Comp. Methods in Mech. Eng.

Assignment # 2

uOttawa

Name: Usama Tariq

Student Number: 7362757

Professor: Catherine Mavriplis

TA: Fabien Giroux

Due Date: January 24, 2019

1-Introduction

In this assignment the similarities and differences between the Taylor and Lagrange numerical approximations are studied using a function of choice. In this case the function chosen was $y = \sin(x)$. Using successive levels of degrees (number of derivatives for Taylor and number of points for Lagrange) the accuracy is measured and explained.

2-Taylor Interpolation

The general equation that governs Taylor interpolations is:

$$\check{f}(x) = \sum_{n=0}^{\infty} (x-a)^n \frac{f^n(a)}{n!}$$

In this assignment a is centered at 0 and the x range is -10 to 10. As more terms are added to the Taylor approximation the results are expected to improve. The code for which can been seen in Appendix A

2.1- Results of Taylor Interpolation

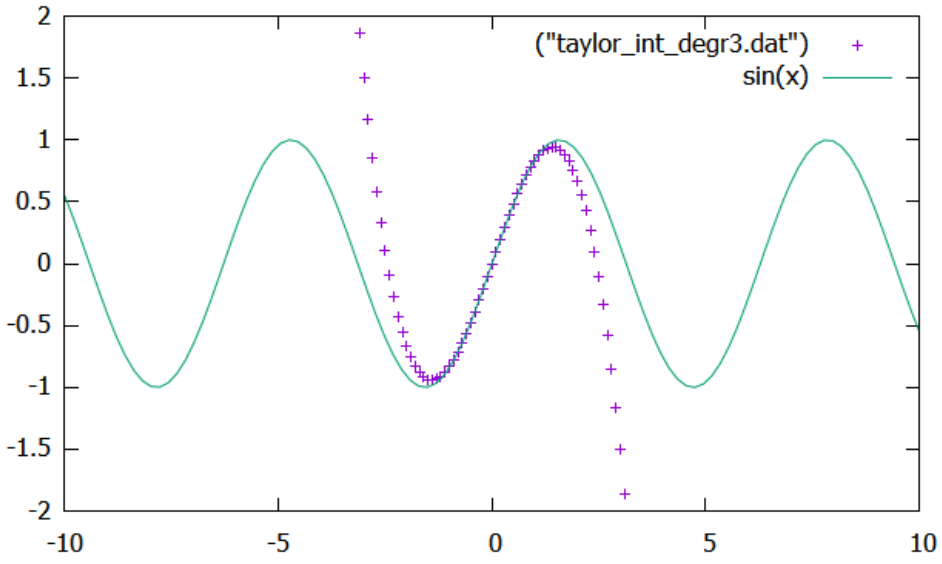

Figure 1 Taylor Approximation of sin(x) at degree 3

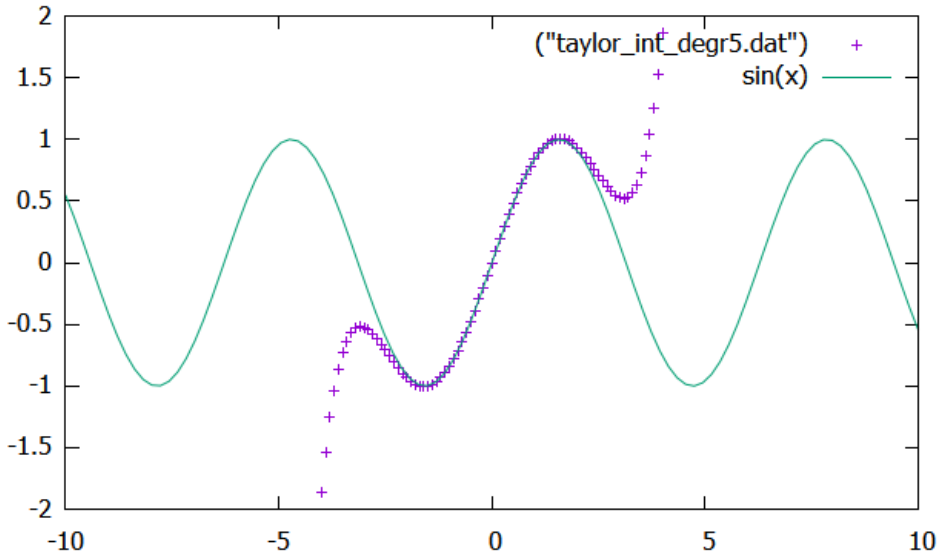

Figure 2 Taylor Approximation of sin(x) at degree 5

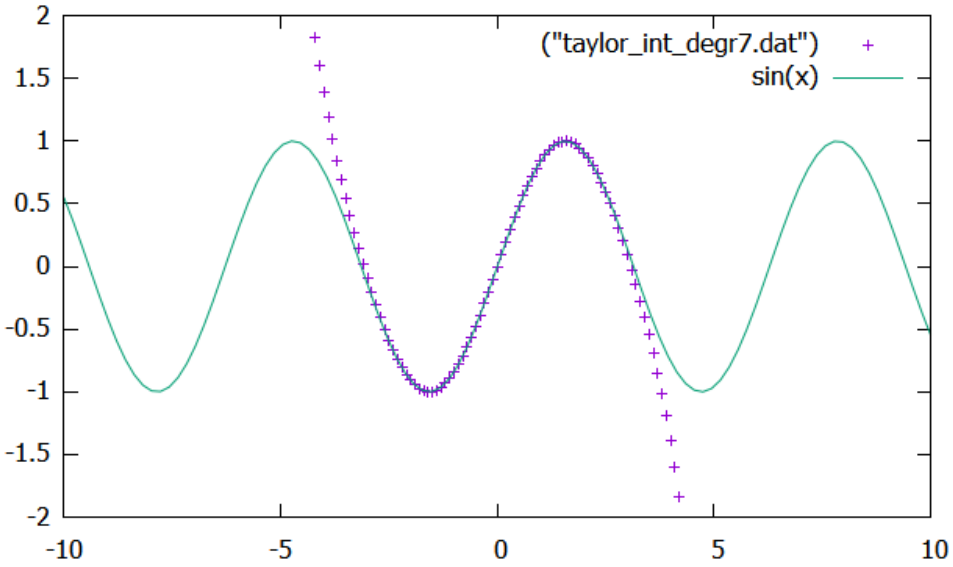

Figure 3 Taylor Approximation of sin(x) at degree 7

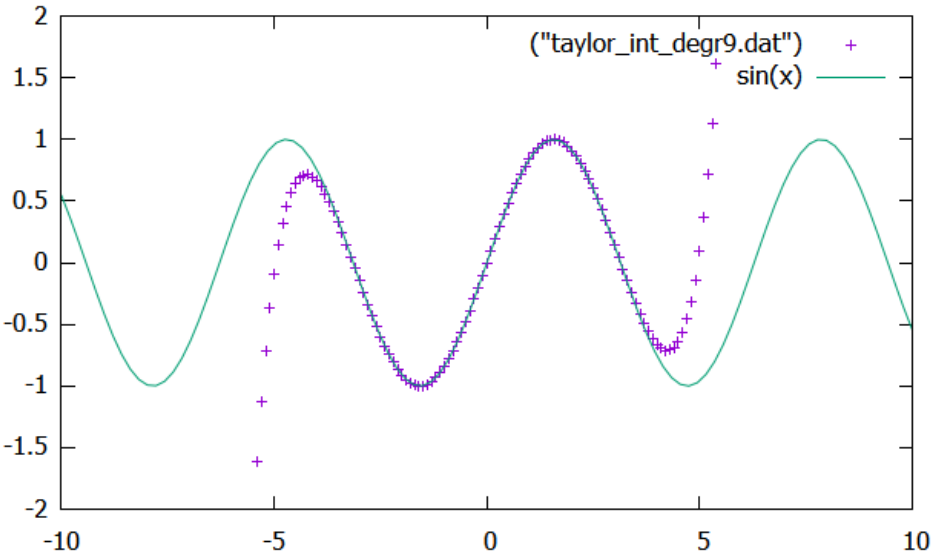

Figure 4 Taylor Approximation of sin(x) at degree 9

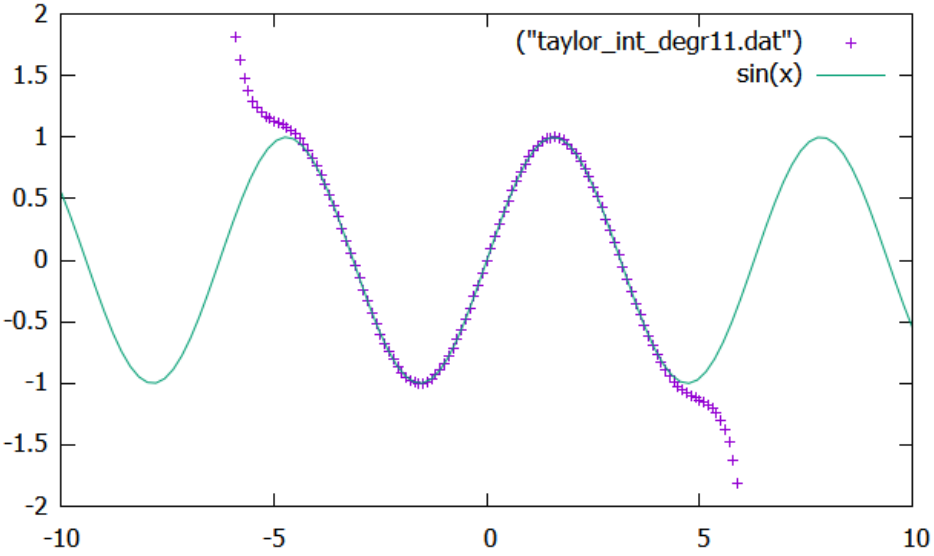

