Homework 2. Numerical Methods

Carlos Giovanny Encinia Gonzalez

August 2021

Problem 1

(1) Find the fourth Taylor polynomial $P_4(x)$ for the function $f(x) = xe^x$ about $x_0 = 0$.
a: Find an upper bound for $|f(x) - P_4(x)|$, for $0 \leq x \leq 0.4$, ie find an upper bound of $|R_4(x)|$ for $0 \leq x \leq 0.4$
b: Approximate $\int_0^{0.4} f(x)\,dx$ using $\int_0^{0.4} P_4(x)\,dx$

Answer

For to find the Mclaurin expantion whit fourth order we make:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)x^2}{2} + \frac{f'''(0)x^3}{6} + \frac{f^4(0)x^4}{24} \quad (1)$$

Now we going to calculate the superior order derivatives of the function.

$$f'(x) = e^{x^2}(1 + 2x^2)$$
$$f''(x) = 2e^{x^2}(3x + 2x^3)$$
$$f'''(x) = 2e^{x^2}(12x^2 + 4x^4 + 3)$$
$$f^4(x) = 2e^{x^2}(40x^3 + 8x^5 + 30x)$$

Evaluating the function in 0 we obtain

$$f(0) = 0$$
$$f'(0) = 1$$
$$f''(0) = 0$$
$$f'''(0) = 6$$
$$f^4(0) = 0$$

Now we will substituing thevalues in (1)

$$f(x) = x + x^3$$

Now we will solve (a):

$$R_4(x) = \frac{f^5(\xi(x))x^5}{5!}$$

We know that the function is crecient in $0 \leq x \leq 0.4$ so

$$|f^5(\xi(x))| \leq e^{\frac{2}{5}} \leq e^{\frac{1}{2}} \leq \sqrt{3} \leq 2$$

so substituing in $|R_4(x)|$

$$|R_4(\frac{2}{5})| = \frac{2(\frac{2}{5})^4}{5!} = \frac{2}{9375}$$

so, an upper bound for $|f(x) - P_4(x)|$ is

$$\boxed{2/9375 = 0.00021}$$

Now we going to solve (b):

$$\int_0^{\frac{2}{5}} f(x)\,dx \approx \int_0^{\frac{2}{5}} P_4(x)\,dx$$

$$\int_0^{\frac{2}{5}} P_4(x)\,dx = \int_0^{\frac{2}{5}} (x + x^3)\,dx$$

$$\int_0^{\frac{2}{5}} (x + x^3)\,dx = (\frac{x^2}{2} + \frac{x^4}{4})\Big|_0^{\frac{2}{5}}$$

$$\int_0^{\frac{2}{5}} P_4(x)\,dx = \frac{4}{25(2)} + \frac{16}{625(4)}$$

$$\boxed{\int_0^{\frac{2}{5}} f(x)\,dx \approx \frac{54}{625} = 0.0864}$$

Problem 2

Implement a function to compute

$$f(x) = \frac{1}{\sqrt{x^2-1}-x}$$

When evaluating the previous function we can lose accuracy, transform the right hand side to avoid error (or improve the accuracy). Implement the transformed expression and compare the results with the original function. Note: use values of x greater than 10000.

Answer

Now we will transform the function, the first idea is to multiply by a factor in the numerator and denominator such taht the rest in the denominator will vanish. So

$$f(x) = \frac{1}{\sqrt{x^2-1}-x} * \frac{\sqrt{x^2-1}+x}{\sqrt{x^2-1}+x}$$

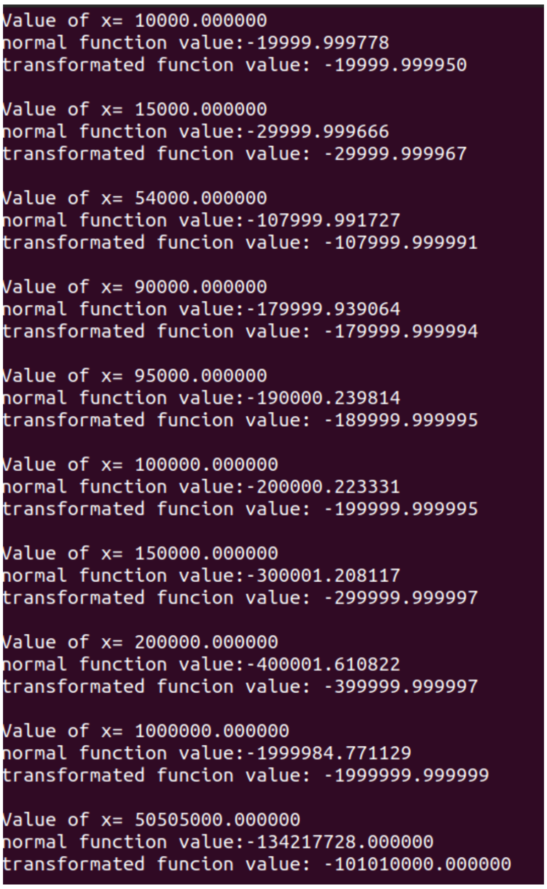
$$f(x) = \frac{\sqrt{x^2-1}+x}{|x^2-1|-x^2}$$

