

## Math 2280-2, Project #1, Problem #1

```
> restart:
```

We use table 2.1.11 for the world population data

```
> with(linalg): with(plots):  
> pops:=matrix(9,2,[[1960,3.049],[1965,3.358],[1970,3.721],[1975,  
4.103],[1980,4.473],[1985,4.882],[1990,5.249],[1995,5.679],  
[2000,6.127]]);  
#matrix of populations (billions)
```

```
>
```

```
pops := 
$$\begin{bmatrix} 1960 & 3.049 \\ 1965 & 3.358 \\ 1970 & 3.721 \\ 1975 & 4.103 \\ 1980 & 4.473 \\ 1985 & 4.882 \\ 1990 & 5.249 \\ 1995 & 5.679 \\ 2000 & 6.127 \end{bmatrix}$$

```

(1)

Here we compute a table with the population and the central differences approximation to the derivative

```
> for i from 1 to 7 do  
  lspoints[i]:=[pops[i+1,2],  
  (pops[i+2,2]-pops[i,2])/(10*pops[i+1,2])]:  
od:
```

```
> A:=matrix(7,2):  
B:=vector(7):  
> for i from 1 to 7 do  
  A[i,1]:=lspoints[i][1]:  
  A[i,2]:=1:  
  B[i]:=lspoints[i][2]:  
od:
```

```
> evalm(A);evalm(B);  
#check work
```

```

$$\begin{bmatrix} 3.358 & 1 \\ 3.721 & 1 \\ 4.103 & 1 \\ 4.473 & 1 \\ 4.882 & 1 \\ 5.249 & 1 \\ 5.679 & 1 \end{bmatrix}$$

```

```
[0.02001191185, 0.02002149960, 0.01832805264, 0.01741560474, 0.01589512495,  
0.01518384454, 0.01546046839]
```

(2)

```
> c:=linsolve(transpose(A)* (A) ,transpose(A)*B);
#least squares solution
```

```
c := [ -0.002379351595  0.02816897209 ]
```

(3)

```
> b:=c[1]; # slope
a:=c[2]; # ordinate at the origin
b := -0.002379351595
a := 0.02816897209
```

(4)

```
> pict1:=pointplot({seq(
  lspoints[i],i=1..7)}):
> line:=plot(a+b*P,P=2..6, color=black):
> display({pict1,line}, title="data fitting for logistic eq
parameters");
```

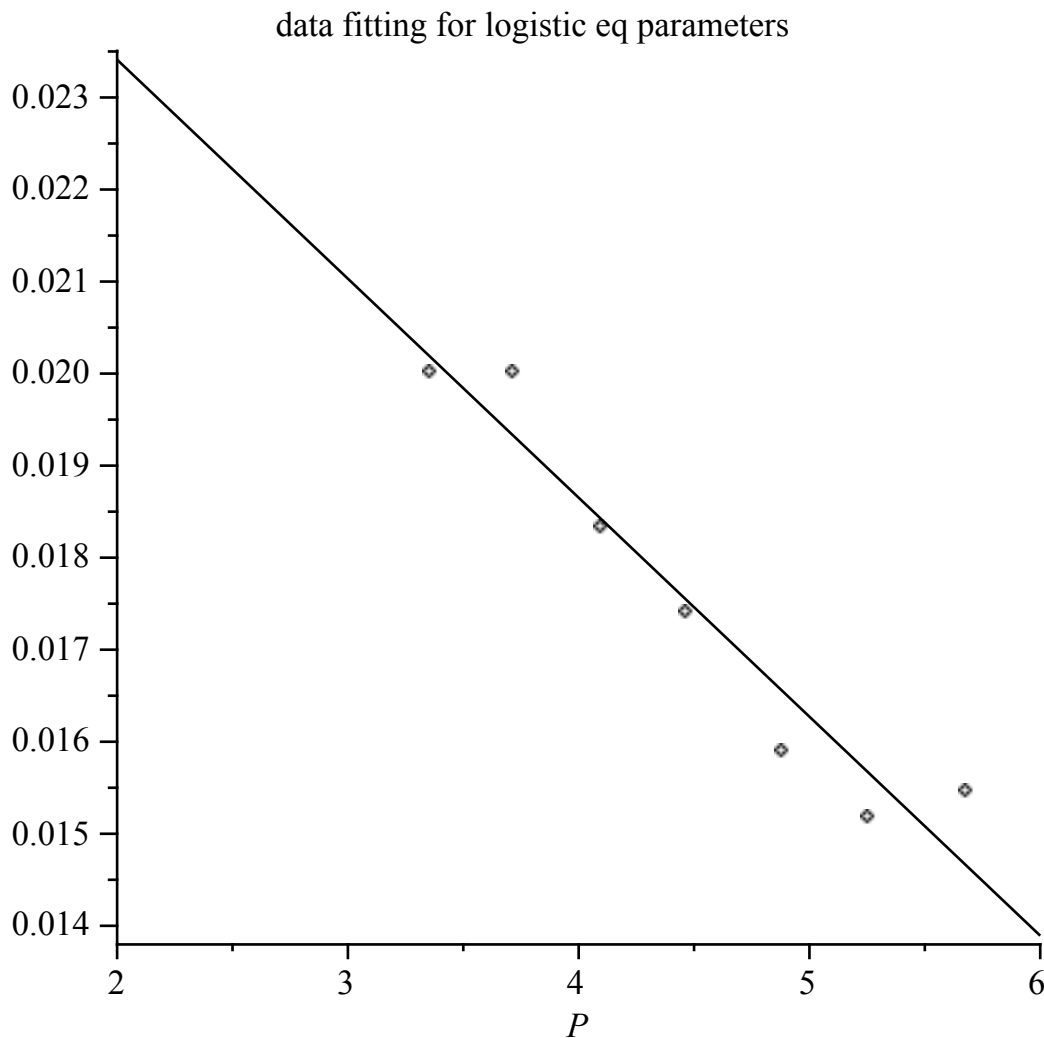


Figure 2.1.10 shows that the logistic model is a pretty good one for the U.S. population in the 1800's. Let's use the "a" and "b" which we found with the least squares fit, use the population in 1900 for our initial condition, and see how the logistic model works when we try to extend it to the 1900's:

```
> with(DEtools): #load the DE package
> deqtn1:=diff(x(t),t)=a*x(t) + b*x(t)^2;
#logistic eqtn with our parameters
```