Figure 5 Taylor Approximation of sin(x) at degree 11

It can be seen that as the number of derivates increases in the Taylor series, the accuracy improves. In Figure 6, a trusted theoretical plot for $\sin(x)$ can be compared to the plots obtained in this assignment. There is a striking resemblance, however this can be further investigated analytically.

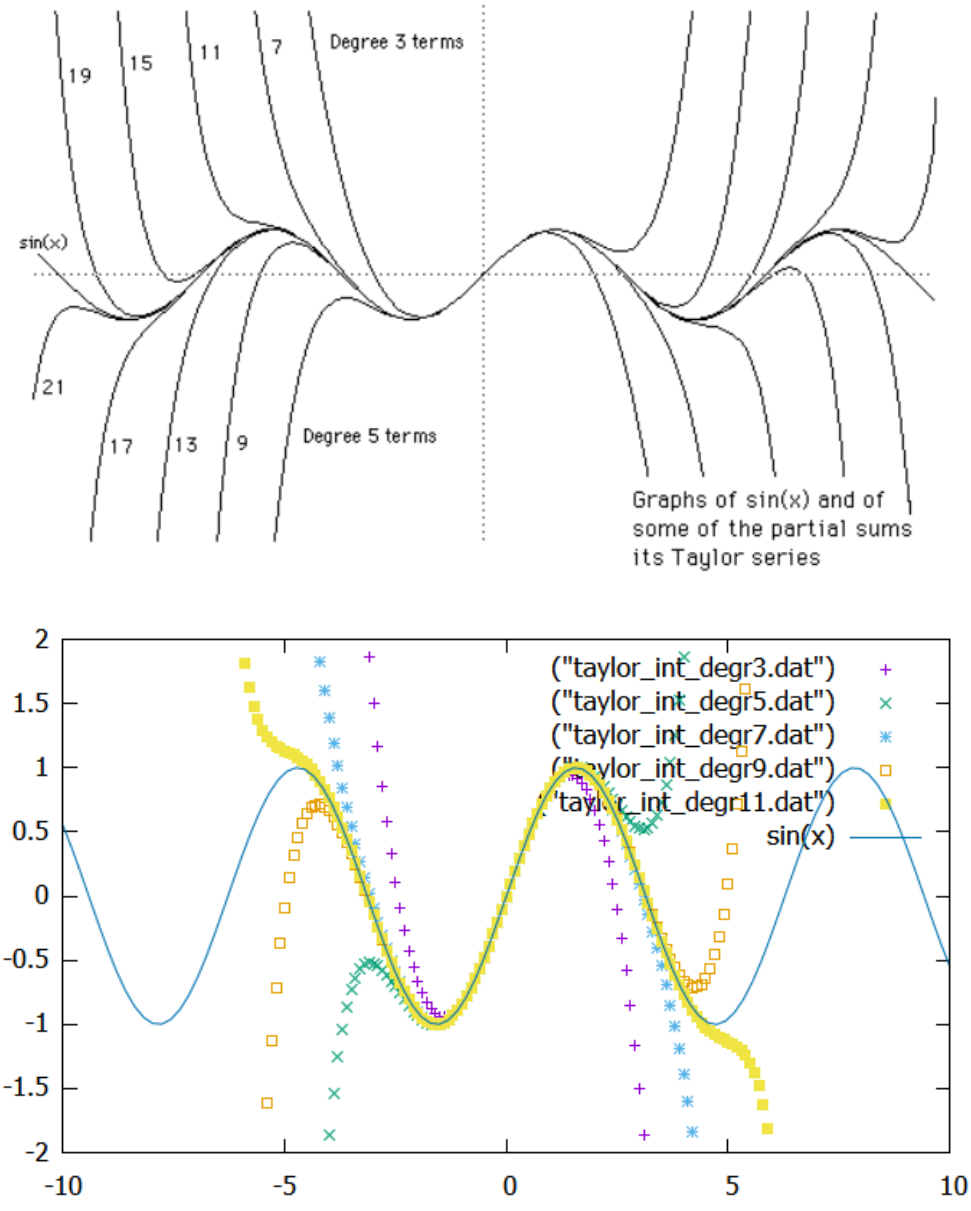

Figure 6 Checking the accuracy of the code for Taylor Approximations

In Table 1, as the degree increases the percent error decreases, this is expected. Also, it is also noted that as you move further away from the origin ($a = 0$) the level of accuracy decreases and only gets better as more derivatives are taken.

