Due Friday, November 27, 24:00

Problem 18 (Bonus)

Attempt this problem only if you really like genetic algorithms. There are too many parameters to play with, and you can get non-optimal results even due to simple mistakes. Depending on your luck, it can be a quick project or can take days. There are python files on the internet that tries to solve this same problem, remember my policy, feel free to utilize internet as a resource, but copying someone's code is plagiarism.

One of the most famous problems of classical mechanics is finding the brachistochrone, the curve that makes an object moving on it under gravity take the minimum time between two given points (x_i, y_i), (x_f, y_f). The solution is typically calculated analytically using calculus of variations (remember your PHYS201), but if you are twisted enough by this course you can also view it as an optimization problem. After all, we are looking for the curve with the *minimum* time, so the objective function is the time spent on the curve $y(x)$:

$$t = \int_{x_i}^{x_f} \sqrt{\frac{1 + (dy/dx)^2}{2gy}} dx \ .$$

However, realize that every point on the curve is a parameter to vary, hence, this is an infinite dimensional problem!

(a) Let us simplify the problem by approximating the curve with its value on N equidistant points on the x axis in addition to the known end points which you can take to be $(0, 0)$ and $(1, -1)$. This means the curve is specified by N parameters which are the values $y_j \equiv y(x_i + jh)$ for $j = 1, \ldots, N$. Use a genetic algorithm to evolve parameter vectors of length N (chromosomes) to find the optimal solution. We already wrote functions to calculate derivatives and integrals in terms of discretely sampled points in the previous problem set, so you can use them to express the objective function in terms of our parameter vector. I leave it to you to test for the optimal N, starting distribution, mutation rate, selection rate and number of generations. You can mutate your vectors using the Gaussian normal distribution `numpy.random.normal`. It is not easy to determine a good stopping criteria for genetic algorithms.

- You can evolve the population for a fixed number of generations (you may need thousands), increasing the number, if needed, after trial and error.
- You can stop if the best value of the objective function in your population has not changed in a fixed number of simulations (assuming you are not going to get any better).

The sample problem in your textbook decreases the mutation rate as the generations increase (see page 115). Try both a fixed and adaptive rate.

Compare your result with the famous solution in the literature. If you think it helps, feel free to use any builtin function such as `scipy.optimize.differential_evolution`, but implementing the algorithm yourself is not much work.

The way you mutate your vectors are crucial in this problem, the algorithm can get stuck far from the solution if you just randomly mutate each y_j independently. This is because random uncorrelated changed can make your curve jagged, making it obviously worse, so once you happen to have a relatively smooth curve, any mutation is rejected, even if this smooth curve is far from optimal.[1]

(b) Repeat the previous part, but this time instead of using a piecewise linear approximation, approximate the function with a truncated Fourier series

$$y(x) \approx y_i + \frac{y_f - y_i}{x_f - x_i}(x - x_i) + \sum_{j=1}^{N} A_j \sin\left(\frac{j\pi}{x_f - x_i} x\right)$$

[1] This is clearly demonstrated in this website, though they perform something called *mating* in addition to mutation.

where the parameter vector to be evolved is that of the A_j values. This approach always provides smoother functions, so see if it is better than the previous one.

Problem 19 Celestial mechanics (6 points)

The quintessential dynamical system is the two-body problem in astronomy which is exactly solvable. We can reduce the equations of motion to that of a single particle, but we will treat the bodies separately in our numerical solution:

$$m_i \frac{d^2\vec{r}_i}{dt^2} = -\frac{Gm_1m_2}{|\vec{r}_1+\vec{r}_2|^2}\hat{r}_i$$

where $i = 1, 2$, $\hat{v}$ is the unit vector in the direction of $\vec{v}$, and we have assumed that the center of mass (CM) is at the origin. Remember that in the CM reference frame the two point-like masses move on elliptical orbits that lie on a common plane, hence $\vec{r}_i$ can be taken to be two dimensional without loss of generality.

Let us examine the Sun and Jupiter this way ignoring other objects in the Solar system. You can look up the velocity of Jupiter and its distance to the Sun at perihelion (closest approach) from NASA's fact sheet: $v_p = 13.72$km/s and $r_p = 7.405 \times 10^8$km. $m_J = 1.898 \times 10^{27}$kg and $m_S = 1.988 \times 10^{30}$kg uniquely determine the initial conditions in the CM frame.

