Math 128A, Summer 2019

PSET #5 (due Wednesday 7/31/2019)

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 $y(0) = s > 0$.

Solution. We proceed normally:

$$\begin{aligned}
\frac{dy}{dt} &= \frac{1}{2}\left[y(t) + y(t)^3\right] \\
\int \frac{dy}{y(t) + y(t)^3} &= \int \frac{dt}{2} \\
\int \left[\frac{1}{y} - \frac{y}{1+y^2}\right] dy &= \frac{t}{2} \\
2\ln|y| - \ln\left|1+y^2\right| + c &= t \\
\ln \frac{y^2}{1+y^2} + \ln(e^c) &= t \\
k\frac{y^2}{1+y^2} &= e^t \\
y^2 &= \frac{1}{\frac{1}{k}e^{-t} - 1}
\end{aligned}$$

Recall that our initial condition is $y(0) = s > 0$, and from our given $y'(t) := \frac{1}{2}\left[y(t) + y(t)^3\right]$ and $y(t)$ are both positive for all $t > 0$. With these, we have:

$$\begin{aligned}
y(0)^2 := s^2 &= \frac{1}{\frac{1}{k}e^{-0} - 1} \\
k &= \frac{1}{1 + \frac{1}{s^2}} = \frac{s^2}{s^2+1},
\end{aligned}$$

which gives our exact solution to the IVP:

$$\implies y(t) = \left[\left(1 + s^{-2}\right)e^{-t} - 1\right]^{-1/2}$$

□

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

Solution. From our solution $y(t) = \left[\left(1 + s^{-2}\right)e^{-t} - 1\right]^{-1/2}$, when $t := \ln\left(1 + s^{-2}\right)$, we have:

$$\lim_{t \to \ln(1+s^{-2})} y(t) = y\left(\ln\left(1 + s^{-2}\right)\right) = \left[\left(1 + s^{-2}\right)\frac{1}{1+s^{-2}} - 1\right]^{-1/2} = \frac{1}{0} = +\infty,$$

where we have $+\infty$ because as noted above, for $t > 0$ we have $y(t) > 0$ and $y'(t) > 0$. □

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

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

Solution. Similar to as done in lecture, define $v_j := [u_{j+1}; u_j]$ so that:

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

Now define matrix $T := \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$, where its eigenvalues are given by:

$$\begin{aligned} \det \begin{bmatrix} 1-\lambda & 1 \\ 1 & -\lambda \end{bmatrix} &:= 0 \\ \lambda^2 - \lambda - 1 &= 0 \\ \implies \lambda &= \frac{1 \pm \sqrt{1+4}}{2} \end{aligned}$$

where we recognize one of the roots is the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$, and the other is $1 - \varphi$. Equivalently, notice $(1 - \varphi) = \frac{-1}{\varphi}$, as Strain uses in lecture (and the rest of the class likely uses). (In retrospect, the given problem is identically the Fibonacci relation.) Because we have two distinct eigenvalues, our 2×2 matrix T is diagonalizable. Define the matrix of eigenvalues $X := \begin{bmatrix} \varphi & 0 \\ 0 & 1-\varphi \end{bmatrix}$, so that we write via similar matrices:

$$T^n = [SXS^{-1}]^n = SX^nS^{-1} = S \begin{bmatrix} \varphi & 0 \\ 0 & 1-\varphi \end{bmatrix}^n S^{-1} = S \begin{bmatrix} \varphi^n & 0 \\ 0 & (1-\varphi)^n \end{bmatrix} S^{-1},$$

so that we have:

$$\begin{aligned} v_n = \begin{bmatrix} u_{n+1} \\ u_n \end{bmatrix} &= SXS^{-1}v_{n-1} \\ &= SX^2S^{-1}v_{n-2} \\ &= \quad \vdots \\ &= SX^nS^{-1}v_0 = \underbrace{SX^nS^{-1}}_{2\times 2} \begin{bmatrix} u_1 \\ u_0 \end{bmatrix} \\ &= S \begin{bmatrix} \varphi & 0 \\ 0 & 1-\varphi \end{bmatrix} S^{-1} \begin{bmatrix} u_1 \\ u_0 \end{bmatrix}, \end{aligned}$$

which gives us the desired closed-form:

$$u_n = x\varphi^n + y(1-\varphi)^n,$$

for some x, y. We solve this via the patented Strain method, which is to know an equation or solution exists and is unique, and hence testing sufficient points gives us particular values or expressions we need to get the exact solution. In this case, we want to write u_n in terms of the initial conditions u_0, u_1, so we solve for x, y by setting $n := 0$ and $n := 1$.

$$u_0 = x + y \tag{1}$$
$$u_1 = x\varphi + y(1-\varphi). \tag{2}$$

Taking $(2) - \varphi(1)$, we have:

$$u_1 - \varphi u_0 = (1-2\varphi)y \implies y = \frac{u_1 - \varphi u_0}{1-2\varphi}$$
$$x = u_0 - y = \frac{(1-2\varphi)u_0}{1-2\varphi} - \frac{u_1 - \varphi u_0}{1-2\varphi} \implies x = \frac{(1-\varphi)u_0 - u_1}{1-2\varphi}$$

Bringing these back into our closed-form expression, we have:

$$\begin{aligned} u_n &= x\varphi^n + y(1-\varphi)^n \\ &= \frac{(1-\varphi)u_0 - u_1}{1-2\varphi}\varphi^n + \frac{u_1 - \varphi u_0}{1-2\varphi}(1-\varphi)^n, \end{aligned}$$

where $\varphi := \frac{1+\sqrt{5}}{2}$.

□

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

Solution. Taking the limit of our expression above as j approaches $+\infty$, let us define:

$$\begin{aligned} L &:= \lim_{j\to\infty} u_j = \lim_{j\to\infty} \frac{(1-\varphi)u_0 - u_1}{1-2\varphi}\varphi^j + \frac{u_1 - \varphi u_0}{1-2\varphi}(1-\varphi)^j \\ &= \lim_{j\to\infty} \frac{(1-\varphi)u_0 - u_1}{1-2\varphi}\varphi^j + \lim_{j\to\infty} \frac{u_1 - \varphi u_0}{1-2\varphi}(1-\varphi)^j \\ &= \frac{(1-\varphi)u_0 - u_1}{1-2\varphi} \underbrace{\lim_{j\to\infty} \varphi^j}_{+\infty} + \frac{u_1 - \varphi u_0}{1-2\varphi} \underbrace{\lim_{j\to\infty} (1-\varphi)^j}_{0} \end{aligned}$$

Hence for L to be bounded (a finite number), we need the left term to equal zero, which is achieved by setting:

$$\begin{aligned} 0 &= \frac{(1-\varphi)u_0 - u_1}{1-2\varphi} \\ 0 &= (1-\varphi)u_0 - u_1, \end{aligned}$$

so we conclude that for all u_0, u_1 with $u_1 = (1-\varphi)u_0$, the value u_j remains bounded by a constant as $j \to \infty$. To see this explicitly, letting $u_1 := (1-\varphi)u_0$ gives:

$$\begin{aligned} u_j &= 0 + \frac{u_1 - \varphi u_0}{1-2\varphi}(1-\varphi)^j \\ &= \frac{(1-\varphi)u_0 - \varphi u_0}{1-2\varphi}(1-\varphi)^j \\ &= \frac{(1-2\varphi)u_0}{1-2\varphi}u_0(1-\varphi)^j \\ &= u_0(1-\varphi)^j, \end{aligned}$$

where for all $j > 1$, $|u_j| < |u_0|$ and hence $|u_0|$ is a bound for $|u_j|$. □

Problem 3. (a) Write, test and debug a Matlab function `euler.m` that approximates the final solution vector $y(b)$ of the vector initial value problem

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

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 $h := \frac{b-a}{n}$ and $u_0 := y_a$.

Solution. Our implementation in R is as follows:

```
euler <- function(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 (that we never use)
  # u : OUTPUT approximation to the final solution vector y(b)
  x <- eulerfull(a,b,ya,f,r,n)
  u <- x[,ncol(x)]
}

eulerfull <- function(a, b, ya, f, r, n) {
  # RETURNS FULL HISTORY UP TO SOLUTION
  h <- (b - a)/n # fixed time-step
  u <- ya # initial conditions
  ### u.store used to generate movement plot along unit circle
  u.store <- matrix(data=NA, nrow = length(ya), ncol = n+1)
  u.store[,1] <- ya
  t <- a # start of interval
  for (j in 1:n) {
    u <- u + h * f(t,u,r)
    u.store[,j+1] <- u
    t <- t + h
  }
  u.store
}
```

In the interest of space, we include some code for the next sub-problem, here.

```
f3b <- function(t, z, r) {
  # called in Euler as f(t,y,r)
  x <- z[1]; y <- z[2]; u <- z[3]; v <- z[4];
  denom <- x**2 + y**2
  c(u,v, -x/denom, -y/denom) # return
}

err.3b <- function(N) {
  # want to tabulate max error for N = 1000, 2000, ..., 16000.
  # initial condition given by problem 3b
  z0 <- c(1, 0, 0, 1) #ya
  a <- 0
  b <- 4*pi
  z.4pi <- euler(a,b,z0,f3b,r,N)
  # max error of
  #   x_N - cos(t_N) = x - 1, y_N - sin(t_N) = y - 0,
  #   u_N + sin(t_N) = u + 0, v_N - cos(t_N) = v - 1;
  #   coincidentally (or by periodicity 4pi same as 0)
  max(abs(z.4pi - z0)) #return max error
}
```

