Pat O'Sullivan

Mh4718 Week 7

Week 7

0.0.0.1 Taylor's Theorem.

Taylor's theorem is commonly used for the evaluation of functions such as $\cos(x), \sin(x), \exp(x), \log(x)$
Recap:

Taylor's Theorem: If $f(x)$ is $n+1$ times differentiable over and open interval I containing x then:

$$f(x) = f(a)+f'(a)(x-a)+\frac{f^{(2)}(a)}{2!}(x-a)^2+\frac{f^{(3)}(a)}{3!}(x-a)^3+\cdots+\frac{f^{(n)}(a)}{n!}(x-a)^n+R_n$$

where $R_n = \dfrac{f^{(n+1)}(c)}{n+1!}(x-a)^{n+1}$ for some c between a and x.
$f(a) + f'(a)(x-a) + \dfrac{f^{(2)}(a)}{2!}(x-a)^2 + \dfrac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots + \dfrac{f^{(n)}(a)}{n!}(x-a)^n$
is a polynomial of degree n and is called the *Taylor polynomial* for f around a.
R_n is called the remainder term and is a measure of how good an approximation the Taylor polynomial is of the function.

If the $f^{(n)}$ is bounded as $n \to \infty$ then $R_n \to 0$ as $n \to \infty$ because the denominator $n+1!$ becomes large much faster than $|(x-a)^n|$.
For many important functions, including $\cos(x), \sin(x), \exp(x), \log(x)$ this is what happens.
In the case of $\cos(x), \sin(x), \exp(x)$ it is also best to choose $a = 0$ since we can evaluate all the necessary derivatives at this point. Thus we use the polynomial:
$f(0) + f'(0)x + \dfrac{f^{(2)}(0)}{2!}x^2 + \dfrac{f^{(3)}(0)}{3!}x^3 + \cdots + \dfrac{f^{(n)}(a)}{n!}x^n$ to approximate these functions.

The Taylor polynomials in particular are:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \text{ for } \exp(x)$$
$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \pm \frac{x^n}{n!} \text{ for } \cos(x)$$
$$x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \pm \frac{x^n}{n!} \text{ for } \sin(x)$$

The following program which displays the value of the Taylor polynomial of degree 6 around 0 for the function e^x and also outputs the value of the C++ *exp* function is clearly too cumbersome. It would be very difficult to use a Taylor polynomial of very high degree using this design:

```
#include <iostream>
#include <cmath>
#include <iomanip>
double Exp(double x)
{
    return 1+x+1.0/2*x*x+1.0/(3*2)*x*x*x+1.0/(4*3*2)*x*x*x*x
            +1.0/(5*4*3*2)*x*x*x*x*x+1.0/(6*5*4*3*2)*x*x*x*x*x*x;
}
using namespace std;
void main()
{
    double x=1;
    cout<<setprecision(30);
    cout<<"Taylor poly of degree 6 ="<<Exp(x)<<endl;
    cout<<"Microsoft Function value ="<<exp(x)<<endl;
}
```

It is clearly more efficient replace the evaluation of the Taylor polynomial with a loop calculating a sum in the usual way. We can have something like:

```
    double Taylor =0;
    int n=6;
    for(int i =0;i<=n;i++)
    {
        Taylor+=1.0/fact(i)*pow(x,i);
    }
    return Taylor;
```

instead of

```
return 1+x+1.0/2*x*x+1.0/(3*2)*x*x*x+1.0/(4*3*2)*x*x*x*x
            +1.0/(5*4*3*2)*x*x*x*x*x+1.0/(6*5*4*3*2)*x*x*x*x*x*x;
```

if we first define a factorial function `fact(i)`.

This will allow us to compute a Taylor polynomial of higher degrees just by changing the value of the variable `n`

The following program implements the above suggestions and leaves it open to us to increase the degree of the Taylor polynomial just by altering the value of the variable `n`in the definition of the function `Exp`

```
#include <iostream>
#include <cmath>
#include <iomanip>
int fact(int n)
{
    int p=1;
    for(int i=1;i<=n;i++)
    {
        p*=i;
    }
    return p;
}
double Exp(double x)
{
    double Taylor =0;
    int n=6;
    for(int i =0;i<=n;i++)
    {
        Taylor+=1.0/fact(i)*pow(x,i);
    }
    return Taylor;
}
using namespace std;
void main()
{
    double x=1;
    cout<<setprecision(30);
    cout<<"Taylor poly of degree 6 ="<<Exp(x)<<endl;
    cout<<"Microsoft Function value ="<<exp(x)<<endl;

}
```

The problem with this method is that the maximum value of `n` that we can use is quite small. The upper limit will be imposed by the factorial function `fact(n).` This function becomes large so quickly it produces integer overflow once `n` is greater than 12.

The following program produces agreement up to 15 decimal places with the value of the Microsoft version of the exponential function.

```
#include <iostream>
#include <cmath>
#include <iomanip>
int fact(int n)
{
```

```
    int p=1;
    for(int i=1;i<=n;i++)
    {
        p*=i;
    }
    return p;
}
double Exp(double x)
{
    double Taylor =0;

    for(int i =0;i<=12;i++)
    {
        Taylor+=1.0/fact(i)*pow(x,i);
    }
    return Taylor;
}
using namespace std;
void main()
{
    double x=1;
    cout<<setprecision(20);
    cout<<"Taylor poly of degree 12 ="<<Exp(x)<<endl;
    cout<<"Microsoft Function value ="<<exp(x)<<endl;

}
```

Using Horner's metod to evaluate the Taylor polynomial instead produces total agreement with the Microsoft function (at least for the value x=0.1.)

```
#include <iostream>
#include <cmath>
#include <iomanip>
int fact(int n)
{
    int p=1;
    for(int i=1;i<=n;i++)
    {
        p*=i;
    }
    return p;
}
double ExpH(double x)
{
    double a[13] ;
double b[13];

    for(int i =0;i<=12;i++)
    {
        a[i]=1.0/fact(i);
    }

b[12]=a[12];
for(int k=11;k>=0;k--)
```

```
{
b[k]=a[k]+x*b[k+1];
}
    return b[0];
}
using namespace std;
void main()
{
    double x=0.1;
    cout<<setprecision(20);
    cout<<"Taylor poly of degree 12 (Horner's method) ="<<ExpH(x)<<endl;
    cout<<"Microsoft Function value ="<<exp(x)<<endl;

}
```

0.1 Solving Differential Equations

We will consider only differential equations which can be written in the form: $\dfrac{dy}{dx} = F(x, y)$.
Examples:

- $\dfrac{dy}{dx} = y,$ that is $F(x, y) = y$.
- $\dfrac{dy}{dx} = \sqrt{1 - y^2},$ that is $F(x, y) = \sqrt{1 - y^2}$.
- $\dfrac{dy}{dx} = \dfrac{2y}{x},$ that is $F(x, y) = \dfrac{2y}{x}$.