Patrick E. Farrell
University of Oxford

2023–2024
January 29, 2024

Computational Mathematics Projects

Contents

General advice on computational projects

In Hilary term you will undertake two of the three computational projects listed below. Each project is worth up to twenty marks, which are split between mathematical content, programming skill, and clarity of exposition. The marks will count towards the Preliminary examinations; they carry the weight of one third of a paper. The deadlines for the projects are

- 12:00 (noon), Monday, week 6: online submission of first project;
- 12:00 (noon), Monday, week 9: online submission of second project.

The projects are to be submitted online via Inspera. The projects can be done in any order (e.g. one may submit project C in week 6 and project A in week 9). It is recommended that students familiarise themselves with Inspera and all data necessary for submission (e.g. single sign-on and examination candidate number) well in advance of the deadlines. The deadlines are strict and penalties for late submission apply.

Your submissions should consist of one or more `.py` files and a `.html` file produced from the main `.py` file via `publish`, as described in chapter 5. The examiners will primarily scrutinise the published `.html` file, but may modify and execute your code. The files for your submission should be gathered into exactly one `.zip` or `.tar.gz` file for upload.

A key difference from the problem sheets is that the project reports should be more expository. The mathematics behind and intent of the code written for the project must be made as clear as possible, since marks are awarded for mathematical understanding.

When you complete your projects, you must not upload them on the internet e.g. on web forums or in public code repositories. This is to assist in the prevention of plagiarism.

All projects must be your own unaided work. You will be asked to make a declaration to this effect when you submit them. The University's plagiarism policy applies in full, with potential penalties for plagiarism ranging from deduction of marks to expulsion from the University.

Frequently asked questions

Do I need to do any background reading for the projects?

No. References to the literature are included for any students who may be interested in learning more, but are not required to solve the problems.

Can we use TeX for our documentation?

It is not expected at this stage that students know TeX/LaTeX. However, if you are familiar with it, you are encouraged to use TeX notation for writing equations. TeX notation written in comments is rendered appropriately by `publish.py`. TikZ diagrams are not supported; any diagrams required by the projects should be rendered with matplotlib.

Are we allowed to use Jupyter Notebooks?

Again, it is not expected that students are familiar with Jupyter Notebooks. It is possible to use Jupyter Notebooks for your code development if you prefer, but your code should ultimately be submitted as a `.py` file. You can convert Jupyter Notebooks to plain Python with `jupyter nbconvert`.

How important is code optimisation?

The code should not be egregiously inefficient (e.g. having drastically worse scaling in time or memory than a straightforward implementation). Beyond that, do not invest too much effort into it; code clarity is more important than running as fast as possible. As long as the code runs in reasonable time, it suffices. The reference solutions for the projects each take no more than ten minutes or so to execute on modest hardware.

Should we determine the complexity of our algorithms in time and/or memory?

You do not need to do this unless explicitly requested.

How should I structure my code?

A good general structure for your code is to first write out the solution idea in comments, then give the (commented) implementation, then show the code is correct with examples. In many cases the question will indicate what examples to run your code on.

A *Primality testing*

(*This project relates to material in the Trinity term Prelims course* M1: Groups and Group Actions*, and in the Part A option* ASO: Number Theory.)

Computing whether a natural number is prime, and identifying its factors, are core tasks in number theory and in cryptography. As Carl Friedrich Gauss wrote in article 329 of his magnum opus *Disquisitiones Arithmeticae* (1801) (translated by Arthur A. Clarke, 1965[1]):

> The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length. Nevertheless we must confess that all methods that have been proposed thus far are either restricted to very special cases or are so laborious and prolix that even for numbers that do not exceed the limits of tables constructed by estimable men, i.e. for numbers that do not require ingenious methods, they try the patience of even the practiced calculator. ... Further, the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated.

[1] C. F. Gauss. *Disquisitiones Arithmeticae*. Yale University Press, 1965. Translation by A. A. Clarke

The two tasks Gauss mentions—primality *testing* and number *factorisation*—are rather different. An algorithm is known for primality testing (due to Agrawal, Kayal, and Saxena[2]) that deterministically gives the correct answer with a runtime that is a polynomial function of the number of digits of the input. By contrast, the existence or nonexistence of a polynomial-time algorithm for factorising a number remains a major open problem in mathematics, and indeed most of the cryptography in practical use today relies centrally on its computational difficulty.

[2] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004

In this project you will investigate algorithms for testing whether a number is prime or not. For more details on this subject, see the book of Crandall and Pomerance[3].

[3] R. Crandall and C. Pomerance. *Prime Numbers: a Computational Perspective*. Springer-Verlag New York, second edition, 2010

A.1 Trial division

Trial division, the algorithm we met in Code block 7.11 and Exercise 7.6, was first described in Fibonacci's *Liber Abaci* (1202)[4]. In Chapter 5, Fibonacci writes

> If it is even, then he recognises its composition. However if odd, then it will be composite or prime ...Odd numbers truly are composed of odds alone. Whence the components of them by odds are investigated, for which we take the beginning. Therefore when in the figure of first place of any odd number there is the number 5, one will know 5 to be a factor. However, if another odd figure will appear in the first place, then one indeed takes the residue of number when divided by 9; if a zephir [i.e. zero] results, then 9 is a factor, and if 3 or 6 is the residue, then 3 is a factor; however if the residue will show none of these, one divides by 7; and if there will be an excess, then one again divides the number by 11; and if there is an excess, then he divides again by 13, and always he goes on dividing in order by prime numbers until he will find a prime number by which he can divide, and thence he will come to the square root; if he will be able to divide by none of them, then one will judge the number to be prime.

[4] L. Pisano (Fibonacci). *Liber Abaci*. Springer Science & Business Media, 2003. Translation by L. E. Sigler

We will employ some convenient notation for this question. Define $a : b$ as

$$a : b := [a, b] \cap \mathbb{Z}. \tag{A.1.1}$$

We denote numbers known to be prime by p, and the number whose primality we wish to determine by n.

Question A.1. Modify your code for Exercise 7.6 (which implements an efficient variant of trial division) to return `(flag, ndivisions)`, where `flag = True` if the input is prime and `False` otherwise, and `ndivisions` is the count of the number of divisions performed. Print the output of the function applied to all $n \in 2 : 20$. How many divisions are performed to test the primality of 9999991111111?

Question A.2. Compute the number of divisions performed for all numbers $n \in 2 : 10^5$. By means of a plot, verify that trial division takes about $\sqrt{n}/3$ divisions in the worst case to test a number n for primality.

Expressing $\sqrt{n}/3$ in terms of the logarithm of n, we see that the work involved in trial division is exponential in the number of digits. This exponential dependence on the number of digits motivates the search for more efficient algorithms.

