Problem 1 (a) For arbitrary real s find the exact solution of the initial value problem

$$y'(t) = \frac{1}{2}\left(y(t) + y(t)^3\right)$$

with $y(0) = s > 0$.

(b) Show that the solution blows up when $t = \log(1 + 1/s^2)$.

Solution 1 (a)

Using separation of variables we solve

$$\begin{aligned}
\frac{1}{2}\int dt &= \int \frac{dy}{y + y^3} \\
\iff \frac{t}{2} &= \int \frac{\left(1+y^2\right) - y^2}{y\left(1+y^2\right)}\, dy = \int \frac{1}{y} - \frac{1}{2}\frac{2y}{1+y^2}\, dy \quad \text{(partial fractions)} \\
&= \log(y) - \frac{1}{2}\log\left(1+y^2\right) + C_1 \\
\iff t &= \log\left(y^2\right) - \log\left(1+y^2\right) + C_2 \quad \left(C_2 = 2C_1 \text{ and } 2\log y = \log\left(y^2\right)\right) \\
\iff e^t &= C_3\frac{y^2}{y^2+1} \quad \left(\text{passing both sides to } \exp \text{ and } C_3 = e^{C_2}\right) \\
\iff 1 + \frac{1}{y^2} = \frac{y^2+1}{y^2} &= C_4 e^{-t} \quad \text{(taking reciprocals, with } C_4 = 1/C_3\text{)} \\
\iff y^2 &= \frac{1}{C_4 e^{-t} - 1}.
\end{aligned}$$

When $y(t) > 0$, then $y' = (y+y^3)/2 > 0$ hence y is increasing. Thus $y(0) = s > 0$ means that only the positive square root above is our solution. Solving for C_4 we have

$$s^2 = y(0)^2 = \frac{1}{C_4 e^0 - 1} \implies C_4 = 1 + \frac{1}{s^2}$$

hence

$$\boxed{y(t) = \sqrt{\frac{1}{\left(1+\frac{1}{s^2}\right)e^{-t} - 1}}}.$$

(b)

As observed above, when $y(0)$ is positive, $y(t)$ is strictly increasing, so it makes qualitative sense that the solution would grow towards $+\infty$. We can find the point when this happens by setting the denominator inside the radical equal to zero:

$$\left(1 + \frac{1}{s^2}\right)e^{-t} - 1 = 0 \iff e^t = 1 + \frac{1}{s^2} \iff \boxed{t^* = \log\left(1 + \frac{1}{s^2}\right)}.$$

As t goes from 0 to t^*, $\left(1 + \frac{1}{s^2}\right)e^{-t}$ decreases from $1 + \frac{1}{s^2}$ down to 1 hence $y(t)$ increases from s up to $+\infty$.

Problem 2 (a) Find the general solution of the difference equation

$$u_{j+2} = u_{j+1} + u_j.$$

(b) Find all initial values u_0 and u_1 such that u_j remains bounded by a constant as $j \to \infty$.

Solution 2 (a)

Defining the 2D vector

$$v_j = \begin{bmatrix} u_{j+1} \\ u_j \end{bmatrix}$$

we get the order one vector-difference equation

$$v_{j+1} = \begin{bmatrix} u_{j+2} \\ u_{j+1} \end{bmatrix} = \begin{bmatrix} u_{j+1} + u_j \\ u_{j+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} v_j = Av_j.$$

This matrix has characteristic polynomial

$$\chi_A(t) = \det \begin{bmatrix} 1-t & 1 \\ 1 & -t \end{bmatrix} = (1-t)(-t) - 1 = t^2 - t - 1.$$

This has roots $t = \frac{1 \pm \sqrt{5}}{2}$. Defining $\varphi = \frac{1+\sqrt{5}}{2}$ (the so-called "golden ratio") as one of the roots, the roots are φ and $1-\varphi$. Since the roots are distinct, A is diagonalizable hence we can write

$$A = V \begin{bmatrix} \varphi & 0 \\ 0 & 1-\varphi \end{bmatrix} V^{-1}.$$

This in turns allows us to compute

$$A^n = V \begin{bmatrix} \varphi & 0 \\ 0 & 1-\varphi \end{bmatrix}^n V^{-1} = V \begin{bmatrix} \varphi^n & 0 \\ 0 & (1-\varphi)^n \end{bmatrix} V^{-1}.$$

Using this

$$v_n = Av_{n-1} = A^2 v_{n-2} = \cdots = A^n v_0 \Longrightarrow \begin{bmatrix} u_{n+1} \\ u_n \end{bmatrix} = V\Lambda^n V^{-1} \begin{bmatrix} u_1 \\ u_0 \end{bmatrix}.$$

Since V^{-1} and V just produce linear combinations of the entries, this means that

$$u_n = C\varphi^n + D\,(1-\varphi)^n$$

for some C, D which are linear combinations of u_0, u_1.

To determine their values, plug in $n = 0, 1$ to see

$$\begin{aligned} u_0 &= C + D \\ u_1 &= C\varphi + D\,(1-\varphi) \\ \Longrightarrow u_1 - \varphi u_0 &= D\,(1-2\varphi) = -\sqrt{5}D \\ \Longrightarrow u_1 - (1-\varphi)\,u_0 &= C\,(2\varphi - 1) = \sqrt{5}C \end{aligned}$$

Putting it all together this means

$$\boxed{u_n = \frac{u_1 - (1-\varphi)\, u_0}{\sqrt{5}} \varphi^n + \frac{u_1 - \varphi u_0}{-\sqrt{5}} (1-\varphi)^n}.$$

NOTE: One could similarly find the roots φ and $1-\varphi$ by considering the auxiliary equation for the recurrence:

$$u_n = t^n \Longrightarrow t^{n+2} = t^{n+1} + t^n \Longrightarrow t^2 = t + 1.$$

(b)

Since

$$\varphi \approx 1.618033988749895, \quad 1 - \varphi \approx -0.618033988749895$$

we know

$$\lim_{n\to\infty} \varphi^n = \infty, \quad \lim_{n\to\infty} (1-\varphi)^n = 0$$

hence u_n will only remain bounded if the φ^n term vanishes, which requires

$$0 = \frac{u_1 - (1-\varphi)\, u_0}{\sqrt{5}} \Longleftrightarrow \boxed{u_1 = (1-\varphi)\, u_0}.$$

This gives a line in $u_0 u_1$ space and for these points, the other term becomes

$$\frac{u_1 - \varphi u_0}{-\sqrt{5}} = \frac{(1-2\varphi)\, u_0}{-\sqrt{5}} = u_0 \Longrightarrow u_n = 0 \cdot \varphi^n + u_0 \cdot (1-\varphi)^n.$$

Problem 3 (a) Write, test and debug a matlab function

```
function u = euler(a, b, ya, f, r, n)
% a,b: interval endpoints with a < b
% n: number of steps with h = (b-a)/n
% ya: vector y(a) of initial conditions
% f: function handle f(t, y, r) to integrate
% r: parameters to f
% u: output approximation to the final solution vector y(b)
```

which approximates the final solution vector $y(b)$ of the vector initial value problem

$$y' = f(t, y, r)$$

$$y(a) = y_a$$

by the numerical solution vector u_n of Euler's method

$$u_{j+1} = u_j + hf(t_j, u_j, r) \qquad j = 0, 1, \ldots, n-1$$

with $h = (b-a)/n$ and $u_0 = y_a$.

