Ellipse and its normals

Large computer simulations play a key role in all fields of scientific research, from artificial photosynthesis [8] to astrophysics [9]. These simulations might model a real world phenomenon, train and evaluate the performance of a machine learning model, or use numerical algorithms to solve an equation which can't be solved through analytical methods, even if a solution can be shown to exist. In this report we will investigate the performance of a numerical method on a given a problem when we know how many solutions there should be, then we will consider some improvements which can be made to this method, and finally we will discuss the environmental and economic impacts of running such computer simulations in today's context of exascale computing.

Theory

An ellipse can be defined in cartesian coordinates by

$$\frac{x^2}{a} + \frac{y^2}{b} = 1,$$

or parametrically in polar coordinates by

$$x = a\cos t,\ y = b\sin t,$$

where $t \in [0, 2\pi)$, a is the semi-major axis, and b is the semi-minor axis ($a \geq b$). A normal is a line which is perpendicular to the tangent at the point where it meets the curve. Given some $(x_1, y_1) \in \mathbb{R}^2$ we want to know how many normals of the ellipse pass through this point.

The first approach we will consider is to try and find the points where the normals meet the ellipse. Specifically, we will try to find an expression for the azimuthal angles (values of t) of these points.

We can consider the distance between the point (x_1, y_1) and the curve given parametrically by $(x(t), y(t))$ as a function of the parameter. It can then be shown (`proof1.pdf` in [1]) that if t_0 gives a stationary point of the distance function, then the line between $(x(t_0), y(t_0))$ and (x_1, y_1) is a normal.

In the case of an ellipse, this means all t satisfying

$$a\sin t(x_1 - a\cos t) = b\cos t(y_1 - b\sin t) \qquad (1)$$

each give us a normal which passes through (x_1, y_1).

Numerical Solution

It is very difficult (perhaps impossible) to find the solutions of Eq. (1) analytically, in general, so we will solve it numerically for specific values of a and b, and for a range of x_1 and y_1. Specifically, we will use `scipy.optimize.fsolve` in python, which is a wrapper for `HYBRD` in the Fortran library `MINPACK` [2]. `HYBRD` implements Powell's dog leg method (also known as the Hybrid method) for finding solutions of non-linear least squares problems [3]. The algorithm is a combination of the Gauss-Newton algorithm (which is itself an extension of the Newton-Raphson method) and gradient descent.

We will find the solutions of Eq. (1) in the interval $[0, 2\pi)$ given starting estimates of $0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$, over a grid of 10^8 points where $x \in [-2.2, 2.2]$ and $y \in [-5, 5]$[1], for an ellipse with semi-major axis $a = 2$ and semi-minor axis $b = 1$.

The function `fsolve` returns as many solutions as we give estimates, however we don't always expect to get four solutions, so we have to choose at what precision we consider two numerical solutions to be "the same". Fig. 1 shows the number of solutions for the grid of points and ellipse when we assume solutions which are the same to 9 d.p. are the same, and Fig. 2 shows this for 3 d.p.

Both figures show a similar shape for the number of points with four normals (yellow region), however Fig. 1 has a larger continuous region of yellow, along with much more noise outside this region than Fig. 2. At first glance this noise may suggest some sort of fractal pattern (e.g. like the Mandelbrot set), however this is not the case. In Fig. 2 we can see a continuous, smooth yellow region, outlined in orange, which is the ellipse's evolute. The evolute of a curve is the locus of the

[1] Technically, we only perform the calculations for the first quadrant, since the distances from e.g. $(1, 2)$ and $(-1, 2)$ to the ellipse are the same.