(5)

$$\text{deqtn1} := \frac{d}{dt} x(t) = 0.02816897209 x(t) - 0.002379351595 x(t)^2 \quad (5)$$

```
> P:=dsolve({deqtn1,x(0)=5.679},x(t));
#take 1995 as t=0 and solve the initial value
#problem
```

$$P := x(t) = \frac{31994318499822}{2702467541601 + 2931326876399 e^{-\frac{2816897209}{100000000000} t}} \quad (6)$$

```
> f:=s->evalf(subs(t=s,rhs(P))); #extract the right-hand side
#from the above expression to make your solution function
```

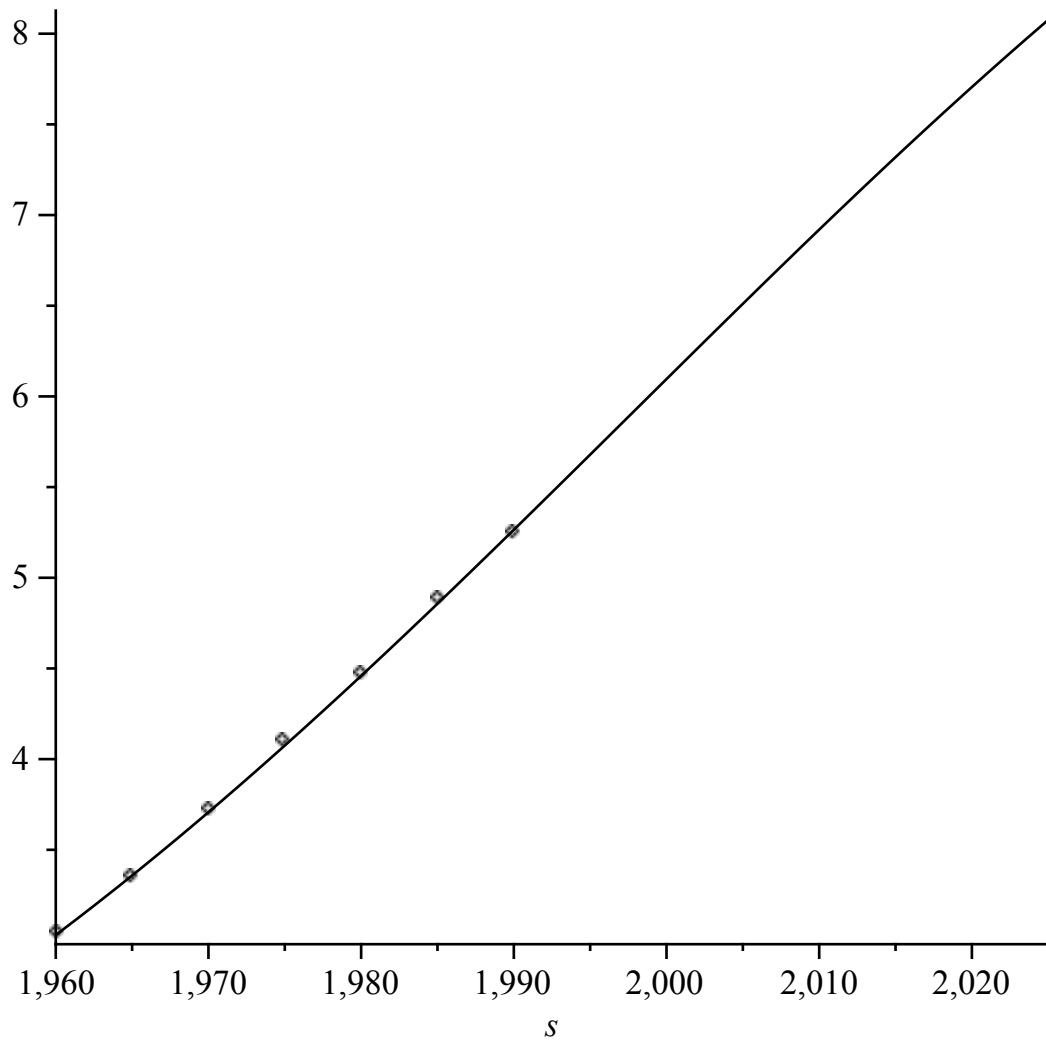
```
f:=s->evalf(subs(t=s,rhs(P))) (7)
```

```
> f(s); #check that those weird subs and rhs commands really
work
```

$$\frac{3.199431850 \cdot 10^{13}}{2.702467542 \cdot 10^{12} + 2.931326876 \cdot 10^{12} e^{-0.02816897209 s}} \quad (8)$$

We can see how the model works by plotting our model against the real data.

```
> actual:=pointplot({seq([pops[i,1],pops[i,2]],i=1..7)}):
model:=plot(f(s-1995),s=1960..2025,color=black):
display({actual,model});
```



We can now estimate the population in the year 2025 by using our model. Notice how we don't evaluate  $x(2025)$ , which would give us a wrong answer because the model so that  $t=0$  corresponds to 1995.

```
> printf("Estimated population in 2025 is %5.3f billion\n",f(2025  
-1995));  
Estimated population in 2025 is 8.076 billion
```

```
>
```

## Math 2280-2, Project #1, Problem 2

### (a) Improved Euler's method

```

> x0:=0; xn:=1.; # first and last points in the interval
  y0:=1; # initial condition
  n:=100; # number of steps
  h:=(xn-x0)/n; # step size

```

$x_0 := 0$

$x_n := 1.$

$y_0 := 1$

$n := 100$

$h := 0.010000000000$

(1)

```

> f:=(x,y)->y; # slope function (rhs in DE dy/dx = f(x,y), here f
  (x,y)=y)
  x:=x0; y:=y0;
  printf("%4s, %15s, %15s, %15s\n", "x_i", "y_i", "exp(x_i)", "error")
  ;
  for i from 1 to n do
    k1:= f(x,y): # left hand slope
    k2:= f(x+h,y+h*k1): # approximation to right hand slope
    k:=(k1+k2)/2: # averaged slope
    y:= y + h*k: # improved Euler update
    x:= x+h: # increase x
    if frac(i/10)=0 then
      printf("%4.1f, %15.10f, %15.10f, %15.10f\n",x,y,exp(x),abs(y-
  exp(x)));
    end if:
  end do: # i do loop