Here we can observe that we have two options x^2-1 and $1-x^2$ both have diferents range in where is defined, but the range where is defined $1-x^2$ is the range where the $f(x)$ is not defined, so

$$f(x) = -(x+\sqrt{x^2-1})$$

The implemention algorithm contains two function, in the next image we show the diferent results, the normal function is the original function and the transformated function is the function with the simplification. Also we will show a table with the results, f(x) is the normal function.

Value x	f(x)	g(x)
10000	-19999.999778	-19999.999950
15000	-29999.999666	-29999.999967
54000	-107999.991727	-107999.999991
90000	-179999.939054	-179999.999994
95000	-190000.239814	-189999.999995
100000	-200000.223331	-199999.999995
150000	-300001.208117	-299999.999997
200000	-400001.610822	-399999.999997
1000000	-1999984.771129	-1999999.999999
50505000	-134217728.0000	-101010000.0000



```
Value of x= 10000.000000
normal function value:-19999.999778
transformated funcion value: -19999.999950

Value of x= 15000.000000
normal function value:-29999.999666
transformated funcion value: -29999.999967

Value of x= 54000.000000
normal function value:-107999.991727
transformated funcion value: -107999.999991

Value of x= 90000.000000
normal function value:-179999.939064
transformated funcion value: -179999.999994

Value of x= 95000.000000
normal function value:-190000.239814
transformated funcion value: -189999.999995

Value of x= 100000.000000
normal function value:-200000.223331
transformated funcion value: -199999.999995

Value of x= 150000.000000
normal function value:-300001.208117
transformated funcion value: -299999.999997

Value of x= 200000.000000
normal function value:-400001.610822
transformated funcion value: -399999.999997

Value of x= 1000000.000000
normal function value:-1999984.771129
transformated funcion value: -1999999.999999

Value of x= 50505000.000000
normal function value:-134217728.000000
transformated funcion value: -101010000.000000
```

The original function gives relts with some errors and with the number 50505000 the result is totally wrong, this is because it can not be represent in the flat number, on the other hand the transformated function is more precice, this is because avoid the complicate operation in the denominator of the original function.

Code 1: Problem 2

```c
//Giovanny Encinia
//17/08/2021
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define ONE 1
#define SQUARE(x) (pow(x, 2))
#define SIZE(x) (sizeof(x)/sizeof(x[0]))

double function(double x)
{
    /*Calculate the function 1/(sqrt(x^2 -1)) -xevaluated in x
    and return the result*/
    double result;
    result = ONE / (sqrt(SQUARE(x) - ONE) - x);

    return  result;
}

double function_op(double x)
{
    /*Calculate the function x-sqrt(x^2 - 1)evaluated in x
    and return the result*/
    double result;
    result = -(x + sqrt(SQUARE(x) - ONE));

    return result;
}

int main()
{
    int i = 0;
    double x[10] = {
                       10000, 15000, 54000, 90000,
                       95000, 100000, 150000, 200000,
                       1000000, 50505000
                     };

    while(i < SIZE(x))
    {
        printf("Value of x= %f\n", x[i]);
        printf("normal function value:%f\n", function(x[i]));
        printf("transformated funcion value: %f\n\n", function_op(x[i]));
        i++;
    }

    return 0;
}
```

Problem 3

Implement a function to compute the exponential function by using the Taylor/Maclaurin series

