j}$ and $S_{f\,j}$.

Integration of the equations: the Crank-Nicolson method

We introduce this update rule with a generic second-order parabolic differential equation,

$$\frac{\partial u}{\partial \tau} = F\left(u, \hat{S}, \tau, \frac{\partial u}{\partial \hat{S}}, \frac{\partial^2 u}{\partial \hat{S}^2}\right), \tag{3.15}$$

where F is an arbitrary function and u is the variable to be solved for. The most straightforward discretisation of the time derivative of this equation, in terms of the discretised u, is

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta\tau} = F_{i,j}, \tag{3.16}$$

where $F_{i,j}$ represents the value of function F with τ, $\hat{S}$, u and the finite-difference discretised derivatives of u evaluated at $\hat{S} = i\Delta\hat{S}$, $\tau = i\Delta\tau$. The derivatives with respect to $\hat{S}$ in (3.15) are discretised using centred finite-difference formulae, namely

$$\left(\frac{\partial u}{\partial \hat{S}}\right)_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta\hat{S}}, \tag{3.17}$$

$$\left(\frac{\partial^2 u}{\partial \hat{S}^2}\right)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta\hat{S})^2}. \tag{3.18}$$

Importantly, these discretisations mean that $F_{i,j}$ depends not just on $u_{i,j}$ but on the adjacent values $u_{i-1,j}$ and $u_{i+1,j}$ as well. The discretisation (3.16) is known as the forward Euler method, and has the benefit that it is trivial to rearrange into a form where the values at the next time step $u_{i,j+1}$ are given explicitly in terms of the (known) values at the current time step $u_{i,j}$ and $F_{i,j}$,

$$u_{i,j+1} = u_{i,j} + \Delta\tau F_{i,j}. \tag{3.19}$$

However, a serious disadvantage of the Euler method for parabolic equations is that it is unstable for all except very small values of the timestep $\Delta\tau$ (see Press et al., 2007).

Instead we will use the Crank-Nicolson scheme, in which F is evaluated not only from values of u at the current timestep j, but also from values at the next timestep $j+1$.

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta\tau} = \frac{1}{2}\left(F_{i,j} + F_{i,j+1}\right). \tag{3.20}$$

This Crank-Nicolson formulation has the advantage of stability for arbitrarily large timestep (though of course, this is no guarantee that our solution will be *accurate* using such large timesteps!) The cost of this additional stability is that the Crank-Nicolson scheme is *implicit*: due to the term in $F_{i,j+1}$, the values $u_{i,j+1}$ cannot be obtained by a simple rearrangement of the integration scheme. Rearranging the terms known at the current timestep to the right hand side, we instead find

$$u_{i,j+1} - \frac{\Delta\tau}{2}F_{i,j+1} = u_{i,j} + \frac{\Delta\tau}{2}F_{i,j}. \tag{3.21}$$

The values at the next timestep $u_{i,j+1}$ must be obtained by a numerical method, either by solution of a linear system (if F is linear in u) or by iterative solution to a set of nonlinear algebraic equations (if F is nonlinear in u).

Our free-boundary Black-Scholes problem (3.7) is not quite of the form of the typical parabolic PDE (3.15) , since the nonlinear term,

$$\frac{\partial V}{\partial \hat{S}}\frac{dS_f}{d\tau}\frac{\hat{S}}{S_f} \tag{3.22}$$

involves the time derivatives. When discretising this term we follow the overall principle of the Crank-Nicolson method, namely discretising time derivatives as in the left hand side of (3.20), and evaluating the value of our unknowns and their other (i.e. non-temporal) derivatives as the average of the values at steps j and $j+1$.

Since our equations are nonlinear, we will use a Newton-Raphson iteration to solve them at each timestep.

The Newton method

Given a nonlinear algebraic equation

$$\mathbf{R}(\mathbf{u}) = \mathbf{0}, \tag{3.23}$$

for some vector function $\mathbf{R}$, the Newton(-Raphson) method is an iterative scheme for finding $\mathbf{u}$, consisting of an initial guess $\mathbf{u}^{(0)}$ and the iteration

$$J(\mathbf{u}^{(k)})(\mathbf{u}^{(k+1)} - \mathbf{u}^{(k)}) = -\mathbf{R}(\mathbf{u}^{(k)}) \qquad (k \geq 0), \tag{3.24}$$

where J is the Jacobian matrix of $\mathbf{R}$, with components

$$J_{mn} = \frac{\partial R_m}{\partial x_n}. \tag{3.25}$$

Defining $\delta\mathbf{u}^{(\mathbf{k+1})} := \mathbf{u}^{(k+1)} - \mathbf{u}^{(k)}$,

$$J\,\delta\mathbf{u}^{(\mathbf{k+1})} = -\mathbf{R}, \tag{3.26}$$

where the terms J and $\mathbf{R}$ are evaluated at $\mathbf{u}^{(n)}$. For an initial guess $\mathbf{u}^{(0)}$ sufficiently close to a solution $\mathbf{u}$ of (3.23), it can be shown that successive iterations $\mathbf{u}^{(k)}$ converge rapidly (in fact, quadratically) to $\mathbf{u}$ (Press et al., 2007).

3.1.3 Matrix equation

We now look at how this applies to the Black-Scholes problem at hand, (3.7)–(3.12).

In the discussion of the Crank-Nicolson method we had only one independent variable, u, whereas in the Black-Scholes problem (3.7)–(3.12) we have two independent variables, V and S_f. We accomodate these two variables within the framework of the Crank-Nicolson method by augmenting the spatially-discretised values of V with the value of S_f at timestep j. Our vector of unknowns $\mathbf{u}$ is therefore the discretised form of V augmented with the value of S_f, and contains $N+2$ elements, namely

$$\mathbf{u} = \left(V_0, V_1, V_2, \ldots, V_{N-1}, V_N, S_f\right)^T. \tag{3.27}$$

When evaluating J, note that with this augmented vector of unknowns, $F_{i,j+1}$ depends not just on $V_{i,j+1}$, $V_{i-1,j+1}$ and