(b) Use `euler.m` to approximate the solution $z(T)$ at $T = 4\pi$ of the initial value problem

$$z' = \begin{bmatrix} x \\ y \\ u \\ v \end{bmatrix}' = f(t, z) = \begin{bmatrix} u \\ v \\ -x/(x^2+y^2) \\ -y/(x^2+y^2) \end{bmatrix}$$

with initial conditions

$$z = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

at $t = 0$ which cause the solution to move in a unit circle forever. Measure the maximum error

$$E_N = \max(|x_N - \cos t_N|, |y_N - \sin t_N|, |u_N + \sin t_N|, |v_N - \cos t_N|)$$

after 2 revolutions ($T = 4\pi$) with time steps $h = T/N$ for $N = 1000$, 2000, $\ldots$, 16000. Estimate the constant C such that the error behaves like Ch. Measure the CPU time for each run and estimate the total CPU time necessary to obtain the solution to three–digit, six–digit and twelve–digit accuracy. Plot the solutions.

(c) Use `euler.m` with $s = [512, 64, 8, 1]$ and $N = [10^3, 10^4, 10^5, 10^6]$ to verify conclusion (b) of problem 1.

Solution 3 (a)

The code `euler.m` is embedded in this pdf file, whose output has an extra `uHist` holding the entire history of u.

(b)

For this problem, we want to approximate two fit lines, one for error (dependent on step size) and one for CPU time (dependent on number of steps).

In general, to fit the data $(x_i, y_i)_i$ on the line

$$y_i = mx_i$$

we seek to minimize $g(m) = \sum_i (y_i - mx_i)^2$. This occurs when

$$0 = g'(m) = \sum_i 2(y_i - mx_i)(-x_i) \Longrightarrow m = \frac{\sum_i x_i y_i}{\sum_i x_i^2}.$$

So the fit lines

$$E = Ch, \qquad T = SN$$

have slopes

$$C = \frac{\sum_i h_i E_i}{\sum_i h_i^2}, \qquad S = \frac{\sum_i N_i T_i}{\sum_i N_i^2}$$

(here E is error, h is stepsize, T is CPU time and N is the number of steps).

Using `ps08ErrorPlot.m` we create a log-log plot of errors in figure 1. With $N = 1k, 2k, \ldots 16k$, we get $C \approx 75$. Tracking the CPU time for each run we plot figure 2, which gives $\boxed{S \approx 1.2 \cdot 10^{-4}}$ (this will vary a great deal depending on your computer).

Putting these all together, we'd like to be able to predict T for a given desired d-digit error $E = 10^d$. Since $h = \frac{4\pi}{N}$, we've got

$$10^{-d} = E \approx C\frac{4\pi}{N} \Longrightarrow N \approx 4\pi C \cdot 10^d \Longrightarrow T \approx SN \approx 4\pi SC \cdot 10^d.$$

Thus for three-digit, six-digit and twelve-digit accuracy we've got

$$\begin{aligned} \text{three-digit} &\Longrightarrow \boxed{1.13 \cdot 10^2 \text{ seconds}} \\ \text{six-digit} &\Longrightarrow \boxed{1.13 \cdot 10^5 \text{ seconds}} \\ \text{twelve-digit} &\Longrightarrow \boxed{1.13 \cdot 10^{11} \text{ seconds}} \end{aligned}$$

$\boxed{\text{What is the significance of this?}}$ When the error for the method is "linear" (i.e. $\mathcal{O}(h)$), every extra decimal digit costs 10 times as much CPU time.

The phase diagrams of $x(t) \approx x_n$ (first component of z) against y_n (second component) for $N = 1k, 2k, 4k, 8k, 16k$ is generated by `unitCircle.m`, and shown below:

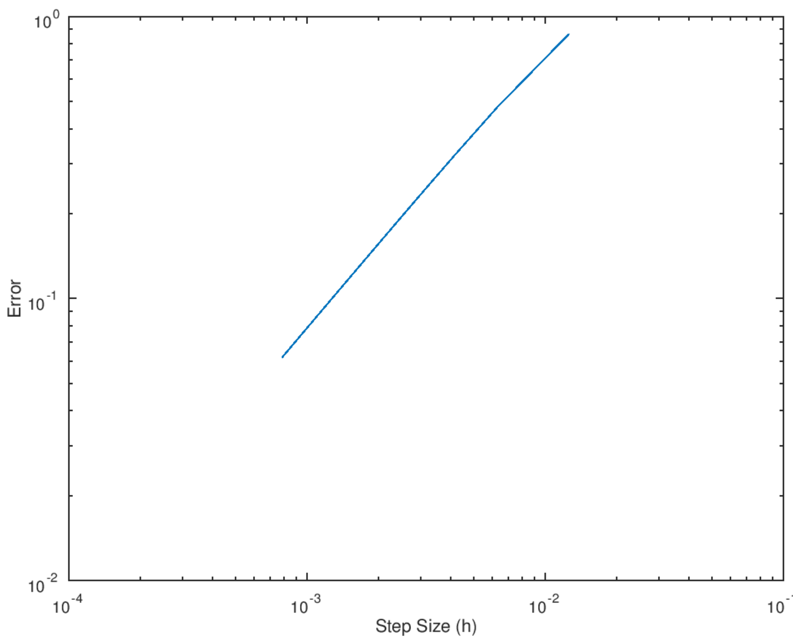


Figure 1: Error against step size.

