Sakari Cajanus
82036R
sakari.cajanus@aalto.fi

Exercise Round 1

S-114.1100 Computational Science

September 18, 2011

Problem 1. Plots and conclusions

The function

$$(a) \qquad f_1(x) = \frac{e^x - 1}{x}$$

has potential loss of significance near $x = 0$. This is because the exponential function e^x has limit

$$\lim_{x \to 0} e^x = 1$$

and we end up substracting almost equal values. The error is further amplified by the division by small x. For example, when $x = 1 \times 10^{-3}$

$$e^x = 1.001000500166708...$$

Now we have from the substraction $e^x - 1$

$$1 - \frac{1}{e^x} = 0.000999500166625...$$

This lies between values $2^{-9} = 0.001953125$ and $2^{-10} = 0.0009765625$ so at least 9 but at most 10 bits are lost. It is also worth pointing out that for x near zero, the series expansion

$$\frac{e^x - 1}{x} = 1 + \frac{x}{2!} + \frac{x^2}{3!} + \frac{x^3}{4!} + \ldots$$

converges to 1 quite rapidly: to avoid loss of significance, this is what we should use. In

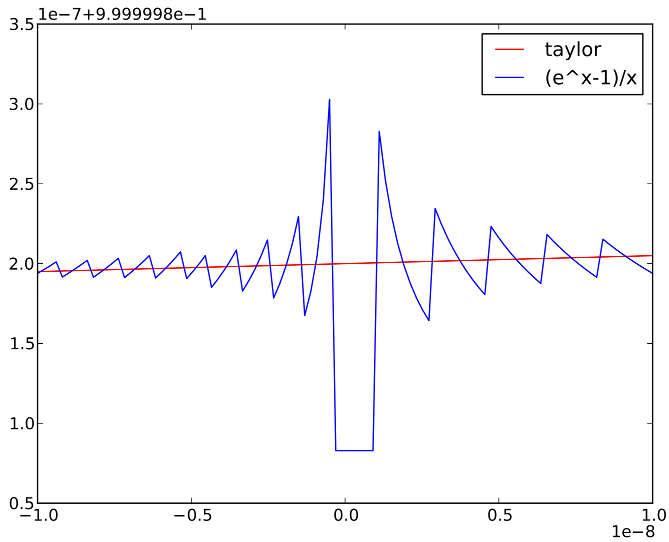

Figure 1: Approximation by 4th degree Taylor polynomial and function itself.

Figure 1 you can see both the Taylor approximation $(n = 4)$ and function values with the values calculated in 100 points between $x = -1 \times 10^{-9}$ and $x = 1 \times 10^{-9}$. The behaviour of the function when the approximation is not used is heavily dependant on which points it's values are calculated: If we'd choose different number of points or different interval, the function would look quite different.

Estimate of the error of the Taylor approximation is given by the $(n+1)$th term in the series. In this case, for $x = 1 \times 10^{-9}$, it is

$$R(n+1) = \frac{x^4}{5!} = 8.333\ldots \times 10^{-39}.$$

It is also worth mentioning that python has built-in function *expm1(x)* that is meant to be used for calculating $e^x - 1$ when $x < \log 2$.

With the function

$$(b) \qquad f_2(x) = \frac{e^x - e^{-x}}{2x}$$

we face the similar problem. Both terms in the numerator converge to one as x approaches zero:

$$\lim_{x \to 0} e^x = 1$$
$$\lim_{x \to 0} e^{-x} = 1$$

As before, we can write the function as a series expansion

$$\frac{e^x - e^{-x}}{2x} = 1 + \frac{x^2}{3!} + \frac{x^4}{5!} + \frac{x^6}{7!} + \ldots$$

Using just the first two terms $(n = 4)$, as before, we get following figure 2 as x get values between $x = -1 \times 10^{-6}$ and $x = -1 \times 10^{-6}$.

We can calculate the error estimate of the approximation at $x = 1 \times 10^{-6}$ using the next term in series

$$R(n+1) = \frac{x^4}{5!} = 8.333\ldots \times 10^{-27}.$$

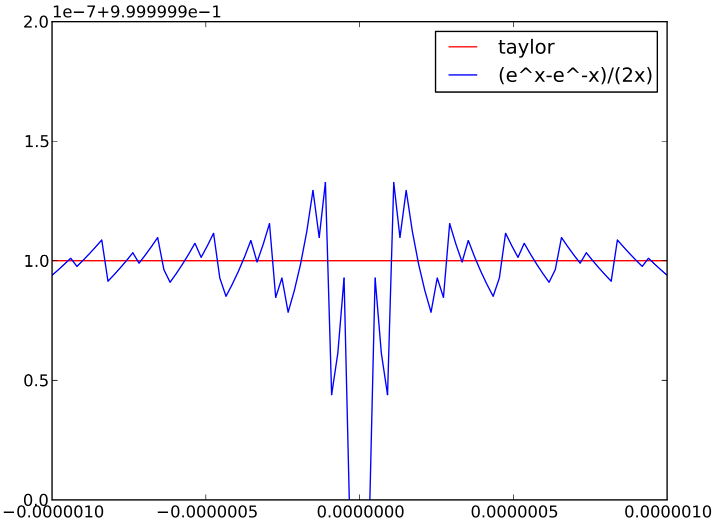

Figure 2: Approximation by 4th degree Taylor polynomial and function itself.