$V_{i+1,j+1}$, but also on S_f. We can therefore write (3.26) as

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & . & . & . & . & d_0 \\ a_1 & b_1 & c_1 & 0 & . & . & . & . & d_1 \\ 0 & a_2 & b_2 & c_2 & . & . & . & . & d_2 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & a_i & b_i & c_i & . & d_i \\ . & . & . & . & . & . & . & . & . \\ 0 & . & . & . & . & a_{N-1} & b_{N-1} & c_{N-1} & d_{N-1} \\ 0 & . & . & . & . & . & a_N & b_N & d_N \\ 0 & . & . & . & . & 1 & -4 & 3 & d_{N+1} \end{pmatrix} \begin{pmatrix} \delta V_0 \\ \delta V_1 \\ \delta V_2 \\ . \\ . \\ . \\ \delta V_{N-1} \\ \delta V_N \\ \delta S_f \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ . \\ . \\ . \\ . \\ . \\ e_{N+1} \end{pmatrix}, \tag{3.28}$$

where all but the last row of this equation are of the form

$$a_i\,\delta V_{i-1} + b_i\,\delta V_i + c_i\,\delta V_{i+1} + d_i\,\delta S_f = e_i. \tag{3.29}$$

Boundary conditions

The first and last few rows of (3.28) are altered by the boundary conditions of our problem. We have one boundary condition at $\hat{S} = 0$, (3.8), and two at $\hat{S} = 1$, (3.9) and (3.10). For these, the values of a, b, c, d and e are known and are as follows. At $\hat{S} = 0$, we have

$$V_{0,j+1}^{(k+1)} = V_{0,j}^{(k)} + \delta V_0^{(k+1)} = 0, \tag{3.30}$$

and so $b_0 = 1$, $c_0 = 0$, $d_0 = 0$ and $e_0 = -V_{0,j}^{(k)}$. At $\hat{S} = 1$,

$$V_{N,j+1}^{(k+1)} = S_{f\,j+1}^{(k+1)} - X, \tag{3.31}$$

or

$$\delta V_N^{(k+1)} - \delta S_f^{(k+1)} = S_{f\,j+1}^{(k)} - V_{N,j+1}^{(k)} - X. \tag{3.32}$$

Using a backward finite-difference approximation,

$$\left(\frac{\partial u}{\partial \hat{S}}\right)_{N,j+1} = \frac{\frac{3}{2}u_{N,j+1} - 2u_{N-1,j+1} + \frac{1}{2}u_{N-2,j+1}}{\Delta\hat{S}}, \tag{3.33}$$

we can generate an equation for the derivative condition (3.9),

$$\frac{\frac{3}{2}\left(V_{N,j+1}^{(k)} + \delta V_N^{(k+1)}\right) - 2\left(V_{N-1,j+1}^{(k)} + +\delta V_{N-1}^{(k+1)}\right) + \frac{1}{2}\left(V_{N-2,j+1}^{(k)} + \delta V_{N-2}^{(k+1)}\right)}{\Delta\hat{S}\left(S_{f,j+1}^{(k)} + \delta S_f\right)} = 1 \tag{3.34}$$

which rearranges to give

$$3\delta V_N^{(k+1)} - 4\delta V_{N-1}^{(k+1)} + \delta V_{N-2}^{(k+1)} - 2\left(\Delta\hat{S}\right)\delta S_f^{(k+1)} = 2\left(\Delta\hat{S}\right)S_{f,j}^{(k)} - \left(3V_{N,j}^{(k)} - 4V_{N-1,j}^{(k)} + V_{N-2,j}^{(m)}\right). \tag{3.35}$$

The matrix J has only a few entries below the diagonal, namely the a_i terms and the three entries $1, -4, 3$ in the last row. Row operations can be used to remove these terms, resulting in an upper-triangular matrix that can be solved by back-substitution. For most rows in the matrix (containing the terms a_i, b_i, c_i, d_i), the procedure is to subtract from row i of the matrix the quantity a_i/b'_{i-1} times row $i-1$ of the matrix. Denoting entries in the new matrix with primes, in the first row of the matrix we have $b'_0 = b_0$, $c'_0 = c_0$ and $d'_0 = d_0$. For $1 \le i \le N$ we have

$$a'_i = a_i - b'_{i-1}\frac{a_i}{b'_{i-1}} = 0, \tag{3.36}$$

$$b'_i = b_i - c'_{i-1}\frac{a_i}{b'_{i-1}}, \tag{3.37}$$

$$c'_i = c_i, \tag{3.38}$$

and

$$d'_i = d_i - d'_{i-1}\frac{a_i}{b'_{i-1}}. \tag{3.39}$$

The terms e_i on the right hand side are transformed to

$$e'_i = e_i - e'_{i-1}\frac{a_i}{b'_{i-1}}, \tag{3.40}$$

where for the $i = 1$ row we use $e'_0 = e_0 = -V^{(k)}_{0,j}$. The final row of the matrix requires subtraction of appropriate multiples of the previous three rows, to remove the terms 1, −4, 3, with corresponding changes to d_{N+1} and e_{N+1}.

Having performed these row operations, (3.28) is now in the form

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & . & . & . & . & d_0 \\ 0 & b'_1 & c'_1 & 0 & . & . & . & . & d'_1 \\ 0 & 0 & b'_2 & c'_2 & . & . & . & . & d'_2 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & b'_i & c'_i & 0 & . & d'_i \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & 0 & b'_{N-1} & c'_{N-1} & d'_{N-1} \\ . & . & . & . & . & . & 0 & b'_N & d'_N \\ 0 & . & . & . & . & . & 0 & 0 & d'_{N+1} \end{pmatrix} \begin{pmatrix} \delta V_0 \\ \delta V_1 \\ \delta V_2 \\ . \\ . \\ . \\ \delta V_{N-1} \\ \delta V_N \\ \delta S_f \end{pmatrix} = \begin{pmatrix} -V^{(m)}_{0,j} \\ e'_1 \\ e'_2 \\ . \\ . \\ . \\ . \\ . \\ e'_{N+1} \end{pmatrix} \tag{3.41}$$

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:

$$\delta S_f = \frac{e'_{N+1}}{d'_{N+1}}. \tag{3.42}$$

and the value of the δV_N using

$$\delta V_N = \frac{e'_N - d'_N \delta S_f}{b'_N}. \tag{3.43}$$

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

$$\delta V_i = \frac{e'_i - d'_i \delta S_f - c'_i \delta V_{i+1}}{b'_i} \tag{3.44}$$

This solution for $\delta\mathbf{u}^{(k+1)}$ allows us to calculate $\mathbf{u}^{(k+1)}$, and proceed with the next Newton step, step $k+2$. To detect whether the solution has converged we can either evaluate the residuals $\mathbf{R}(\mathbf{u})$ and check that these are sufficiently close to zero. Alternatively, we can terminate when δV_i and δS_f are all less than some threshold. A typical tolerance in either case is 10^{-8}.