□

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

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

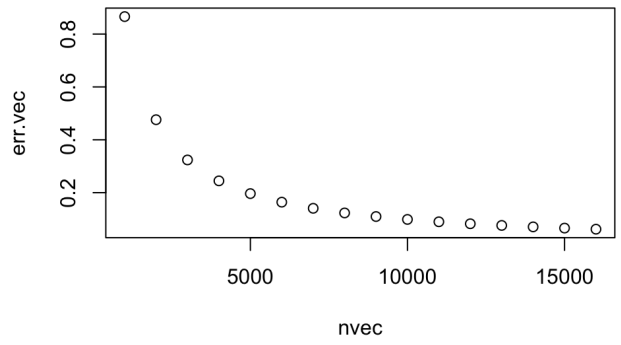
with initial conditions $z = \begin{bmatrix} 1;0;0;1 \end{bmatrix}$ at $t = 0$ which cause the solution to move in a unit circle forever. (1) Measure the maximum error

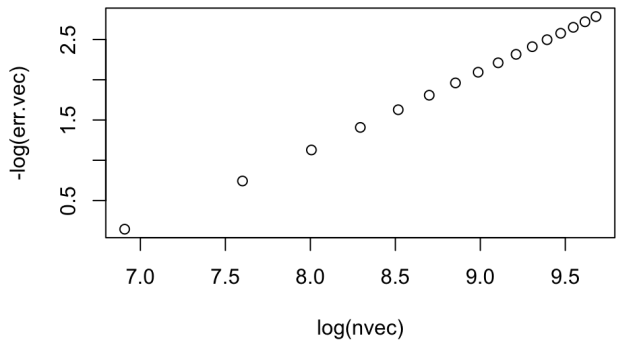
$$E_N = \max\{|x_N - \cos(t_N)|, |y_N - \sin(t_N)|, |u_N + \sin(t_N)|, |v_N - \cos(t_N)|\}$$

after 2 revolutions ($T = 4\pi$) with time steps $h := T/N$ for $N := 1000, 2000, \ldots, 16000$.

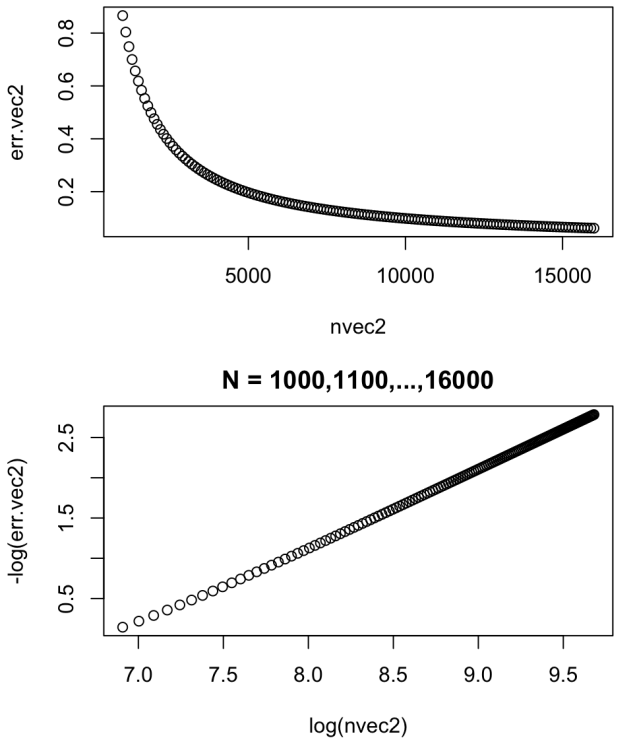
(2) Estimate the constant C such that the error behaves like Ch.
(3) Measure the CPU time for each run and estimate the total CPU time necessary to obtain the solution to 3-digit, 6-digit, and 12-digit accuracy.
Plot the solutions.

Solution. First we tabulate the maximum error after $T = 4\pi$, which gives the following error plots. We are asked for $N := 1000, 2000, \ldots, 16000$ and we plot the result. Because we want to estimate (2) the constant C for which the error behaves like Ch, it appears that plotting $-\log(E_N)$ against $\log(N)$ gives a nearly linear result.





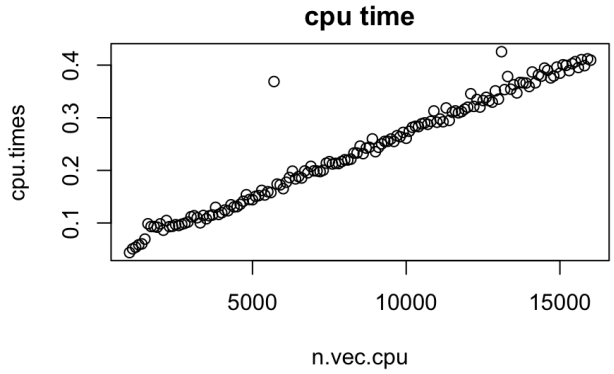
To see this more clearly (at more points), we proceed in taking $N := 1000, 1100, \ldots, 16000$ simply out of curiosity, which produces:



(2) If we take greater credibility towards the higher concentration of points, taking the slope between two points gives us a slope $C = 72.69776370527644$. We can take a different two points or a least-squares fit, but this should suffice for our purposes. Each plot corresponds to the value of N at the top, for example $N = 1000$ in the first plot.

```
# slope:
len <- length(nvec2)
slope <- ( log(err.vec2)[15] - log(err.vec2[len]) ) / (nvec2[len] - nvec2[15])
# Ch = 1/slope, our h here is 100
C <- 1/slope/100; options(digits=16); C;
```

(3) Now we plot the solutions and measure the cpu time for each pass of euler. Generating a ridiculous amount of plots for $N = 1000, 1100, 1200, \ldots, 16000$ and measuring the cpu time required, we have:

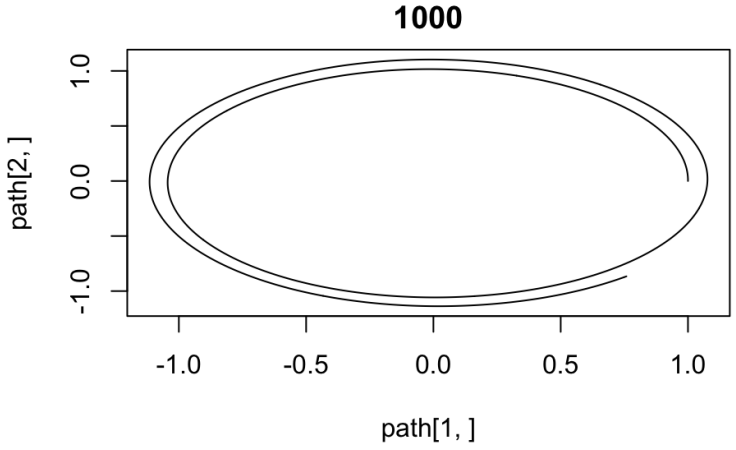


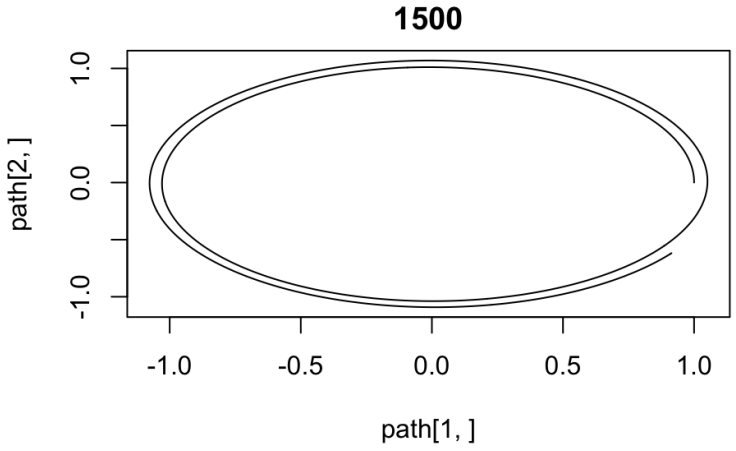
There appears to be a general linear relationship here (as we should expect).