Table 1 Numerical Analysis of Taylor Approximations at Specific Points

	-5	*-2*	*-1*	*0.1*	*0.2*	*0.5*
$Sin(x)$	0.95892427	-0.9092974	-0.84147	0.09983342	0.19867	0.47943
Degree 3	15.8333	-0.66667	-0.83333	0.0998333	0.198667	0.479167
Error (%)	1551.15228	26.6829554	0.967471	0.00011684	0.00117	0.05393
Degree 5	-10.2083	-0.93333	-0.84166	0.0998334	0.198669	0.479426
Error (%)	1164.55747	2.64298264	0.022462	0	0.00017	0
Degree 7	5.29266	-0.90793	-0.84146	0.0998334	0.198669	0.479426
Error (%)	451.937222	0.15038279	0.001305	0	0.00017	0
Degree 9	-0.08963	-0.90934	-0.84147	0.0998334	0.198669	0.479426
Error (%)	109.346932	0.00468199	0.000117	0	0.00017	0
Degree 11	1.13362	-0.90929	-0.84147	0.0998334	0.198669	0.479426
Error (%)	18.2178854	0.00081677	0.000117	0	0.00017	0

In Figure 7 the percent errors from Table 1 are summarized and it can be clearly seen that as the degree is increased, the point at which the error goes to 0 starts to merge further and further away from the origin giving us a larger range for accuracy.

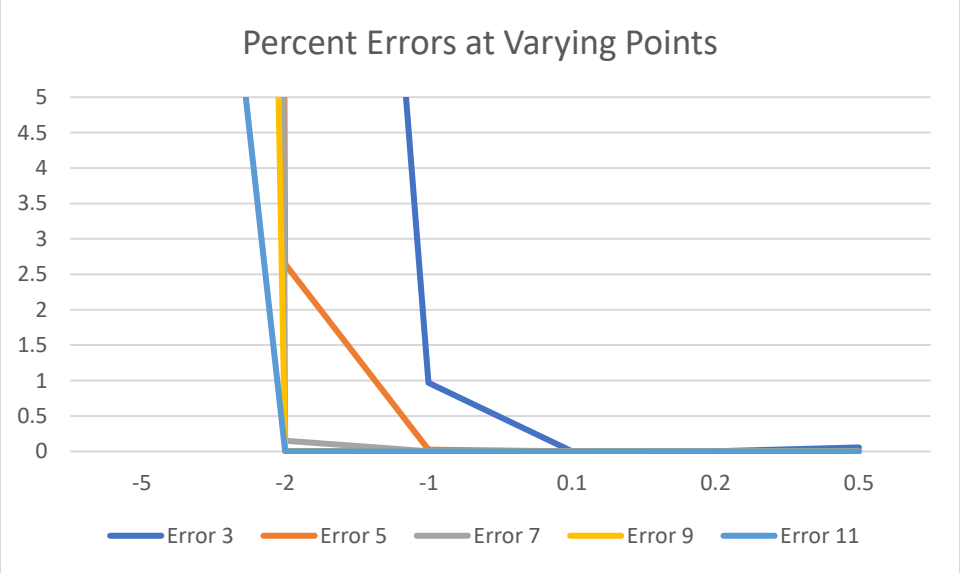

Figure 7 Percent Errors of Taylor Approximations at different locations

2.2- Errors in Taylor Approximation Code

There was an error that occurred as the degree started to go above 15. The accuracy actually started to decrease, this is obviously an error and the reason must be some bug in the code that hadn't been fully sorted out.