(a) It is usually good practice to scale our quantities so that we do not have to deal with very large or very small numbers in simulations. The natural length scale in this problem is r_p, the natural mass scale is the reduced mass $\mu = m_S m_J/(m_S m_J)$, and the natural time scale is $t_0 \equiv \sqrt{r_p^3/[G(m_S+m_J)]}$. In other words, we expect the unitless quantitites $|\vec{\rho}_i| \equiv |\vec{r}_i/r_p|$ and $\tau \equiv t/t_0$ to be of order unity, and know that $\mu_i \equiv m_i/\mu$ are relatively close to unity[2] Explain the choice of t_0. Show that the equations of motion reduce to

$$\mu_i \frac{d^2\vec{\rho}_i}{d\tau^2} = -\frac{1}{|\vec{\rho}_1+\vec{\rho}_2|^2}\hat{\rho}_i$$

Notice that we could use m_J as our natural mass scale, or $\sqrt{r_p^3/(Gm_S)}$ as the time scale with similar effects, but our specific choices lead to a particularly simple expression for our unitless quantities. Proceed with these variables for the numerical calculation. Another way of looking at our scaling is saying that our unit of time is t_0, unit of mass is μ, and unit of length is r_p. You should convert the quantities in initial conditions from SI to these in your simulation.

(b) Implement the function `time_step(vec, t, rhs, dt, method)` [3] that evolves a vector `vec` of positions and velocities with derivative `rhs` (the right hand side of the ODE, a function of `vec` and `t`) from `t` to `t+dt`. Implement the `method` options for forward Euler, symplectic Euler, implicit Euler and Runge-Kutta 4. Evolve the system from the above initial conditions for ~ 10 orbits with dt $= 1/64$, and save the dynamical variables. Plot the orbits (on the $x-y$ plane), and the conserved quantities energy, angular momentum and linear momentum (vs time). Test the speed of each method and compare. The true orbits are exact ellipses with no precession, what about numerical orbits? How well are the conserved quantities conserved? Now, decrease the time steps separately for each method until you have "acceptable" orbits, and report how the methods compare in terms of speed for these cases. All these schemes are converging with smaller `dt`, so you should get better results, but has the qualitative behavior of the conserved quantities changed?

I hope the above results demonstrate the importance of using a numerical method that respects the physical structure of the system under investigation. Note that all Euler methods are linearly convergent, but their behaviors are widely different. Perhaps even more strikingly, symplectic and implicit

[2] Of course $m_J/m_S \sim 10^{-3}$ is still quite small, but this is a physically meaningful and unit-independent number, you cannot get rid of this smallness since it is a ratio. The situation is much better than 10^{24} and 10^{30} which you would use in SI units.

[3] Remember that this `t` is in our particular units, i.e. it is τ.

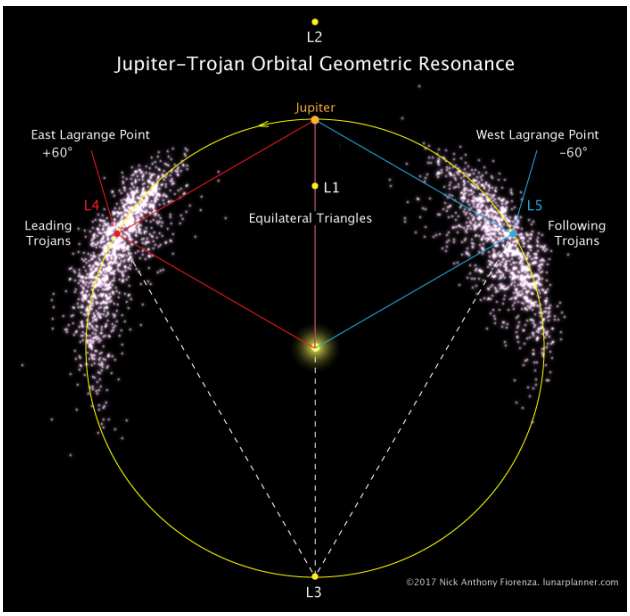

Figure 1: The five Lagrange points of the Sun-Jupiter system. Only L_4 and L_5 are stable, and there are large groups of small objects around these points called the Jupiter Trojans. Adapted from lunarplanner.com.