When the Newton method has converged, we are in possession of our solution ($V_{i,j+1}, S_{f\,j+1}$ at timestep $j+1$, and can proceed to the next Crank-Nicolson step to evaluate the solution at time step $j+2$.

3.2 Exercises

1. Write a function `GaussianSolver` to implement the Gaussian elimination on the vectors a, b, c, d, and e (all of length $N+2$) that store the nonzero components of the linear system (3.28). Your function should output the solution vector ($\delta V_i, \delta S_f$), either through a parameter passed by reference or by returning it.
2. Check your algorithm with a test matrix, and compare the solution with that obtained another source (e.g. MATLAB).
3. Now evaluate the components a_i, b_i etc. that occur in (3.28). To do this you should first calculate the residuals function $\mathbf{R}$ in (3.26), by discretising the equations according to the Crank-Nicolson method. Then evaluate the Jacobian matrix, J. Most of the terms in the equations are linear (with the exception of the nonlinear term (3.22)) and thus their derivatives are correspondingly straightforward.

4. Set up variables for 'old' and 'new' values of the option value, `v_old` and `v_new`, and free boundary `S_f_new` and `S_f_old`, as well as the vector of values of $\hat{S}$ at each grid point i. Then implement the initial conditions of the option value and free boundary at $\tau = 0$, with $X = 1$, $r = 0.05$, and $d = 0.025$. Display the values to screen or output them to file, and check that they are as you expect. How might you transform the results back into the original variable S?
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_old` and $V_{i,1}$ = is `v_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_new. The algorithm is given by
   - Loop until converged:
     - set up the linear algebra problem to find δV_i and δS_f, setting appropriate values for a, b etc.
     - solve the linear algebra problem using your `GaussianSolver` function to find the correction
     - 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 convergence condition, and exit loop if converged.

   Check that your solution seems viable (e.g. check that boundary conditions are met, the option value is positive).
6. Now iterate the scheme through $jmax$ timesteps, and put the entire algorithm into a function (perhaps a class member 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. Modify your solver function to accept the `Option` class as an argument.

3.3 Convertible Bond

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

$$V(S, \tau = 0) = \max(S, B), \quad \text{and} \quad S_f(\tau = 0) = B, \tag{3.45}$$

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

$$V(S = 0, \tau) = Be^{-r\tau}, \tag{3.46}$$

$$V(S = S_f, \tau) = S, \tag{3.47}$$

$$\left.\frac{\partial V}{\partial S}\right|_{S=S_f} = 1. \tag{3.48}$$

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.

3.4 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 25 marks available for correct numerical results and code. The other 25 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

- 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 **[3]**

  - Produce a table to show the convergence of the option value as the number of steps in $\hat{S}$ and τ increase. (You might also wish to investigate how the computation time varies as $\Delta\hat{S}$ and $\Delta\tau$ vary.) **[3]**
  - Plots of the free boundary $S_f(\tau)$ for different values of r and d **[3]**
  - What happens when $d = 0$? **[2]**

- In part 2, investigate the American call option with $d > r$.
  - Explain why the solution does not always converge when $d > r$? **[2]**
  - You will need to alter the timesteps in this case so that the following scaling is implemented. Simply divide time into $jmax$ steps as usual, however when using $\Delta\tau$ in the numerical calculation instead use

$$\Delta\tau = \frac{(2j-1)\,T}{jmax^2}$$

    [2]
  - Plot the value of the option and the payoff for some arbitrary values **[2]**
  - Produce a table to show the convergence **[2]** of the option value to the 'exact' solution (and computation times) as the number of steps in $\hat{S}$ and τ increase **[2]**
  - Plot the free boundary $S_f(\tau)$ for different values of r and d **[2]**

- In the final part, investigate the convertible bond by plotting the free boundary $S_f(\tau)$ for different values of r and d. **[2]**

Bibliography

P. Johnson. *Improved Numerical Techniques for Occupation-Time Derivatives and Other Complex Financial Instruments*. PhD thesis, The University of Manchester, 2008. URL http://eprints.ma.man.ac.uk/1357/.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3rd edition, 2007. ISBN 978-0-521-88068-8.

G. D. Smith. *Numerical Solution of Partial Differential Equations*. Oxford University Press, 1985.

P. Wilmott, S. Howison, and J. Dewynne. *The Mathematics of Financial Derivatives*. Cambridge University Press, 1995.

Project 4

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 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 use an object orientated approach to test different revenue management systems on a randomly generated set of customers. By averaging the results of your model across several realisations (each with different random bookings) we can obtain the expected value of various measures of the car park, such as revenue and occupancy. With these measures of car park performance we can evaluate new strategies for revenue management of the car park.

This project is relatively simple mathematically but involves more interpretation of results and pricing strategies. As such, it is most suitable for students with a background in finance or economics.

4.1 Theory – The Carpark Model

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

$$B^i = \begin{pmatrix} t_b \\ t_a \\ t_d \end{pmatrix} \tag{4.1}$$

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 time steps. Take the interval in time $t \in [a, b]$ and split it into K equally sized steps of length Δt so that

$$t^k = a + k\Delta t. \quad \text{for} \quad k = 0, 1, \ldots, K \tag{4.2}$$

In this project we set the basic unit of time to be a day, and $\Delta t = 1$ represents a time period of one day. Furthermore we

set the start time $a = 0$, and so $t^k = k$. Then let C^k denote the number of cars present in the car park at any time during the day k, where $t \in [t^k, t^{k+1})$. We may write

$$C^k = \sum_i f(B^i) \quad \text{where} \quad f(B) = \begin{cases} 1 & \text{if} \quad t^k \le t_a < t^{k+1} \\ 1 & \text{if} \quad t^k < t_d \le t^{k+1} \\ 1 & \text{if} \quad t_a < t^k \quad \text{and} \quad t_d > t^{k+1} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