```

$f := (x, y) \rightarrow y$

$x := 0$

$y := 1$

| x_i, | y_i,          | exp(x_i),     | error        |
|------|---------------|---------------|--------------|
| 0.1, | 1.1051690900, | 1.1051709180, | 0.0000018280 |
| 0.2, | 1.2213987170, | 1.2214027580, | 0.0000040410 |
| 0.3, | 1.3498521080, | 1.3498588080, | 0.0000067000 |
| 0.4, | 1.4918148260, | 1.4918246980, | 0.0000098720 |
| 0.5, | 1.6487076330, | 1.6487212710, | 0.0000136380 |
| 0.6, | 1.8221007140, | 1.8221188000, | 0.0000180860 |
| 0.7, | 2.0137293870, | 2.0137527070, | 0.0000233200 |
| 0.8, | 2.2255114730, | 2.2255409280, | 0.0000294550 |
| 0.9, | 2.4595664900, | 2.4596031110, | 0.0000366210 |
| 1.0, | 2.7182368590, | 2.7182818280, | 0.0000449690 |

## (b) Runge Kutta

```
> x0:=0; xn:=1.; # first and last points in the interval
y0:=1; # initial condition
n:=20; # number of steps
h:=(xn-x0)/n; # step size

x0:=0
xn:=1.
y0:=1
n:=20
h:=0.050000000000

> f:=(x,y)->y; # slope function (rhs in DE dy/dx = f(x,y), here f
(x,y)=y)
x:=x0; y:=y0;
printf("%4s, %15s, %15s, %15s\n", "x_i", "y_i", "exp(x_i)", "error")
;
for i from 1 to n do
  k1:= f(x,y): # left hand slope
  k2:= f(x+h/2,y+h*k1/2): # midpoint slope: first approx.
  k3:= f(x+h/2,y+h*k2/2): # midpoint slope: second approx.
  k4:= f(x+h,y+h*k3): # right hand slope approx.
  k:=(k1+2*k2+2*k3+k4)/6: # Simpson's integration rule
  y:= y + h*k: # RK4 update
  x:= x+h: # increase x
  if frac(i/2)=0 then
    printf("%4.1f, %15.10f, %15.10f, %15.10f\n", x,y,exp(x),abs(y-
exp(x)));
  end if:
end do: # i loop
```

$f := (x, y) \rightarrow y$

$x := 0$

$y := 1$

| $x_i$ , | $y_i$ ,       | $\exp(x_i)$ , | error        |
|---------|---------------|---------------|--------------|
| 0.1,    | 1.1051709130, | 1.1051709180, | 0.0000000050 |
| 0.2,    | 1.2214027460, | 1.2214027580, | 0.0000000120 |
| 0.3,    | 1.3498587880, | 1.3498588080, | 0.0000000200 |
| 0.4,    | 1.4918246680, | 1.4918246980, | 0.0000000300 |
| 0.5,    | 1.6487212290, | 1.6487212710, | 0.0000000420 |
| 0.6,    | 1.8221187460, | 1.8221188000, | 0.0000000540 |
| 0.7,    | 2.0137526370, | 2.0137527070, | 0.0000000700 |
| 0.8,    | 2.2255408390, | 2.2255409280, | 0.0000000890 |
| 0.9,    | 2.4596030000, | 2.4596031110, | 0.0000001110 |
| 1.0,    | 2.7182816920, | 2.7182818280, | 0.0000001360 |

(c) With 100 points we get **5 digits** of precision with improved Euler's method and **2 digits** with plain Euler's method. With RK4 on 20 points we get **7 digits**.

## Math 2280-2, Project #1, Problem 3 (2.4.29 in textbook)

(a) We need to solve  $\frac{dy}{dx} + \frac{1}{7x}y = 0$  with initial condition  $y(-1) = 1$ .

We use separation of variables which gives:  $\int \frac{1}{y} dy = -\int \frac{1}{7x} dx + C$ , or equivalently

$\ln|y| = -\frac{1}{7}\ln|x| + C$ . Since near  $x = -1$  we have  $y > 0$  and  $x < 0$ , we get  $\ln(y) = -\frac{1}{7}\ln(-x) + C$ ,

whereby using the initial condition we get  $C = 0$ . Thus  $y(x) = \frac{1}{(-x)^{\frac{1}{7}}}$ . (the formula in the book is

incorrect, as a fractional power of a negative number gives an imaginary number).

(b) We use Euler's method with a step size that is too large ( $h=0.5$ ) and that misses the singularity of the solution

```

> restart:          # Clear Maple's internal memory
with(LinearAlgebra): with(plots):
> x0:=-1.; xn:=0.5; # first and last points in the interval
y0:=1;             # initial condition
n:=10;             # number of steps
h:=(xn-x0)/n;      # step size
                    x0 := -1.
                    xn := 0.5
                    y0 := 1
                    n := 10
                    h := 0.1500000000                                (1)
> f:=(x,y)-> -y/(7*x); # slope function (rhs in DE dy/dx = f(x,y))
)
x:=x0; y:=y0;      # initialize x,y
xs1:=Vector(n+1): ys1:=Vector(n+1): xs1[1]:=x0: ys1[1]:=y0:
printf("%15s, %15s, %15s\n", "x_i", "y_i", "Y(x_i)"):
for i from 1 to n do
  k:= f(x,y): # current slope
  y:= y + h*k: # new y value (Euler's method)
  x:= x+h:    # increase x
  printf("%15.2f, %15.10f, %15.10f\n", x,y,1/abs(x)^(1/7));
  xs1[i+1]:=x: ys1[i+1]:=y;
od: # end for i loop
>
                    f := (x,y) -> -1/7 * y/x
                    x := -1.
                    y := 1
                    x_i,          y_i,          Y(x_i)
-0.85,          1.0214285710,          1.0234886020
-0.70,          1.0471788710,          1.0522740280
-0.55,          1.0792353670,          1.0891583980
-0.40,          1.1212834980,          1.1398522810
-0.25,          1.1813522570,          1.2190136540
-0.10,          1.2826110220,          1.3894954940
 0.05,          1.5574562410,          1.5341274050