Euler methods converge more slowly than RK4, but still have better long term qualitative behavior. Convergence is very important, but it is not the only important aspect of a numerical method.

(c) The restricted three body problem is solving the dynamics of three gravitationally interacting bodies where one of the bodies has negligible mass. In this case, the more massive bodies behave as in the two-body problem, and the third one moves under their gravitational force. The Sun-Jupiter system together with a small third body is an excellent fit to this description. Among the most curious aspects of this problem are the five Lagrange points where the small body would stay fixed relative to the other two bodies in their rotating frame (see the figure above). The first three Lagrange points, out of a total of five, lie on the line connecting the Sun and Jupiter and they are unstable. You can calculate the exact positions of these by finding the roots of some polynomials (see Wikipedia). L_4 and L_5 form equilateral triangles with the Sun and Jupiter, and they are stable. Note that the Sun is not at the origin of your coordinate system, it is also rotating around the origin which is the CM. Establishing the stability properties of Lagrange points is not easy, so we will analyze orbits of objects near these points numerically.

First, let us simplify the Sun-Jupiter system by assuming that they are on a circular orbit with separation r_p. Together with the masses above, you can analytically calculate the common angular velocity Ω of the system, and the positions of both the Sun and Jupiter obey

$$x_i = R_i \cos \Omega t$$
$$y_i = R_i \sin \Omega t$$

for constants R_i where $i = J, S.$, i.e. $R_J = r_p m_S/(m_S + m_J)$ and $R_S = -r_p m_J/(m_S + m_J)$. We will use these analytical formula to evolve the position of the third body numerically.

Calculate the positions of the Lagrange points in the initial configuration of the Sun-Jupiter system. Put a small mass in each point with an initial velocity so that the objects move with angular velocity Ω w.r.t. the center of mass in the inertial frame, i.e. they do not move in the rotating reference

frame. Evolve each object for 10 periods of the two-body system using Verlet integration (second order symplectic integration).

Perform the numerical integration in the inertial frame, but after you obtain the positions, it is easier to see the movement of the small bodies in the rotating frame, around the Lagrange points. You can transform the inertial frame coordinates $\vec{\rho}_i$ to the rotating ones using

$$\vec{\rho}_{\rm rot} = \begin{pmatrix} \cos\Omega t & \sin\Omega t \\ -\sin\Omega t & \cos\Omega t \end{pmatrix} \vec{\rho}_{\rm in} - \vec{\rho}_{L,\rm in}$$

where we also translated by the constant rotating frame coordinate of the Lagrange point $\vec{\rho}_{L,\rm in}$. So, if the body is initially at the Lagrange point with the correct velocity, analytically $\vec{\rho}_{\rm rot}(t) = \vec{\rho}_{\rm rot}(0) = 0$, i.e. the change in $\vec{\rho}_{\rm rot}(t)$ shows you the stability of the object. Plot the orbits in terms of $\vec{\rho}_{\rm rot}$ for all Lagrange points, and discuss the results in light of the above information on stability.

For L_4, repeat the above steps for initial positions slightly away from the Lagrange point and try to have an estimate for the region of stability (see the above figure for the real region). Show your results in plots.

(d) Pick one stable case and one unstable case in part (c). Repeat the time evolution for $h/2$ and $h/4$. Analyze the error in the coordinates and the conserved quantities. Is the convergence behavior what you expect?

You can find many Python implementations of Verlet integration online, but it is not part of SciPy as far as I know.[4] Julia has symplectic integrators and a very good differential equations package if you are willing to learn the language, which I hear is surprisingly easy. There is an advanced monograph by Hairer, Lubich and Wanner, *Geometric Numerical Integration*, if you want to learn more about structure-preserving algorithms, e.g. symplectic integrators, in general. Note that this text is well above the level of this class.