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Since we cannot add infinite terms, we can approximate this expansion by

$$e^x \approx \sum_{k=0}^{n} \frac{x^k}{k!}$$

Answer

Code 2: Problem 3

```
//Giovanny Encinia
// 08/18/2021
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define SIZE 25// terms number of the mclaurin's serie
#define ONE 1
#define  ZERO 0
#define E_R(e_x, result) (fabs(e_x - result)/fabs(e_x))
#define DELTA 0.00001
#define X 10.5 // value x to evaluate

long factorial(long k, long *memo)
{
    /*This function calculates the factorial with memoization*/
    long result;

    if(memo[k])
    {
        result = memo[k]; // search in the array memo
    }
    else
    {
        result = k*factorial(k-ONE, memo);
    }

    memo[k] = result; // save the new result in memo

    return result;
}

long double power(long double x, int k, long double *memo_pow)
{
    /*Calculate power of x with memoization*/
    long double result;
```

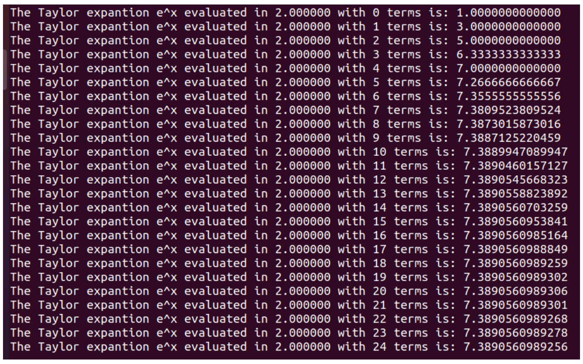
```
    if(memo_pow[k])
    {
        result = memo_pow[k];//search
    }
    else
    {
        result = power(x, k -ONE, memo_pow) * x;// kernel recursion
    }

    memo_pow[k] = result;// save in memo

    return result;
}

double e_mclaurin(long double x, int n, long *memo, \
                  long double *memo_pow, long double *memo_taylor)
{
    /*mclaurine^x serie with memoization*/
    long double sum;

    if(memo_taylor[n])
    {
        sum = memo_taylor[n];
    }
    else
    {
        sum = e_mclaurin(x, n - ONE, memo, memo_pow, memo_taylor) \
        + power(x, n, memo_pow) / factorial(n, memo);
    }

    memo_taylor[n] = sum;

    return sum;
}

int main()
{
    long *memo\
     = (long*)calloc(SIZE, sizeof(long));//factorial
    long double *memo_pow\
     = (long double *)calloc(SIZE, sizeof(long double));//power
    long double *memo_taylor\
     = (long double *)calloc(SIZE, sizeof(long double));//term serie

    int i = ZERO;
    long double x;
    long double result = ZERO;
    long double e_x;
    //x = X;
```

```c
    //e_x = exp(X);
    printf("Give me a value of x\n");
    scanf("%Lf", &x);
    e_x = exp(x);

    if(memo == NULL || memo_pow == NULL || memo_taylor == NULL)
    {
        printf("Memory not asigned, full memory\n");
    }
    else
    {
    memo[ZERO] = memo[ONE] = ONE;// 0! = 1, 1! = 1
    memo_pow[ZERO] = ONE; // x^0 = 1
    memo_taylor[ZERO] = ONE;

    //check the relative error or the terms number
    while(i < SIZE)
    {
        result = e_mclaurin(x, i, memo, memo_pow, memo_taylor);
        i++;
        if(E_R(e_x, result) <= DELTA) // check relative error
        {
            break;
        }
    }

    if(E_R(e_x, result) > DELTA) //check solution
    {
        printf("solution not found\n");
        printf("relative error big with n <= %d\n", i);
    }
    else
    {
        printf("solution found with Er=%lf\n", E_R(e_x, result));
        printf("n = %d, f(%Lf) = %.9Lf\n", i, x, result);
    }

    }

    free(memo);
    free(memo_pow);
    free(memo_taylor);

    return ZERO;
}
```

The first program was modified to show how the algoritm converge to the solution, in the Figure (1), when the value of x is 2 the result qith a small relative error is around the termn 13 in contrast qhen the value of x is been increasing the serie diverge Figure(2), because the number in the denominator is very large, and the machine can not represent the number correctly. In the algorithm the maximum number of terms is 24 but this can be change to any other number of terms, but the function have a large value when x is increasing and this make tht the algorithm diverges.

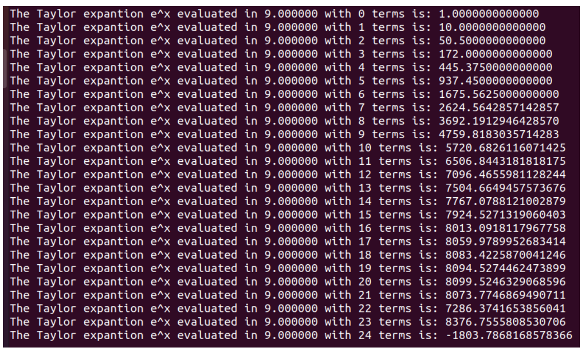
Inside the algoritm was utiliiced memoization for not to calculate the repeat power and factorials also was used in the taylor calculation for found a result quickly.