```

|       |               |              |
|-------|---------------|--------------|
| 0.20, | 0.8899749949, | 1.2584989510 |
| 0.35, | 0.7946205312, | 1.1618047200 |
| 0.50, | 0.7459702946, | 1.1040895140 |

(C) Using Eulers method with 50 points (h=0.03)

```
> x0:=-1.; xn:=0.5; # first and last points in the interval
y0:=1; # initial condition
n:=50; # number of steps
h:=(xn-x0)/n; # step size
x0 := -1.
xn := 0.5
y0 := 1
n := 50
h := 0.030000000000
f:=(x,y)->-y/(7*x); # slope function (rhs in DE dy/dx = f(x,y))
)
x:=x0; y:=y0; # initialize x,y
xs2:=Vector(n+1); ys2:=Vector(n+1); xs2[1]:=x0; ys2[1]:=y0;
printf("%15s, %15s, %15s\n", "x_i", "y_i", "Y(x_i)");
for i from 1 to n do
k:= f(x,y): # current slope
y:= y + h*k: # new y value (Euler's method)
x:= x+h: # increase x
if frac(i/5)=0 then
# display current values and compare to true sol.
printf("%15.2f, %15.10f, %15.10f\n",x,y,1/abs(x)^(1/7));
end if;
xs2[i+1]:=x; ys2[i+1]:=y;
od: # end for i loop
```

(2)

$$f:=(x,y) \rightarrow -\frac{1}{7} \frac{y}{x}$$

x := -1.

y := 1

| x_i,   | y_i,          | y(x_i)       |
|--------|---------------|--------------|
| -0.85, | 1.0230525210, | 1.0234886020 |
| -0.70, | 1.0511872810, | 1.0522740280 |
| -0.55, | 1.0870173480, | 1.0891583980 |
| -0.40, | 1.1357655930, | 1.1398522810 |
| -0.25, | 1.2103714660, | 1.2190136540 |
| -0.10, | 1.3612350040, | 1.3894954940 |
| 0.05,  | 1.8721276780, | 1.5341274050 |
| 0.20,  | 1.4711426700, | 1.2584989510 |
| 0.35,  | 1.3506362150, | 1.1618047200 |
| 0.50,  | 1.2807638240, | 1.1040895140 |

Euler's method with h=0.006 (n=250)

```
> x0:=-1.; xn:=0.5; # first and last points in the interval
y0:=1; # initial condition
n:=250; # number of steps
h:=(xn-x0)/n; # step size
x0 := -1.
xn := 0.5
y0 := 1
n := 250
```

```

                                h := 0.006000000000
> f:=(x,y)->-y/(7*x); # slope function (rhs in DE dy/dx = f(x,y)
)
x:=x0; y:=y0; # initialize x,y
xs3:=Vector(n+1); ys3:=Vector(n+1); xs3[1]:=x0; ys3[1]:=y0:
printf("%15s, %15s, %15s\n", "x_i", "y_i", "y(x_i)"):
for i from 1 to n do
  k:= f(x,y): # current slope
  y:= y + h*k: # new y value (Euler's method)
  x:= x+h: # increase x
  if frac(i/25)=0 then
    # display current values and compare to true sol.
    printf("%15.2f, %15.10f, %15.10f\n",x,y,1/abs(x)^(1/7));
  end if;
  xs3[i+1]:=x: ys3[i+1]:=y;
od: # end for i loop

```

$$f := (x, y) \rightarrow -\frac{1}{7} \frac{y}{x}$$

$$x := -1.$$

$$y := 1$$

| $x_i$ , | $y_i$ ,       | $y(x_i)$     |
|---------|---------------|--------------|
| -0.85,  | 1.0234003870, | 1.0234886020 |
| -0.70,  | 1.0520538520, | 1.0522740280 |
| -0.55,  | 1.0887235900, | 1.0891583980 |
| -0.40,  | 1.1390189010, | 1.1398522810 |
| -0.25,  | 1.2172352250, | 1.2190136540 |
| -0.10,  | 1.3834694200, | 1.3894954940 |
| 0.05,   | 1.0084717740, | 1.5341274050 |
| 0.20,   | 0.8210324540, | 1.2584989510 |
| 0.35,   | 0.7571469171, | 1.1618047200 |
| 0.50,   | 0.7192300717, | 1.1040895140 |