A.2 The Fermat test

The trial division algorithm is based on the definition of primality, i.e. that n has no factors other than one and n. Ideally, we would base a test for primality on alternative condition that is equivalent to primality, i.e. that is necessary and sufficient for primality. Finding such conditions is tricky. Instead, we will base our next algorithm on a condition that is merely *necessary* for primality—all prime numbers satisfy the condition, but some composite numbers may also. The condition is inspired by Fermat's Little Theorem, which you will meet in Hilary Term *Groups and Group Actions*.

Theorem A.2.1 (Fermat, 1640)**.** *Let $p \in \mathbb{N}$ be prime. Let $a \in \mathbb{N}$ such that $\gcd(a, p) = 1$ (i.e. a is not a multiple of p). Then*

$$a^{p-1} \equiv 1 \mod p. \tag{A.2.1}$$

This inspires the following procedure. Let n be the number we wish to test for primality.

1. Choose $a \in 2 : (n-2)$.
2. Calculate the greatest common divisor $\gcd(a, n)$;[5] if the greatest common divisor is not 1, then n is composite.
3. Calculate $a^{n-1} \mod n$; if it is not congruent to 1, then n is composite.
4. If it is congruent to 1, then the test is inconclusive.

[5] The greatest common divisor can be efficiently computed using Euclid's algorithm, as in code block 4.9.

We refer to conducting this procedure for a single a as a *Fermat trial*; a *Fermat test* is to do this for a set of candidate values of a.

Of course, since a Fermat test relies on a condition that is only *necessary* for primality, it can only prove that n is *not prime*—in other words, this is actually a compositeness test. Nevertheless, if n passes enough trials, it might give us confidence that n is probably prime.

Question A.3. Write a function to implement the Fermat trial for given n and a.

Write another function to apply the Fermat test with all a in a given list; if no list is supplied, use as default value all $a \in 2 : (n-2)$ in ascending order. This latter function should return a tuple `(flag, ntrials)` where `flag = False` if the Fermat test has shown n to not be prime and `True` otherwise[6], and where `ntrials` is the number of Fermat trials performed. Print the output of the function applied to the natural numbers $n \in 2 : 20$, using in each case all $a \in 2 : (n-2)$ in ascending order.

[6] In other words, a number with flag `True` might still be composite.

[*Hint: the greatest common divisor can be computed using* `math.gcd`.]

[*Hint: in Python, the* `pow` *function takes an optional third argument.* `pow(x, y, z)` *calculates* $x^y \mod z$.]

Question A.4. Compute the first 5 odd numbers n where the Fermat test proves compositeness with just one trial, i.e. with $a = 2$.

Question A.5. For how many odd $n \in 3 : 10,000$ does the Fermat test prove compositeness with at most five trials (using $a \in 2 : \min(6, n-2)$)? What proportion of odd composite numbers in $3 : 10,000$ does this represent?

Question A.6. A *Carmichael number*, also called an absolute Fermat pseudoprime, is a composite number which passes the Fermat trial for any $a \in 2 : (n-1)$ with $\gcd(a, n) = 1$. Compute the Carmichael numbers up to $10,000$.

[*Hint: the first Carmichael number is* 561.]

In 1994, Alford, Granville & Pomerance proved that there are infinitely many Carmichael numbers[7]; for large enough n, there are at least $n^{2/7}$ Carmichael numbers in $1 : n$. This fact limits the utility of the standalone Fermat test; for the Fermat test to work on a Carmichael number, only those bases that share a factor with n will detect its compositeness, and choosing a few a's will likely not help us. However, in the words of Carl Pomerance, ''using the Fermat congruence is so simple that it seems a shame to give up on it just because there are a few counterexamples'' [8]. It is often used in combined algorithms to quickly test for compositeness with a handful of choices of a before subjecting n to more complicated algorithms.

[7] W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703, 1994

[8] F. Bornemann. PRIMES is in P: a breakthrough for ''Everyman''. *Notices of the American Mathematical Society*, 50(5):545–553, 2003

A.3 *Miller–Rabin primality test*

The Miller–Rabin test is a refinement of the Fermat test. Like the Fermat test, we will computationally determine whether a specific property that must hold for primes holds for the n in question. A deterministic version was introduced by Miller in 1976[9], with its correctness dependent on the (unproven) extended Riemann hypothesis; Rabin introduced a probabilistic version in 1980[10].

[9] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976

[10] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980

To motivate the Miller–Rabin trial, suppose $p > 4$ is prime. Let $a \in 2 : (p-2)$ with $\gcd(a, p) = 1$. From Fermat's Little Theorem, we know that

$$a^{p-1} - 1 \equiv 0 \mod p. \tag{A.3.1}$$

Since $p - 1$ is even, the left-hand side is the difference of two squares, so we can write

$$\left(a^{\frac{p-1}{2}} - 1\right)\left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p. \tag{A.3.2}$$

If $(p-1)/2$ is still even we can expand the left-most term further, as

$$\left(a^{\frac{p-1}{4}} - 1\right)\left(a^{\frac{p-1}{4}} + 1\right)\left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p. \tag{A.3.3}$$

Repeating this process yields

$$\left(a^{\frac{p-1}{2^s}} - 1\right)\left(a^{\frac{p-1}{2^s}} + 1\right) \cdots \left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p \tag{A.3.4}$$

for some s such that $p - 1 = 2^s d$ with d odd. The left-hand side is divisible by p; we wish to assert that p must divide one of the factors. To do so we invoke Euclid's lemma, given as Proposition 30 in Book VII of Euclid's Elements (approximately 300 B.C., translated by Richard Fitzpatrick, 2008[11]):

> If two numbers make some number by multiplying one another, and some prime number measures the number so created from them, then it will also measure one of the original numbers.

[11] Εκλεδης. Στοιχεα. 300 B.C. Translation by R. Fitzpatrick. Independently published

In modern language, we would express this as

Lemma A.3.1. *If a prime p divides the product ab of two integers a and b, then p must divide at least one of a or b.*

By the primality of p, we can therefore conclude that at least one of the following conditions must hold:

1. $a^d \equiv 1 \mod p$,
2. $a^{2^r d} \equiv -1 \equiv p - 1 \mod p$ for some $r \in 0 : (s-1)$,

where again $p - 1 = 2^s d$.

Now let $n > 4$ be the number whose primality we wish to test. Write $n - 1 = 2^s d$. For a given $a \in 2 : (n-2)$, the Miller–Rabin trial for $n > 4$ proceeds as follows. If one of the following conditions holds:

1. $a^d \equiv 1 \mod n$,
2. $a^{2^r d} \equiv -1 \equiv n - 1 \mod n$ for some $r \in 0 : (s-1)$,

then the trial is inconclusive. If none of these conditions hold, then the trial yields the conclusion that n is composite.

As with the Fermat test, the Miller–Rabin *test* consists of one or more Miller–Rabin trials with different choices of a. Choosing several bases at random gives a probabilistic primality test; Miller gave a clever choice of deterministic bases that guarantees correctness, subject to the extended Riemann hypothesis.