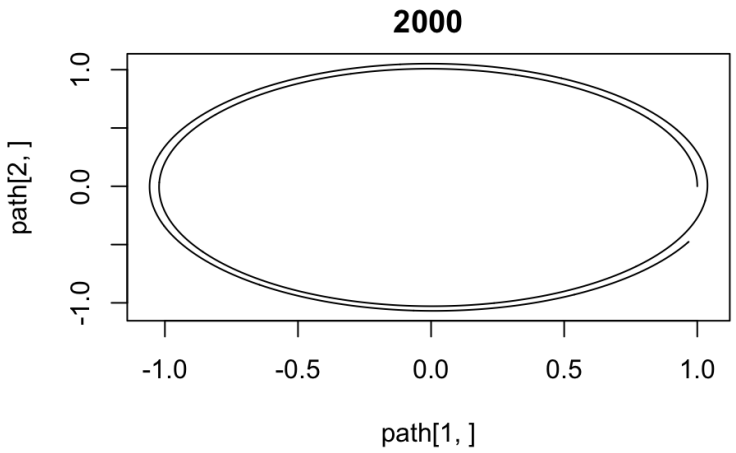
Notice in our appended code that we **do indeed** include cpu time spent plotting into our time measure. Initially, the resulting cpu time versus number of steps was very erratic, but now including function call time as well as plotting time produces a much more linear result. As a result, our (upcoming) estimation in getting p-digit accuracy includes the time required to plot that resulting path for the solution. Additionally, we note that cpu time required for each plot is substantially lower (about a factor of 10 less) for plotting the path as a line rather than the individual points. We first want an estimator for the cpu time per number of steps, which appears to be about 2.6×10^{-5} seconds per step.
Now we wish to estimate the time needed for $3, 6, 12$-digit accuracy. We have found $C = 72.69776$ earlier so that we can use the estimation, for T_i the time required for i-digit accuracy:

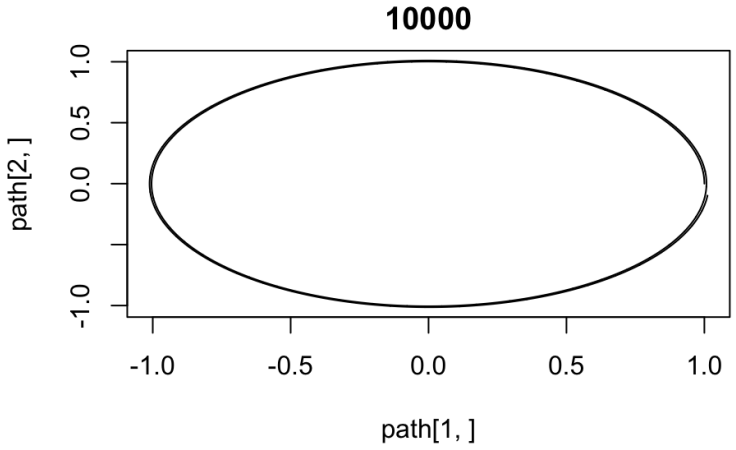
$$T_3 = (2.6 \times 10^{-5})(4\pi)(72.69776)(10^3) = 23.75222186983713 \text{ seconds}$$
$$T_6 = (2.6 \times 10^{-5})(4\pi)(72.69776)(10^6) = 23752.22186983713 \text{ seconds}$$
$$T_12 = (2.6 \times 10^{-5})(4\pi)(72.69776)(10^{12}) = 23752221869.83713 \text{ seconds} = 753.18 \text{ years}$$

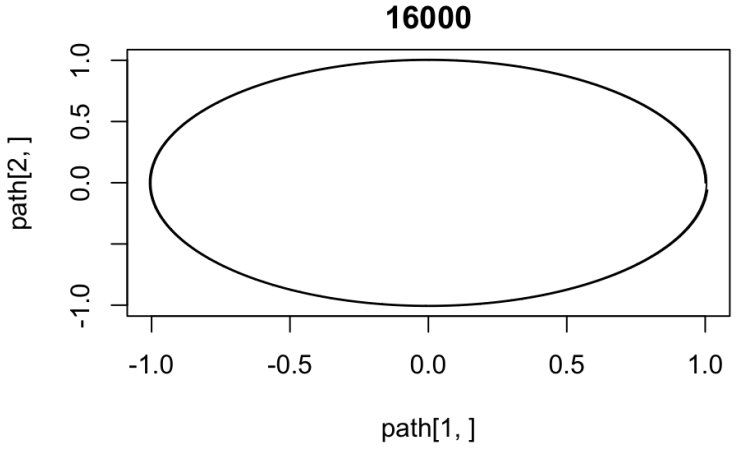
Now plotting the solution paths involves calling `eulerfull`. We select a notable few from our bunch of plots for the above $N = 1000, 1100, 1200, \ldots, 16000$.






2000
path[2,]
path[1,]
1.0
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0


10000
path[2,]
path[1,]
1.0
0.5
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0


16000
path[2,]
path[1,]
1.0
0.5
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0

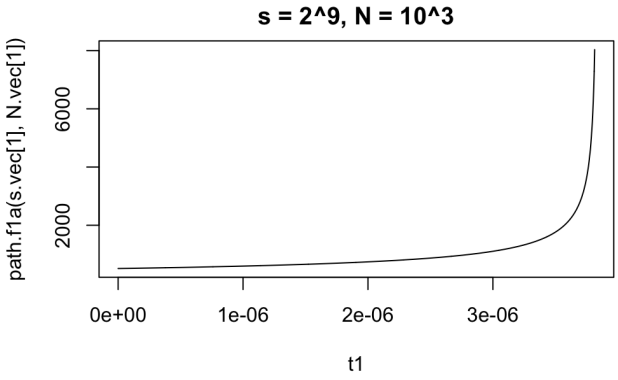
The code to generate these plots is included here:

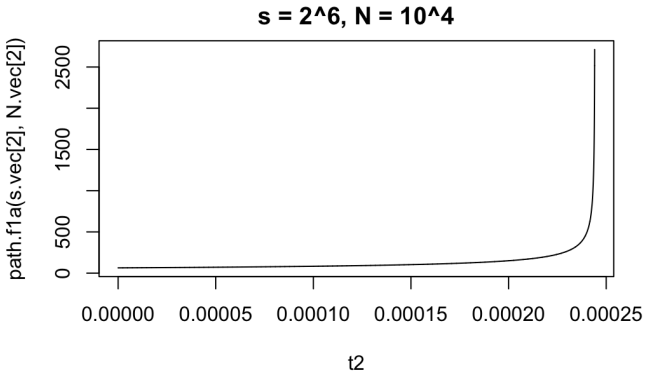
```
# measuring cpu time
n.vec.cpu <- seq(1000,16000,by=100)

# if we want to try to get exact cpu times for certain digits of accuracy
tol.1 <- 10**(-3)
tol.2 <- 10**(-6)
tol.3 <- 10**(-12)

h.vec <- c()
ya <- c(1,0,0,1)
path <- matrix(data = NA, nrow = length(ya), ncol = length(n.vec.cpu)) # initialize path matrix
cpu.times <- c()

for (n in n.vec.cpu) {
  h.vec <- c(h.vec, 4*pi/n)
  # start clock for
  time.start <- Sys.time()
  path <- eulerfull(0, 4*pi, c(1,0,0,1), f3b, r, n)
  plot(path[1,], path[2,], main=n, type='l')
  time.stop <- Sys.time()
  cpu.times <- c(cpu.times, (time.stop - time.start) )
}
plot(n.vec.cpu, cpu.times, main='cpu time')

## Plot Errors
nvec <- seq(1000,16000,by=1000)
err.vec <- c()
for (n in nvec) {
  err.vec <- c(err.vec, err.3b(n))
}
plot(nvec, err.vec)
plot(log(nvec),-log(err.vec))


## trying more n values:
nvec2 <- seq(1000, 16000, by=100)
err.vec2 <- c()
for (n in nvec2) {
  err.vec2 <- c(err.vec2, err.3b(n))
}
plot(nvec2, err.vec2)
plot(log(nvec2), -log(err.vec2), main = 'N = 1000,1100,...,16000')

# slope:
len <- length(nvec2)
slope <- ( log(err.vec2)[15] - log(err.vec2[len]) ) / (nvec2[len] - nvec2[15])
# Ch = 1/slope, our h here is 100
C <- 1/slope/100
C
```

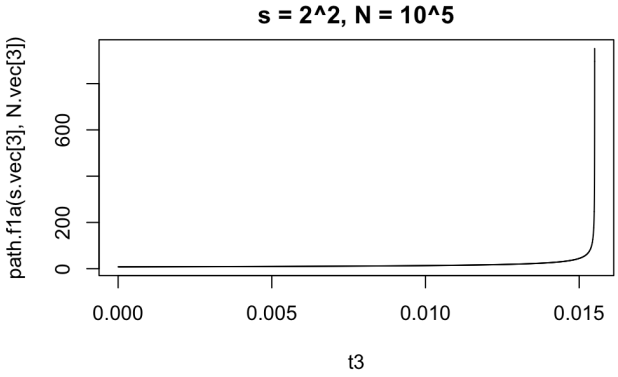
□

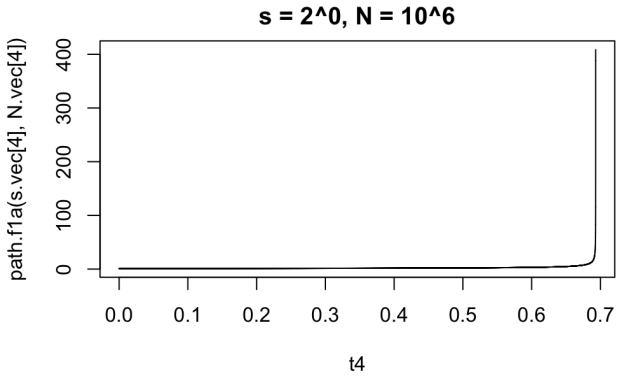
(c) Use `euler.m` with $s := [512, 64, 8, 1]$ and $N := [10^3, 10^4, 10^5, 10^6]$ to verify conclusion (b) of Problem 1, which is that the solution blows up when $t := \log\left(1 + \frac{1}{s^2}\right)$.