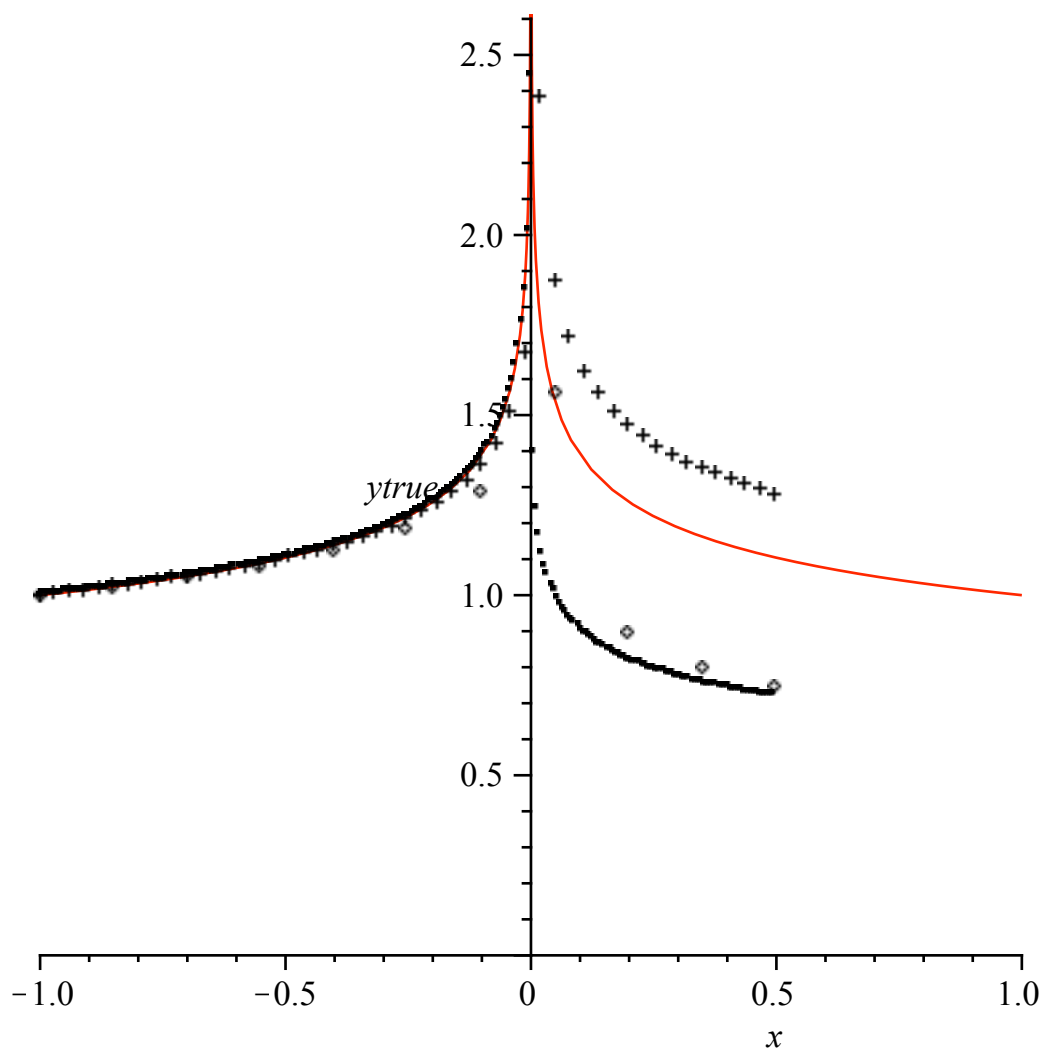
>

We can guess that there is something wrong with the solution since after crossing zero, the approximations do not agree. This is easier to see by plotting the solution and its approximations (this was not required). The true solution blows up at the origin, but Euler's method continues oblivious of the presence of this singularity, because the step size was too large! The difference between the approximations is also more striking in the plot than in the listings.

```

> unassign('x','y'): # we've used these variables before, forget
  about them
ytrue:=x->1/abs(x)^(1/7):
p1:=pointplot(Matrix([xs1,ys1]),symbol=diamond):
p2:=pointplot(Matrix([xs2,ys2]),symbol=cross):
p3:=pointplot(Matrix([xs3,ys3]),symbol=point):
p4:=plot(ytrue(x),x=-1..1,ytrue=0..2.6):
display({p1,p2,p3,p4});

```



## Math 2280-2, Project #1, Problem 4

The objective of this problem is to implement a different second order method, called Runge Kutta of order 2.

### Improved Euler's method

```
> restart:          # Clear Maple's internal memory
with(LinearAlgebra): with(plots): Digits:=16:
```

We will now execute improved Euler's method for several values of n and see the error for y(1) with respect to the true value which is pi.

```
> f:=(x,y)->4/(1+x^2); # slope function => solution to y'(x)=f
(x,y) is y(x) 4*atan(x)
x0:=0; xn:=1.; # first and last points in the interval
y0:=0; # initial conditional
x:=x0; y:=y0; # initialize x,y
printf("%15s,%15s,%15s,%15s\n",
      "n", "y", "true", "error"); # fixed column width
printing
mmax:=7: # number of
experiments to do
errors:=Vector(mmax): ns:=Vector(mmax): # vectors to store the
errors and values of n
for m from 1 to mmax do # loop over
experiments
  n:=2^m: # the number of points
for this experiment
  h:= (xn-x0)/n:
  x:=x0: y:=y0:
  for i from 1 to n do
    k1:= f(x,y): # left hand slope
    k2:= f(x+h,y+h*k1): # approximation to right hand slope
    k:= (k1+k2)/2: # averaged slope
    y:= y+k*h: # improved Euler update
    x:= x+h: # increase x
  od: # end for i loop
  printf("%15f,%15.10f,%15.10f,%15.10f\n",
      n,y,evalf(Pi),abs(y-evalf(Pi))): # fixed column width
printing
  errors[m]:=abs(y-evalf(Pi)): ns[m]:=n: # store errors and ns
for later use
od: # end for m loop
```

>

$$f := (x, y) \rightarrow \frac{4}{1 + x^2}$$

x0 := 0

xn := 1.

y0 := 0

x := 0

y := 0

| n,         | y,             | true,         | error        |
|------------|----------------|---------------|--------------|
| 2.000000,  | 3.10000000000, | 3.1415926536, | 0.0415926536 |
| 4.000000,  | 3.1311764706,  | 3.1415926536, | 0.0104161830 |
| 8.000000,  | 3.1389884945,  | 3.1415926536, | 0.0026041591 |
| 16.000000, | 3.1409416120,  | 3.1415926536, | 0.0006510415 |
| 32.000000, | 3.1414298932,  | 3.1415926536, | 0.0001627604 |

```

64.000000, 3.1415519635, 3.1415926536, 0.0000406901
128.000000, 3.1415824811, 3.1415926536, 0.0000101725

```

Instead of having the table we plot these numbers in a log-log scale and we fit the log-log data with a line (similar to what was done in the US population example)

```

> errplot1:=pointplot(map(log,Matrix([ns,errors]))): # error
plot for future use
> A:=Matrix([map(log,evalf(ns)),Vector(mmax,1.)]):
b:=map(log,errors):
> alpha1:=LinearSolve(Transpose(A). A,Transpose(A). b): # alpha=
[slope,intercept]
> errfitplot1:=
plot(alpha1[1]*t+alpha1[2],t=log(2)..log(2^mmax)): # linear
plot for future use