Figure 1: Convergence



```
The Taylor expantion e^x evaluated in 2.000000 with 0 terms is: 1.0000000000000
The Taylor expantion e^x evaluated in 2.000000 with 1 terms is: 3.0000000000000
The Taylor expantion e^x evaluated in 2.000000 with 2 terms is: 5.0000000000000
The Taylor expantion e^x evaluated in 2.000000 with 3 terms is: 6.3333333333333
The Taylor expantion e^x evaluated in 2.000000 with 4 terms is: 7.0000000000000
The Taylor expantion e^x evaluated in 2.000000 with 5 terms is: 7.2666666666667
The Taylor expantion e^x evaluated in 2.000000 with 6 terms is: 7.3555555555556
The Taylor expantion e^x evaluated in 2.000000 with 7 terms is: 7.3809523809524
The Taylor expantion e^x evaluated in 2.000000 with 8 terms is: 7.3873015873016
The Taylor expantion e^x evaluated in 2.000000 with 9 terms is: 7.3887125220459
The Taylor expantion e^x evaluated in 2.000000 with 10 terms is: 7.3889947089947
The Taylor expantion e^x evaluated in 2.000000 with 11 terms is: 7.3890460157127
The Taylor expantion e^x evaluated in 2.000000 with 12 terms is: 7.3890545668323
The Taylor expantion e^x evaluated in 2.000000 with 13 terms is: 7.3890558823892
The Taylor expantion e^x evaluated in 2.000000 with 14 terms is: 7.3890560703259
The Taylor expantion e^x evaluated in 2.000000 with 15 terms is: 7.3890560953841
The Taylor expantion e^x evaluated in 2.000000 with 16 terms is: 7.3890560985164
The Taylor expantion e^x evaluated in 2.000000 with 17 terms is: 7.3890560988849
The Taylor expantion e^x evaluated in 2.000000 with 18 terms is: 7.3890560989259
The Taylor expantion e^x evaluated in 2.000000 with 19 terms is: 7.3890560989302
The Taylor expantion e^x evaluated in 2.000000 with 20 terms is: 7.3890560989306
The Taylor expantion e^x evaluated in 2.000000 with 21 terms is: 7.3890560989301
The Taylor expantion e^x evaluated in 2.000000 with 22 terms is: 7.3890560989268
The Taylor expantion e^x evaluated in 2.000000 with 23 terms is: 7.3890560989278
The Taylor expantion e^x evaluated in 2.000000 with 24 terms is: 7.3890560989256
```