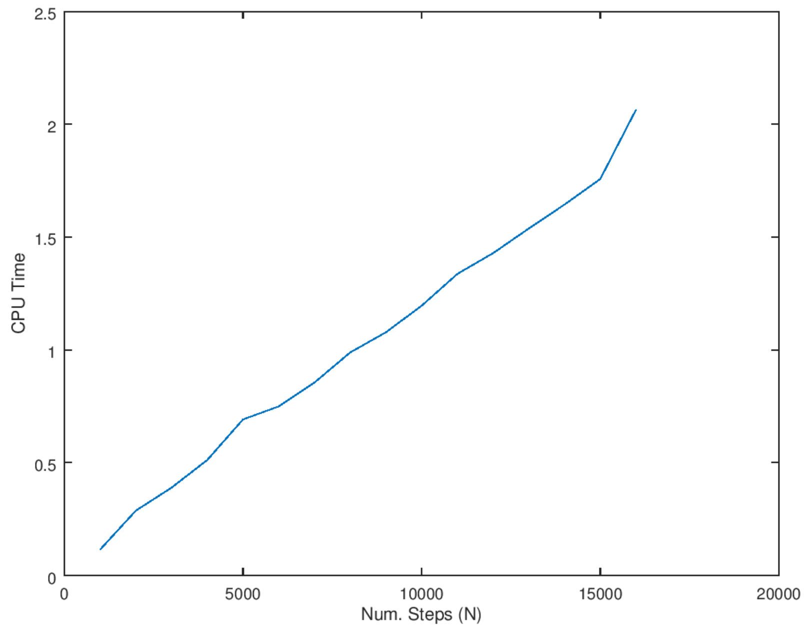
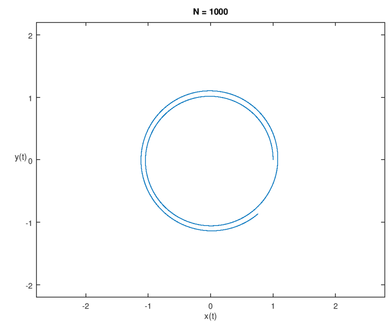
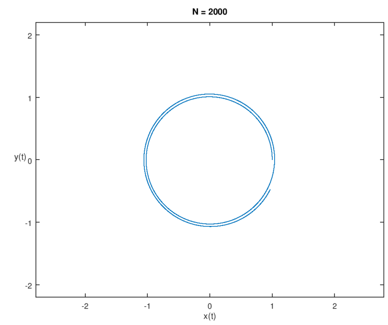
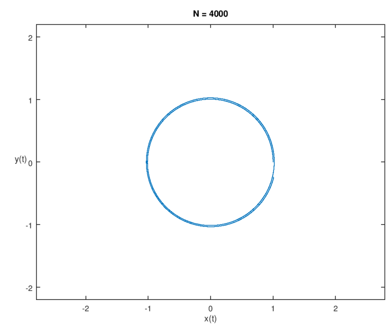
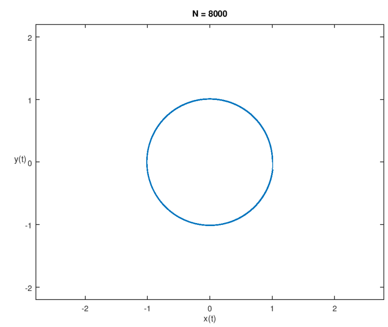


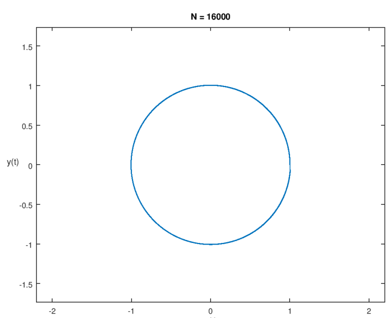
Figure 2: CPU time against number of steps.





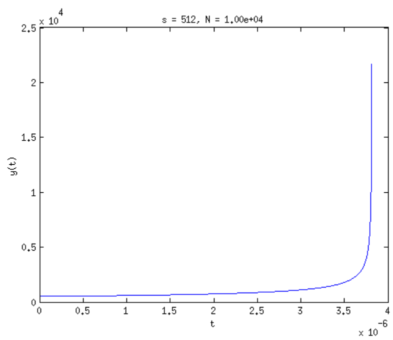


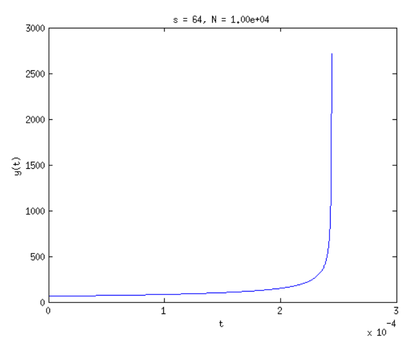


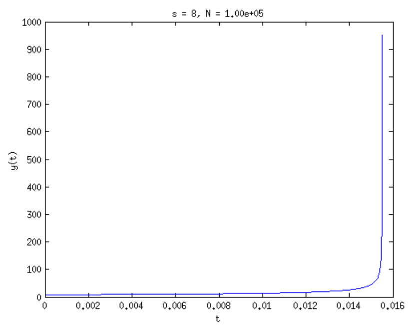


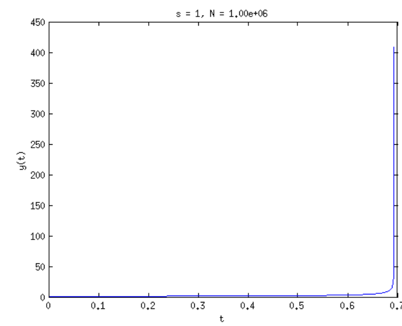
(c)

In order to track the solution as it approaches the singularity (point where it blows up), we implement `makeIVPBlowUpPlots.m`, which further calls `ivpBlowUp.m`, and allows us to specify the starting value s and the number of steps N to take. Running this we produce the following plots:


s = 512, N = 1.00e+04
y(t)
t


s = 64, N = 1.00e+04
y(t)
t


s = 8, N = 1.00e+05
y(t)
t


s = 1, N = 1.00e+06
y(t)
t

Problem 4 (See GGK 10.1) The position $(x(t), y(t))$ of a satellite orbiting around the earth and moon is described by the *second-order* system of ordinary differential equations

$$x'' = x + 2y' - b\frac{x+a}{((x+a)^2+y^2)^{3/2}} - a\frac{x-b}{((x-b)^2+y^2)^{3/2}}$$

$$y'' = y - 2x' - b\frac{y}{((x+a)^2+y^2)^{3/2}} - a\frac{y}{((x-b)^2+y^2)^{3/2}}$$

where $a = 0.012277471$ and $b = 1 - a$. When the initial conditions

$$x(0) = 0.994$$

$$x'(0) = 0$$

$$y(0) = 0$$

$$y'(0) = -2.00158510637908$$

are satisfied, there is a periodic orbit with period $T = 17.06521656015796$.

(a) Convert this problem to a 4×4 *first-order* system $u' = f(t, u, r)$, $u(0) = u_0$, by introducing

$$u = [x, x', y, y'] = [u_1, u_2, u_3, u_4]$$

as a new vector unknown function and defining f appropriately.

(b) Use `euler.m` to approximate $u(T)$ and plot the error vs. N for $N = 1000$, 2000, ..., 1024000 steps. Measure the CPU time for each run and estimate the total CPU time necessary to obtain an orbit which is periodic to three–digit, six–digit and twelve–digit accuracy.