```

## RK2

```

> x:=x0; y:=y0; # initialize x,y
printf("%15s,%15s,%15s,%15s\n",
      "n","y","true","error"): # fixed column width
printing
mmax:=7: # number of
experiments to do
errors:=Vector(mmax): ns:=Vector(mmax): # vectors to store the
errors and values of n
for m from 1 to mmax do # loop over
experiments
n:=2^m: # the number of points
for this experiment
h:= (xn-x0)/n:
x:=x0: y:=y0:
for i from 1 to n do
k1:= f(x,y): # left hand slope
k2:= f(x+h/2,y+h*k1/2): # approximation to right hand slope
y:= y + h*k2: # improved Euler update
x:= x+h: # increase x
od: # end for i loop
printf("%15f,%15.10f,%15.10f,%15.10f\n",
      n,y,evalf(Pi),abs(y-evalf(Pi))): # fixed column width
printing
errors[m]:=abs(y-evalf(Pi)): ns[m]:=n: # store errors and ns
for later use
od: # end for m loop

```

```

>
x:=0
y:=0
n, y, true, error
2.000000, 3.1623529412, 3.1415926536, 0.0207602876
4.000000, 3.1468005184, 3.1415926536, 0.0052078648
8.000000, 3.1428947296, 3.1415926536, 0.0013020760
16.000000, 3.1419181743, 3.1415926536, 0.0003255207
32.000000, 3.1416740338, 3.1415926536, 0.0000813802
64.000000, 3.1416129986, 3.1415926536, 0.0000203451
128.000000, 3.1415977399, 3.1415926536, 0.0000050863

```

Again we fit the log-log data with a line.

```

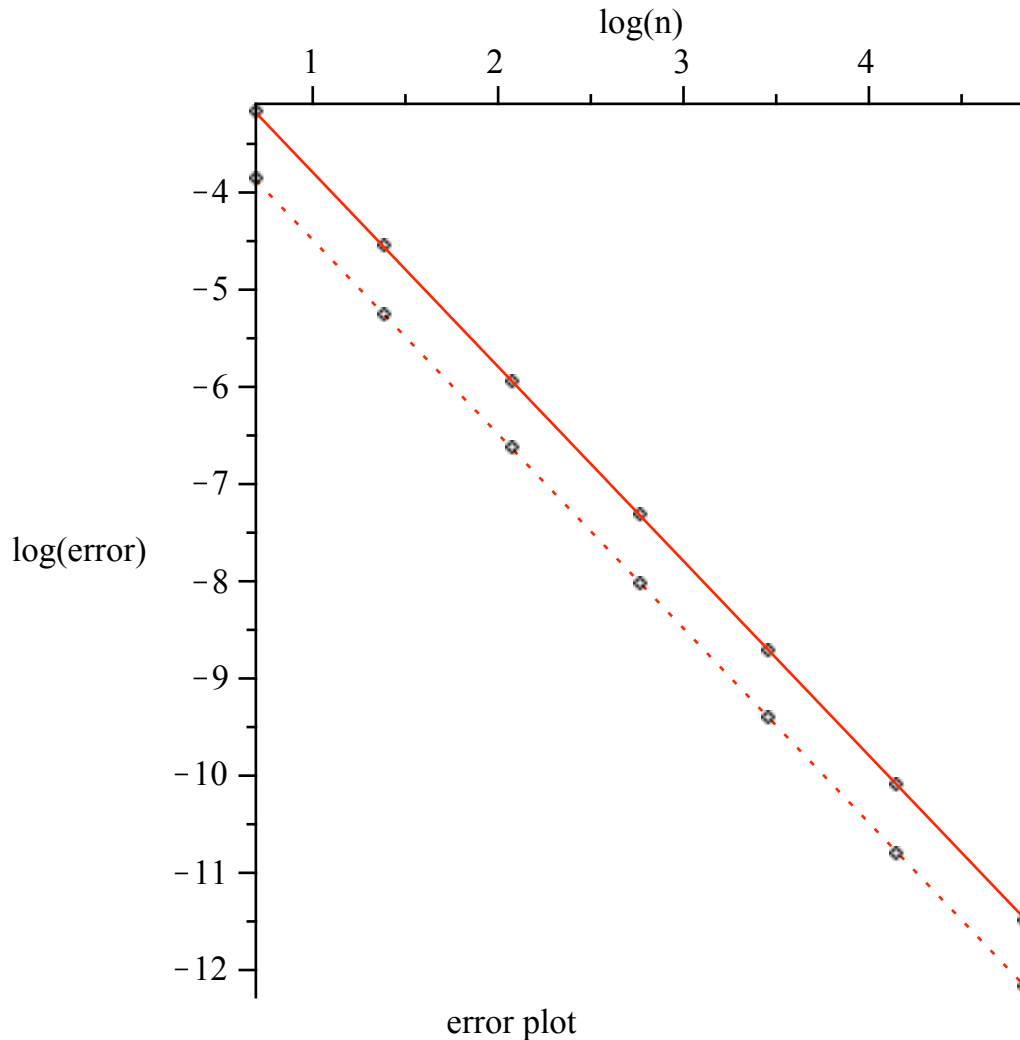
> errplot2:=pointplot(map(log,Matrix([ns,errors]))): # error
plot for future use
> A:=Matrix([map(log,evalf(ns)),Vector(mmax,1.)]):
b:=map(log,errors):
> alpha2:=LinearSolve(Transpose(A). A,Transpose(A). b): # alpha=

```

```
[slope,intercept]
> errfitplot2:=
  plot(alpha2[1]*t+alpha2[2],t=log(2)..log(2^mmax),linestyle=
dot): # linear plot for future use
```

(b) Plot of both methods compared (Euler: full line, RK2: dotted line)

```
> display({errfitplot1,errplot1,errfitplot2,errplot2},caption=
"error plot",labels=["log(n)","log(error)"]);
```



(c) We look at the slopes of the lines we fitted through the error plots for both methods:

```
> alpha2[1], alpha1[1];
-1.999447519927240, -1.999720249100162
```

(1)

since both are very close to -2, we can say that RK2 is a second order accurate method, as improved Euler's method.