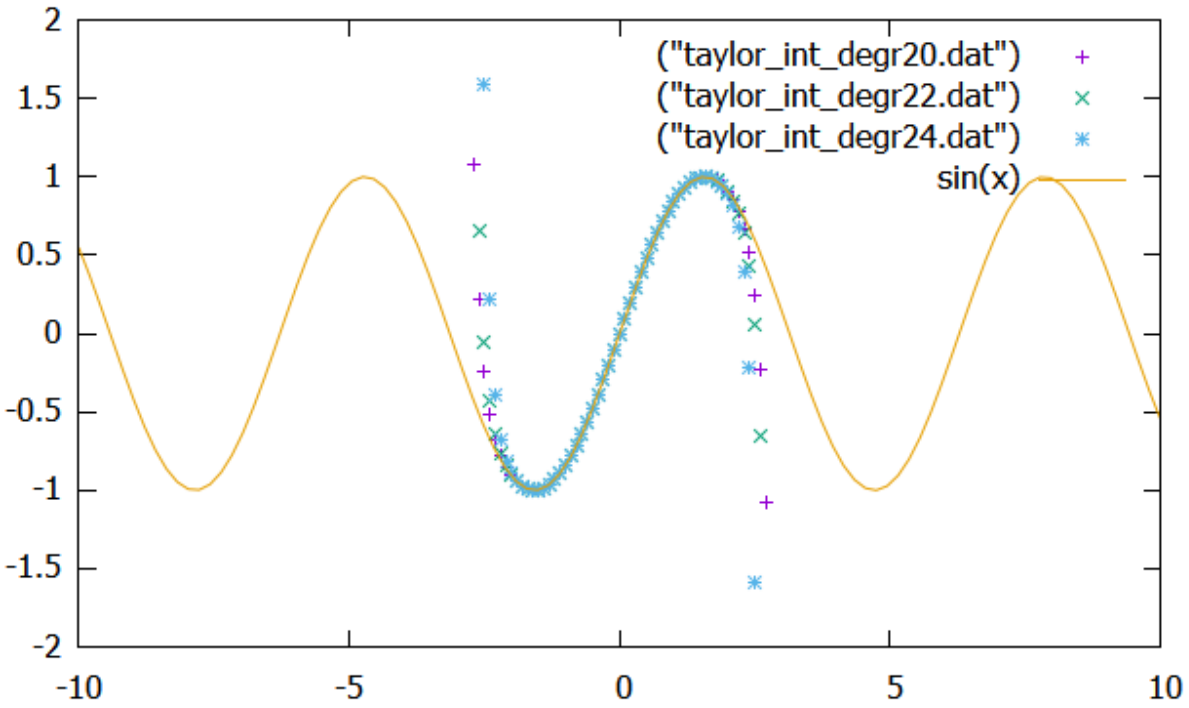

Figure 8 Taylor Approximation breaks down at very high degrees

3- Lagrange Interpolation

The Lagrange interpolation is another numerical method that can be used to approximate a function. The function associated with this method is as follows:

$$\check{f}(x) = \sum_{i=0}^{n} l_j(x_i) f(x_i)$$

Where $l_j(x_i)$ is:

$$l_j(x_i) = \prod_{m=0, m\neq j}^{k} \frac{x_i - x_m}{x_j - x_m}$$

Using the Lagrange Interpolation method for which the code can be seen in Appendix A, the following plots were obtained.