Question A.7. Write a function to implement the Miller–Rabin trial for given n and a.

Write another function to apply the Miller–Rabin test with all a in a given list; if no list is supplied, use as default value the single trial $a = 2$. This latter function should return a tuple `(flag, ntrials)` where `flag = False` if the Miller–Rabin test has shown n to not be prime and `True` otherwise[12], and where `ntrials` is the number of Miller–Rabin trials performed. Print the output of the function applied to the natural numbers $n \in 5 : 20$, using in each case only $a = 2$.

[12] In other words, a number with flag `True` might still be composite.

Question A.8. Using only the single trial with base $a = 2$, what is the minimal odd composite number n for which the test does not conclude that n is composite?

Question A.9. Using only the trials $a \in \{2, 3\}$, what is the minimal odd composite number n for which the test does not conclude that n is composite?

Adding a single additional trial to the Miller–Rabin test greatly extends the range of natural numbers for which the test is guaranteed to be accurate. For example, using $a \in \{2, 3, 5\}$ is guaranteed to give the correct answer for $n < 25,326,001$; using $a \in \{2, 3, 5, 7\}$ is guaranteed to give the correct answer for $n < 3,215,031,751$; using $a \in \{2, 3, 5, 7, 11\}$ is guaranteed to give the correct answer for $n < 2,152,302,898,747$[13].

[13] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980; and G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993

Question A.10. Using trials $a \in \{2, 3, 5, 7, 11, 13, 17\}$[14], how much faster or slower is the Miller–Rabin test than trial division to verify the primality of $n = 9999991111111$?

[14] With these bases, the Miller–Rabin test is guaranteed to be correct for this n.

A.4 Concluding remarks

The current state of the art for deterministic primality testing is to combine one Miller–Rabin trial (with base $a = 2$) with another test we have not discussed, the *Lucas probable prime test*. This combination is known as the Baillie–Pomerance–Selfridge–Wagstaff (or Baillie–PSW) test[15], and is the algorithm behind the primality testing algorithms in Mathematica, Maple, PARI/GP, SageMath, and other symbolic algebra systems. The reason for the popularity of this algorithm is that no composite number is known that (falsely) passes the Baillie–PSW test. The construction of such a composite number, or a proof that no such number exists, would solve a major open question in computational number theory.

[15] R. Baillie and S. S. Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980; and C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980

B The Kepler problem

(*This project relates to material in Prelims* M4: Dynamics, A7: Numerical Analysis, *and* B7.1: Classical Mechanics.)

The glorious triumph of Newton's twin discoveries of calculus and Newtonian mechanics was that it allowed us to make *physical predictions* by *solving differential equations*. Newton's second law provides an initial value problem for a second-order differential equation that, if solved, describes the motion of the given system for all future time. One of the first examples of the first written treatment of calculus, Problema II, Solutio Casus II, Ex. I of Newton (1671)[1], is to solve

$$\dot{y} := \frac{\mathrm{d}y}{\mathrm{d}t} = 1 - 3t + y + t^2 + ty \tag{B.0.1}$$

which Newton does by means of an infinite series. Indeed, in the bitter dispute with Leibniz over the discovery of calculus, Newton wrote[2]

> ...and by the Answer of Mr. Leibnitz to the first of those Letters, it is as certain that he had not then found out the Reduction of Problems either to differential Equations or to converging Series.

Of course, most differential equation initial value problems cannot be solved exactly, and so numerical methods for their approximate solution were (and remain) of pressing concern[3]. In this project you will investigate numerical algorithms for computing approximate solutions of ordinary differential equation initial value problems. For more details on this subject, see the books of Hairer, Wanner & Nørsett[4], or Hairer, Lubich & Wanner[5].

We will focus our investigations on the two-body Kepler problem, modelling the orbit of a single planet around a star. As discussed in Chapter 9, the two-body problem is amenable to symbolic analysis that does not extend to three or more bodies. However, it is best to study the qualitative and quantitative accuracy of our numerical algorithms on the simplest possible case; if an algorithm does not work for two bodies, we cannot reasonably expect it to work for three or more.

[1] I. Newton. *The Method of Fluxions and Infinite Series*. Henry Woodfall; and sold by John Nourse, 1671. Translated from the Author's Latin Original Not Yet Made Publick. To which is Subjoin'd, a Perpetual Comment Upon the Whole Work, By John Colson. Published in 1736.

[2] I. Newton. An account of the book entituled Commercium Epistolicum Collinii et Aliorum, de Analysi Promota. *Philosophical Transactions of the Royal Society of London*, 342:173–224, 1715

[3] Even for ordinary differential equation initial value problems, important mathematical and algorithmic advances continue to this day.

[4] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993

[5] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006

B.1 Equations of motion and invariants

Following section I.2 of Hairer et al.[6], we choose one of the two bodies to be the origin of our coordinate system. Since two-body motion remains in a plane (a fact you will prove in Dynamics), we consider the position $\mathbf{q} = (q_1, q_2)$ and momentum $\mathbf{p} = (p_1, p_2)$ as two-dimensional vector-valued functions of time. Setting all physical constants to one for convenience, the Hamiltonian for the system is

$$H(\mathbf{p}, \mathbf{q}) = \frac{1}{2}\left(p_1^2 + p_2^2\right) - \frac{1}{\sqrt{q_1^2 + q_2^2}}. \tag{B.1.1}$$

[6] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006

Question B.1. Symbolically calculate with sympy the resulting system of ordinary differential equations. (Recall (13.0.4).)

As a Hamiltonian system, the equations of motion you derive in Question B.1 must structurally preserve the Hamiltonian, the total energy of the system. The Kepler problem has other invariants, however[7]. Kepler's second law is equivalent to the statement that the angular momentum is conserved, which is

$$\mathbf{L}(\mathbf{p}, \mathbf{q}) = \begin{pmatrix} p_1 \\ p_2 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} q_1 \\ q_2 \\ 0 \end{pmatrix}. \tag{B.1.2}$$

[7] In fact, the Kepler problem is *maximally superintegrable*—it has as many invariants of motion as is possible to have. The concept of a system being integrable is rather subtle; Oxford's Nigel Hitchin, emeritus Savilian Professor of Geometry, gives a useful introduction in the first few pages of

N. J. Hitchin, G. B. Segal, and R. S. Ward. *Integrable systems: Twistors, loop groups, and Riemann surfaces*, volume 4 of *Oxford Graduate Texts in Mathematics*. Clarendon Press, 1999

Question B.2. Prove by symbolic substitution with sympy that the Hamiltonian is conserved, i.e. that its value does not change over time.

Question B.3. Prove by symbolic substitution with sympy that the angular momentum is conserved.