## Math 2280-2, Project #1, Problem 5

We first solve the crossbow example 2.3.3 using improved Euler's method with  $n=100$

```
> with(LinearAlgebra): with(plots): Digits:=16:
> t0:=0.; tn:=10.; # first and last points in the interval
v0:=49.; # initial velocity
n:=100; # number of steps
h:=(tn-t0)/n; # step size
f:=(t,v)-> -(0.0011)*v*abs(v) -9.8; # slope function (rhs in DE
dy/dx = f(x,y))
v:=v0: t:=t0:
ts1:=Vector(n+1): vs1:=Vector(n+1): ts1[1]:=t0: vs1[1]:=v0:
printf("%15s, %15s\n", "t", "v"):
for i from 1 to n do
  k1:= f(t,v): # left hand slope
  k2:= f(t+h,v+h*k1): # approximation to right hand slope
  k:=(k1+k2)/2: # averaged slope
  v:= v + h*k: # improved Euler update
  t:= t+h: # increase x
  ts1[i+1]:=t: vs1[i+1]:=v:
  if frac(i/10)=0 then
    printf("%15.1f, %15.5f\n", t,v):
  end if:
od: # end for i loop
```

$t_0 := 0.$

$t_n := 10.$

$v_0 := 49.$

$n := 100$

$h := 0.100000000000000000$

$f := (t, v) \rightarrow -0.0011 v |v| - 9.8$

| t,    | v         |
|-------|-----------|
| 1.0,  | 37.15469  |
| 2.0,  | 26.24277  |
| 3.0,  | 15.94529  |
| 4.0,  | 6.00406   |
| 5.0,  | -3.80200  |
| 6.0,  | -13.51045 |
| 7.0,  | -22.93557 |
| 8.0,  | -31.89841 |
| 9.0,  | -40.25568 |
| 10.0, | -47.90661 |

And now with 200 points

```
> t0:=0.; tn:=10.; # first and last points in the interval
v0:=49.; # initial velocity
n:=200; # number of steps
h:=(tn-t0)/n; # step size
f:=(t,v)-> -(0.0011)*v*abs(v) -9.8; # slope function (rhs in DE
dy/dx = f(x,y))
v:=v0: t:=t0:
ts2:=Vector(n+1): vs2:=Vector(n+1): ts2[1]:=t0: vs2[1]:=v0:
printf("%15s, %15s\n", "t", "v"):
for i from 1 to n do
  k1:= f(t,v): # left hand slope
  k2:= f(t+h,v+h*k1): # approximation to right hand slope
  k:=(k1+k2)/2: # averaged slope
  v:= v + h*k: # improved Euler update
```

```

t:= t+h: # increase x
ts2[i+1]:=t: vs2[i+1]:=v:
if frac(i/20)=0 then
  printf("%15.1f, %15.5f\n",t,v):
end if:
od: # end for i loop

t0 := 0.
tn := 10.
v0 := 49.
n := 200

h := 0.05000000000000000000

f := (t, v) → -0.0011 v |v| - 9.8

t, v
1.0, 37.15474
2.0, 26.24292
3.0, 15.94555
4.0, 6.00444
5.0, -3.80159
6.0, -13.51019
7.0, -22.93545
8.0, -31.89845
9.0, -40.25588
10.0, -47.90696

```

We see that the two approximations agree with each other to two digits.

Here is a plot of both along with the true solution (from the file crossbow.mw that we used in class). All are indistinguishable!

```

> rho2:=0.0011: g:=9.8:
C1:=arctan(v0*sqrt(rho2/g)): C2:=arctanh(v0*sqrt(rho2/g)):
tmax1:=C1/sqrt(rho2*g): tmax2:=C2/sqrt(rho2*g):
v3:=t->piecewise(t<tmax1,sqrt(g/rho2)*tan(C1-t*sqrt(rho2*g)),t>=
tmax1,sqrt(g/rho2)*tanh(C2-(tmax2+t-tmax1)*sqrt(rho2*g)));

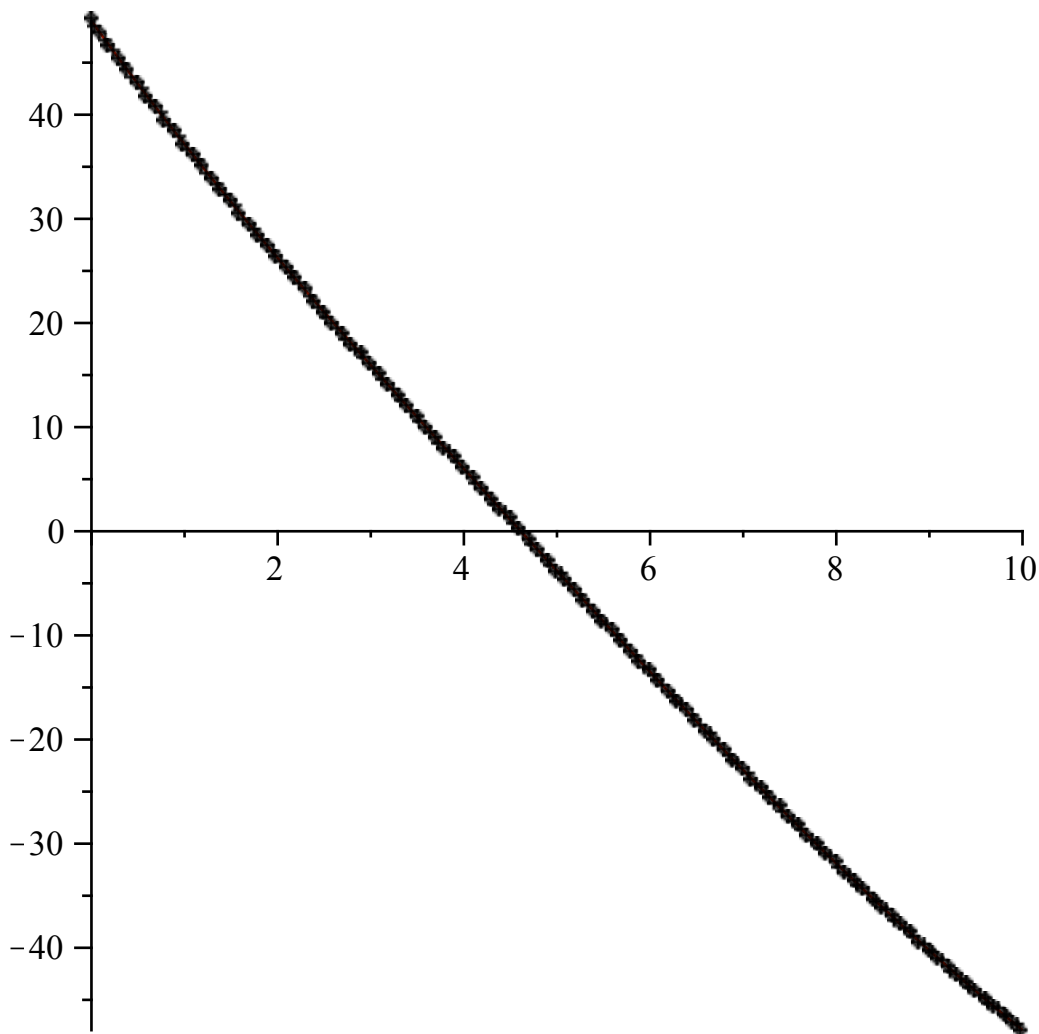
```