Solution 4 (a)

We have

$$\boxed{u_1'} = f_1(t,u) = x' = \boxed{u_2}$$

$$\boxed{u_3'} = f_3(t,u) = y' = \boxed{u_4}$$

$$\begin{aligned}\boxed{u_2'} &= f_2(t,u) = x'' \\ &= \boxed{u_1 + 2u_4 - b\frac{u_1+a}{\left((u_1+a)^2+u_3^2\right)^{3/2}} - a\frac{u_1-b}{\left((u_1-b)^2+u_3^2\right)^{3/2}}}\end{aligned}$$

$$\begin{aligned}\boxed{u_4'} &= f_4(t,u) = y'' \\ &= \boxed{u_3 - 2u_2 - b\frac{u_3}{\left((u_1+a)^2+u_3^2\right)^{3/2}} - a\frac{u_3}{\left((u_1-b)^2+u_3^2\right)^{3/2}}}\end{aligned}$$

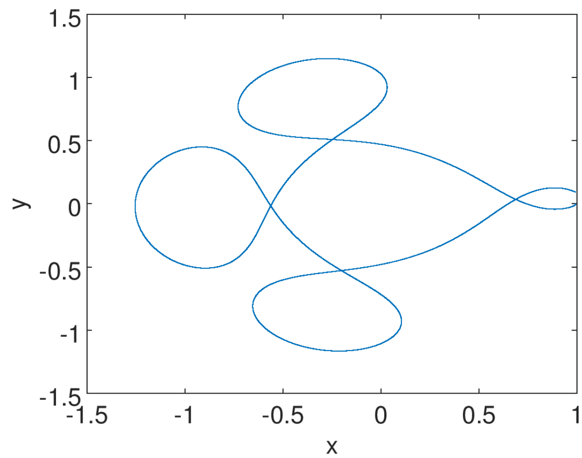


Figure 3: A sample trajectory, computed with $N = 1024000$.

(b)

We use the embedded `satellite.m` to compute trajectories and errors as shown below. A sample trajectory in shown in figure 3. The log-log plot of error with respect to N is in figure 4.

By the error formula of Euler's method,

$$e(T) \leq \frac{hM}{2L}(e^{LT} - 1)\,,$$

where h is the step size, M is the bound for second-order derivative of f, L is the Lipshitz constant, T is the total time length. Our problem statement asks us to change only h, our error should shrink proportionally with $h = t/N$. This is verified in figure 4. For $N = 1024000$, we use 240 cpu seconds to obtain error of 0.0925. Since cpu time is proportional to N, we have below estimation:

error	N	cpu time
9.25×10^{-2}	1.02×10^{6}	240
10^{-3}	9.4×10^{7}	2.2×10^{4}
10^{-6}	9.4×10^{10}	2.2×10^{7}
10^{-12}	9.4×10^{16}	2.2×10^{13}

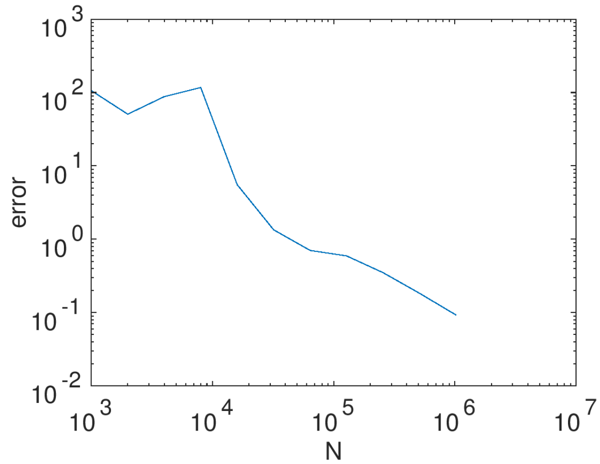


Figure 4: Error v.s. N.

Problem 5 Suppose $y(t)$ is the exact solution of the initial value problem

$$y'(t) = f(t, y(t)),$$

$$y(0) = y_0,$$

and $u(t)$ is any approximation to $y(t)$ with $u(0) = y(0)$. Define the error $e(t) = y(t) - u(t)$.

(a) Show that $e(t)$ satisfies the initial value problem

$$e'(t) = f(t, u(t) + e(t)) - u'(t)$$

$$e(0) = 0$$

(b) Suppose $f(t, y) = \lambda y$ for some constant λ. Solve the initial value problem from (a) exactly to show that $u(t) + e(t) = y(t)$.

Solution 5 (a)

Since the error is defined to be $e(t) = y(t) - u(t)$, taking derivatives of both sides gives us:

$$\begin{aligned} e'(t) &= y'(t) - u'(t) \\ &= f(t, y(t)) - u'(t) \\ &= f(t, u(t) + e(t)) - u'(t) \end{aligned}$$

And clearly, $e(0) = y(0) - u(0) = 0$.

(b)

If $f(t,y) = \lambda y$, the IVP becomes

$$e'(t) = \lambda(u(t) + e(t)) - u'(t)$$

To show that $u(t) + e(t) = y(t)$, we first note that $u(0) + e(0) = u(0) = y(0)$, and furthermore:

$$\begin{aligned} u'(t) + e'(t) - y'(t) &= \cancel{u'(t)} + \lambda(u(t) + e(t)) - \cancel{u'(t)} - \lambda y(t) \\ &= \lambda(u(t) + e(t) - y(t)) \end{aligned} \tag{1}$$

Therefore the function $z(t) = u(t) + e(t) - y(t)$ satisfies the equation:

$$z'(t) = \lambda z(t), \quad z(0) = 0$$

Which we can solve as follows:

$$z'(t)/z(t) = \lambda \implies \ln(z(t)) = \lambda t + c \implies z(t) = ke^{\lambda t}$$

Enforcing the initial condition $z(0) = 0$, we get $k = 0$. Therefore $z(t) = 0 = u(t) + e(t) - y(t)$ for all t. So we can conclude $u(t) + e(t) = y(t)$.

Problem 6 Define a family of explicit Runge-Kutta methods parametrized by order p, by applying $p-1$ passes of deferred correction to p steps of Euler's method. I.e. starting from u_n define the uncorrected solution by

$$u^1_{n+j+1} = u^1_{n+j} + hf(t_{n+j}, u^1_{n+j})$$

for $0 \le j \le p-1$. Let $u(t) = U_1(t)$ be the degree-p polynomial that interpolates the $p+1$ values u^1_{n+j} at the $p+1$ points $t = t_{n+j}$ for $0 \le j \le p$. Solve the error equation from question 5 by Euler's method, yielding approximate errors e^1_{n+1}, e^1_{n+2}, ..., e^1_{n+p}. Produce a second-order accurate corrected solution

$$u^2_{n+j} = u^1_{n+j} + e^1_{n+j}$$

for $1 \le j \le p$. Repeat the procedure to produce u^2_{n+j}, ..., u^p_{n+j}.

(a) Verify that $p = 1$ gives Euler's method.