This function $f(B)$ simply describes the fact that a car is regarded as being in a car park for a full day if it either arrives during that day, leaves during that day, or arrives on a previous day and leaves on a later day. We can also calculate the duration of stay for a booking as the number of days for which a single car is present in the car park

$$D^i = \sum_k f(B^i) \tag{4.4}$$

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

Now let the price per day 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 kth day is

$$R^k = \sum_i f(B^i) p(D^i) \tag{4.5}$$

and a typical pricing function (in GBP per day) may look like

$$p(D^i) = \beta + \frac{\alpha}{D^i} \tag{4.6}$$

where D^i is the (integer) duration in days and α and β are positive constants. The cost to customer i for their entire booking is therefore

$$D^i p(D^i) = \beta D^i + \alpha, \tag{4.7}$$

which can be interpreted constant fee of α per booking, plus parking at a rate of β per day. We can calculate the total revenue for the car park as

$$V^k = \sum_k \sum_i f(B^i) p(D^i) \tag{4.8}$$

4.1.1 Generating events from a Poisson Process

To generate the set of bookings we suppose that bookings are generated by a Poisson process, i.e. they are generated independently and continuously at some average rate λ_b bookings per day. For such a process, the time between subsequent bookings is given by an exponential distribution. If u^n is an independent random draw from a uniform distribution $(0 < u^n \le 1)$ then the function to generate the time of the next event t^n of a Poisson process with intensity λ_b, given t^{n-1} the time of the last event is:

$$t^n = t^{n-1} - \frac{1}{\lambda_b} \log(u^n). \tag{4.9}$$

4.1.2 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 λ, 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 previous booking was made t_b^i we may generate the next booking as:

$$B^{i+1} = \begin{pmatrix} t_b^{i+1} = t_b^i - (1/\lambda_b)\log(u^n) \\ t_a^{i+1} = t_b^{i+1} - (1/\lambda_a)\log(u^{n+1}) \\ t_d^{i+1} = t_a^{i+1} - (1/\lambda_d)\log(u^{n+2}) \end{pmatrix} \tag{4.10}$$

(assume that the first booking occurs at $t_b^0 = 0$). The sequence of booking times t_b is generated by a Poisson process with

rate λ_b. For each booking, the time between booking and arrival, and the time between arrival and departure, are each exponentially distributed with mean $1/\lambda_a$ and $1/\lambda_b$ respectively. Note that this booking model is not very realistic, 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.

4.1.3 Calculated expected values of car park measures

We can use the random bookings that we generate as input to our car park model in section 4.1, and in doing so evaluate various measures of our car park, such as the revenue, and occupancy (number of cars in the car park) for a given day. These values will depend on the parameters intrinsic to the car park model (for example, the size of the car park, and the pricing function), but will also depend on the particular booking times chosen by our random booking process. To study the effect of car park parameters (size, pricing function etc.) on our measurements (of revenue, occupancy, etc.) we would like these measurements to be independent of the randomness in the booking times. To do this by taking the *expected value*, or average, of our measurement over all the possible random bookings. For very simple measurements it may be possible to calculate this expected value algebraically, for most measurements we will need to calculate these expected values numerically, using a so-called Monte Carlo technique.

The idea behind Monte Carlo methods is very straightforward. Suppose we wish to measure the expected value of R^k, the revenue of the car park on the kth day. We simply run our entire car park model some number of times N, generating different revenues $R^k_{(1)}, R^k_{(2)}, R^k_{(3)}, \ldots, R^k_{(N)}$ each time we run the model. The expected value of the revenue is approximated by the mean of these samples,

$$E[R^k] = \frac{1}{N}\sum_{n=1}^{N} R^k_{(n)}. \tag{4.11}$$

When N is large this expected value $E[R^k]$ is approximately normally distributed (from the central limit theorem), and an estimate for its standard deviation is

$$s = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}\left(R^k_{(n)} - E[R^k]\right)^2}. \tag{4.12}$$

This standard deviation decreases with increasing N, so by choosing sufficiently large N our expected value becomes relatively insensitive to the particular random booking times.

Note that we cannot take N arbitrarily large – it may take too long to simulate the car park model a very large number of times – so the choice of N is a compromise between program run time and accuracy.

4.1.4 Optimal Revenue Management

The explanation of Revenue Management in this project will be brief but we will lay out the basics relevant to our simplified model. 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 car park model, we assume that there exists a set of low price customers that take advantage of the discount for staying in the car park for a long period. (By discount we mean that, while the total price of a long stay is greater than that of a short stay (4.7), the price *per day* of a long stay is less than that of a short stay (4.6).) 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 parking 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 (4.6) with positive constants α and β.

At first we will decide whether to reject bookings by simply assigning a constant proportion of the car park to each set of customers. By varying this proportion we can then investigate when the maximum revenue is generated, and thus find the optimal proportion to assign to each set of customers.

4.2 Exercises

4.2.1 The Customers class

- Create a function that returns the time of the next event for a Poisson process with intensity λ. You may use the internal C++ random number generator `rand()` to generate samples from a uniform distribution. Alternatively, look at the Mersenne Twister code provided for the Gillespie Algorithm project in this booklet. Make sure that you enforce the constraint $u^n > 0$ by excluding the case where `rand()` generates a value of `0`.
- Enter the following data structure for bookings into your code:

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

(Recall that a `struct` is simply a class with all members public by default.) Complete the implementation for the "less than" operator `<` so that bookings may be ordered by `bookingTime`. You may also wish to overload stream outputs using `operator<<`.

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

```
class Customers
{
private:
    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 `generateBookings` method. The first three arguments should be the intensity parameters for the Poisson process. Using `startTime` as your initial time, keep generating bookings until you reach `finishTime`. See section 4.1.2 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 $\lambda_d = 1/7$ with $t_b \in [0, 150]$ and write them to screen. These will be the low-price leisure customers. (Note that although we restrict $t_b \leq 150$, we may have bookings where $t_a > 150$ and/or $t_d > 150$).
- Generate bookings using $\lambda_b = 25$, $\lambda_a = 1$, and $\lambda_d = 2$ with $t_b \in [0, 150]$ and compare them to the previous booking set. These will be the high-price business customers.
- Write a member function of `Customers` to add bookings from another `Customers` class. The function signature should look like:

```
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` function, in the `<algorithm>` include library. For a `std::vector vectorBookings` we write

```
std::sort(vectorBookings.begin(),vectorBookings.end());
```

which sorts the vector by booking time, so long as the less than operator has been overloaded correctly. Display your bookings to screen, and check that the sets have combined correctly.

4.2.2 The Car Park Class

- By writing a class, or otherwise, create storage for the number of cars present in the car park in each period k and also the revenue they generate (you could combine both using your own data structure).
- Now create a function `CarsPresent` that takes as input:
  - a list of bookings stored in a `Customers` class
  - the start time
  - the number of days

  and gives as output
  - a vector containing the number of cars present in the car park for each day.