$$v3 := t \rightarrow \text{piecewise} \left( t < t_{max1}, \sqrt{\frac{g}{\rho_2}} \tan(C1 - t \sqrt{\rho_2 g}), t_{max1} \leq t, \sqrt{\frac{g}{\rho_2}} \tanh(C2 - (t_{max2} + t - t_{max1}) \sqrt{\rho_2 g}) \right) \quad (1)$$

```

> unassign('t'):
p1:=pointplot(Matrix([ts1,vs1]),symbol=diamond):
p2:=pointplot(Matrix([ts2,vs2]),symbol=cross):
p3:=plot(v3(t),t=0..10):
display({p1,p2,p3});

```



(a) to obtain the value where  $v=0$ , we run improved Euler's method and select the time for which  $v$  is closest to 0. Since all computations are already stored in a vector here, we can compute the minimum in Maple (as done below). Normally (in higher dimensional problems) you do not want to store the value of your solution at all points, so the numerical method's loop could be modified to find the time for which  $v$  is minimum (with no additional memory cost).

```
> # find the index of the minimum value in the vector vs2
res:=rtable_scanblock( abs(vs2), [rtable_dims(vs2)], (val, ind,
res) -> `if`(val<res[2],[ind,val],res), [[1],abs(vs2[1])]);
printf("approximation = %4.2f seconds\n",ts2[res[1][1]]);
```

```
>
```

```
res := [[93], 0.1163332210318962]
```

```
approximation = 4.60 seconds
```

(b) To estimate the impact velocity after 9.41 seconds, we could use improved Euler with  $t_n=9.41$ , to get:

```
> t0:=0.; tn:=9.41; # first and last points in the interval
v0:=49.; # initial velocity
n:=200; # number of steps
h:=(tn-t0)/n; # step size
f:=(t,v)-> -(0.0011)*v*abs(v) -9.8; # slope function (rhs in DE
dy/dx = f(x,y))
v:=v0: t:=t0:
```

```
for i from 1 to n do
  k1:= f(t,v): # left hand slope
  k2:= f(t+h,v+h*k1): # approximation to right hand slope
  k:=(k1+k2)/2: # averaged slope
  v:= v + h*k: # improved Euler update
  t:= t+h: # increase x
```

```
od: # end for i loop
```

```
printf("v( %4.2f ) approx. equals to %4.2f m/s\n",t,v);
```

```
    t0:=0.
```

```
    tn:=9.41
```

```
    v0:=49.
```

```
    n:=200
```

```
    h:=0.047050000000000000
```

```
    f:=(t,v)→-0.0011 v|v| - 9.8
```

```
v( 9.41 ) approx. equals to -43.48 m/s
```

```
[>
```

## Math 2280-2, Project #1, Problem #6

This problem shows that even an accurate numerical method like Runge-Kutta 4, is susceptible to problems that are unstable.

```
> x0:=0.; xn:=4.; # first and last points in the interval
y0:=1; # initial condition
n:=80; # number of steps
h:=(xn-x0)/n; # step size

x0 := 0.
xn := 4.
y0 := 1
n := 80
h := 0.050000000000
```

(1)

```
> x:=x0; y:=y0;
f:=(x,y)->5*y - 6*exp(-x); # slope function (rhs in DE dy/dx = f
(x,y))
printf("%15s, %15s, %15s\n", "x_i", "y_i", "y(x_i)");
for i from 1 to n do
  k1:= f(x,y): # left hand slope
  k2:= f(x+h/2,y+h*k1/2): # midpoint slope: first approx.
  k3:= f(x+h/2,y+h*k2/2): # midpoint slope: second approx.
  k4:= f(x+h,y+h*k3): # right hand slope approx.
  k:=(k1+2*k2+2*k3+k4)/6: # Simpson's integration rule
  y:= y + h*k: # RK4 update
  x:= x+h: # increase x
  if frac(i/8)=0 then
    printf("%15.2f, %15.10f, %15.10f\n", x,y,exp(-x));
  end if;
od: # end for i loop

x := 0.
y := 1
```

$$f := (x, y) \rightarrow 5y - 6e^{-x}$$

| x_i,  | y_i,             | y(x_i)       |
|-------|------------------|--------------|
| 0.40, | 0.6703113029,    | 0.6703200460 |
| 0.80, | 0.4492585034,    | 0.4493289641 |
| 1.20, | 0.3006696723,    | 0.3011942119 |
| 1.60, | 0.1980182369,    | 0.2018965180 |
| 2.00, | 0.1066781969,    | 0.1353352832 |
| 2.40, | -0.1210208487,   | 0.0907179533 |
| 2.80, | -1.5036578640,   | 0.0608100626 |
| 3.20, | -11.5185681600,  | 0.0407622040 |
| 3.60, | -85.3806995000,  | 0.0273237224 |
| 4.00, | -631.0329796000, | 0.0183156389 |

>