These two invariants are also conserved by n-body problems. The case $n = 2$ has one further invariant, the so-called Laplace–Runge–Lenz (LRL) vector:

$$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \begin{pmatrix} p_1 \\ p_2 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 0 \\ 0 \\ q_1 p_2 - q_2 p_1 \end{pmatrix} - \frac{1}{\sqrt{q_1^2 + q_2^2}} \begin{pmatrix} q_1 \\ q_2 \\ 0 \end{pmatrix}. \tag{B.1.3}$$

Because of its more complicated form, this invariant is much less well known, and has thus been rederived independently many times (by, among others, Hermann, Bernoulli, Laplace, Hamilton, and Gibbs).

The LRL vector points from the star being orbited to the point of closest approach, the periapsis. The LRL vector was crucial to Pauli's quantum mechanical analysis of the hydrogen atom, which is a two-body problem with a central force governed by an inverse square law, discussed in Chapter 11.

Question B.4. Prove by symbolic substitution with sympy that the LRL vector is conserved.

The conservation of H, $\mathbf{L}$, and $\mathbf{A}$ encode the crucial geometric property of Kepler's problem, that the planet should orbit in an ellipse. (Roughly speaking, H and $\mathbf{L}$ encode the shape of the ellipse, while $\mathbf{A}$ encodes its orientation.) An important way to study the effectiveness of numerical algorithms for solving differential equations is to consider how they preserve the geometric properties encoded in the equations; algorithms that honour the underlying geometry are studied in the field of *geometric numerical integration*. In our case, we are particularly interested to what extent numerical algorithms yield discretisations that conserve H, $\mathbf{L}$, and $\mathbf{A}$, given in (B.1.1), (B.1.2), and (B.1.3); if these invariants are conserved, then the discrete solution will be the correct ellipse, but if they are not, the approximate trajectory will deviate from this. Measuring the conservation error therefore gives insight into whether a particular numerical approximation is unphysical or not.

B.2 *Euler's method*

The simplest possible method for solving initial value problems is the *forward Euler method*. Suppose we are solving

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \tag{B.2.1a}$$

$$\mathbf{y}(t_0) = \mathbf{y}_0. \tag{B.2.1b}$$

Let $t_1 = t_0 + \Delta t$ with $\Delta t > 0$ small. Euler's suggestion is to make the approximation that $f(t, \mathbf{y})$ is constant over $[t_0, t_1]$, with value $f(t_0, \mathbf{y}_0)$. Integrating the equation over this interval then yields that

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \Delta t f(t_0, \mathbf{y}_0). \tag{B.2.2}$$

More generally, denoting

$$t_n = t_0 + n\Delta t, \quad \mathbf{y}_n \text{ our approximation for } \mathbf{y}(t_n), \tag{B.2.3}$$

the forward Euler scheme is to iteratively compute

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t f(t_n, \mathbf{y}_n). \tag{B.2.4}$$

The forward Euler method was proposed by Euler in *Institutiones Calculi Integralis* (the Foundations of Integral Calculus) in 1768[8]. In Volume I, Section II, Chapter 7 (translated by Ian Bruce in 2010), Euler writes

[8] L. Euler. *Institutiones Calculi Integralis*. Academia Imperialis Scientiarum, 1768. Translation by I. Bruce. Independently published

> **Concerning the approximate integration of differential equations.**
>
> **Problem 85**. To assign an approximate value to the complete integral of any differential equation.
>
> **Solution**. Let x and y be two variables, between which the differential equation is proposed, and this equation shall have a form of this kind, so that $\frac{dy}{dx} = V$ with V being some function of x and y. Now since the complete integral is desired, this has to be interpreted thus, so that while x is given a certain value, for example $x = a$, the other variable y is given a certain value, for example $y = b$. Hence in the first place we are to treat the question, so that we can find the value of y, when the value of x is attributed a value differing a little from a, on putting $x = a + \omega$ so that we may find y. But since shall be the smallest possible amount, the value of y will differ minimally from b ; from which, while x only is changed from a as far as to $a + \omega$, the quantity V is allowed to be looked on as being constant. Whereby on putting $x = a$ and $y = b$ there is made $V = A$ and from this very small change we will have $\frac{dy}{dx} = A$ and thus on integrating, $y = b + A(x - a)$, clearly with a constant of this kind to be added so that on putting $x = a$ there becomes $y = b$. Hence we may put in place $x = a + \omega$ and there becomes $y = b + A$.
>
> Hence just as here from the values given initially $x = a$ and $y = b$ we find approximately the following $x = a + \omega$ and $y = b + A$, thus from these in a like manner it is allowed to progress through another very short interval, as long as it arrives finally at values however far from the starting value.

This is a natural first idea; if Δt is small enough, the error in the value of $\mathbf{y}(t_\star)$ for some fixed $t_\star$ converges linearly as Δt is reduced (i.e. if you halve the timestep, the error in the approximation also halves). However, the forward Euler scheme does not pay heed to the geometric structure of our problem, with disastrous physical consequences, as we shall soon see.

Question B.5. Write a function to execute the forward Euler scheme to approximate the solution of the Kepler problem with initial conditions

$$\mathbf{p}(t = 0) = [0, 2], \quad \mathbf{q}(t = 0) = [0.4, 0], \tag{B.2.5}$$

for a specified timestep and number of timesteps.

On a single figure, plot the trajectory computed with forward Euler with

1. 50 steps of timestep $\Delta t = 0.1$;

2. 100 steps of timestep $\Delta t = 0.05$;

3. 200 steps of timestep $\Delta t = 0.025$.

For comparison, on the same figure plot also the true orbit, given in parametric form by

$$q_1 = -0.6 + \cos s, \quad q_2 = 0.8 \sin s, \quad s \in [0, 2\pi]. \tag{B.2.6}$$

Comment on your results.

Question B.6. For the finest discretization, on separate figures plot the computed values of H, L, and θ, where

$$L := \|\mathbf{L}\|, \quad \theta := \arg \mathbf{A} \tag{B.2.7}$$

as a function of time, where arg returns the azimuthal angle from the positive x-axis of the vector projected to the x-y plane. Are the invariants conserved by the forward Euler discretisation? What are the implications of this for the numerical approximation to the planet's orbit?

These results motivate the search for more sophisticated algorithms.

B.3 *Explicit midpoint method*

Over an interval $[t_n, t_{n+1}]$, the forward Euler method approximates the slope of the tangent to the solution $\mathbf{y}(t)$ via its (known) value at the left end-point. A natural objection to this procedure is that it *violates the symmetry inherent in the equations*: whereas Newton's laws of motion are symmetric forward or backward in time[9], the forward Euler method is not symmetric in time. More precisely, exchanging $t_n \leftrightarrow t_{n+1}$, $\mathbf{y}_n \leftrightarrow \mathbf{y}_{n+1}$, and $\Delta t \leftrightarrow -\Delta t$ in (B.2.4), we do not recover (B.2.4)[10].