  You will need to use equation (4.3) to calculate the cars present in the car park for each day from the booking times. (Is there a faster way to calculate D^i than simply summing over f as in (4.4)?)
- What is the expected number of cars in the car park at $t = 150$ using either of the sets of customers? Choose what you think is a sensible number of simulations N when calculating the expected value, and give your answer to an appropriate number of digits.
- Write in a price function of the form shown in equation (4.6) with $\alpha = 10$ and $\beta = 5$ and use it to calculate the revenues generated at each day, and also the total revenue.
- Now modify your car park class to include a capacity (number of spaces in the car park). Write a function that takes a booking and checks if space is available in the car park for the entire duration of that booking. Use this in your `CarsPresent` function to reject bookings if there is no space.
- Using a combined set of both customer types, what is the expected time that a car park with 50 spaces will first reach capacity? As before, think about how many simulations N you should run when calculating the expectation.

4.2.3 Revenue management with a fixed allocation strategy

- For the combined set of customers, calculate the total expected revenue for car parks with capacity $10, 20, 30, \ldots, 100$.
- Now create two different car parks for each set of customers so that the sum of the capacity of each car park is equal to capacity of the combined car park. Investigate what is the optimal proportion of the car park to reserve for each set of customers. For example, you could create a plot of the optimal proportion as a function of total car park size. Optionally, can you estimate the error in your calculation of the optimal proportion?
- You may wish to extend this idea by generating a contour plot of the revenue as a function of both the car park size and the proportion allocated to business/leisure customers.
- How does the optimal proportion of the car park to allocate vary with the booking rates λ_b for each type of customer?

4.2.4 Booking rejection strategies

So far we have used a very simple strategy for rejecting bookings, by allocating a fixed proportion of the car park to each type of customer. Alternatively, we could generate a rejection rule based on the number of spaces left in the car park, 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 car park 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 (4.6) 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 car park had 100 spaces left, then the car park is unlikely to fill, the value of the spaces displaced is low (zero if there is no chance that the car park will fill up) and we should accept the booking. How we should calculate the notional value of the space that is displaced is left for you to discover.

Investigate booking rejection strategies of this type. The most interesting cases are likely to be found for car parks that are large enough that they will not usually be filled by the highest-paying customers alone, but small enough that some customers cannot be accommodated. Can you design a method that improves on the optimal revenue generated when a fixed proportion of the car park is allocated to each type of customer?

4.3 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 25 marks available for correct numerical results and code. The other 25 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

- Detail any classes that you have used in the problem. **[3]**
- Show some test cases to demonstrate that your car park model is working as expected. (For example, you could generate a few bookings manually and check that the number of cars in the car park for each day is as you expect, both for unlimited capacity and when the car park is small enough that some bookings are rejected.) **[3]**
- For the unlimited capacity case, analyse the expected number of cars in the car park, and also the expected revenue. **[4]**
- After including capacity to the problem, analyse the expected time that a car park with 50 spaces will first reach capacity. **[5]**
- Analyse of the optimal proportion of the car park that should be reserved for the business customers, as described in §4.2.3 **[5]**
- Describe your investigation of new booking strategies in §4.2.4, including numerical results from the strategies you chose. (If you did not have time to complete this part of the project, discuss what you think might be good strategies). Do your new rejection strategies generate higher revenue than that when an (optimal) fixed proportion of the car park is allocated to each type of customer? **[5]**

Bibliography

A. Papayiannis, P. Johnson, D. Yumashev, S. Howell, N. Proudlove, and P. Duck. Continuous-time revenue management in carparks. *working paper*, 2012. URL http://eprints.ma.man.ac.uk/1806/01/covered/MIMS_ep2012_42.pdf.

Project 5

Random Numbers, Stochastic Simulation and the Gillespie Algorithm

Author: Mark Muldoon

The chemistry I learned in school involves reactions among huge numbers of molecules, so huge that chemists have a special unit, the *mole*[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 base class. Such a class provides an interface 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 `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 `IntegerRing`, `Gaussian-IntegerRing` and `PolynomialRing`, each with its own version of the ring operations.

5.1 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

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

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 (5.3) 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.

5.1.1 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.

$$S + E \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C \xrightarrow{k_2} P + E$$

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 one expects that, eventually, 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:

$$\begin{aligned}
\frac{de}{dt} &= -k_1 es + (k_{-1} + k_2)c \\
\frac{ds}{dt} &= -k_1 es + k_{-1} c \\
\frac{dc}{dt} &= k_1 es - (k_{-1} + k_2)c \\
\frac{dp}{dt} &= k_2 c
\end{aligned} \tag{5.1}$$

Note that there are two sorts of terms on the right hand sides of the equations above, linear ones such as the $k_2 C$ 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 (5.1) are used very widely and with great success.

One can simplify (5.1) 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

$$e_0 \equiv e(0) + c(0) = e(t) + c(t) \qquad \text{so} \qquad e(t) = e_0 - c(t).$$

These observations mean that its sufficient to study the simpler system

$$\begin{aligned}
\frac{ds}{dt} &= -k_1 e_0 s + (k_1 s + k_{-1})c \\
\frac{dc}{dt} &= k_1 e_0 s - (k_1 s + k_{-1} + k_2)c
\end{aligned} \tag{5.2}$$

The standard account of (5.2) investigates the initial value problem in which, at the outset, there is very 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

State Num.	N_S	N_E	N_C	N_P
1	3	2	0	0
2	2	1	1	0
3	1	0	2	0
4	2	2	0	1
5	1	1	1	1
6	0	0	2	1
7	1	2	0	2
8	0	1	1	2
9	0	2	0	3

Table 5.1: The states of the Michaelis-Menten system compatible with $\boldsymbol{n}(0) = (3, 2, 0, 0)$. The numbering scheme here is used in Figure 5.1 as well.

called the *quasi-steady state hypothesis*) and so the dynamics are well described by

$$c(t) = \frac{e_0 s(t)}{s(t) + K_m} \tag{5.3}$$

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

$$s(t) + K_m \ln\left(\frac{s(t)}{s_0}\right) = s_0 + k_2 e_0 t. \tag{5.4}$$

The solution (5.4) 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

$$\tau = \frac{1}{k_1(s_0 + K_m)} \tag{5.5}$$

during which complex is formed rapidly and so one cannot assume $dc/dt = 0$.