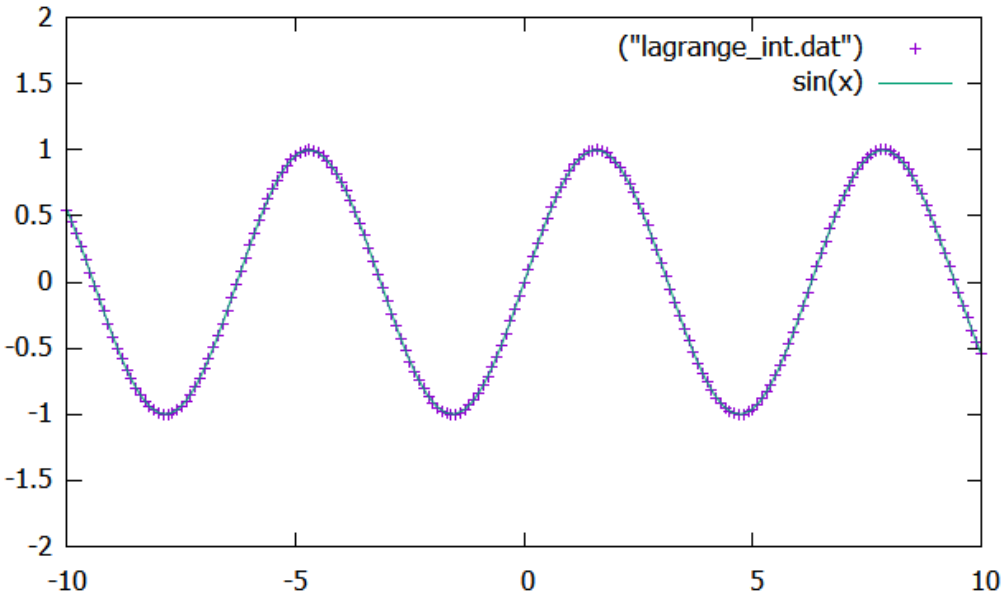

Figure 9 Lagrange Method of sin(x) using 200 points

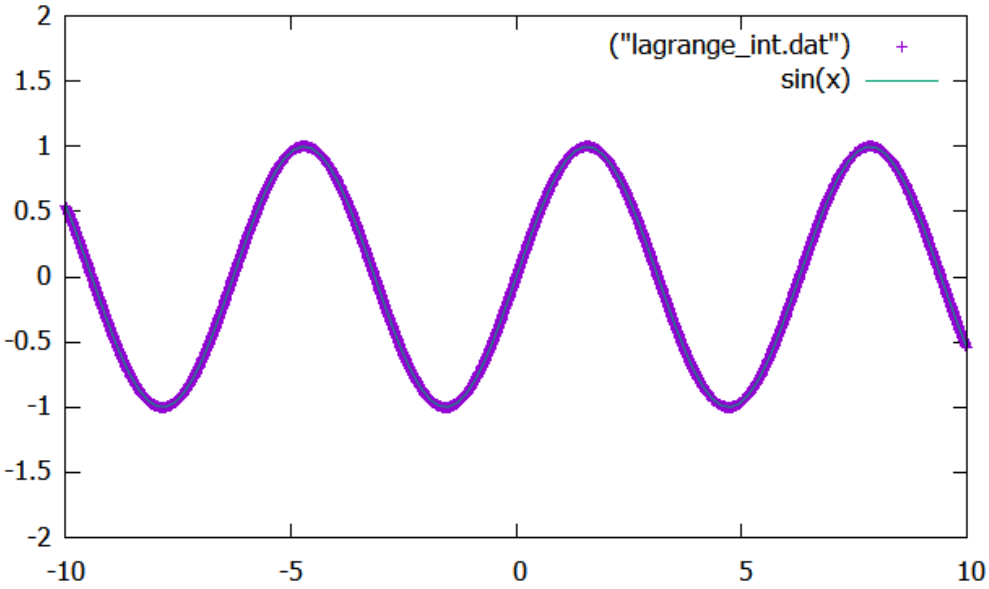

Figure 10 Lagrange Method of sin(x) using 2000 points

In table 2, as the number of points was increased the accuracy remained quite constant with the overall percent error staying close to zero no matter how further you went from the origin. The reason for this seems to be the larger range of $y(x)$ values that are sampled.

Table 2 Lagrange Interpolation of sin(x) summarized at Different Points

	-5	*-2*	*-1*	*0.1*	*0.2*	*0.5*
Sin	0.958924	-0.909297	-0.84147	0.099833	0.19867	0.47943
200 Points	0.958924	-0.909297	-0.84147	0.099833	0.19866	0.47942
Error (%)	0	0	0.00011	0	0.0005	0.00083
2000 Points	0.958924	-0.909297	-0.84147	0.099833	0.19866	0.47942
Error (%)	0	0	0.000119	0	0.0005	0.00083

4- Discussion

Both Taylor and Lagrange methods are appropriate to use for approximating functions however I think it's more important to know when to use which method. From this assignment I learned that Taylor approximations can be good to use when not enough data for the original function is known but we do know data for it's derivates (like the speed and acceleration). Even then it is important to know where you want the information of the original function. For example, in the case of $\sin(x)$ as we got further from the origin the accuracy decreased heavily. This leads me to believe Taylor series are good for when you don't know too much about a function. Using this method, it is possible to derive an approximate an equation, accurate to a certain extent.

In contrast, the Lagrange interpolation was much more accurate but the reason for this was we used a larger sample size of the original function. This implies that Lagrange interpolation method must be used when we are given a set of data points on a function for which we want to approximate a function on.

In Figure 11 and 12 the differences between the two methods are summarized.