This consideration of symmetry prompts us to seek a numerical method where the slope of our approximation matches the right-hand side of the ordinary differential equation *at the midpoint of the interval*, i.e. to find $\mathbf{y}_{n+1}$ such that

$$\frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} = f\left(\frac{t_n + t_{n+1}}{2}, \frac{\mathbf{y}_n + \mathbf{y}_{n+1}}{2}\right). \tag{B.3.2}$$

The numerical method defined by (B.3.2) is clearly symmetric in time by construction. However, it has a substantial disadvantage: to compute $\mathbf{y}_{n+1}$, we need to solve (B.3.2), which in general is a nonlinear

[9] Indeed, the invariance of the laws of physics in time is precisely what leads to the conservation of energy: Noether's Theorem, one of the most beautiful in all of mathematics, asserts that every differentiable symmetry of a physical system has a corresponding conservation law.

E. Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918

[10] Instead, we recover

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t f(t_{n+1}, \mathbf{y}_{n+1}). \tag{B.3.1}$$

This scheme is known as *backward Euler*.

equation. (Notice that the right-hand side depends on the unknown $\mathbf{y}_{n+1}$.) The method defined by (B.3.2) is described as *implicit*, in contrast to *explicit* methods like forward Euler (B.2.4), where one can directly compute $\mathbf{y}_{n+1}$ from $\mathbf{y}_n$. This method is therefore known as *implicit midpoint*.

Moreover, implicit midpoint has a more fundamental geometric property: the associated discrete system is *symplectic*, just as our Hamiltonian system is. Explaining symplecticity is beyond the scope of this project, but for our purposes the following rough idea will suffice. Imagine an ordinary differential equation with $\mathbf{y} \in \mathbb{R}^2$. The initial data are thus an element of $\mathbb{R}^2$. Imagine taking a (measurable) set A of many different initial conditions in $\mathbb{R}^2$, and integrating the equations of motion up to an arbitrary fixed time $T > 0$ for each $a \in A$. This procedure will give another set of points $B \subset \mathbb{R}^2$. Symplecticity means that the *area is preserved* under this procedure: the area of B is exactly the area of A, for any terminal time T. Symplecticity is crucial to the geometry of Hamiltonian mechanics; implicit midpoint preserves symplecticity, whereas forward Euler does not.

Implicit midpoint is therefore a very powerful and popular integrator[11]. However, in this project we will confine our attention to explicit schemes, as implementing implicit schemes requires a good knowledge of the numerical solution of nonlinear equations, which you will study in Trinity Term *Constructive Mathematics*. An explicit alternative is to instead approximate

$$\frac{\mathbf{y}_n + \mathbf{y}_{n+1}}{2} \approx \mathbf{y}_n + \frac{\Delta t}{2} f(t_n, \mathbf{y}_n), \tag{B.3.3}$$

i.e. we estimate the average of the initial and final states with a half-step of forward Euler. This yields the *explicit midpoint* method:

$$\frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} = f\left(\frac{t_n + t_{n+1}}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f(t_n, \mathbf{y}_n)\right). \tag{B.3.4}$$

Unlike implicit midpoint, explicit midpoint is neither symmetric nor symplectic, but it is nevertheless a substantial improvement over forward Euler: on halving the timestep, the error in $\mathbf{y}_n$ decreases by a factor of 4, instead of the factor of 2 yielded by forward Euler.

[11] It is especially popular for problems where implicit integrators are necessary for other reasons, such as the parabolic partial differential equations you will meet in Part A *A1: Differential Equations I*.

Question B.7. Write a function to execute the explicit midpoint scheme to approximate the solution of the Kepler problem. Make an analogous plot as in Question B.5, with the same initial conditions, timestep Δt, and number of steps.

Question B.8. Compare on a plot a run of forward Euler with a run

of explicit midpoint. Make the comparison fair by ensuring that both schemes require the same number of evaluations of the right-hand side f. Comment on your results.

Question B.9. Comment on the conservation properties of explicit midpoint. (No plot necessary.)

B.4 *Newton–Störmer–Verlet method*

Is it possible to devise an explicit scheme for our problem that is both symmetric and symplectic?

Our intuition for the explicit and implicit midpoint schemes was that we wished the *slope* of our approximation at the midpoint of the time interval $[t_n, t_{n+1}]$ to match that specified by the ODE. This makes sense for a generic first-order system of ODE. But we are not solving just any first-order system of ODE—we are solving a first-order reformulation of a problem that is fundamentally second-order (recall Newton's second law). In other words, our equations are of the particular form

$$\dot{\mathbf{p}} = g(\mathbf{q}), \tag{B.4.1a}$$

$$\dot{\mathbf{q}} = \mathbf{p}, \tag{B.4.1b}$$

which is the first-order reformulation of

$$\ddot{\mathbf{q}} = g(\mathbf{q}). \tag{B.4.2}$$

Since g tells us *the second derivative of* $\mathbf{q}$, this instead suggests we should find a numerical approximation that matches the *second derivative* at $\mathbf{q}_n$ with the right-hand side evaluated there. That is, given $\mathbf{q}_{n-1}$ and $\mathbf{q}_n$, we compute the next value $\mathbf{q}_{n+1}$ such that the second derivative of the quadratic function that passes through $(t_{n-1}, \mathbf{q}_{n-1})$, $(t_n, \mathbf{q}_n)$, and $(t_{n+1}, \mathbf{q}_{n+1})$ is $g(\mathbf{q}_n)$). After some algebra, the scheme that results is

$$\mathbf{q}_{n+1} - 2\mathbf{q}_n + \mathbf{q}_{n-1} = (\Delta t)^2 g(\mathbf{q}_n). \tag{B.4.3}$$

The method (B.4.3) has been invented many times, with different names used in different communities[12]. Its most common name is the Verlet method, invented by Loup Verlet in 1967[13] in the context of molecular dynamics. It is sometimes called the Störmer method, as Carl Störmer[14] used higher-order variants of it to compute the motion of ionised particles in the Earth's magnetic field to understand the *aurora borealis*[15]. Loup Verlet subsequently became interested in the history of science and discovered the method that had made

[12] This historical account is drawn from E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the Störmer–Verlet method. *Acta Numerica*, 12:399–450, 2003

[13] L. Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical Review*, 159(1):98, 1967

[14] C. Störmer. Sur les trajectoires des corpuscules électrisés. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 1907

[15] For more details, see Section III.10 of E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993

him famous had been employed by Newton in 1687 to prove Kepler's second law, the conservation of angular momentum in the two-body Kepler problem—in Theorem I of Book I of the *Principia*. We therefore refer to it here as the Newton–Störmer–Verlet method.