5.1.2 Markov formulation

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 (5.2) and even (5.1) are poor approximations. 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

$$\boldsymbol{n}(t) = (N_S(t), N_E(t), N_C(t), N_P(t))$$

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 5.1 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 5.1 illustrates the nine states from the 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 Δt, the probability that, say, a complex will form is proportional to some rate $r(\boldsymbol{n}(t))$:

$$P(\text{reaction while } t_0 \le t \le t_0 + \Delta t) \approx r(\boldsymbol{n}(t_0)) \times \Delta t. \tag{5.6}$$

We'll defer the problem of determining the rates $r(\boldsymbol{n}(t))$ for a moment and concentrate on what (5.6) 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 Δt is

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

where the sum and product range over all the reactions that are possible when the system is in the state $\boldsymbol{n}(t)$. In the limit $\Delta t \to 0$ the expression above implies that transitions away from a state with populations $\boldsymbol{n}_0$ occur at a rate

$$R(\boldsymbol{n}_0) = \sum_j r_j(\boldsymbol{n}_0) \tag{5.7}$$

or, equivalently, the gaps between such transitions are exponentially distributed with mean $\tau = 1/R(\boldsymbol{n}_0)$. Further, given that such a transition has occurred, the probability that it went via some particular reaction is

$$P(\text{reaction } k \,|\, \text{transition away from } \boldsymbol{n}_0) = r_k(\boldsymbol{n}_0)/R(\boldsymbol{n}_0) = \frac{r_k(\boldsymbol{n}_0)}{\sum_j r_j(\boldsymbol{n}_0)}. \tag{5.8}$$

Equations (5.7) and (5.8) are the heart of the Gillespie algorithm, which I'll explain in detail in section 5.1.3 below. But before we can apply these equations we need to work out the relationship between the rates $r_j(\boldsymbol{n}_0)$ 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 $\boldsymbol{n}(t)$, the dynamics implied by (5.7) and (5.8) reduce to (5.1). 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

$$r(\boldsymbol{n}) = \alpha_{SE} N_S N_E. \tag{5.9}$$

Now consider what this implies about the large-$\boldsymbol{n}$ limit. The reaction whose transition rate is (5.9) produces one molecule of the complex and so this implies

$$\begin{aligned}
\frac{dc}{dt} &= \lim_{\Delta t \to 0} \frac{\Delta c}{\Delta t} \\
&= \frac{1}{\Delta t}\left(\frac{\Delta N_C}{\mathcal{L}v}\right) \\
&= \frac{1}{\Delta t}\frac{(\alpha_{SE} N_S N_E)\Delta t}{\mathcal{L}v} \\
&= \alpha_{SE}\mathcal{L}v\left(\frac{N_S}{\mathcal{L}v}\right)\left(\frac{N_E}{\mathcal{L}v}\right) \\
&= (\alpha_{SE}\mathcal{L}v)\, ce
\end{aligned} \tag{5.10}$$

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 ΔN_C with $(\alpha_{SE} N_S N_E)\Delta t$ which, given rate formula (5.9), is the expected number of complex formations: it tends to zero with Δt.

Comparison of the last line above with (5.1) 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 5.2.

Reaction	δN_S	δN_E	δN_C	δN_P	Rate
Complex formation	-1	-1	1	0	$k_1(N_S N_E)/(\mathcal{L}v)$
Complex dissociation	1	1	-1	0	$k_{-1}N_C$
Product formation	0	1	-1	1	$k_2 N_C$

Table 5.2: 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} and k_2 are as in (5.1) while $\mathcal{L}$ is Avogadro's number and v is the volume of the system.

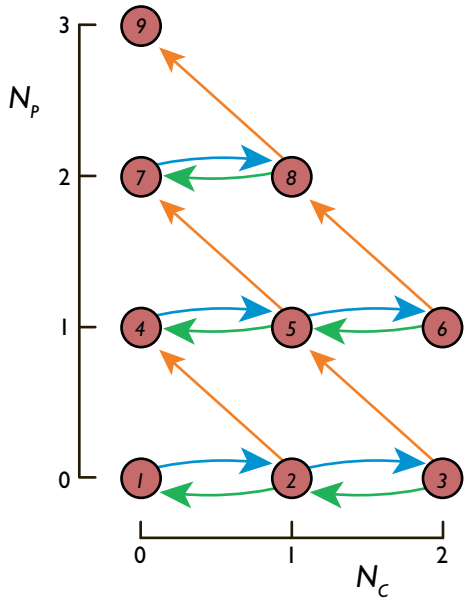

Figure 5.1: The diagram above shows the states and transitions for a Gillespie model of the Michaelis-Menten system. The states—which are numbered as in Table 5.1—are specified by pairs (N_C, N_P). Transitions corresponding to complex formation are shown as curved, blue arrows while those arising from dissociation of the complex are curved and green. The straight, orange arrows represent state transitions due to formation of the product.

5.1.3 The Gillespie algorithm

Once we have the information in Table 5.2 the Gillespie algorithm is easy to describe. Assume the various chemical species have initial populations $\boldsymbol{n}_0$. We'll need keep track of the time t, a vector of populations $\boldsymbol{n}$ and a vector of rates $\boldsymbol{r} = (r_1, r_2, \dots)$. Also associated with the j-th reaction is a vector $\delta\boldsymbol{n}_j$ that gives the corresponding changes in the populations.

(1) *Initialization:*

$t \leftarrow 0$

$\boldsymbol{n}(0) \leftarrow \boldsymbol{n}_0$

(2) *Compute reaction rates:*

$\boldsymbol{r} \leftarrow \boldsymbol{r}(\boldsymbol{n})$

(3) *Choose* δt: Generate a random interval δt drawn from the exponential distribution with mean $\tau = 1/R(\boldsymbol{n})$ where $R(\boldsymbol{n})$ is as in (5.7).

(4) *Update the time:*

$t \leftarrow t + \delta t$

(5) *Choose a reaction:* Use the probabilities (5.8) to decide at random which reaction occurs.