(b) For $p = 2$ express your method as a Runge-Kutta method in the form

$$k_1 = f(t_n, u_n)$$

$$\begin{aligned} k_2 &= f(t_n + c_2 2h, u_n + 2ha_{21}k_1) \\ k_3 &= f(t_n + c_3 2h, u_n + 2h(a_{31}k_1 + a_{32}k_2)) \\ u_{n+2} &= u_n + 2h(b_1k_1 + b_2k_2 + b_3k_3). \end{aligned}$$

Find all the constants c_i, a_{ij} and b_j and arrange them in a Butcher array.

(c) For $p = 2$, ignore the t argument of $f(t, u)$ and Taylor expand $k_2(h)$ and $k_3(h)$ to $O(h^2)$. Show that your method has local truncation error $\tau = O(h^2)$ and find the coefficient of the $O(h^2)$ term.

(d) For arbitrary p, verify that your method is equivalent to using fixed point iteration to solve an implicit Runge-Kutta method.

Solution 6 (a)

When $p = 1$, we do not apply any deferred correction steps. Therefore, we simply have

$$u_{n+j+1} = u_{n+j} + hf(t_{n+j}, h_{n+j})$$

And that is Euler's Method.

(b)

When $p = 2$, we first apply Euler's method twice to get:

$$\begin{aligned} u^1_{n+1} &= u_n + hf(t_n, u_n) \\ u^1_{n+2} &= u^1_{n+1} + hf(t_{n+1}, u^1_{n+1}) \end{aligned}$$

Next we apply deferred correction. As in problem 3, we know that

$$\begin{aligned} e'(t) &= f(t, u(t) + e(t)) - u'(t) \\ e(t_n) &= 0 \end{aligned}$$

so applying Euler's method:

$$\begin{aligned} e^1_{n+1} &= e(t_n) + h(f(t_n, e(t_n) + u(t_n)) - u'(t_n)) \\ &= hf(t_n, u_n) - hu'(t_n) \end{aligned}$$

Next, we estimate $u'(t_n)$ via Lagrange interpolation. So, let $U_1(t)$ be the Lagrange polynomial going through $(t_n, u_n), (t_{n+1}, u^1_{n+1}), (t_{n+2}, u^1_{n+2})$, then:

$$U_1(t) = \frac{(t-t_n)(t-t_{n+1})}{(t_{n+2}-t_n)(t_{n+2}-t_{n+1})}u^1_{n+2} + \frac{(t-t_n)(t-t_{n+2})}{(t_{n+1}-t_n)(t_{n+1}-t_{n+2})}u^1_{n+1} + \frac{(t-t_{n+1})(t-t_{n+2})}{(t_n-t_{n+1})(t_n-t_{n+2})}u_n$$

$$U_1'(t) = \frac{(t-t_n)+(t-t_{n+1})}{(t_{n+2}-t_n)(t_{n+2}-t_{n+1})}u^1_{n+2} + \frac{(t-t_n)+(t-t_{n+2})}{(t_{n+1}-t_n)(t_{n+1}-t_{n+2})}u^1_{n+1} + \frac{(t-t_{n+1})+(t-t_{n+2})}{(t_n-t_{n+1})(t_n-t_{n+2})}u_n$$

So,

$$U_1'(t_n) = -\frac{1}{2h}u^1_{n+2} + \frac{2}{h}u^1_{n+1} - \frac{3}{2h}u_n$$

$$\boxed{e^1_{n+1} = hf(t_n, u_n) + \frac{1}{2}(u^1_{n+2} - 4u^1_{n+1} + 3u_n)}$$

Similarly, applying the next Euler's step:

$$e^1_{n+2} = e^1_{n+1} + h(f(t_{n+1}, e^1_{n+1} + u^1_{n+1}) - u'(t_{n+1}))$$

$$\boxed{e^1_{n+2} = e^1_{n+1} + h(f(t_{n+1}, e^1_{n+1} + u^1_{n+1}))) + \frac{u_n - u^1_{n+2}}{2}}$$

Now, we update our new estimates for u_{n+2}:

$$u^2_{n+2} = u^1_{n+2} + e^1_{n+2}$$

So, let's simplify what we have done. Let

$$\begin{aligned} k_1 &= f(t_n, u_n) \\ k_2 &= f(t_{n+1}, u^1_{n+1}) \\ k_3 &= f(t_{n+1}, e^1_{n+1} + u^1_{n+1})) \end{aligned}$$

So

$$\begin{aligned}
u_{n+2}^2 &= u_{n+2}^1 + e_{n+2}^1 \\
&= u_{n+2}^1 + e_{n+1}^1 + hk_3 + \frac{u_n - u_{n+2}^1}{2} \\
&= u_{n+2}^1 + hk_1 + \frac{1}{2}(u_{n+2}^1 - 4u_{n+1}^1 + 3u_n) + hk_3 + \frac{u_n - u_{n+2}^1}{2} \\
&= u_{n+2}^1 - 2u_{n+1}^1 + 2u_n + hk_1 + hk_3 \\
&= hk_2 - u_{n+1}^1 + 2u_n + hk_1 + hk_3 \\
&= u_n - hk_1 + hk_2 + hk_1 + hk_3 \\
&= u_n + hk_2 + hk_3
\end{aligned}$$

Furthermore, we see that

$$\begin{aligned}
k_2 &= f(t_{n+1}, u_{n+1}^1) \\
&= f(t_n + h, u_n + hk_1) \\
k_3 &= f(t_{n+1}, e_{n+1}^1 + u(t_{n+1})) \\
&= f(t_n + h, hk_1 + \frac{1}{2}(u_{n+2}^1 - 4u_{n+1}^1 + 3u_n) + u_{n+1}^1) \\
&= f(t_n + h, hk_1 + \frac{1}{2}(u_{n+1}^1 + hk_2 - 2u_{n+1}^1 + 3u_n)) \\
&= f(t_n + h, u_n + \frac{1}{2}(hk_2 + hk_1))
\end{aligned}$$

Therefore, we conclude that $c_2 = a_{21} = c_3 = b_2 = b_3 = \frac{1}{2}$, $a_{31} = a_{32} = \frac{1}{4}$, and $b_1 = 0$.

$$\begin{array}{c|ccc}
0 & 0 & & \\
\frac{1}{2} & \frac{1}{2} & 0 & \\
\frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\
\hline
 & 0 & \frac{1}{2} & \frac{1}{2}
\end{array}$$

(c)

Let us Taylor expand $k_2(h)$ and $k_3(h)$ to get (ignoring the t argument):

$$\begin{aligned}
k_2(h) &= f(u_n + hk_1) \\
k_3(h) &= f(u_n + \frac{1}{2}(hk_2 + hk_1)) \\
k_2(h) &= k_2(0) + hk_2'(0) + \frac{h^2}{2}k_2''(0) + O(h^3) \\
k_3(h) &= k_3(0) + hk_3'(0) + \frac{h^2}{2}k_3''(0) + O(h^3)
\end{aligned}$$