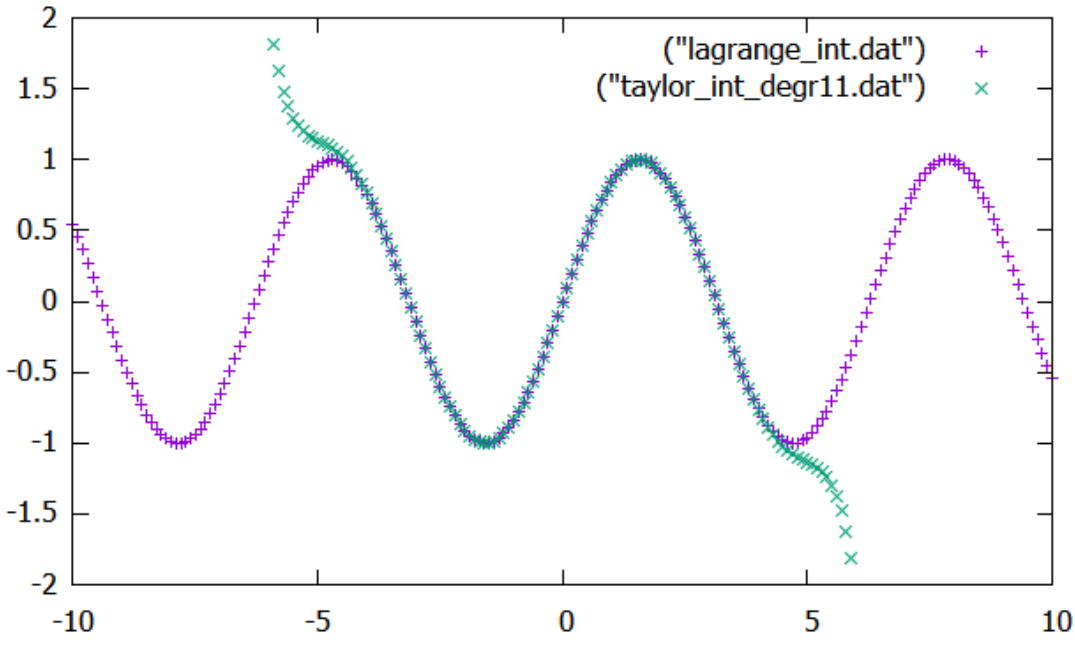

Figure 11 The difference between Lagrange and Taylor Approximation at their highest level of accuracy

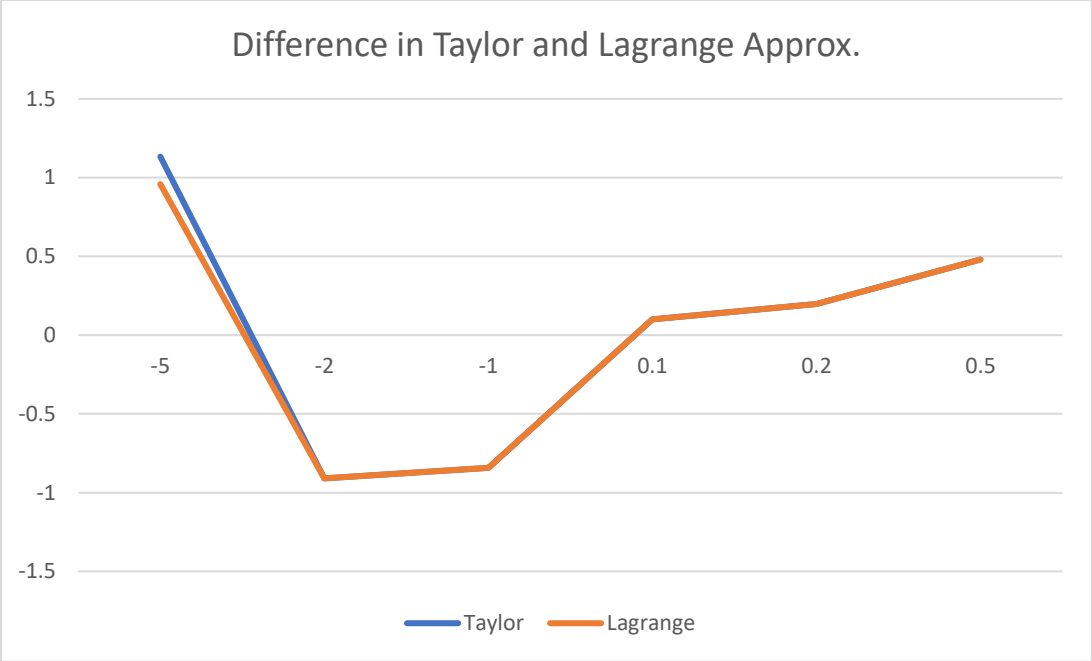

Figure 12 A closer look in the differences from Figure 11

Appendix A - C++ Code

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <fstream>
#include <functional>
#include <math.h>

// Using typedefs for all functions
using function_type = std::function<double(double)>;

// For calculating the factorials
int factorial (int n){
  if (n > 1){
    return n * factorial(n-1);
  }
  else{return 1;}
}

// Plotting the results in .dat format
void plot (std::vector<double> z, std::string filename){
  std::ofstream fout(filename);
  if(!fout){
    throw std::runtime_error("Could not open file: " + filename);
  }

  double cnt = 0;
  for (auto it = z.begin(); it != z.end(); it++){
    fout << (-10 + (cnt/10)) << " " << *it << '\n';
    std::cout << (-10 + (cnt/10)) << " " << *it << std::endl;
    cnt++;
  }
}

void plot_lagrange (std::vector<double> z, std::string filename){
  std::ofstream fout(filename);
  if(!fout){
    throw std::runtime_error("Could not open file: " + filename);
  }

  double cnt = 0;
  for (auto it = z.begin(); it != z.end(); it++){
    fout << (-10 + (cnt/100)) << " " << *it << '\n';
    std::cout << (-10 + (cnt/100)) << " " << *it << std::endl;
    cnt++;
  }
}