(6) *Update the populations:* Say we chose reaction k in step (5)

$\boldsymbol{n} \leftarrow \boldsymbol{n} + \delta\boldsymbol{n}_k$

(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 5.2. 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 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

$$X + X \xrightarrow{k} D \tag{5.11}$$

then the corresponding rate in a Gillespie simulation is

$$r = N_X(N_X - 1) \times \left(\frac{k}{\mathcal{L}v}\right). \tag{5.12}$$

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

5.1.4 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, 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* (Knuth, 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 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

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

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

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

Exponentially distributed numbers

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

$$\begin{aligned} P(a \le t \le b) &= \int_a^b f(t)\,dt \\ &= \int_a^b Re^{-Rt}\,dt \\ &= e^{-Ra} - e^{-Rb} \end{aligned} \tag{5.13}$$

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

$$F(T) = P(t \le T) = \int_0^T Re^{-Rt}\,dt = 1 - e^{-RT}.$$

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

$$t = F^{-1}(u) = \frac{-\ln(1-u)}{R} = -\tau \ln(1-u) \tag{5.14}$$

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 $\lambda = Cv$,

$$P(k \text{ molecules in volume } v) = \frac{\lambda^k}{k!} e^{-\lambda}, \tag{5.15}$$

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 5.1.

```
double PoissonValue(double lambda)
{
    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 5.1: Knuth's algorithm for the Poisson distribution

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 $0 \le j \le N$ 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 5.1.4. Begin by defining the discrete analogue of the cumulative density function

$$s_k = \sum_{j=0}^{k} p_j$$

for $0 \le k \le N$ 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

$$n = \min \{k \,|\, 0 \le k \le N \text{ and } s_k \ge u\}. \tag{5.16}$$

5.2 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 of 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.

Note that the code for this project involves multiple source code (`.cpp`) and header(`.h`) files. To compile these you must include all the `.cpp` files in you Visual Studio / XCode project, or when using g++ at the command line add all the `.cpp` files (but not the `.h` files) to your compilation command, e.g.

```
g++ main.cpp MersenneTwister.cpp GillespieSimulation.cpp -o gillespie.exe
```

5.2.1 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 5.2 contains enough information to update populations, but doesn't know how to compute rates: that's devolved to derived classes.

Listing 5.3 provides an example of a derived class for the dimerization reaction (5.11), and an associated dissociation reaction. There is a small subtlety in the implementation of the rate formula (5.12) because one will get nonsense if one computes `(n_monomer - 1)` when `n_monomer == 0` and `n_monomer` is a `unsigned` type. These dimerization and dimer dissociation reactions are not part of the Michaelis-Menten system that we study later, but the classs provide an examples of how reactions can expressed by deriving from the `Reaction` class.

```
//**********************************************************
//  Starting from the virtual base class for reactions,
// derive classes for the two reactions involved in
// reversible dimerization.
//
//  mrm:    Whalley Range 30 September 2009
//
#ifndef _DimerizationReactions_ // There can be only one
#define _DimerizationReactions_
//_________________________________________
#include <vector>
#include <cassert>

#include "Reaction.h"
//____________________________________________________________

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, unsigned x_idx, unsigned d_idx)
    {
        rate_const = 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;
```

```
//*********************************************************
//  This header defines one of a circle of classes
// designed to support Gillespie simulations. To actually
// build a simulation one should have a look at
// GillespieSimulation.h as well.
//
//  mrm:    Whalley Range 29 September 2009
//
#ifndef _Reaction_ // There can be only one
#define _Reaction_
//_____________________________________________
#include <vector> // for std::vector<>
#include <cassert> // for assert()

// 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
{
    // All reactions have these
public:
    double rate_const; // from the ODE
    std::vector<unsigned> rateDependsOn;
    std::vector<unsigned> 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 do_reaction(std::vector<unsigned> &n_particles) const
    {
        for (unsigned j = 0; j < speciesAffected.size(); ++j) {
            unsigned crnt_species = speciesAffected[j];
            n_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<unsigned> &pop, double volume) const = 0;
};


#endif // _Reaction_
```

Listing 5.2: The virtual base class for reactions.

```
    }

    // Compute the rate
    double rate(const std::vector<unsigned> &pop, double volume) const
    {
        unsigned monomer_idx = rateDependsOn[0];
        unsigned n_monomer = pop[monomer_idx];

        // Take care lest (n_monomer == 0), so
        // (n_monomer - 1) would lead to underflow.
        double rate = 0.0;
        assert(volume > 0.0);
        if (n_monomer > 0) {
            rate = (rate_const / (volume * Avogadro_const));
            rate *= n_monomer * (n_monomer - 1);
        }

        return rate;
    }
};

class DimerDissociation : public Reaction // D -> X + X
{
public:
    // The creator requires the indices (in the population vector)
    // of the species X and D.
    DimerDissociation(double k, unsigned x_idx, unsigned d_idx)
    {
        rate_const = k; // Install the rate constant

        // Note which species the rate depends on
        rateDependsOn.resize(1);
        rateDependsOn[0] = d_idx;

        // Note which species the reaction affects
        speciesAffected.resize(2);
        speciesAffected[0] = d_idx;
        speciesAffected[1] = x_idx;

        // Note the consequences of the reaction
        deltaN.resize(2);
        deltaN[0] = -1;
        deltaN[1] = 2;
    }

    // Compute the rate
```

```
    double rate(const std::vector<unsigned> &pop, double volume) const
    {
        unsigned dimer_idx = rateDependsOn[0];
        unsigned n_dimer = pop[dimer_idx];

        return rate_const * n_dimer;
    }
};

#endif // _DimerizationReactions_
```

Listing 5.3: A derived class for dimerizations

5.2.2 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 5.4 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 the course website.

```
//************************************************************
//  This header describes an object used to do Gillespie
// simulations. To actually set up a simulation one needs
// to study ReactionKinetics.h as well.
//
//  mrm:    Whalley Range 29-30 September 2009
//
#ifndef _GillespieSimulation_ // There can be only one
#define _GillespieSimulation_
// Standard headers
#include <vector>

// My headers
#include "PseudoRandomGenerator.h"
#include "Reaction.h"

class GillespieSimulation
{
public:
    PseudoRandomGenerator prng; // Random number machinery

    // // Constructors
    GillespieSimulation()
    {
        volume = 0;
    }

    GillespieSimulation(
```

```
        double vol,
        const std::vector<Reaction*> &rx,
        const std::vector<unsigned> &pop);

    // Accessors
    void set_volume(double vol);
    double get_volume() const
    {
        return volume;
    }

    void set_populations(const std::vector<unsigned> &pop);
    void get_populations(std::vector<unsigned> &pop) const
    {
        pop = n_particles;
    }

    // The dynamics
    void advance_time(double time_step);
    bool is_stopped() const
    {
        // Check whether the sum of all rates is zero.
        if (rate_sum.empty()) return true;
        return (rate_sum.back() == 0.0);
    }

private:
    // Privade member data, set via public accessor methods

    double volume; // in litres
    std::vector<unsigned> n_particles; // populations
    std::vector<Reaction*> reaction; // list of reactions

    std::vector<double> rate; // the rates
    std::vector<double> rate_sum; // used like a CDF

    // Private functions for the dynamics
    double time_to_next_event();
    unsigned choose_next_reaction();
    void perform_reaction(unsigned rx_num);

    // A utility to manage the rates and their sums
    void update_rates_n_sums();
};

#endif // _GillespieSimulation_
```

Listing 5.4: A class of objects to do Gillespie simulation

5.2.3 Remarks

- Much of the code found in these pages is available from the course website.
- My code makes frequent use of `assert()`. The idea is to lard your code with expressions like

```
# include <cassert>

// etc ...