Problem 20 Some pitfalls in numerical ODE solutions (6 points)

(a) The fireball at the end of a match grows as oxygen is obtained from the surface and consumed within the volume. A simple model of the growth of the radius of the fireball is given by

$$\frac{dr}{dt} = c_2 r^2 - c_3 r^3$$

where c_i are positive constants. The fire would grow in the beginning, and then asymptotically reach a steady radius and stay in that shape. We will analyze this system following MATLAB's notes on stiff differential equations using Python. Our discussion is self-contained, but I suggest you read the link which has valuable information on the nature of stiffness.

Show that you can define scaled variables $\rho \equiv r/r_0$ and $\tau \equiv t/t_0$ so that the equation becomes

$$\frac{d\rho}{d\tau} = \rho^2 - \rho^3 \ ,$$

and has the exact solution

$$\rho(t) = \left[W(ae^{a-t}) + 1\right]^{-1} \ , \ a = \delta^{-1} - 1$$

for $\rho(0) = \delta$. W is our old friend *Lambert W function*.

[4] There is a separate module, PyDSTool, which still does not have symplectic integrators, but it has more varied tools for simulating dynamical systems. It may be useful for your research, especially on the biology front.

(b) Solve the ODE numerically for $\rho(0) = \delta$ and $0 < t < (2/\delta)$ using the builtin `scipy.integrate.solve_ivp`[5]. Try absolute error tolerances between 10^{-4} and 10^{-6}, which can be set by the optional `atol` parameter. This function uses a high order Runge-Kutta method by default, but has several other options. The documentation says:
You should use the 'RK45' or 'RK23' method for non-stiff problems and 'Radau' or 'BDF' for stiff problems [9]. If not sure, first try to run 'RK45'. If needs unusually many iterations, diverges, or fails, your problem is likely to be stiff and you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

Try RK45, BDF and LSODA for $\delta = 2^{-8}$. For stiff solvers, run the function both with and without the Jacobian of the right-hand-side of the ODE using the optional input argument `jac`. Explain the advantages or disadvantages of the methods, and proceed to $\delta = 2^{-12}, 2^{-16}$ with the method you deem to be ideal.

For each run, plot your results together with the analytical solution. `scipy.integrate.solve_ivp` also gives you a report about how many times it had to call the right-hand-side etc. Your decision about the best method to use should depend on such output and the speed of the code as well as the agreement of the result with the known analytical solution.

(c) Consider a particle of mass m moving in the perturbed harmonic potential[6]

$$V(x) = \frac{1}{2}m\omega^2 x^2 + m\omega^2 A e^{-x^2/2\ell^2} \ .$$

The natural time scale of the problem is ω^{-1}, so it is wise to switch to the variable $\tau \equiv \omega t$ which leads to the equation of motion

$$\frac{d^2x}{d\tau^2} = -x + \frac{A}{\ell^2} x e^{-x^2/2\ell^2}$$

Set $A = 1$ and $x(\tau = 0) = 1$ in relevant units. For $\ell = 2^{-8}$ plot $V(x)$, and use `time_step` from Problem 20 with RK4 to evolve the particle in time with a fixed step size of your choosing, You should note that the particle should be confined to $x > 0$, make sure this is the case for your choice of dt. Now change ℓ to successively smaller values while keeping dt constant until the confinement condition is not satisfied. Explain the result. Repeat the problem using *solve_ivp* with default parameters. At what ℓ value does the function fail (if at all)? Adaptive time stepping is indispensable for modern ODE solvers.

This problem aimed to demonstrate to you some cases where your go-to method might fail, possibly without an obvious hint. Note that the differential equation in the first problem does not have any obviously large parameters which are usually associated with stiffness. However, quoting the above MATLAB link: *"An ordinary differential equation problem is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results."* In the second problem, if your fixed step size is larger than the smallest length scale in the problem, you will miss all phenomena occurring on such scales, or you may observe interesting events such as numerical tunneling through a potential barrier. Because of many cases like these, it is essential to test the output of your numerical results for convergence, and compare them to analytical knowledge if there is any.

[5]This is a relatively new function in SciPy, and it is recommended over the old `scipy.integrate.odeint` by the official documentation. However, the same documentation also says that the older ones might be faster in some cases, so they are worth using. The main upgrade is having a better interface rather than superior numerics.

[6]I thank Özgür Oktel for suggesting this problem.