It turns out that the direct implementation of (B.4.3) suffers from a numerical instability in the presence of (inevitable) rounding errors[16]. A more stable equivalent implementation arises for the first-order system (B.4.1). Given $\mathbf{p}_0$ and $\mathbf{q}_0$, the Newton–Störmer–Verlet scheme is to compute

[16] For more details, see pg. 472 of Hairer et al. (1993).

$$\mathbf{p}_{n+\frac{1}{2}} = \mathbf{p}_n + \frac{\Delta t}{2} g(\mathbf{q_n}), \tag{B.4.4}$$

$$\mathbf{q}_{n+1} = \mathbf{q}_n + \Delta t \mathbf{p}_{n+\frac{1}{2}}, \tag{B.4.5}$$

$$\mathbf{p}_{n+1} = \mathbf{p}_{n+\frac{1}{2}} + \frac{\Delta t}{2} g(\mathbf{q_{n+1}}). \tag{B.4.6}$$

Like explicit and implicit midpoint, Newton–Störmer–Verlet is second-order accurate: halving the timestep quarters the error. But unlike explicit midpoint, it is both symmetric and symplectic, with crucial qualitative advantages in the simulation of the class of problems to which it applies.

Question B.10. Write a function to execute the Newton–Störmer–Verlet scheme to approximate the solution of the Kepler problem. Make an analogous plot as in Question B.5, with the same initial conditions, timestep Δt, and number of steps.

Question B.11. Compare on a plot a run of explicit midpoint with a run of Newton–Störmer–Verlet, both employing $\Delta t = 0.05$ for 1200 steps. Comment on your results.

Question B.12. Discuss the conservation properties of Newton–Störmer–Verlet. Provide evidence for your assertions, and relate your results to your answer for the previous question.

B.5 *Concluding remarks*

Hamiltonian problems (like the Kepler problem) possess an extremely rich mathematical structure. The associated differential equations are symplectic, conserve the Hamiltonian, and possibly conserve other

invariants also. However, when discretising, one must in general make a choice of structure to preserve: an approximate integrator cannot generally preserve both symplecticity and the Hamiltonian[17]. For chaotic systems, preserving symplecticity is probably the right choice, as it is crucial for their statistical behaviour; the inevitable discretisation errors mean that any individual trajectory is not particularly meaningful, but symplecticity ensures their aggregation is.

However, for other systems, it may be preferable to choose approximate schemes that exactly conserve the invariants of the system. For example, for the Kepler problem, the trajectory of such an approximation would be confined to exactly the same ellipse as that of the true solution, which is very appealing. More generally, the design of such structure-preserving discretisations for physical systems such as the Navier–Stokes equations of fluid mechanics or the Einstein field equations of general relativity is a major focus of ongoing research in the numerical analysis of differential equations.

[17] G. Zhong and J. E. Marsden. Lie–Poisson Hamilton–Jacobi theory and Lie–Poisson integrators. *Physics Letters A*, 133(3):134–139, 1988

C Percolation

(*This project relates to material in Prelims and Part A courses on* Probability *and* Statistics*, and Part A* Simulation and Statistical Programming.)

Statistical mechanics is the branch of mathematics that applies statistical and probabilistic methods to large assemblies of microscopic entities. For example, one might consider a gas as composed of a very large number of molecules moving in all directions and colliding with each other. From this viewpoint we wish to derive macroscopic properties like its pressure or temperature. Of course, keeping track of the state of each individual molecule among the trillions of trillions in a typical cubic metre is simply impossible. One therefore instead develops a theory where one considers the *probability distribution* of the molecules of the gas; given knowledge of the probability distribution, we can at any time calculate the number of molecules of a certain velocity range in a certain volume of space. In 1860, Maxwell calculated the equilibrium distribution for a gas at a given temperature, now known as the Maxwellian[1]; in 1872 Boltzmann derived the equation governing the time evolution of the probability distribution function, now known as the Boltzmann equation[2].

The Boltzmann equation is a nonlinear integro–differential equation, where the unknown is a function in six dimensions (three of position, three of velocity). It is therefore not terribly surprising that it is rather hard to solve. Computational simulations are crucial to gaining mathematical and physical insight into most systems of statistical mechanics.

A major goal of statistical mechanics is to understand *phase transitions*. A phase transition is an abrupt, discontinuous change in the properties of a system. For example, if we take our gas and cool it, at a critical temperature it will (usually) turn into a liquid, with its density and volume changing discontinuously. Phase transitions are of enormous mathematical, physical, and economic importance. For example, some materials (known as superconductors) exhibit a phase transition at a critical temperature, below which they offer no resistance to electrical current. The discovery of a practical superconducting material where the critical temperature is above room temperature would

[1] J. C. Maxwell. V. Illustrations of the dynamical theory of gases. Part I. On the motions and collisions of perfectly elastic spheres. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 19(124):19–32, 1860

[2] L. Boltzmann. Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen. *Sitzungsberichte der Akademie der Wissenschaften zu Wien*, 66:275–730, 1872

trigger a second industrial revolution[3].

One route to understanding phase transitions is to consider simple mathematical models that exhibit them. A prominent class of such mathematical models is studied in *percolation theory*[4]. Percolation theory describes the properties of a graph as nodes or edges are added. Hugo Duminil-Copin won the Fields Medal in 2022 for his work on percolation theory; a popular account of his work was published in Quanta magazine[5].

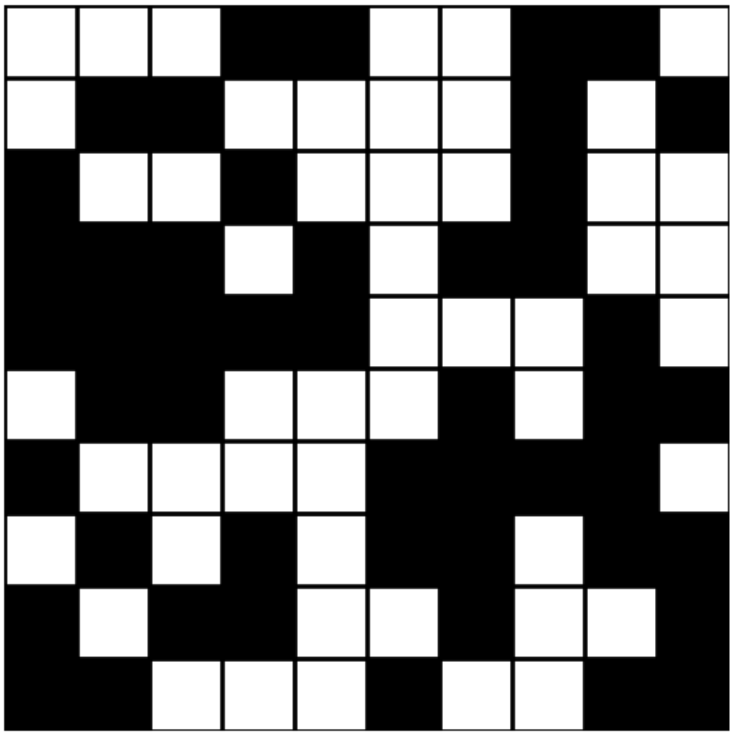

Figure C.1: A 10×10 grid sampled with vacancy probability $p = 0.5$. Open squares are white; closed squares are black.