Now systematically compute all these derivatives (similar to how we did RK order conditions):

$$\begin{aligned}
k_2(h) &= f(u_n + hk_1) \\
k_2(0) &= f(u_n) = k_1 \\
k_2'(h) &= k_1 f'(u_n + hk_1) \\
k_2'(0) &= k_1 f'(u_n) = ff' \\
k_2''(h) &= k_1^2 f''(u_n + hk_1) \\
k_2''(0) &= k_1^2 f''(u_n) = f^2 f''
\end{aligned}$$

Put it all together:

$$k_2(h) = f + hff' + \frac{h^2}{2} f^2 f'' + O(h^3)$$

Similarly for k_3:

$$\begin{aligned}
k_3(h) &= f(u_n + \frac{1}{2}(hk_2 + hk_1)) \\
k_3(0) &= f(u_n) = k_1 \\
k_3'(h) &= \frac{1}{2}(k_2 + hk_2' + k_1) f'(u_n + \frac{1}{2}(hk_2 + hk_1)) \\
k_3'(0) &= \frac{1}{2}(k_2(0) + k_1) f'(u_n) \\
&= k_1 f'(u_n) = ff' \\
k_3''(h) &= \frac{1}{2}(k_2' + hk_2'' + k_2') f'(u_n + \frac{1}{2}(hk_2 + hk_1)) + \frac{1}{4}(k_2 + hk_2' + k_1)^2 f''(u_n + \frac{1}{2}(hk_2 + hk_1)) \\
k_3''(0) &= k_2'(0) f'(u_n) + \frac{1}{4}(k_2(0) + k_1)^2 f''(u_n) \\
&= k_1 (f'(u_n))^2 + \frac{1}{4}(2k_1)^2 f''(u_n) \\
&= ff'^2 + f^2 f''
\end{aligned}$$

Put it all together:

$$k_3(h) = f + hff' + \frac{h^2}{2}(f(f')^2 + f^2 f'') + O(h^3)$$

Now the truncation error is:

$$
\begin{aligned}
\tau &= \frac{y_{n+2} - y_n}{2h} - b_1 k_1 - b_2 k_2 - b_3 k_3 \\
&= \frac{\cancel{y_n} + (2h)y_n' + \frac{(2h)^2}{2!}y_n'' + \frac{(2h)^3}{3!}y_n''' + O(h^4) - \cancel{y_n}}{2h} - \frac{1}{2}k_2 - \frac{1}{2}k_3 \\
&= f + hf'f + \frac{2}{3}h^2 y_n''' + O(h^3) - \frac{1}{2}\underbrace{\left(f + hff' + \frac{h^2}{2}f^2 f'' + O(h^3)\right)}_{k_2} + \dots \\
&\quad \dots - \frac{1}{2}\underbrace{\left(f + hff' + \frac{h^2}{2}(f(f')^2 + f^2 f'') + O(h^3)\right)}_{k_3} \\
&= \cancel{f + hf'f} + \frac{2}{3}h^2(f(f')^2 + f^2 f'')\cancel{- f - hff'} - \frac{1}{2}h^2 f^2 f'' - \frac{1}{4}h^2 f(f')^2 + O(h^3) \\
&= \frac{5}{12}h^2 f(f')^2 + \frac{1}{6}h^2 f^2 f'' \\
&= O(h^2)
\end{aligned}
$$

(d)

(See IDEC Handout - Fixed point equivalent)
Since u^2 is built from u^1 by:

$$
\begin{aligned}
e_{n+j+1} &= e_{n+j} + h[f(t_{n+j}, u^1_{n+j} + e_{n+j}) - U'(t_{n+j})] \\
u^2_{n+j} &= u^1_{n+j} + e_{n+j}
\end{aligned}
$$

deferred correction is a fixed point iteration of the form

$$
\begin{bmatrix} u^2_{n+1} \\ u^2_{n+2} \\ \vdots \\ u^2_{n+p} \end{bmatrix} = \begin{bmatrix} u^1_{n+1} \\ u^1_{n+2} \\ \vdots \\ u^1_{n+p} \end{bmatrix} + \begin{bmatrix} e_{n+1} \\ e_{n+2} \\ \vdots \\ e_{n+p} \end{bmatrix} = G\left(\begin{bmatrix} u^1_{n+1} \\ u^1_{n+2} \\ \vdots \\ u^1_{n+p} \end{bmatrix}\right)
$$

or $U^2 = G(U^1)$.

In the limit where $U^k \to U$, U must satisfy $U = G(U)$, or

$$E = U^2 - U^1 = 0$$

Equivalently:

$$e_{n+j} \equiv 0$$

so that

$$0 = 0 + h[f(t_{n+j}, u_{n+j}) - U'(t_{n+j})]$$

and

$$\boxed{U'(t_{n+j}) = f(t_{n+j}, u_{n+j}) \qquad 1 \leq j \leq p}$$

Here $U(t)$ is the interpolating polynomial satisfying

$$U(t_{n+j}) = u_{n+j}$$

so that

$$U(t) = \sum_{j=0}^{p} L_j(t) u_{n+j}$$

and

$$U'(t_{n+j}) = \frac{1}{h} \sum_{k=0}^{p} d_{jk} u_{n+k}$$

for some dimensionless differentiation constants d_{jk}. Thus deferred correction is a fixed point iteration for solving

$$\frac{1}{h} \sum_{k=0}^{p} d_{jk} u_{n+k} = f(t_{n+j}, u_{n+j}) \qquad 1 \leq j \leq p \tag{2}$$

It remains to show that (1) is an implicit Runge-Kutta method with p stages

$$k_j = f(t_{n+j}, u_{n+j}) \qquad 1 \leq j \leq p$$

We have the below since differentiating a constant gives 0.

$$\sum_{k=0}^{p} d_{jk} = 0$$

Hence:

$$\sum_{k=0}^{p} d_{jk} u_n = 0$$

So combining with (1), u must satisfy:

$$\sum_{k=0}^{p} d_{jk}(u_{n+k} - u_n) = hf(t_{n+j}, u_{n+j}) = hk_j$$

since the $k = 0$ gives us $d_{j0}(u_n - u_n) = 0$, giving use a square system of equations to solve for $(u_{n+k} - u_n)$:

$$\sum_{k=1}^{p} d_{jk}(u_{n+k} - u_n) = hk_j$$

If we define c_{ij} to be the elements of the inverse matrix $C = D^{-1}$ to the square $p \times p$ matrix D with elements d_{ij} and apply to both sides, we can extract the updates:

$$u_{n+k} - u_n = h \sum_{j=1}^{p} c_{kj} k_j$$

and rewrite k_j as:

$$k_j = f(t_{n+j}, u_{n+j}) = f(t_{n+j}, u_n + ph \sum_{r=1}^{p} (c_{jr}/p) k_r)$$

So we identify this as a p-stage implicit Runge-Kutta method with stepsize ph.