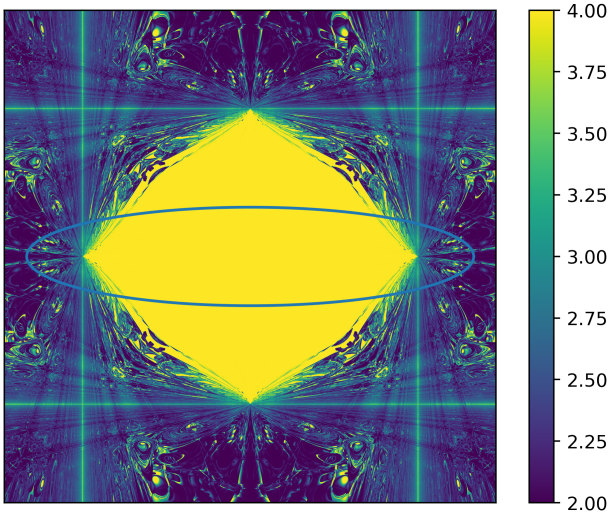

Figure 1: The number of numerical solutions of Eq. (1) (with $a = 2$ and $b = 1$) which are different to 9 deicmal places, for a grid of 10^8 points with $-2.2 \leq x \leq 2.2$ and $-5 \leq y \leq 5$. The ellipse is shown in blue.

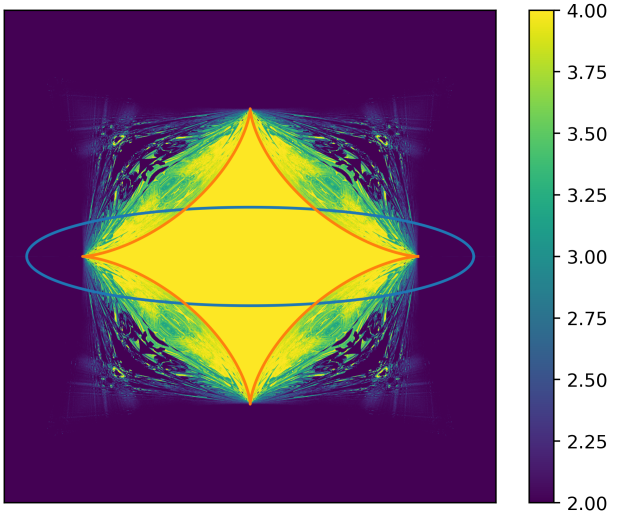

Figure 2: The number of numerical solutions of Eq. (1) (with $a = 2$ and $b = 1$) which are different to 3 decimal places, for a grid of 10^8 points with $-2.2 \leq x \leq 2.2$ and $-5 \leq y \leq 5$. The ellipse is shown in blue and its evolute in orange.

centres of its osculating circles (an osculating circle at a point is a circle which has both the same tangent and curvature as the curve at that point), or it can alternatively be defined as the envelope of the normals of the curve. This latter definition suggests a link to the problem we are investigating here, and it can be proved that for any point on the interior of the evolute, there are four normals of the ellipse which pass through it, three for any point on the edge, and two for all other points; the details of the proof can be found in [4].

The question of *how many* normals of the ellipse pass through a particular point thus has a closed solution, however if we wish to find the normals this does not help us. We have seen one numerical method for finding the normals, however it is clearly inaccurate, so we will investigate why it fails, and consider ways of improving it.

Upon closer inspection, we can see (`fullAnimation.mp4` in [1]) that minimum and maximum turning points of the function can be found with reasonable accuracy by `fsolve`, but it has trouble with points which are nearly inflection points, (an example is shown in Fig. 3). The algorithm's weaker performance on inflection points comes about due to the fact that it implements gradient descent, i.e. looks for minima/maxima.

Optimisation

There are other classes of root-finding algorithms, many of which don't rely on gradient descent. For example, there is the Bisection method (see [5], 4.2), which takes an interval in which a root is known to exist and essentially performs a binary search [7] until a small enough interval is reached. This method belongs to the class of bracketing methods, which generally use the intermediate value theorem to find roots. There are also "fixed point methods" of root finding, an example being the Secant method ([5], 4.3.4) which is a finite difference version of Newton's method ([5], 4.3.3), but is generally faster as it doesn't require explicit derivative calculations.