The specific percolation model we consider in this project is the following. Consider an $n \times n$ grid of squares, where each site can be either *open* or *closed*, as in Figure C.1. We say that a site is *full* if it is open and can be connected to an open site in the top row via chain of neighbouring (left, right, up, down) open sites. If there is a full site on the bottom row, we say the system percolates (see Figures C.2 and C.3 for examples). In other words, is there a connected open component spanning from the top of the grid to the bottom? You might imagine this to model the question of whether water poured on the top will percolate to the bottom (hence the name), or whether a fire started at one end of a forest will propagate tree-by-tree to the other.

A natural question to ask about this system is: if each site is open with *vacancy probability* p, what is the probability that the system percolates, $C(p)$? As we will see, the percolation probability $C(p)$

[3] The current record at atmospheric pressure is held by the cuprate of mercury, barium, and calcium, which has a critical temperature around -140 °C.

[4] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. Taylor & Francis, second edition, 1994

[5] Hugo Duminil-Copin wins the Fields medal. *Quanta Magazine*, 2022

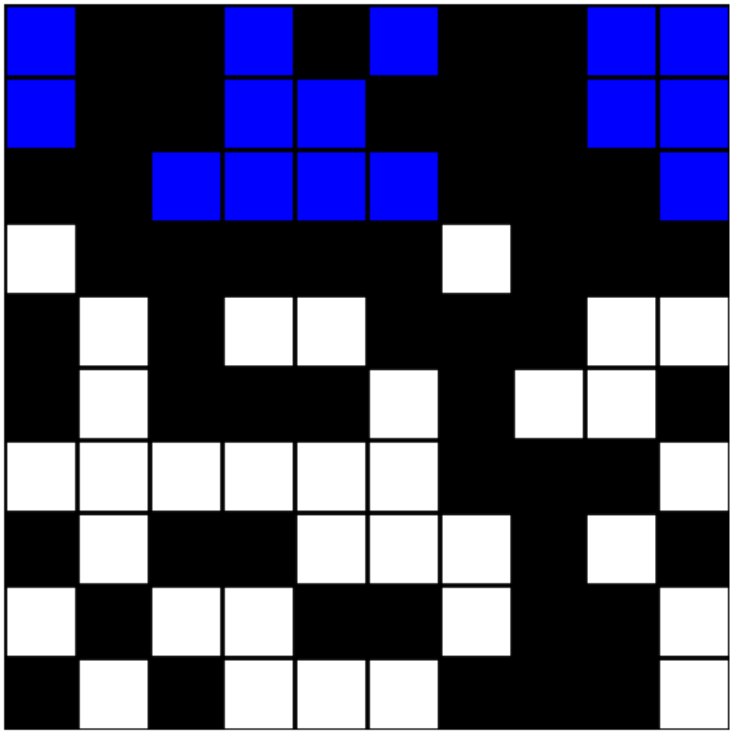

Figure C.2: A 10 × 10 grid sampled with vacancy probability $p = 0.5$. Full squares are blue. The system does not percolate.

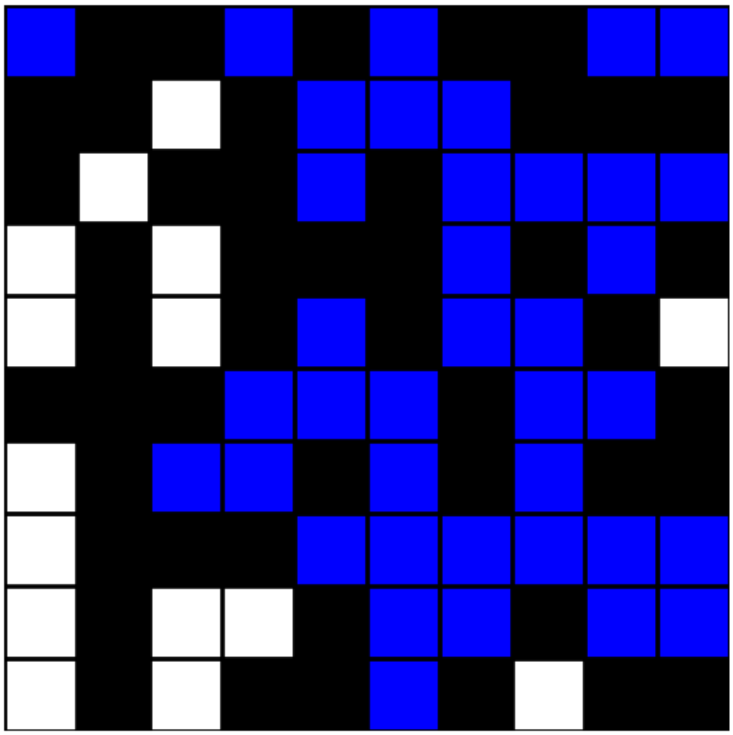

Figure C.3: A 10 × 10 grid sampled with vacancy probability $p = 0.5$. Full squares are blue. The system does percolate.

exhibits a phase transition in the vacancy probability. For low values of p, the system does not percolate; at a critical $p = p_c$ the system very rapidly switches to always percolating[6]. No analytical results are known characterising p_c. As described by Newman & Ziff[7],

> Percolation is one of the best-studied problems in statistical mechanics. ...It is one of the simplest and best understood examples of a phase transition in any system, and yet there are many things about it that are still not known. For example, despite decades of effort, no exact solution for the site percolation problem yet exists on the simplest two-dimensional lattice, the square lattice, and no exact results are known on any lattice in three dimensions or above. Because of these and many other gaps in our current understanding of percolation, numerical simulations have found wide use in the field.

[6] In fact, in the limit of infinite grid size, the percolation probability for a given p is either zero or one, by Kolmogorov's zero-one law. For small n the transition becomes smoother.

[7] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001

By means of such numerical simulations, Newman & Ziff computed that the critical probability p_c for large n was approximately $p_c \approx 0.59274621$[8].

[8] M. E. J. Newman and R. M. Ziff. Efficient Monte Carlo algorithm and high-precision results for percolation. *Physical Review Letters*, 85(19):4104–4107, 2000

In this project you will investigate Monte Carlo algorithms for estimating the percolation probability $C(p)$. Monte Carlo methods are one of the most important and prominent tools for modern statistical inference. In a Monte Carlo simulation, we randomly draw inputs from a suitable probability distribution (in this case, the binomial distribution), perform deterministic computations (in this case, decide whether the grid percolates or not), and aggregate the results (to compute $C(p)$).

C.1 *Representing the state*

Our first task is to generate suitable random samples of our grids.

Question C.1. Write a function `make_grid(n, p)` to make an $n \times n$ numpy array of Boolean values, with each site `True` with probability p and `False` otherwise.

[*Hint: this should take one line of numpy code.*]

Draw a few samples to ensure that the empirical probability of a site being open is approximately p.