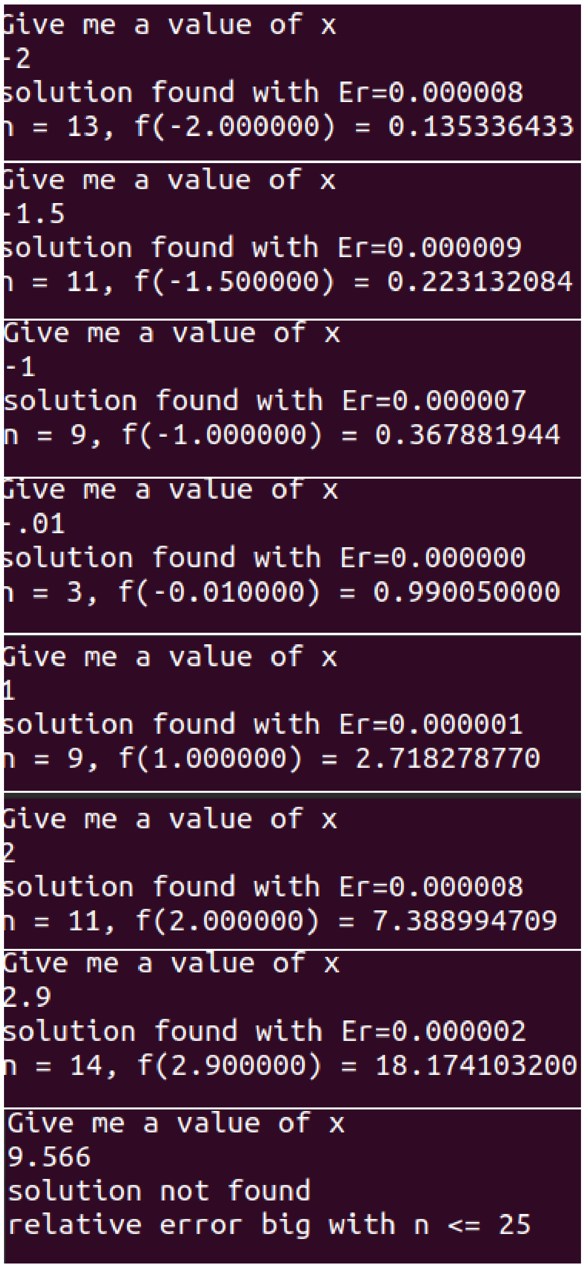
Figure 2: Divergence



```
The Taylor expantion e^x evaluated in 9.000000 with 0 terms is: 1.0000000000000
The Taylor expantion e^x evaluated in 9.000000 with 1 terms is: 10.0000000000000
The Taylor expantion e^x evaluated in 9.000000 with 2 terms is: 50.5000000000000
The Taylor expantion e^x evaluated in 9.000000 with 3 terms is: 172.0000000000000
The Taylor expantion e^x evaluated in 9.000000 with 4 terms is: 445.3750000000000
The Taylor expantion e^x evaluated in 9.000000 with 5 terms is: 937.4500000000000
The Taylor expantion e^x evaluated in 9.000000 with 6 terms is: 1675.5625000000000
The Taylor expantion e^x evaluated in 9.000000 with 7 terms is: 2624.5642857142857
The Taylor expantion e^x evaluated in 9.000000 with 8 terms is: 3692.1912946428570
The Taylor expantion e^x evaluated in 9.000000 with 9 terms is: 4759.8183035714283
The Taylor expantion e^x evaluated in 9.000000 with 10 terms is: 5720.6826116071425
The Taylor expantion e^x evaluated in 9.000000 with 11 terms is: 6506.8443181818175
The Taylor expantion e^x evaluated in 9.000000 with 12 terms is: 7096.4655981128244
The Taylor expantion e^x evaluated in 9.000000 with 13 terms is: 7504.6649457573676
The Taylor expantion e^x evaluated in 9.000000 with 14 terms is: 7767.0788121002879
The Taylor expantion e^x evaluated in 9.000000 with 15 terms is: 7924.5271319060403
The Taylor expantion e^x evaluated in 9.000000 with 16 terms is: 8013.0918117967758
The Taylor expantion e^x evaluated in 9.000000 with 17 terms is: 8059.9789952683414
The Taylor expantion e^x evaluated in 9.000000 with 18 terms is: 8083.4225870041246
The Taylor expantion e^x evaluated in 9.000000 with 19 terms is: 8094.5274462473899
The Taylor expantion e^x evaluated in 9.000000 with 20 terms is: 8099.5246329068596
The Taylor expantion e^x evaluated in 9.000000 with 21 terms is: 8073.7746869490711
The Taylor expantion e^x evaluated in 9.000000 with 22 terms is: 7286.3741653856041
The Taylor expantion e^x evaluated in 9.000000 with 23 terms is: 8376.7555808530706
The Taylor expantion e^x evaluated in 9.000000 with 24 terms is: -1803.7868168578366
```

In figure (3) the algoritm change and now you have to insert a value of x, then the algorithm try to find the result with the less relative error, if it does not find the result after n terms the algoritjm finish. Also we can observe that the values around 0 needs less iterations to find a result with a small relative error.

Figure 3: Results



```
Give me a value of x
-2
solution found with Er=0.000008
n = 13, f(-2.000000) = 0.135336433
Give me a value of x
-1.5
solution found with Er=0.000009
n = 11, f(-1.500000) = 0.223132084
Give me a value of x
-1
solution found with Er=0.000007
n = 9, f(-1.000000) = 0.367881944
Give me a value of x
-.01
solution found with Er=0.000000
n = 3, f(-0.010000) = 0.990050000
Give me a value of x
1
solution found with Er=0.000001
n = 9, f(1.000000) = 2.718278770
Give me a value of x
2
solution found with Er=0.000008
n = 11, f(2.000000) = 7.388994709
Give me a value of x
2.9
solution found with Er=0.000002
n = 14, f(2.900000) = 18.174103200
Give me a value of x
9.566
solution not found
relative error big with n <= 25
```