Solution. We are asked to take $s := [2^9, 2^6, 2^3, 2^0]$ and $N := [10^3, 10^4, 10^5, 10^6]$. In the interest of space, we consider the pairwise value assignments: $(s, N) := (2^9, 10^3), (2^6, 10^4), (2^3, 10^5), (2^0, 10^6)$. We generate plots of the path `euler` takes on for each of these 4 cases:









and we are content that the solution blows up when $t = \log\left(1 + 1/s^2\right)$. Please excuse my attrocious plotting code.

```r
# verifying Problem 1 blows up for
# t := log(1 + 1/s^2)

f1a <- function(t,y,r) {
  (1/2) * (y + y^3) # fixed step size in time
}

path.f1a <- function(s,N) {
  # s : initial value y(0) = s > 0
  # N : step count
  OwO <- log(1 + 1 / s^2) # notices bulge (blows up)
  eulerfull(0, OwO, s, f1a, r, N)
}

s.vec <- c(2^9, 2^6, 2^3, 2^0)
N.vec <- c(10^3, 10^4, 10^5, 10^6)

getowo <- function(s) { log(1 + 1 / s^2, base=exp(1)) } # smh
owo1 <- getowo(s.vec[1])
owo2 <- getowo(s.vec[2])
owo3 <- getowo(s.vec[3])
owo4 <- getowo(s.vec[4])
t1 <- seq(0,owo1, by=owo1/N.vec[1])
t2 <- seq(0,owo2, by=owo2/N.vec[2])
t3 <- seq(0,owo3, by=owo3/N.vec[3])
t4 <- seq(0,owo4, by=owo4/N.vec[4])
plot(t1, path.f1a(s.vec[1], N.vec[1]), type='l', main='s = 2^9, N = 10^3')
plot(t2, path.f1a(s.vec[2], N.vec[2]), type='l', main='s = 2^6, N = 10^4')
plot(t3, path.f1a(s.vec[3], N.vec[3]), type='l', main='s = 2^2, N = 10^5')
plot(t4, path.f1a(s.vec[4], N.vec[4]), type='l', main='s = 2^0, N = 10^6')
```

□

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 the problem to a 4×4 **first-order** system $u' = f(t, u, r)$ with $u(0) = u_0$, 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.

Solution. We are given (to introduce) $u = [x, x', y, y'] = [u_1, u_2, u_3, u_4]$, so we have

$$u_1 := x, \qquad u_2 := x' = u_1'$$
$$u_3 := y, \qquad u_4 := y' = u_3'.$$

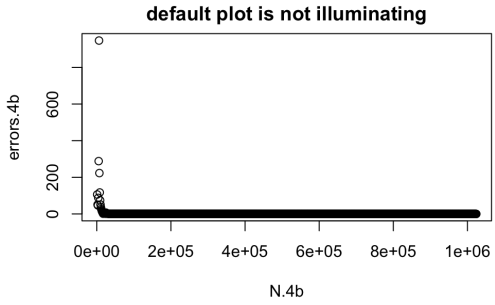
Hence substituting these into the given second-order system of ODEs yields:

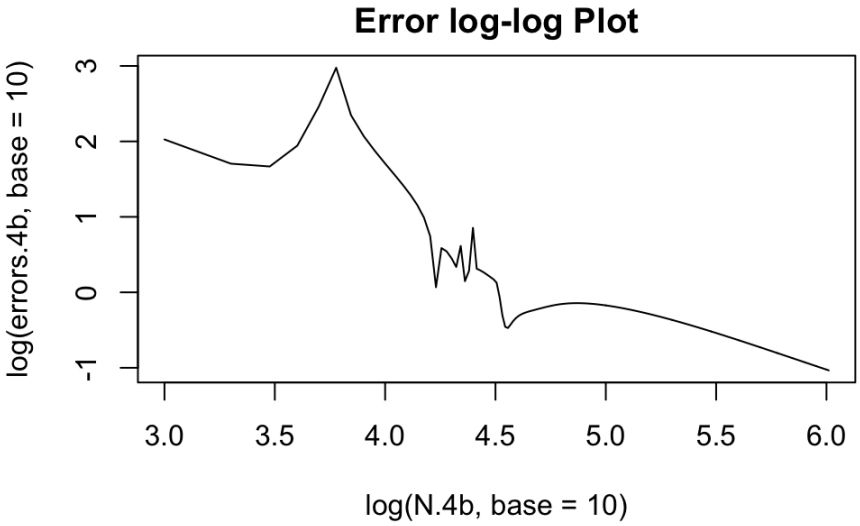
$$\begin{aligned}
f_1(t,u,r) := u_1' &= u_2 = x' \\
f_2(t,u,r) := u_2' = x'' &= x + 2y' - b\frac{x+a}{[(x+a)^2+y^2]^{3/2}} - a\frac{x-b}{[(x-b)^2+y^2]^{3/2}} \\
&= u_1 + 2u_4 - b\frac{u_1+a}{[(u_1+a)^2+u_3^2]^{3/2}} - a\frac{u_1-b}{[(u_1-b)^2+u_3^2]^{3/2}} \\
f_3(t,u,r) := u_3' &= u_4 = y' \\
f_4(t,u,r) := u_4' = y'' &= y - 2x' - b\frac{y}{[(x+a)^2+y^2]^{3/2}} - a\frac{y}{[(x-b)^2+y^2]^{3/2}} \\
&= u_3 - 2u_2 - b\frac{u_3}{[(u_1+a)^2+u_3^2]^{3/2}} - a\frac{u_3}{[(u_1-b)^2+u_3^2]^{3/2}}
\end{aligned}$$

□

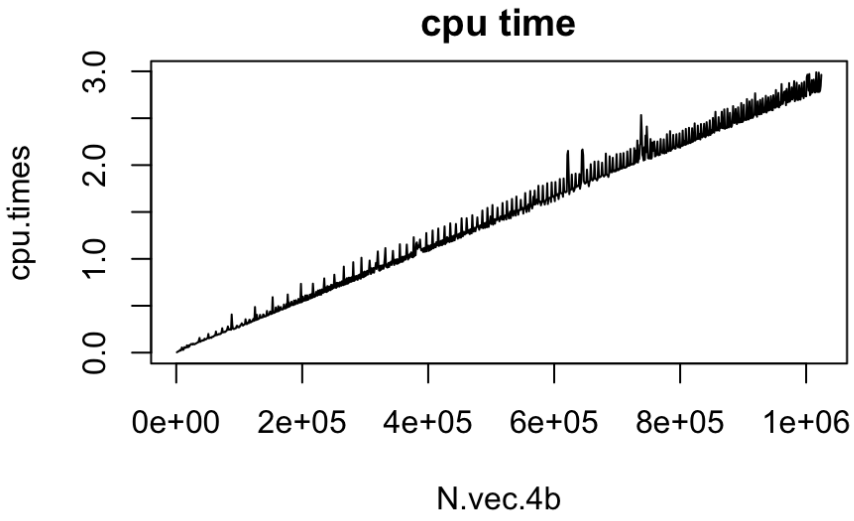
(b) Use `euler.m` to approximate $u(T)$ and plot the error versus N, for $N := 1000, 2000, \ldots, 2000k, \ldots, 1024000$ steps. Measure the CPU time for each run and estimate the total CPU time necessary to obtain an orbit which is periodic to 3-digit, 6-digit, and 12-digit accuracy.

Solution. First we plot the errors versus N (and) similarly notice that there is a nicer log-log relationship. Notwithstanding the obviously nonlinear result with the error log-log plot, we assume a linear model to be able to easily perform an estimation for CPU time required for $3, 6, 12$-digit accuracies. Because the nature of this linear interpolation scheme is inappropriate for our given data, we claim taking the points $(10^N, 10^E) = \{(3.0, 2), (6.0, -1)\}$ can't possibly create more harm. First we verify that `min(errors.4b)` returns 0.09245539327662651 as the true





value of error. This is the point being estimated by our $(6.0, -1)$ in that this precisely occurs at $N := 10^6 = 1024000$, where our CPU took about 2.8927 seconds. Content with results, we proceed with a similar approach as taken in the previous problem, in that assuming all other factors constant, scaling N (and hence stepsize h) causes an inverse linear scaling effect between error and stepsize.



Measuring the CPU time for each run took about 45 minutes in total (weak cpu + my Euler stores all history).