Instead of relying on just one algorithm, another possible approach is to start with a slower algorithm which is guaranteed or very likely to converge, e.g. bisection, then once we have a rough estimate of where the root must exist, we can use this as an initial guess for a faster algorithm.

Aside from the algorithm used to compute

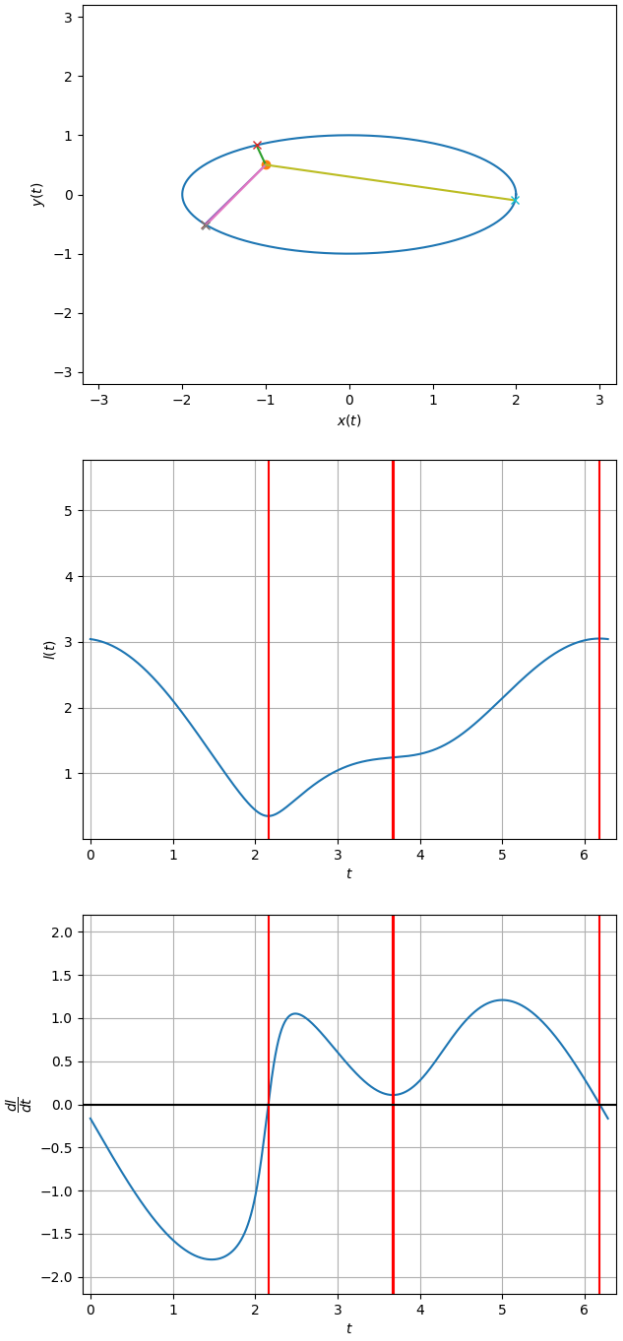

Figure 3: Top: The ellipse with $a = 2$ and $b = 1$ (in blue), the points which have been numerically calculated to give stationary points of $l(t)$ (crosses), the point (-1, 0.5) (orange dot), and the resulting lines which connect the points on the ellipse to the target point. Middle: $l(t)$ for $t \in [0, 2\pi]$ (blue) and the values of t satisfying $dl/dt = 0$ (red). Bottom: dl/dt (blue) and the values of t satisfying $dl/dt = 0$ (red).

the roots, the numerical approach taken here has other downfalls. For example, the entire $10000 \times 10000 \times 4$[2] array has to be stored in memory for the duration of the program's running. This isn't a huge issue for a standard computer as it is roughly 380MB when storing numbers as 8-bit integers (or about 3GB if using full precision floats, which is the default in NumPy). However, if the problem required a much larger array, or more precision, it would be very inefficient to store all of it in memory, when only a small part at a time is needed.