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

for example,

```
assert( volume != 0.0 ) ;
rate *= 1.0 / volume ;
```

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>` or the course website FAQ to learn how.

5.3 Exercises

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

1. Prove that the transformation (5.14) works. That is, show that if u is distributed uniformly over $[0,1)$ and t and u are related by (5.14), then t is distributed according to (5.13).
2. Prove that Knuth's algorithm in section 5.1.4 really does draw random samples from a Poisson distribution, i.e. prove that
$$p_k = P(\text{algorithm generates } k) = \frac{\lambda^k}{k!} e^{-\lambda}. \tag{5.17}$$
You may find it useful to write down the conditions that must apply to the uniformly distributed numbers u_j that go into the algorithm when the algorithm stops after exactly k times through the `while` loop. These are in the form of products, which can be converted to sums by taking logs of both sides. Then apply the answer of the previous exercise to find the probability distribution of the logarithm of each of the uniform random variables. To combine the probability distributions each of the transformed u_j, you may find the following lemma helpful: it's easy to prove by induction on n.

   Lemma 5.1. *The volume of the set $\Delta^n \in \mathbb{R}^n$ defined by*
$$\Delta^n = \{(t_1, \ldots, t_n) \in \mathbb{R}^n \mid \sum_j t_j \leq \lambda \text{ and } t_j \geq 0 \text{ for all } j\}$$
   is $\lambda^n / n!$.
3. Explain briefly why the algorithm sketched in section 5.1.4 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 5.5 for an outline of the class declaration for `PseudoRandomGenerator`.

```
//*********************************************************
//  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"

// Our derived class for fancier distributions
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
#endif // _PseudoRandomGenerator_
```

Listing 5.5: A skeleton for a random number generator class.

[2] You'll need both `MersenneTwister.h` and `MersenneTwister.cpp` from the course website.

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

$$p_k = \begin{cases} p^k (1-p)^{N-k} \frac{N!}{k!\,(N-k)!} & \text{If } 0 \le k \le N \\ 0 & \text{otherwise} \end{cases}$$

   You can think of p_k as, say, then 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*

```
// 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 ; }
```

   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()`. Write a version of `discrete()` that uses linear search as above, then read the documentation for `lower_bound()` (e.g. at http://en.cppreference.com/w/cpp/algorithm/lower_bound)and write another implemetation that uses this.

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

$$\texttt{rate\_sum[j]} = \sum_{k=0}^{j} \texttt{rate[k]}.$$

   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

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

   where

$$\texttt{R} = \sum_{j} \texttt{rate[j]}$$

   is the sum of all the rates—it's also the last entry in the `rate_sum` vector. Then choose reaction k, where

$$k = \min \{ j \,|\, \texttt{rate\_sum[j]} \ge \texttt{rx\_u} \}.$$

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 in `main.cpp` and `Dimerization.h` 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. (5.11) and Listing 5.3 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.

k	0	1	2	3	4	5
p_k	0.0094	0.0940	0.3076	0.3916	0.1780	0.0194

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).

e_0	Initial enzyme concentration	`2.0e-7`	moles per liter
s_0	Initial substrate concentration	`5.0e-7`	moles per liter
c_0	Initial enzyme complex	`0.0`	moles per liter
p_0	Initial enzyme product	`0.0`	moles per liter
k_1	Rate const. for complex formation	`1.0e6`	$\frac{\text{litres}}{\text{moles} \times \text{secs}}$
k_{-1}	Rate const. for complex dissociation	`1.0e-4`	sec^{-1}
k_2	Rate const. for product formation	`0.1`	sec^{-1}
v	System's volume	`1.0e-15`	liters

   To implement the Michaelis-Menten system in C++, you will need to create reaction classes (derived from the base class `Reaction`) for the complex formation, complex dissociation and product formation. You can use the `Dimerization` and `DimerDissociation` classes provided as a basis for these new reaction classes. As with the `Dimerization` and `DimerDissociation` classes, you will need to write a constructor and an implementation of the `rate` member function to set up these reactions.

   To use your new reaction classes you will need to set up a vector of pointers to reactions and a vector of initial conditions, to pass to a new `GillespieSimulation` class. (You can can base your code for this on the example provided in `main.cpp` for the dimerization reaction).

   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.

5.4 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 25 marks available for correct numerical results and code. The other 25 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

- Brief answers to Exercise 1–3. **[7]**
- Output from the test program or programs that you wrote to do Exercise 5. **[5]**
- A pair of plots to summarise 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 $N_E(t)/N_E(0)$. **[6]**
- 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. **[3]**
- All your code—replete with comments and sensible variable names—in an appendix. **[4]**

Bibliography

Daniel T. Gillespie. A general method for numericaly simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22:403–434, 1976. ISSN 0021-9991.

Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977. doi: 10.1021/j100540a008.

Desmond J. Higham. Modeling and simulating chemical reactions. *SIAM Review*, 50(2):347–368, JUN 2008. doi: 10.1137/060666457.

Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997. ISBN 0-201-89684-2.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/272991.272995.

Leonor Michaelis and Maud Leonora Menten. Die kinetic der invertinwirkung. *Biochemische Zeitschrift*, 49:333–369, 1913. ISSN 0366-0753.

James D. Murray. *Mathematical Biology I: An Introduction*. Springer Verlag, New York, third edition, 2003. ISBN 0-387-95223-3.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3rd edition, 2007. ISBN 978-0-521-88068-8.

Darren James Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC mathematical and computational biology series. Chapman & Hall/CRC, Boca Raton, 2006. ISBN 1-58488-540-8.