Using our values from above (and plots for judgment), we have the following estimates for CPU times corresponding to :

$E = 0.09246 \approx 10^{-1}$	$CPU = 2.8927$ secs
$E = 10^{-3}$	$CPU = 2.6746 \times 10^{2}$ secs
$E = 10^{-6}$	$CPU = 2.6746 \times 10^{5}$ secs
$E = 10^{-12}$	$CPU = 2.6746 \times 10^{11}$ secs

Although not required or prompted, we include the orbit plot for fun:

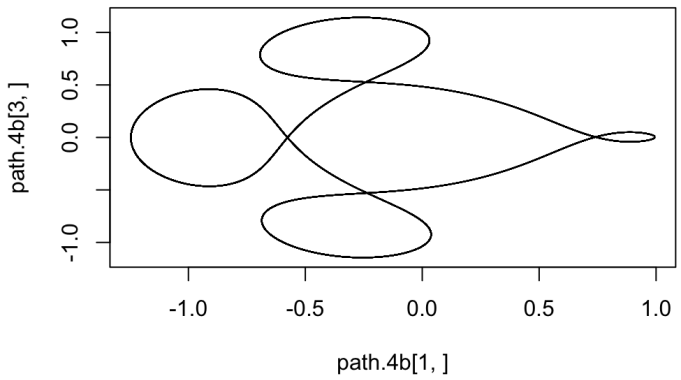


Figure 1: N = 10,240,000

Per usual, our codes to generate the data and plots for this problem are included below:

```
# initialize things given in the problem
N.4b <- seq(1000,1024000, by=1000)
period.T <- 17.06521656015796
ya <- c(0.994, 0, 0, -2.00158510637908)
a <- 0.012277471; b <- 1 - a; # coeffs for formula


f.4b <- function(t,z,r) {
  # old
  u1 <- z[1]; u2 <- z[2]; u3 <- z[3]; u4 <- z[4];
  # new
  v1 <- u2;
  denom1 <- ((u1+a)^2+u3^2)^(3/2)
  denom2 <- ((u1 - b)^2 + u3^2)^(3/2)
  v2 <- u1 + 2*u4 - b * (u1 + a)/denom1 - a * (u1 - b)/denom2
  v3 <- u4;
  v4 <- u3 - 2*u2 - b*u3/denom1 - a*(u3)/denom2 # could also store common denoms
  c(v1, v2, v3, v4) # return new u1,u2,u3,u4 which we call v.
}
```

For CPU measurement:

```
cpu.times <- c()
N.vec.4b <- seq(1000,1024000,by=1000)
ya.4b <- c(0.994, 0, 0, -2.00158510637908)

for (n in N.vec.4b) {
  time.start <- Sys.time()
  path.4b <- eulerfull(0, period.T, ya.4b, f.4b, r, n)
  # plot(path[1,], path[3,], main=n, type='l')
  time.stop <- Sys.time()
  cpu.times <- c(cpu.times, (time.stop - time.start) )
  print(n)
}
plot(N.vec.4b, cpu.times, main='cpu time')
```

For error:

```
N.4b <- seq(1000,1024000, by=1000)
period.T <- 17.06521656015796
ya <- c(0.994, 0, 0, -2.00158510637908)
a <- 0.012277471; b <- 1 - a; # coeffs for formula

f.4b <- function(t,z,r) {
  # u : old
  u1 <- z[1]; u2 <- z[2]; u3 <- z[3]; u4 <- z[4];
  # v : new u
  v1 <- u2;
  denom1 <- ((u1+a)^2+u3^2)^(3/2)
  denom2 <- ((u1 - b)^2 + u3^2)^(3/2)
  v2 <- u1 + 2*u4 - b * (u1 + a)/denom1 - a * (u1 - b)/denom2
  v3 <- u4;
  v4 <- u3 - 2*u2 - b*u3/denom1 - a*(u3)/denom2 # could also store common denoms
  c(v1, v2, v3, v4) # return new u1,u2,u3,u4 which we call v.
}

getErr.4b <- function(n) {
  diff <- abs(eulerfast(0,period.T,ya.4b,f.4b,r,n) - ya)
  sqrt(diff[1]^2 + diff[3]^2) #return sqrt(x^2 + y^2)
}

# generate error data: do this only once, then comment after to prevent overwrite
errors.4b <- c()
for (n in N.4b) {
  errors.4b <- c(errors.4b, getErr.4b(n))
```

```
  print(n)
}

plot(log(N.4b, base=10), log(errors.4b,base=10), type = 'l', main = 'Error log-log Plot')

plot(N.4b, errors.4b, main='default plot is not illuminating')
```

And some auxiliary things that help.

```
# for plotting orbit only (uncomment for plot, otherwise leave commented!!!)
path.4b <- eulerfull(0,period.T,ya,f.4b,r,1024000)
plot(path.4b[1,], path.4b[3,], type='l')

# re-implement a faster euler
eulerfast <- function(a, b, ya, f, r, n) {
  # init
  h <- (b - a)/n # fixed time-step
  u <- ya # initial conditions
  t <- a # start of interval
  for (j in 1:n) {
    u <- u + h * f(t,u,r)
    t <- t + h
  }
  u # outputs u without path
}
```

□

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

$$\begin{aligned} y'(t) &= f(t, y(t)) \\ y(0) &= y_0, \end{aligned}$$

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

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

Solution. We defined the error as $e(t) := y(t) - u(t)$ in the question, so taking the derivative of each side with respect to t yields:

$$\begin{aligned} e'(t) &= y'(t) - u(t) \\ &= f(t, y(t)) - u'(t) \qquad (y'(t) = f(t, y(t)), \text{ given}) \\ &= f(t, e(t) + u(t)) - u'(t) \qquad (y(t) = e(t) + u(t), \text{ from definition of } e(t)), \end{aligned}$$

which is precisely the desired expression for $e'(t)$. Now for $e(0) = 0$, recallw e are given $u(0) = y(0)$, and hence $e(0) := y(0) - u(0) = 0$, as required. □

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

Solution. We proceed just as performed by Strain on Monday (Lecture 21): Suppose $y(t)$ is the exact solution to the above initial value problem, where we showed $e'(t) = f(t, u(t) + e(t)) - u'(t)$ and use this fact. Additionally, we suppose $f(y) = \lambda y$ given in the question, so we also have

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

and we can conclude

$$y(t) = e^{t\lambda} y(0),$$

as this can be easily verified. Now take:

$$\begin{aligned} e'(t) &= \lambda(u(t) + e(t)) - u'(t) \\ &= \lambda e(t) + \lambda u(t) - u'(t) \\ (u(t) + e(t))' &= \lambda(u(t) + e(t)) \\ u(t) + e(t) &= e^{t\lambda}[u(0) + e(0)] = y(t), \end{aligned}$$

and as we have explicitly shown $u(t) + e(t) = y(t)$, we have our desired result.

□

Problem 6. Define a family of explicit Runge-Kutta methods parameterized 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 ($e(t) := y(t) - u(t)$) via Euler's method, yielding approximate errors $e^1_{n+1}, e^1_{n+2}, \ldots, e^1_{n+p}$.

Produce a second-order accurate corrected solution

$$u^2_{n+j} = u^1_{n+j} + e^1_{n+j} + hf(t_n, u^1_{n+1})$$

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

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

Solution. When $p = 1$, the uncorrected solution given above in the problem for $0 \le j \le p-1 = 0 \implies j = 0$ is

$$\begin{aligned} u^1_{n+0+1} &= u^1_{n+0} + hf(t_{n+0}, u^1_{n+0}) \\ u^1_{n+1} &= u^1_n + hf(t_n, u^1_n), \end{aligned}$$

which is precisely Euler's method as desired. □

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

$$\begin{aligned} k_1 &= f(t_n, u_n) \\ k_2 &= f(t_n + c_2 2h, u_n + 2h[a_{2,1}k_1]) \\ k_3 &= f(t_n + c_3 2h, u_n + 2h[a_{3,1}k_1 + a_{3,2}k_2]) \\ u_{n+2} &= u_n + 2h(b_1k_1 + b_2k_2 + b_3k_3). \end{aligned}$$

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

Solution. We proceed via the algorithm specified in the problem. We want to express our Runge-Kutta method in the form $u_{n+2} = u_n + 2h(b_1k_1 + b_2k_2 + b_3k_3)$, with k_1, k_2, k_3 defined as above. Here we apply $p = 2$ steps of Euler's method.

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

Now we apply a single ($p - 1 = 1$) pass of deferred correction. From the previous problem (5), we verified for the error (where we set $e(t_n) := 0$ for our initial estimate):

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

which after applying one step of Euler's method gives:

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

where we need to use the polynomial (given in the problem) $u(t) = U_1(t)$ to interpolate $u'(t_n)$ at three points, $(t_n, u_n), (t_{n+1}, u^1_{n+1}), (t_{n+2}, u^1_{n+2})$. Using the Lagrange basis is easiest for differentiation:

$$\begin{aligned} L_0 &:= \frac{(t-t_{n+1})(t-t_{n+2})}{(t_n-t_{n+1})(t_n-t_{n+2})} &&\implies L_0' := \frac{(t-t_{n+1})+(t-t_{n+2})}{(t_n-t_{n+1})(t_n-t_{n+2})} \\ L_1 &:= \frac{(t-t_n)(t-t_{n+2})}{(t_{n+1}-t_n)(t_{n+1}-t_{n+2})} &&\implies L_1' := \frac{(t-t_n)+(t-t_{n+2})}{(t_{n+1}-t_n)(t_{n+1}-t_{n+2})} \\ L_2 &:= \frac{(t-t_n)(t-t_{n+1})}{(t_{n+2}-t_n)(t_{n+2}-t_{n+1})} &&\implies L_2' := \frac{(t-t_n)+(t-t_{n+1})}{(t_{n+2}-t_n)(t_{n+2}-t_{n+1})} \end{aligned}$$

and

$$\begin{aligned} U_1(t) &:= L_0 u_n + L_1 u^1_{n+1} + L_2 u^1_{n+2} \\ \implies U_1'(t) &:= L_0' u_n + L_1' u^1_{n+1} + L_2' u^1_{n+2}. \end{aligned}$$