Problem 7 Write, test and debug a matlab function

```
function yb = idec(a, b, ya, f, r, p, n)
% a,b: interval endpoints with a < b
% ya: vector y(a) of initial conditions
% f: function handle f(t, y) to integrate (y is a vector)
% r: parameters to f
% p: number of euler substeps / correction passes
% n: number of time steps
% yb: output approximation to the final solution vector y(b)
```

which approximates the final solution vector $y(b)$ of the vector initial value problem

$$y' = f(t, y, r)$$
$$y(a) = y_a$$

by the method you derived in problem 6, with $u_0 = y_a$.

(a) Use `idec.m` with orders $p = 1$ through 7 and $N = 10000$, 20000, 40000 and 80000 steps to approximate the final solution vector $u(T)$ of the initial value problem derived in problem 4. Tabulate the errors

$$E_{pN} = \max_{1 \le j \le 4} |u_j(T) - u_j(0)|.$$

Estimate the constant C_p such that the error behaves like $C_p h^p$.

(b) Measure the CPU time for each run and estimate the total CPU time necessary to obtain an orbit which is periodic to three–digit, six–digit and twelve–digit accuracy.

(c) Plot some inaccurate solutions and some accurate solutions and draw conclusions about values of the order p which give three, six or twelve digits of accuracy for minimal CPU time.

Solution 7 (a)

The solution `idec.m` is embedded. We store our system f in `moonOde.m`.

We provide the function `idecToTheMoon.m` which solves the satellite problem. This code takes `p` as a single argument, and finds the error with that `p` and $N = 10000, 20000, 40000, 80000$. Then it approximates the slopes in our fit lines.

Below is the error table ($E_{pN} = \max_{1 \le j \le 4} |u_j(T) - u_j(0)|$)

N \ p	1	2	3	4	5	6	7
10000	41.803	1.614	2.010	0.8409	8.455×10^{-2}	4.196×10^{-2}	6.268×10^{-4}
20000	2.761	1.507	1.426	0.1327	3.210×10^{-3}	5.600×10^{-4}	2.100×10^{-7}
40000	2.021	1.370	0.388	9.160×10^{-3}	9.800×10^{-5}	8.389×10^{-6}	9.444×10^{-8}
80000	1.740	1.026	4.319×10^{-1}	5.872×10^{-4}	2.911×10^{-6}	4.110×10^{-7}	1.981×10^{-7}

Below are the estimates of C_p such that the error behaves like $C_p h^p$.

p	1	2	3	4	5	6	7
C_p	2.772×10^{4}	1.430×10^{5}	3.180×10^{8}	9.620×10^{10}	5.830×10^{12}	1.700×10^{15}	1.487×10^{16}

Some comments:

- As we double N (equivalently halve h), we see that the corresponding error is not decreasing in some systematic way (particularly for the lower p). This suggests we are not really seeing asymptotic behavior so it's impossible to extrapolate from this data things like how long it takes to get three digit accuracy. This suggests that we should run more cases (increase N) until we start to see convergence.
- $p = 7$ seems to achieve the maximum error level ($\approx 10^{-7}$) almost immediately and doesn't improve. So this suggests we couldn't get something like 12 digit accuracy no matter what unless we use higher-precision arithmetic.

(b)

The CPU time for each run:

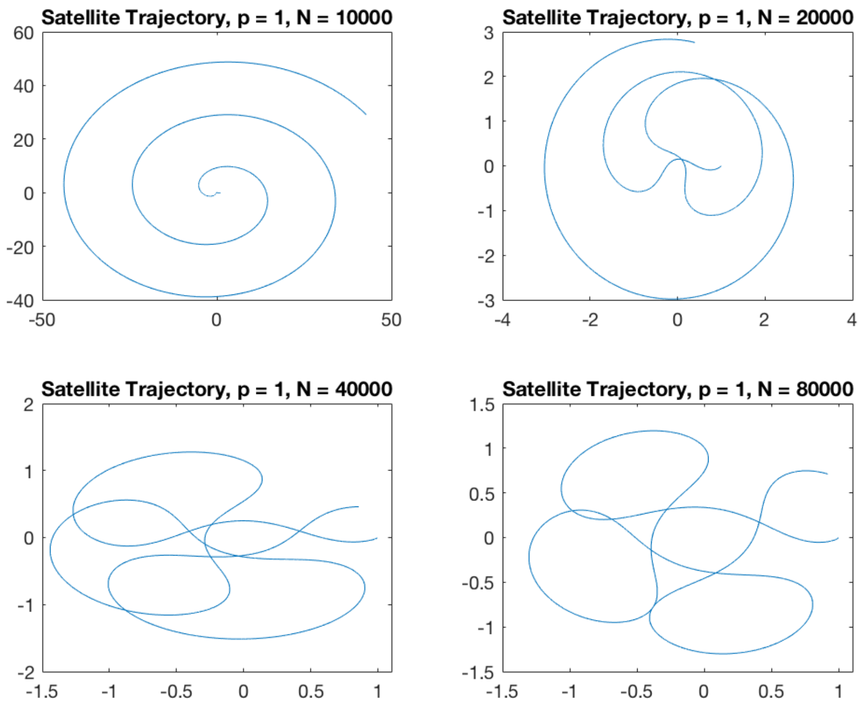
N \ p	1	2	3	4	5	6	7
10000	0.0271	0.1013	0.2261	0.4268	0.6902	1.0206	1.4188
20000	0.0483	0.2001	0.4711	0.8586	1.4402	2.0204	2.8860
40000	0.0968	0.4208	0.9438	1.7397	2.7893	4.1423	5.7881
80000	0.2092	0.8705	1.9132	3.3952	5.5645	8.2326	11.6935

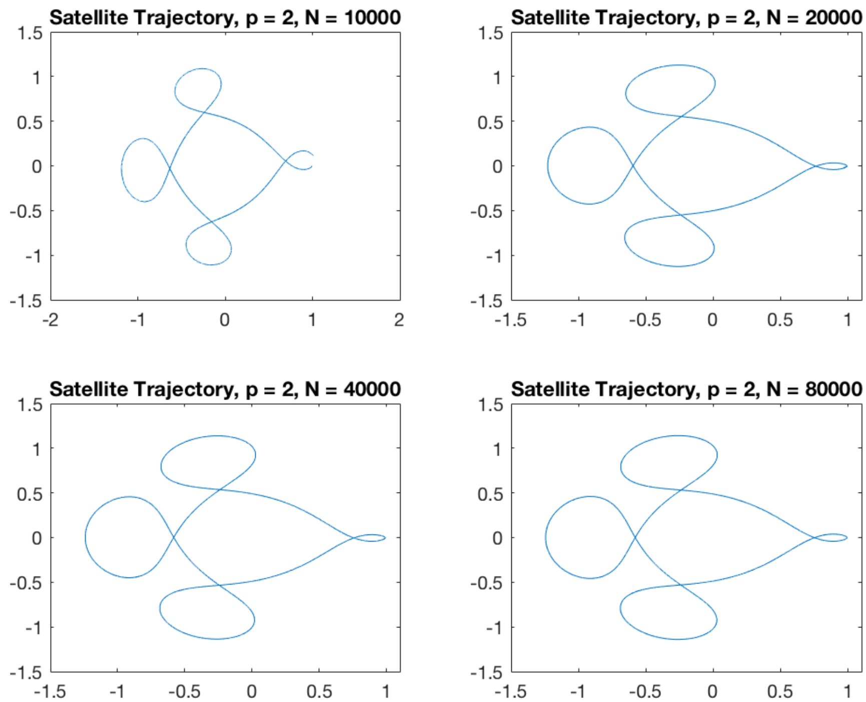
Estimates on CPU time to get specified accuracy (as noted above, these are naive estimates which couldn't actually be achieved):