Problem 2. Plots and conclusions

i	x	f(x)
0	0.577350	-5.384900
1	-5774691.304827	-192568975495066910720.0
2	-3849794.203218	-57057474220760129536.0
3	-2566529.468812	-16905918287632351232.0
4	-1711019.645875	-5009160974113097728.0
5	-1140679.763917	-1484195844181532416.0
	...	
44	-1.762824	-8.715240
45	-0.715653	-4.650875
46	7.953624	490.193685
47	5.356989	143.374340
48	3.672056	40.841946
49	2.636825	10.696613
50	2.098184	2.138817

As a result of very bad starting point choice, the first calculated values of the iteration for both x and the function value $f(x)$ are very small. This is a result of the derivate being close to zero, $-9.32499999884 \times 10^{-07}$. In fact, 50 steps are not sufficient to get to the root after this detour. Function $x^3 - x - 5$ and its tangent at start point are shown in Figure 3. Learning from this, we improve our algorithm by making it use bisection method if the

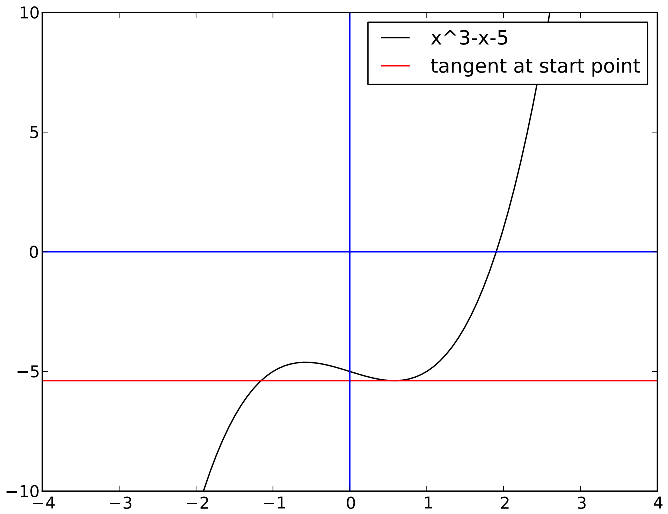

Figure 3: function $x^3 - x - 5$ and our very bad starting point choice.

new x value calculated with Newton's method would take us out of certain bounds, where the root is known to reside. In this case, bounds $a = 0$ and $b = 3$ were used. Here is the output:

```
i                  x               f(x)
0           0.577350          -5.384900
1           1.500000          -5.000000
2           2.369565           5.935163
3           1.994977           0.944903
4           1.908605           0.044005
5           1.904172           0.000112
6           1.904161           0.000000
Found root at 1.904161
```

Nice and quick!

Problem 3. Plots and conclusions

In problem 3 we were asked to examine basins of attraction of three roots in complex plane. The complex polynomial

$$z^3 - 1$$

has three roots:

$$\begin{aligned} z &= 1 \\ z &\approx -0.5 + 0.866025i \\ z &\approx -0.5 - 0.866025i. \end{aligned}$$

These three roots were assigned a different color, and pixels in a 1000×1000 square containing region $-1 \leq \mathrm{Real}(z) \leq 1$ and $-1 \leq \mathrm{Imag}(z) \leq 1$ were assigned this same color if the function starting from the point would reach the root in 100 iterations. Figure 4 shows the results.

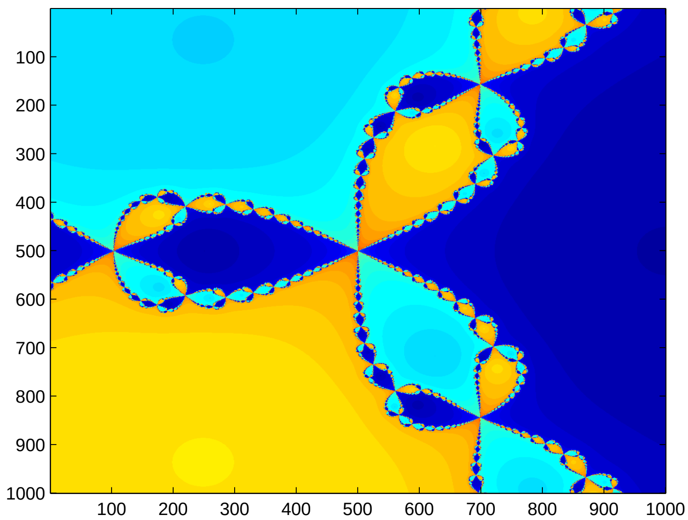

Figure 4: Three basins of attraction featuring ugly colors.