Assuming we use a fixed time-step size $h := t_{n+1} - t_n = t_{n+2} - t_n$, evaluating the polynomial and its derivative at t_n and t_{n+1} gives much simpler expressions:

$$\begin{aligned} L_0'(t_n) &= \frac{(-h)+(-2h)}{(-h)(-2h)} = \frac{-3h}{2h^2} = \frac{-3}{2h} \\ L_1'(t_n) &= \frac{0+(-2h)}{(h)(h)} = \frac{-2h}{h^2} = \frac{-2}{h} \\ L_2'(t_n) &= \frac{0+(-h)}{(2h)(h)} = \frac{-h}{2h^2} = \frac{-1}{2h} \\ L_0'(t_{n+1}) &= \frac{0+(-h)}{(-h)(-2h)} = \frac{-h}{2h^2} = \frac{-1}{2h} \\ L_1'(t_{n+1}) &= \frac{h+(-h)}{(h)(-h)} = 0 \\ L_2'(t_{n+1}) &= \frac{(h)+0}{(2h)(h)} = \frac{h}{2h^2} = \frac{1}{2h} \end{aligned}$$

and plugging these in gives

$$\begin{aligned} U_1'(t_n) &= -\frac{3}{2h} u_n + \frac{2}{h} u^1_{n+1} - \frac{1}{2h} u^1_{n+2} \\ U_1'(t_{n+1}) &= \frac{-1}{2h} u_n + \frac{1}{2h} u^1_{n+2}. \end{aligned}$$

Hence our error equation with our approximation $U_1'(t)$ at t_n, t_{n+1} now gives:

$$\begin{aligned} e^1_{n+1} &= hf(\ t_n, u_n\ ) - hu'(t_n) \\ &= hf(\ t_n, u_n\ ) + \frac{3}{2} u_n - 2u^1_{n+1} + \frac{1}{2} u^1_{n+2} \\ &= hk_1 + \frac{3}{2} u_n - 2u^1_{n+1} + \frac{1}{2} u^1_{n+2} \\ e^1_{n+2} &= e^1_{n+1} + h\left[f(\ t_{n+1}, e^1_{n+1} + u^1_{n+1}\ )\right] + u'(t_{n+1}) \\ &= e^1_{n+1} + h\left[f(\ t_{n+1}, e^1_{n+1} + u^1_{n+1}\ )\right] + \frac{1}{2} u_n - \frac{1}{2} u^1_{n+2} \\ &= e^1_{n+1} + hk_3 + \frac{1}{2} u_n - \frac{1}{2} u^1_{n+2} \end{aligned}$$

We now have all the ingredients to cook up our new estimate u^2_{n+2}:

$$\begin{aligned}
u^2_{n+2} &:= u^1_{n+2} + \underbrace{e^1_{n+2}} \\
&= u^1_{n+2} + \left(\underbrace{e^1_{n+1}} + hk_3 + \frac{1}{2}u_n - \frac{1}{2}u^1_{n+2}\right) \\
&= u^1_{n+2} + \left(hk_1 + \frac{3}{2}u_n - 2u^1_{n+1} + \frac{1}{2}\cancel{u^1_{n+2}}\right) + hk_3 + \frac{1}{2}u_n - \cancel{\frac{1}{2}u^1_{n+2}} \\
&= \underbrace{u^1_{n+2} - u^1_{n+1}}_{hf(t_n,u^1_n)=hk_2} - u^1_{n+1} + 2u_n + hk_1 + hk_3 \\
&= (hk_2) - \underbrace{u^1_{n+1} + u_n}_{-hk_1} + u_n + hk_1 + hk_3 \\
&= u_n - \cancel{hk_1} + hk_2 + \cancel{hk_1} + hk_3 \\
&= u_n + hk_2 + hk_3.
\end{aligned}$$

Recall that we wanted an expression like the following so that we can put them in a Butcher array:

$$u_{n+2} = u_n + 2h(b_1k_1 + b_2k_2 + b_3k_3),$$

so we conclude:

$$b_1 := 0, \qquad b_2 := \frac{1}{2}, \qquad b_3 := \frac{1}{2}.$$

And from earlier,

$$\begin{aligned}
k_1 &= f(t_n, u_n) \\
k_2 &= f(t_n + c_2 2h, u_n + 2ha_{2,1}k_1) = f(t_n + h, u_n + hk_1) \\
k_3 &= f(t_n + c_3 2h, u_n + 2h[a_{3,1}k_1 + a_{3,2}k_2]) = f\left(t_n + h, u_n + \frac{h}{2}k_1 + \frac{h}{2}k_2\right)
\end{aligned}$$

gives the values:

$$\begin{aligned}
a_{2,1} = \frac{1}{2}; \qquad a_{3,1} = \frac{1}{4}; \qquad a_{3,2} = \frac{1}{4} \\
c_2 = \frac{1}{2}; \qquad c_3 = \frac{1}{2}
\end{aligned}$$

Hence our resulting Butcher array for this family of deferred correction explicit Runge-Kutta methods is:

$$\begin{bmatrix}
0: & 0 & \cdot & \cdot \\
\frac{1}{2}: & \frac{1}{2} & 0 & \cdot \\
\frac{1}{2}: & \frac{1}{4} & \frac{1}{4} & 0 \\
\cdot & 0 & \frac{1}{2} & \frac{1}{2}
\end{bmatrix}$$

□

(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.

Solution. Taylor expanding to $O(h^2)$ gives:

$$\begin{aligned}
k_2(h) &= f(u_n + hk_1) = k_2(0) + hk_2'(0) + \frac{h^2}{2!}k_2''(0) + O(h^3) \\
&= f + hff' + \frac{h^2}{2}f^2f'' + O(h^3) \\
k_2'(h) &= k_1 f'(u_n + hk_1) \\
k_2''(h) &= k_1^2 f''(u_n hk_1) \\
k_2(0) &= f(u_n) = k_1 \\
k_2'(0) &= k_1 f'(u_n) = ff' \\
k_2''(0) &= k_1^2 f''(u_n) = f^2 f'' \\
k_3(h) &= f\left(u_n + \frac{1}{2}(hk_1 + hk_2)\right) = k_3(0) + hk_3'(0) + \frac{h^2}{2}k_3''(0) + O(h^3) \\
&= f + hff' + \frac{h^2}{2}\left[f(f')^2 + (f^2)f''\right] + O(h^3) \\
k_3'(h) &= \frac{1}{2}(k_2 + hk_2' + k_1)f'\left(u_n + \frac{1}{2}(hk_1 + hk_2)\right) \\
k_3''(h) &= \frac{1}{2}(k_2' + hk_2'' + k_2')f'\left(u_n + \frac{1}{2}(hk_1 + hk_2)\right) + \frac{1}{4}(k_1 + k_2 + hk_2')^2 f''\left(u_n + \frac{1}{2}(hk_1 + hk_2)\right) \\
k_3(0) &= f(u_n) = k_1 \\
k_3'(0) &= \frac{1}{2}(k_1 + k_2(0))f'(u_n) = k_1 f'(u_n) = ff' \\
k_3''(0) &= k_2'(0)f'(u_n) + \frac{1}{4}(k_2(0) + k_1)^2 f''(u_n) = k_1(f'(u_n))^2 + \frac{1}{4}(2k_1)^2 f''(u_n) = f(f')^2 + (f^2)f''
\end{aligned}$$

Recall that

$$y_{n+2} - y_n = 2h\tau + b_1k_1 + b_2k_2 + b_3k_3,$$

so solving for the local truncation error τ and plugging in above expressions gives:

$$\begin{aligned}
\tau &= \frac{y_{n+2} - y_n}{2h} - b_1k_1 - b_2k_2 - b_3k_3 \\
&= \frac{1}{2h}\left[\left(y_n + (2h)y_n' + \frac{(2h)^2}{2!}y_n'' + \frac{(2h)^3}{3!}y_n''' + O(h^4)\right) - y_n\right] - 0 - \frac{1}{2}k_2 - \frac{1}{2}k_3 \\
&= \frac{2}{3}h^2\left[f(f')^2 + f^2f''\right] - \frac{1}{2}h^2f^2f'' - \frac{1}{4}h^2f(f')^2 + O(h^3) \\
&= \frac{5}{12}h^2f(f')^2 + \frac{1}{6}h^2f^2f'' \\
&= h^2\left[\frac{5}{12}f(f')^2 + \frac{1}{6}f^2f''\right],
\end{aligned}$$

which gives $\tau = O(h^2)$ as desired. □

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

Solution. Given by the `idec` handout, "IDEC can also be viewed as a fixed point iteration for the solution of an implicit Runge-Kutta method. The iteration updates p-vectors $u_{n+j}^k \to u_{n+j}^{k+1}$ by adding e_{n+j}. Hence if it converges as $k \to \infty$, then the limit u_{n+j} is determined by setting $e_{n+j} := 0$." In other words,

$$0 = 0hf(t_{n+j}, u_{n+j}) - hu'(t_{n+j}).$$

This gives us a system fixed point iteration given by:

$$\begin{bmatrix} u_{n+1}^2 \\ u_{n+2}^2 \\ \vdots \\ u_{n+p}^2 \end{bmatrix} := \begin{bmatrix} u_{n+2}^1 \\ u_{n+2}^1 \\ \vdots \\ u_{n+p}^1 \end{bmatrix} + \begin{bmatrix} e_{n+1} \\ e_{n+2} \\ \vdots \\ e_{n+p} \end{bmatrix}$$