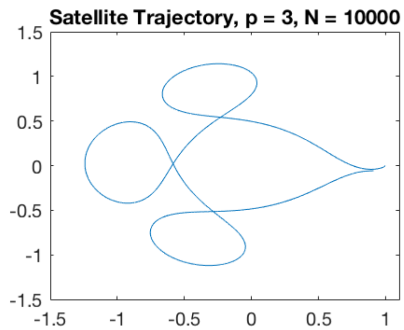
acc \ p	1	2	3	4	5	6	7
3-digit	2.621×10^{2}	14.63	6.0234	2.921	1.744	2.054	2.527
6-digit	2.621×10^{5}	4.626×10^{2}	6.0234×10^{1}	1.6431×10^{1}	6.945	6.496	6.780
12-digit	2.621×10^{11}	4.626×10^{5}	6.0234×10^{3}	5.1959×10^{2}	1.100×10^{2}	6.496×10^{1}	4.879×10^{1}

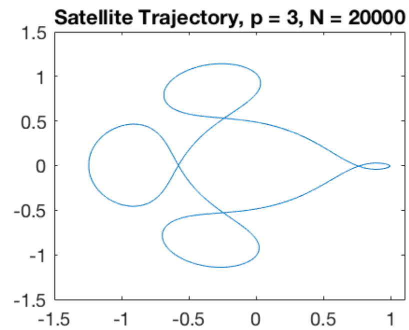
(c)

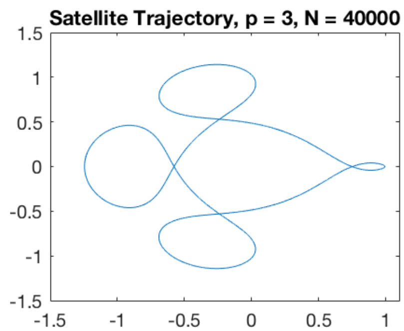
Some plots of the trajectories are provided below (for each p and N, $p > 4$ omitted since they're all really good).

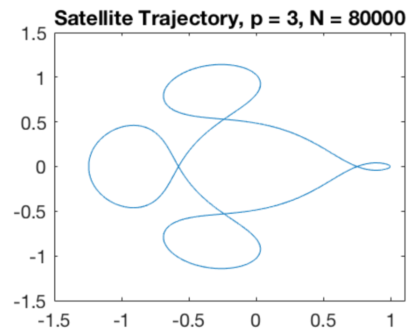

Satellite Trajectory, p = 1, N = 10000
Satellite Trajectory, p = 1, N = 20000
Satellite Trajectory, p = 1, N = 40000
Satellite Trajectory, p = 1, N = 80000

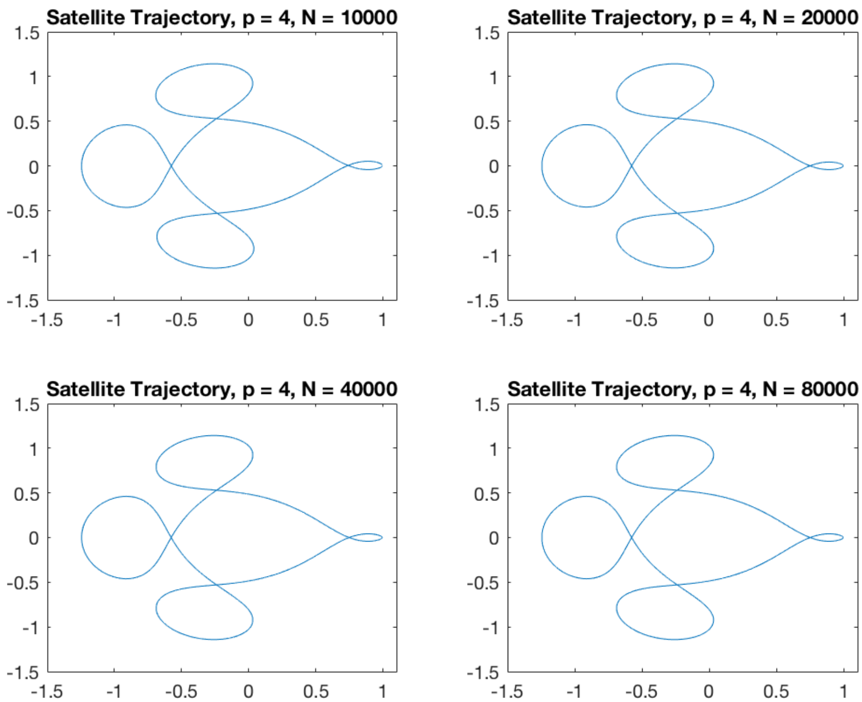

Satellite Trajectory, p = 2, N = 10000
Satellite Trajectory, p = 2, N = 20000
Satellite Trajectory, p = 2, N = 40000
Satellite Trajectory, p = 2, N = 80000


Satellite Trajectory, p = 3, N = 10000
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1


Satellite Trajectory, p = 3, N = 20000
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1


Satellite Trajectory, p = 3, N = 40000
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1


Satellite Trajectory, p = 3, N = 80000
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1



Looking at the data given in (b) - for three digit accuracy, $p = 5$ is fastest. For six digit accuracy, $p = 6$ is fastest. For twelve digit accuracy, $p = 7$ is fastest.

If you run the solution `idecToTheMoon(p)` for some given p, the resulting output will look like the following (for example $p = 2$):

```
>> idecToTheMoon(2)
p =
     2
errors =
   1.614391472108695
   1.507214472488082
   1.370484330041670
   1.026408928083552
errorslope =
     1.430081749578059e+05
durations =
   0.109044391000000
   0.214791086000000
   0.433213913000000
   0.849915735000000
```

```
 threeDigitDuration: 14.7601
   sixDigitDuration: 466.756
twelveDigitDuration: 466756
```