Conclusion

In this report we have looked at the numerical solution to a problem and analysed ways in which could be improved. This sort of analysis is very relevant in high performance computing today.

Large computations carried out by researchers are increasingly requiring more powerful hardware, particularly in fields like data science and machine learning, but also in running simulations of the real world. We can take an inefficient program and run it on a more powerful computer, however it is better to increase the efficiency of programs as this can not only reduce the execution time, but can also help reduce the environmental and economic impacts of carrying out research.

We can consider the benefits of increasing efficiency by looking at the energy consumption of the world's first exascale computer (i.e. capable of 1 exaflops, 10^{18} **fl**oating point **o**perations **p**er **s**econd) Frontier [6], which became operational in 2022. (The following numbers are from [10]). Frontier has a performance of about 50 Gigaflops per Watt and continuously consumes 20MW. In the USA, electricity costs roughly $0.20 per kWh which means that Frontier costs more than $35,000,000 per year to run in just electricity costs. This is about 100,000 tons of CO_2-equivalent annually or 20,000 cars in the USA. From this we can see that increasing the efficiency of programs running on systems like Fron-

[2] We solved the equation numerically then saved four versions of the result depending on the precision used (3, 5, 7, 9 d.p.). The other images can be found in [1].

tier has the potential to save millions of dollars and take the equivalent of thousands of cars off the road, while also allowing research to happen faster.

We have only scratched the surface when it comes to improving the efficiency of large scale computer simulations and computations. Some further discussion can be found in [10, 11]. These problems will have to be tackled not only by those working at the forefront of HPC, but also by the researchers making use of it.

Bibliography

[1] AMA3020 Solo Project Supplementary Materials

https://github.com/AMG3141/AMA3020-Solo-Project-Supplementary-Materials

[2] Burton S. Garbow, Kenneth E. Hillstrom, Jorge J. Moré — *MINIPACK Documentation*, University of Utah

https://www.math.utah.edu/software/minpack/minpack/hybrd.html

(Accessed 2024-02-28)

[3] Wikipedia Contributors — *Powell's dog leg method*, Wikipedia

https://en.wikipedia.org/w/index.php?title=Powell%27s_dog_leg_method&oldid=1143951512

(Accessed 2024-02-28)

[4] Frederick Hartmann and Robert Jantzen — *Apollonius's Ellipse and Evolute Revisited*, Mathematical Association of America

https://maa.org/book/export/html/116798

(Accessed 2024-02-29)

[5] Johnson, Lee W., and R. Dean (Ronald Dean) Riess — *Numerical Analysis. 2nd ed.* Addison-Wesley, 1982. ISBN 0201103923

[6] *Frontier*, Oak Ridge National Laboratory

https://www.olcf.ornl.gov/frontier/

(Accessed 2024-03-02)

[7] Wikipedia Contributors — *Binary search algorithm*, Wikipedia

https://en.wikipedia.org/w/index.php?title=Binary_search_algorithm&oldid=1196513623

(Accessed 2024-03-03)

[8] *Simulation of Artificial Photosynthesis*, Northern Ireland High Performance Computing

https://www.ni-hpc.ac.uk/CaseStudies/SimulationofArtificialPhotosynthesis/

(Accessed 2024-03-03)

[9] *Neutron Star Collision*, Northern Ireland High Performance Computing

https://www.ni-hpc.ac.uk/CaseStudies/NeutronStarCollision/

(Accessed 2024-03-03)

[10] Prof David Keyes, *Efficient Computation through Tuned Approximation*, seminar on 2024-02-21 in Queen's University Belfast. Older version available online at https://youtu.be/KaPPLW_QF5w

[11] Sips, H J — *Programming Languages for High Performance Computers*

https://cds.cern.ch/record/400331/files/p229.pdf

(Accessed 2024-03-03)