Define $U(t)$ similarly as in the previous sub-problem to be the interpolating polynomial at points (t_{n+j}, u_{n+j}), which (again) we write via Lagrange (due to simplicity and symmetry for differentiation). In terms of dimensionless differentiation coefficients d_{jk}, the handout gives:

$$U(t) = \sum_{j=0}^{p} L_j(t) u_{n+j} \qquad U'(t) = \frac{1}{h} d_{jk} u_{n+k}$$

and

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

for $1 \le j \le p$. Additionally, we have $\sum_{k=0}^{p} d_{jk} = 0$ because differentiating a constant returns 0, so we conclude

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

and we must have

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

where the handout claims to recognize the p stages k_j of a Runge-Kutta method. To see this explicitly, we take:

$$\begin{aligned} u_{n+k} - u_n &= h \sum_{j=1}^{p} c_{k,j} k_j, \\ k_j := f(t_{n+j}, u_{n+j}) &= f\left(t_{n+j}, u_n + ph \sum_{m=1}^{p} \frac{k_m c_{j,m}}{p} \right), \end{aligned}$$

which precisely gives our desired relationship between the fixed point iteration and implicit Runge-Kutta method with p stages.

□

Problem 7. Write, test and debug a Matlab function `idec` that 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 (deferred correction), with $u_0 := y_a$.

```
# inherit given information on Orbit from 4b for 7a
period.T <- 17.06521656015796
ya <- c(0.994, 0, 0, -2.00158510637908)
a <- 0.012277471; b <- 1 - a; # coeffs for formula

f.4b <- function(t,z,r) {
  # old
  u1 <- z[1]; u2 <- z[2]; u3 <- z[3]; u4 <- z[4];
  # new
  v1 <- u2;
  denom1 <- ((u1+a)^2+u3^2)^(3/2)
  denom2 <- ((u1 - b)^2 + u3^2)^(3/2)
  v2 <- u1 + 2*u4 - b * (u1 + a)/denom1 - a * (u1 - b)/denom2
  v3 <- u4;
  v4 <- u3 - 2*u2 - b*u3/denom1 - a*(u3)/denom2
  c(v1, v2, v3, v4) # return new u1,u2,u3,u4 which we call v.
}
```

```
idec <- function(a,b,ya,f,r,p,n) {
  # a,b : interval endpoints, a<b
  # ya : vector y(a) of initial conditions
  # f : function handle f(t,y,r) to integrate; y is vector
  # r : parameters to f (which we almost never use)
  # p : number of euler substeps / correction passes
  # n : number of time steps
  # yb : output approximation to the final solution vector y(b)

  # init
  h <- (b - a) / (n - 1)
  dim.ya <- length(ya)
  yb <- matrix(data=0, nrow=dim.ya, ncol=n)
  yb[,1] <- ya
  owo <- matrix(data=0,nrow=p+1,ncol=1)
  for (j in 1:p+1) {
    owo[j,1] <- 1
    for (k in 1:p+1) {
      if (j != k) {
        owo[j,1] <- owo[j,1] / (j - k)
      }
    }
  }

  # fill c.mtx
  c.mtx <- matrix(data=0, nrow=p+1,ncol= p+1)
  for (i in 1:p+1) {
    for (j in 1:p+1) {
      total <- 0
      for (k in 1:p+1) {
        if (j != k) {
          times <- 1
          for (l in 1:p+1) {
            if (( l != k) & (l != j )) {
              times <- (i - l) * times
            }
          }
          total <- total + times
```

```
        }
      }
      c.mtx[i,j] <- owo[j,1] * total
    }
  }

  nwn <- matrix(data=0, nrow = dim.ya, ncol = p+1)
  for (j.idx in 1 : (n - 1)) {
    t <- a + (j.idx - 1) * h
    nwn[,1] <- yb[,j.idx]
    for (i in 1:p) {
      tp <- t + h/p * (i-1)
      nwn[,i+1] <- nwn[,i] + (h/p)*f(tp, nwn[,i], r)
    }
    for (q in 1:(p-1)) {
      err.mtx <- matrix(data=0, nrow = dim.ya, ncol = (p+1))
      for (i in 1:p) {
        tp <- t + h/p * (i-1)
        k.val <- 0
        for (k in 1:(p+1)) {
          k.val <- k.val + c.mtx[i,k] * nwn[,k]
        }
        err.mtx[,i+1] <- err.mtx[,i] + h/p * f(tp, nwn[,i] + err.mtx[,i], r ) - k.val
      }
      for (i in 2:(p+1)) {
        nwn[,i] <- nwn[,i] + err.mtx[,i]
      }
    }
    yb[,j.idx + 1] <- nwn[,p+1]
  }
  # yb <- yb[,n] ## return last
  yb ## return all orbit history
}
```

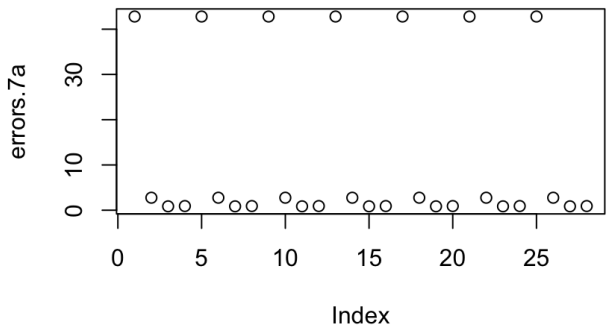
(a) Use `idec.m` with orders $p = 1{:}7$ and $N := 10000, 20000, 40000$, and 80000 steps to approximate the final solution vector $u(T)$ of the initial value problem derived in Problem 4 (satellite orbit around the earth). Tabulate the errors:

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

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

Solution. We implement our Runge-Kutta deferred correction scheme in R (included in later pages). Now we are interested in the error involved with `idec`.

```
getErr.7a <- function(p,n) {
  # max_{1 \leq j \leq 4} ( u_j(T) - u_j(0) )
   diff <- abs(idec(0,period.T, ya.7a, f.4b, r, p, n) - ya.7a)
  max(diff) # return the max of 4 values
}

errors.7a <- c()
for (p in p.7a) {
  for (n in N.7a) {
    errors.7a <- c(errors.7a, getErr.7a(p,n))
    print(n) # debugging
  }
}
```



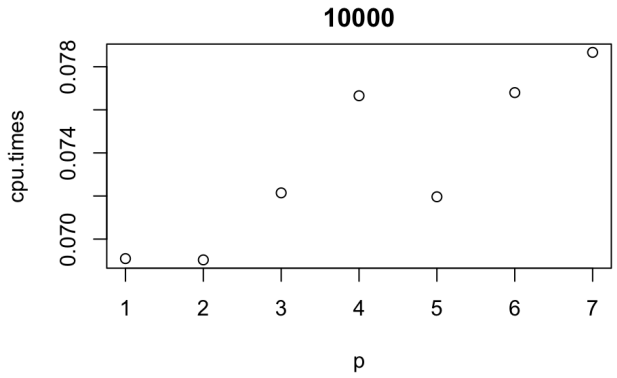
Ironically, my code to plot the errors is throwing errors. Without this, we can only qualitatively talk about the 'visual' error (or lack thereof) in part (c). This plot here cannot be correct, as it shows only the $p = 1$ case, repeated for all p. In hopes for some mercy points, we note that with the correct data, we can proceed just as we have for the previous error and cpu estimation problems.

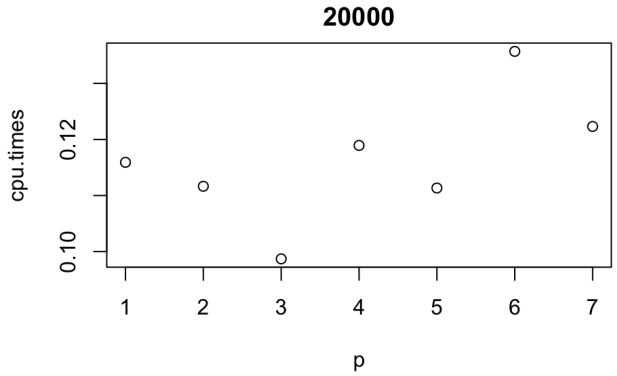
□

(b) Measure the CPU time for each run and estimate the total CPU time necessary to obtain an orbit which is periodic to 3-digit, 6-digit, and 12-digit accuracy.