Problem 4

Riemann sums can be used to estimate the area under the curve y = f (x) in the interval [a, b]. Left- and right-endpoint approximations, with subintervals of the same width, are special kinds of Riemann sums, ie, **Left-Endpoint Approximation:**

$$L_n = f(x_0)\Delta x + f(x_1)\Delta x + \cdots + f(x_{n-1})\Delta x$$

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta x$$

Right-Endpoint Approximation:

$$R_n = f(x_1)\Delta x + f(x_2)\Delta x + \cdots + f(x_n)\Delta x$$

$$R_n = \sum_{i=1}^{n} f(x_i)\Delta x$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x, i = 0, 1, \cdots, n$. Implement functions to compute L_n and R_n . Using the function $f(x) = sin(x)$ over the interval $[0, \frac{\pi}{2}]$, compute L_n and R_n for $n = 10$. Compare the previous results with

$$\int_0^{\frac{\pi}{2}} f(x)\, dx$$

Answer

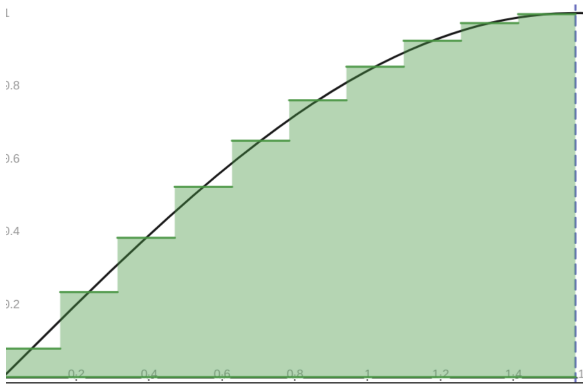
Fisrt we solve the integral analytically

$$\int_0^{\frac{\pi}{2}} f(x)\, dx = -cos(x)|_0^{\frac{\pi}{2}} = 1$$

Figure 4: Riemann R_n L_n

```
R_n = 1.076483, Er = 0.076482803
L_n = 0.919403, Er = 0.080596830
```

Figure 5: R_n

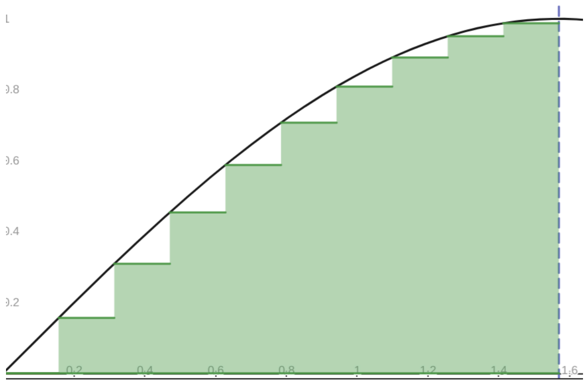


Now we going to implement the algorithm. In the Figure (4), we can observe the result and we can see that the approximation R_n is better tan the L_n because the relative error is less. But this result is not always to be true, si some cases the L_n approximation could be better, this depend of the function type, other thing to cosider is the function to work, for example we expect that with the function $cos(x)$ the better approximation will be L_n, it is easy to see being that is like a reflect of the $sin(x)$ in the interval $[0, \pi/2]$.

When the Δx is very small the result of R_n and L_n would have to be the same.

Anyway this approximation is not the better, being that we need many operations to reach and small relative error. In the figure (5) and (6) show the resppectiove R_n and L_n approximation.

Figure 6: Ln



Code 3: Problem 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define F(x) sin(x)
#define N 10
#define ZERO 0
#define ONE 1
#define PI_2 1.57079632679

#define A ZERO
#define B PI_2
#define DX() ((B-A)/N)

double riemann(int n, int i)
{
    /*This is a general function that calculates the
    riemann''s sum,  it can be Lr or Rn depending of
    the i and the n*/

    double sum = ZERO;
    double x_i;

    x_i = A + i * DX();//initialize the first x_i

    while(i <= n)
    {
        sum += F(x_i);
        x_i += DX();
        i++;

    }

    return  sum * DX();
}

int main()
{
    double rn, ln;

    rn = riemann(N, ONE);//begin with i=1 until n
    ln = riemann(N - ONE, ZERO);//begin with i=0 until n-1
    printf("  R_n = %lf, Er = %.9lf\n", rn, fabs(ONE - rn));
    printf("  L_n = %lf, Er = %.9lf\n", ln, fabs(ONE - ln));

    return 0;
}
```