Our next task is to visualise our grid status. This will be very useful in developing the code for the simulation.

Question C.2.

Write a function `visualise_grid` to visualise a grid produced by `make_grid` with matplotlib. The function should take in a Boolean array. The output should look similar to Figure C.1: plot closed sites in black; plot open sites in white; colour the borders of each square in black.

[*Hint: you will need to consult the matplotlib documentation and other online resources to do this; the relevant matplotlib methods were not discussed in Chapter 7.*]

Use your function to visualise a few sample grids.

C.2 *Calculating percolation*

Once we have represented our grid, we now proceed to calculate whether each site is full or not. As with our grid, we represent the full status of each site with a numpy array of Boolean variables.

Question C.3.
Write a function `visualise_fill` to visualise the fill status of a given grid. The function should take as input two Boolean arrays, the grid and the fill status. The output should look similar to Figures C.2 and C.3. Plot closed sites in black, open unfilled sites in white, and open filled sites in blue. Colour the borders of each square in black.

Apply your function to hand-crafted data (e.g. on a 3×3 grid) to verify it is working correctly. Ensure also that the visualisation code works correctly if the fill status is all `False`.

We now turn to the central task of computing the full status of each site. This is more subtle than it appears. An outline might be the following.

1. Visit each site in the top row.
2. For each visited site, do the following:
   (a) If appropriate, set the site to be full.
   (b) Visit each neighbouring site (left, right, up, down).

This general approach is known in the graph theory literature as *depth-first search*. It is most naturally written as a function that recursively calls itself, but non-recursive implementations are also possible.

Question C.4. Write a function `compute_fill` that takes in a grid produced by `make_grid` and calculates whether each site is full or not.

[*Hint: think carefully about what should happen when a site is visited. For example, if it is already full, it should terminate without further action.*]

[*Hint: it may be useful during your development to visualise the fill state of the grid as you visit each site in the top row.*]

Question C.5. Write a function `percolates` that returns `True` if the given grid percolates, and `False` otherwise.

[*Hint: the core logic can be written with one line of numpy.*]

Draw 10 samples of a 20×20 grid with vacancy probability $p = 0.6$. For each, visualise its fill status, titling each figure with whether that grid percolates or not.

C.3 *Monte Carlo simulation*

With the `percolate` function in hand, we can now conduct our statistical simulation. Our goal here is to calculate $C(p)$, the percolation probability as a function of the vacancy probability.

Question C.6. Take a suitable grid $P \subset [0, 1]$ of p values. (You may wish to increase the resolution for $p \in [0.4, 0.7]$.) For each $p \in P$, draw N samples of a 20×20 grid with vacancy probability p. For each sample, calculate whether the grid percolates or not; the fraction of grids that percolates is our estimate for $C(p)$. Plot $C(p)$ as a function of p.

[*Hint: you will need to choose suitable N and P so that the interpolation error and statistical error due to sampling are acceptable. The curve should appear smooth; if it is not, try increasing N and/or refining P.*]

A word on computational efficiency is in order for question C.6. This question is the most computationally intensive across the three projects[9]. When calling `publish()` as usual, your code actually gets executed twice; normally this is not a problem, but here it may be. Instead, with the latest version of `publish.py` it is possible to execute

[9] On an old laptop, the unoptimised reference solution for question C.6 takes approximately 9 minutes.

(terminal) `python publish.py percolation.py`

which publishes your script, only executing it once.

C.4 *Concluding remarks*

The algorithms investigated in this project can be substantially improved upon. For example, Newman & Ziff propose an entirely different approach to the simulation of site percolation that is several million times faster for 1000×1000[10]. Their approach relies on an alternative representation of the state of the system, explicitly keeping track of each connected cluster as a tree; with this alternative representation, entirely different statistical ensembles and search algorithms are used.

[10] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001

In computational mathematics, there is a constant iteration between programming, computation, and theory; computations motivate new mathematical questions, and mathematical insights make new computations possible.

Bibliography

[1] Hugo Duminil-Copin wins the Fields medal. *Quanta Magazine*, 2022.

[2] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[3] W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703, 1994.

[4] R. Baillie and S. S. Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980.

[5] L. Boltzmann. Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen. *Sitzungsberichte der Akademie der Wissenschaften zu Wien*, 66:275–730, 1872.

[6] F. Bornemann. PRIMES is in P: a breakthrough for ''Everyman''. *Notices of the American Mathematical Society*, 50(5):545–553, 2003.

[7] R. Crandall and C. Pomerance. *Prime Numbers: a Computational Perspective*. Springer-Verlag New York, second edition, 2010.

[8] L. Euler. *Institutiones Calculi Integralis*. Academia Imperialis Scientiarum, 1768. Translation by I. Bruce. Independently published.

[9] L. Pisano (Fibonacci). *Liber Abaci*. Springer Science & Business Media, 2003. Translation by L. E. Sigler.

[10] C. F. Gauss. *Disquisitiones Arithmeticae*. Yale University Press, 1965. Translation by A. A. Clarke.

[11] E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the Störmer–Verlet method. *Acta Numerica*, 12:399–450, 2003.

[12] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006.

[13] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993.

[14] N. J. Hitchin, G. B. Segal, and R. S. Ward. *Integrable systems: Twistors, loop groups, and Riemann surfaces*, volume 4 of *Oxford Graduate Texts in Mathematics*. Clarendon Press, 1999.

[15] G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993.

[16] J. C. Maxwell. V. Illustrations of the dynamical theory of gases. Part I. On the motions and collisions of perfectly elastic spheres. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 19(124):19–32, 1860.

[17] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[18] M. E. J. Newman and R. M. Ziff. Efficient Monte Carlo algorithm and high-precision results for percolation. *Physical Review Letters*, 85(19):4104–4107, 2000.

[19] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001.

[20] I. Newton. *The Method of Fluxions and Infinite Series*. Henry Woodfall; and sold by John Nourse, 1671. Translated from the Author's Latin Original Not Yet Made Publick. To which is Subjoin'd, a Perpetual Comment Upon the Whole Work, By John Colson. Published in 1736.

[21] I. Newton. An account of the book entituled Commercium Epistolicum Collinii et Aliorum, de Analysi Promota. *Philosophical Transactions of the Royal Society of London*, 342:173–224, 1715.

[22] E. Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918.

[23] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

[24] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[25] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. Taylor & Francis, second edition, 1994.

[26] C. Störmer. Sur les trajectoires des corpuscules électrisés. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 1907.

[27] L. Verlet. Computer ''experiments'' on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical Review*, 159(1):98, 1967.

[28] G. Zhong and J. E. Marsden. Lie–Poisson Hamilton–Jacobi theory and Lie–Poisson integrators. *Physics Letters A*, 133(3):134–139, 1988.

[29] Εκλεδης. Στοιχεα. 300 B.C. Translation by R. Fitzpatrick. Independently published.