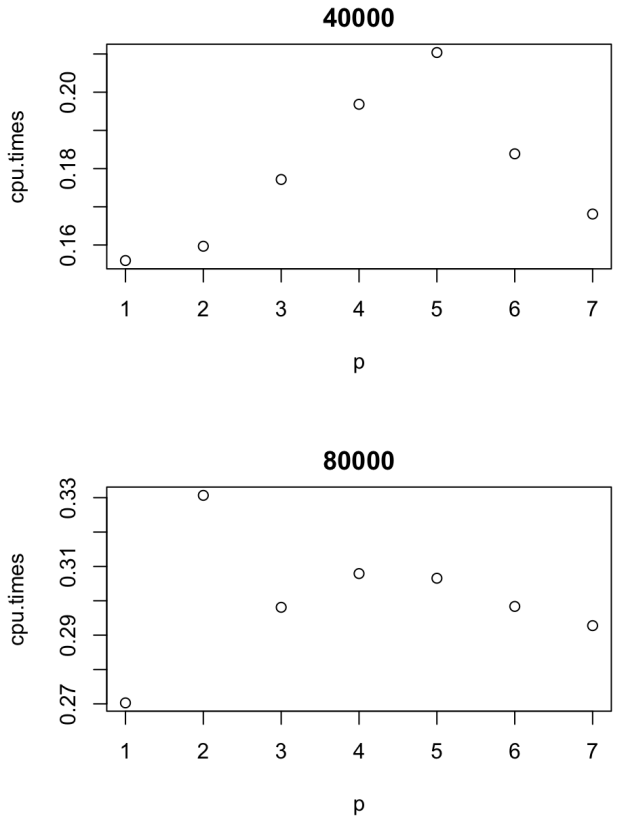
```
# measuring cpu time
# n.vec.cpu <- c(10000,20000,40000,80000)

# orbit
path <- matrix(data = NA, nrow = length(ya.7a), ncol = 7) # initialize path matrix
cpu.times <- c()
n <- 80000 # 10000, 20000, 40000, 80000
for (p in 1:7) {
# start clock
time.start <- Sys.time()
path <- idec(0, period.T, ya.7a, f.4b, r, p, n)
plot(path[1,], path[3,], main=n, type='l')
time.stop <- Sys.time()
cpu.times <- c(cpu.times, (time.stop - time.start) )
}
plot(cpu.times, main='cpu time')
cpu.times
```

Solution. The cpu time plots seem erratic but sometimes with a discernable trend. This may lead to an inaccurate estimation when combined with error estimates from (a).





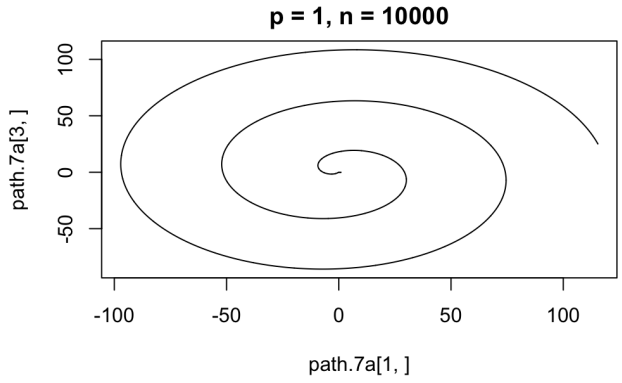
□

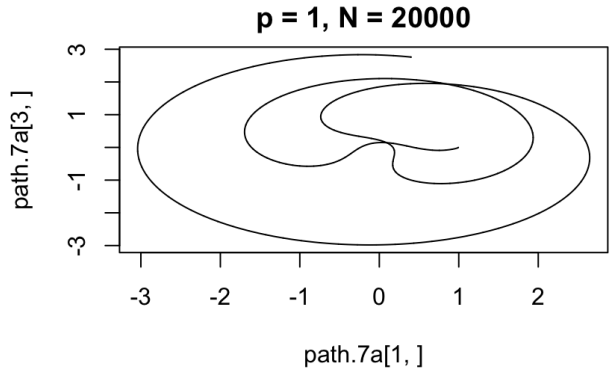


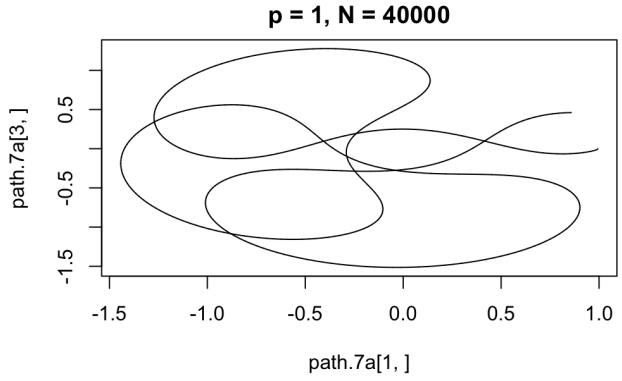
(c) Plot some inaccurate solutions and some accurate solutions, and draw conclusions about values of the order p which give 3-digit, 6-digit, or 12-digit accuracy for minimal CPU time.

```
n.7a <- c(10000, 20000, 40000, 80000)
ya.7a <- c(0.994, 0, 0, -2.00158510637908)
period.T <- 17.06521656015796
p.7a <- 1:7
orbit.idec <- idec(0, period.T, ya.7a, f.4b, r, p,7a[7], n.7a[4])
idec.final <- orbit.idec[,n.7a[1]]
plot(orbit.idec[1,], orbit.idec[3,], type='l', main='p = 7, N = 80000')
```

Solution. With lower values of p, N, we have quite inaccurate solutions. Not only will we fail to have 3-digit accuracy, but also in some cases we are completely off.







Now bringing p from 1 to 2 gives:
And at $p = 4$, we have seemingly identical results across the different N values

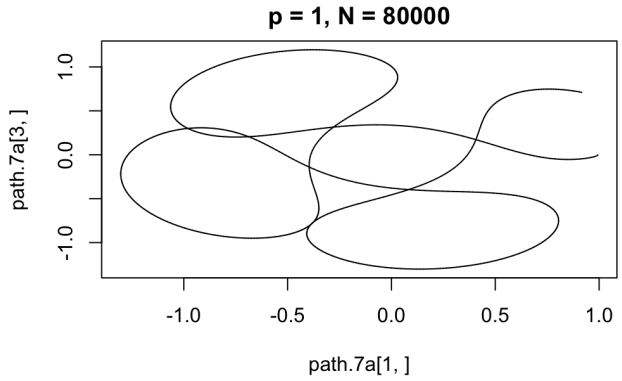
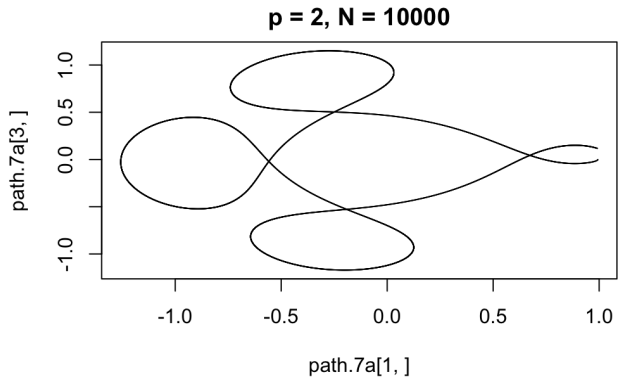
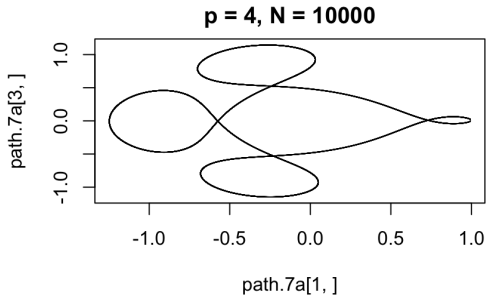


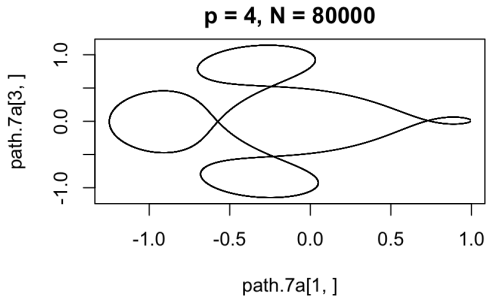
Figure 2: These plots verify part (a) where idec with $p = 1$ is equivalent to Euler (and equally as bad).

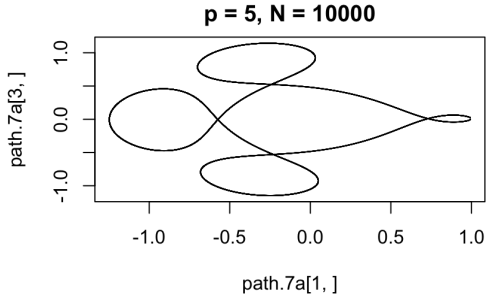


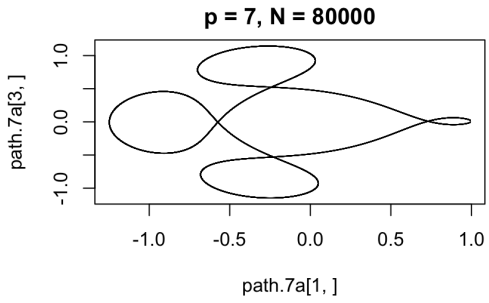


For $p = 5, 6, 7$, we get the same plot and are confident we are converging to the true solution.

□


p = 4, N = 80000
path.7a[3,]
1.0
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0
path.7a[1,]


p = 5, N = 10000
path.7a[3,]
1.0
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0
path.7a[1,]


p = 7, N = 80000
path.7a[3,]
1.0
0.0
-1.0
-1.0
-0.5
0.0
0.5
1.0
path.7a[1,]