////////////////////////////////////////////////////////////////////////////
//////////////////////Taylor Interpolation ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
std::vector<double> taylorInt(function_type y, function_type dydx, double a, double t){

  //tp is a vector which is used to store all the results in
  std::vector<double> tp;

  for (double i = -10; i <= 10; i=i+0.1){
    t = y(a); // resetting the t value as sin(a)
```

```
58.
59.     for (int j = 1; j < 4; j++){
60.
61.       // In this for loop modulus is used to iterate succesively between
62.       // all the if statements to use all the four cases in the derivatives
63.       // sin(x) -> cos(x) -> -sin(x) -> -cos(x) -> sin(x)
64.       if ( ((j-1)%4)+1 == 1){
65.         t = t + ( pow(static_cast<double>(i)-
    a, j) * (dydx(a)) ) / static_cast<double>(factorial(j));
66.       }
67.
68.       if ( ((j-1)%4)+1 == 2){
69.         t = t + ( pow(static_cast<double>(i)-a, j) * (-
    y(a)) ) / static_cast<double>(factorial(j));
70.       }
71.
72.       if ( ((j-1)%4)+1 == 3){
73.         t = t + ( pow(static_cast<double>(i)-a, j) * (-
    dydx(a)) ) / static_cast<double>(factorial(j));
74.       }
75.
76.       if ( ((j-1)%4)+1 == 4){
77.         t = t + ( pow(static_cast<double>(i)-
    a, j) * (y(a)) ) / static_cast<double>(factorial(j));
78.       }
79.     }
80.     tp.push_back(t);
81.   }
82.   return tp;
83. }
84.
85. ////////////////////////////////////////////////////////////////////////////
86. ////////////////////// Lagrangian Interpolation ////////////////////////////
87. ////////////////////////////////////////////////////////////////////////////
88. std::vector<double> lagInt(function_type y){
89.   std::vector<double> li;
90.   std::vector<double> xVals;
91.   std::vector<double> yVals;
92.   double prod = 1;
93.   double sum = 0;
94.
95.   for (double x = -10; x <= 10 ; x = x+0.01){
96.     xVals.push_back(x);
97.   }
98.
99.   for (int i = 0; i < xVals.size(); i++){
100.           prod = 1;
101.           sum = 0;
102.           for (int j = 0; j < xVals.size(); j++){
103.             if (i != j){
104.               prod = prod * ( (xVals.at(i) - xVals.at(j) ) / ( xVals.at(i) - xVals.at(
    j) ) );
105.             }
106.           }
107.           prod = prod * y(xVals.at(i));
108.           sum = sum + prod;
109.           yVals.push_back(sum);
110.         }
111.         return yVals;
112.       }
113.
```

```
114.  ////////////////////////////////////////////////////////////////////////////
115.  ////////////////////// MAIN FUNCTION //////////////////////////////////////
116.  ////////////////////////////////////////////////////////////////////////////
117.  int main(){
118.
119.    // typedefs of sin(x) and cos(x)
120.    auto y = [](double x){
121.      return sin(x);
122.    };
123.
124.    auto dydx = [](double x){
125.      return cos(x);
126.    };
127.
128.    // The Taylor interpolation centered at a = 1
129.    std::cout << "-----------------------------------------------------\n";
130.    std::cout << "-----------------------------------------------------\n";
131.    std::cout << "|                   Taylor Interpolation            |\n";
132.    std::cout << "-----------------------------------------------------\n";
133.    std::cout << "-----------------------------------------------------\n";
134.
135.    double a = 0;
136.    double t = y(a);
137.    std::vector<double> taylor_int_results;
138.    taylor_int_results = taylorInt (y, dydx, a, t);
139.
140.    //Plotting the results
141.    plot (taylor_int_results, "taylor_int_degr.dat");
142.
143.    // Lagrangian Interpolation
144.    std::cout << "-----------------------------------------------------\n";
145.    std::cout << "-----------------------------------------------------\n";
146.    std::cout << "|                   Lagrangian Interpolation        |\n";
147.    std::cout << "-----------------------------------------------------\n";
148.    std::cout << "-----------------------------------------------------\n";
149.
150.    std::vector<double> lag_int_results;
151.    lag_int_results = lagInt(y);
152.    plot_lagrange (lag_int_results, "lagrange_int.dat");
153.  }
```