Appendix 1. Code

```
from __future__ import division
from pylab import *

def f1_taylor(x): #e^x subsituted by series expansion of 4th degree
    return 1 + x/math.factorial(2) + x**2/math.factorial(3)\
            + x**3/math.factorial(4)

def f2_taylor(x): #e^x subsituted by series expansion of 4th degree
    return 1 + x**2/math.factorial(3)

def main():
    print "Let's evaluate!"
    x = linspace(-1*10**-8, 1*10**-8, 100)

    f1_taylorv = f1_taylor(x)
    f1_without_taylor = (exp(x)-1)/x
    f2_taylorv = f2_taylor(x)
    f2_without_taylor = (exp(x)-exp(-x))/(2*x)

    plot(x, f1_taylorv, 'r')
    plot(x, f1_without_taylor, 'b')
    xlim(min(x),max(x))
    legend(('taylor', '(e^x-1)/x'))
    show()

    x = linspace(-1*10**-6, 1*10**-6, 100)
    plot(x, f2_taylorv, 'r')
    plot(x, f2_without_taylor, 'b')
    xlim(min(x),max(x))
    ylim(.9999999,1.0000001)
    legend(('taylor', '(e^x-e^-x)/(2x)'))
    show()

if __name__ == "__main__":
    main()
```

Appendix 2. Code

Appendix 2..1 Only Newton

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXITER 50

double func(double x);
double func_prime(double x);
int main(){
        double x = 0.57735;
        double mindelta = 0.00000001;
        double eps = 0.000001;
        double d = 0;
        int iter=0;
        double fp=1;
        printf("Hello!\n");
        double fx = func(x);
        printf("i                x                f(x)\n");

        for(iter=0; iter<=MAXITER; iter++){
                printf("%d      %10.6f  %12.6f\n",iter,x,fx);
                fp=func_prime(x);
                if(fabs(fp) <= mindelta){
                        printf("Error: derivative too small\n");
                        exit(1);
                }
                d=fx/fp;
                x=x-d;
                if(fabs(d) <= eps){
                        printf("Found root at %f\n", x);
                        break;
                }
                fx=func(x);
        }
}

double func(double x){ return x*x*x-x-5; }
double func_prime(double x){ return 3*x*x-1; }
```

Appendix 2..2 A (hacky) solution using Newton/bisection hybrid method

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXITER 50
#define A 0
#define B 3

double func(double x);
double func_prime(double x);
int main(){
        double x = 0.57735;
        double mindelta = 0.00000001;
        double eps = 0.000001;
        double d = 0;
        int iter=0;
        double fp=1;
        printf(”Hello!\n”);
        double fx = func(x);

        int useNewton=1;
        int a = A;
        int b = B;
        int u = func(a);
        int v = func(b);


        printf(”i               x                f(x)\n”);

        for(iter=0; iter<=MAXITER; iter++){
                if(useNewton==1){
                        printf(”%d      %10.6f  %12.6f\n”,iter,x,fx);
                        fp=func_prime(x);
                        if(fabs(fp) <= mindelta){
                                printf(”Error: derivative too small\n”);
                                exit(1);
                        }
                        d=fx/fp;
                        if(fabs(d) <= eps){
                                printf(”Found root at %f\n”, x);
                                break;
                        }
                        if((x-d)<=B && (x-d)>=A){
```

```
				x=x-d;
				fx=func(x);
			}
			else{
				useNewton=0;
				iter-=1;
			}
		} else {
			if(b-a <= mindelta) break;
			x = 0.5*(a+b);

			fx = func(x);
			if(fabs(fx) <= eps) break;
			if(fx*u < 0) {
				b = x;
				v = fx;
			}
			else {
				a = x;
				u = fx;
			}
			useNewton=1;
		}
	}
}

double func(double x){ return x*x*x-x-5; }
double func_prime(double x){ return 3*x*x-1; }
```

Appendix 3. Code

```
NITER = 100;
threshold = .00000001;
z1=1;
z2=-.5-0.86602540378443864676i;
z3=-.5+0.86602540378443864676i;

[xx,yy] = meshgrid(linspace(-1,1,1000), linspace(-1,1,1000));

solutions = xx(:) + 1i*yy(:);
select = 1:length(solutions);
niters = NITER*ones(numel(xx), 1);
which_root = zeros(numel(xx), 1);

for iteration = 1:NITER
 z = solutions(select);

 solutions(select) = z - ((z.^2).*z - 1) ./ (3*z.^2);

 differ = (z - solutions(select));
 converged = abs(differ) < threshold;
 problematic = isnan(differ);

 niters(select(converged)) = iteration;
 niters(select(problematic)) = NITER+1;
 select(converged | problematic) = []; % drop solved
end

which_root(abs(solutions-z1)<0.00001)=1;
which_root(abs(solutions-z2)<0.00001)=20;
which_root(abs(solutions-z3)<0.00001)=40;

niters = reshape(niters,size(xx));
solutions = reshape(which_root, size(xx));

image(solutions+niters./2.5) % for some extra colors
%based on:
%http://quantombone.blogspot.com/2009/07/
%simple-newtons-method-fractal-code-in.html
```