Fall 2011 MATH566, Homework 1

Issued: 9/7/11, Due: 9/16/11

1. Compute the Taylor series around \( x_0 = 0 \) and remainder up to order \( n \) for the functions
   
   a) \( f(x) = \sin x \)
   
   b) \( f(x) = e^{-x} \)
   
   c) \( f(x) = \log(1 + x) \)

Obtain an estimate of the error in using the Taylor series to evaluate \( f(1) \)

SOLUTION. The Taylor series expansion up to order \( n \) around \( x = 0 \) (MacLaurin series), for \( f \in C^\infty(\mathbb{R}) \) is

\[
T_n(x) = f(0) + \frac{1}{1!}f'(0)x + \frac{1}{2!}f''(0)x^2 + \ldots + \frac{1}{n!}f^{(n)}(0)x^n
\]

The remainder is

\[
R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}x^{n+1}
\]

Evaluation at \( x = 1 \) gives

\[
f(1) = f(0) + \frac{1}{1!}f'(0) + \frac{1}{2!}f''(0) + \ldots + \frac{1}{n!}f^{(n)}(0) + R_n(1)
\]

The error in using the truncated Taylor series to evaluate \( f(1) \) is

\[
e = |R_n(1)|
\]

a) With \( n = 2k + 1 \),

\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!}
\]

\[
R_{2k+2}(x) = (-1)^k \frac{\sin(\xi)}{(2k+2)!} \leq \frac{1}{(2k+2)!}
\]

b) With \( p = n \mod 2 \)

\[
e^{-x} = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \ldots + (-1)^p \frac{x^n}{n!}
\]

\[
R_n(x) = (-1)^{p+1} \frac{e^{-\xi}}{(n+1)!}x^{n+1} \leq \frac{1}{(n+1)!}
\]

c) With \( p = n \mod 2 \)

\[
\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots + (-1)^{p-1} \frac{x^n}{n}
\]

\[
R_n(x) = (-1)^p \frac{1}{n+1} \frac{x^{n+1}}{(1+\xi)^{n+1}} \leq \frac{1}{n+1}
\]
Note: Stirling’s formula \( \log(n!) \approx n \log n - n \) gives the approximation \( n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \). This implies very rapidly decreasing error in the Taylor series truncation of \( \sin 1, e^{-1} \) by the above formulas, but slowly decreasing error for \( \log 1 \), with increasing \( n \).

2. Consider an uniform partition of the interval \([0, 1]\) by points \( x_i = ih, h = 1/n, i = 0, 1, \ldots, n \). Obtain an estimate of the error made by polynomial interpolation of the data set \( D = \{(x_i, f(x_i))\} \) for each of the functions above.

**SOLUTION.** The global polynomial interpolation remainder is

\[
R_n(x) = f^{(n+1)}(\xi) \frac{(x-x_0) \cdots (x-x_n)}{(n+1)!},
\]

Since \( |x - x_i| \leq 1 \) for all \( i \) the error estimates from the Taylor series remainders are valid for the polynomial interpolation remainder also.

3. Carry out a pencil and paper computation to find the Newton form of the polynomial interpolant of the Runge function

\[
f(x) = \frac{1}{1+x^2},
\]
on an uniform partition with \( n = 4 \) subintervals of the domain \([-1, 1]\).

**SOLUTION.** Construct the divided difference table

<table>
<thead>
<tr>
<th>( i )</th>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( [y_i, y_{i-1}] )</th>
<th>( [y_i, y_{i-1}, y_{i-2}] )</th>
<th>( [y_i, y_{i-1}, y_{i-2}, y_{i-3}] )</th>
<th>( [y_{i-1}, y_{i-2}, y_{i-3}; y_{i-4}] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1.0</td>
<td>0.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-0.5</td>
<td>0.8</td>
<td>0.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>1.0</td>
<td>0.4</td>
<td>-0.2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>0.8</td>
<td>-0.4</td>
<td>-0.8</td>
<td>-0.4</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
<td>0.5</td>
<td>-0.6</td>
<td>-0.2</td>
<td>0.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>

The interpolating polynomial is

\[
p_4(x) = 0.5 + 0.6(x+1) - 0.2(x+1)(x+0.5) - 0.4(x+1)(x+0.5)x + 0.4(x+1)(x+0.5)x(x-0.5)
\]

4. Write a computer code to determine the Newton form of the polynomial interpolant of the Runge function above for general \( n \). Compare the \( n = 4, 8, 16 \) interpolations with the exact function by plotting both using a fine partition of \([-1, 1]\) into \( m = 100 \) subintervals. What do you observe?

**SOLUTION.** A solution using the Python language and GnuPlot is presented. First, the numpy module is loaded into Python to allow work with vectors, matrices.

**Python session**

```python
from numpy import *

# Runge function

def f(x):
    return 1. / (1. + x**2)
```

Next, the Runge function is defined
We wish to study how the interpolation behaves as we increase the number of data points. Hence, we build a function to generate \( n + 1 \) evenly spaced data points \((x_i, y_i), i = 0, 1, ..., n\) in interval \( x \in [a, b] \).

```python
def GenData(N, f, a, b):
    h = (b-a)/N
    x = a + h*array(range(N+1))
    y = f(x)
    return x, y
```

```python
xData, yData = GenData(4, f, -1., 1.)
print xData
[-1. -0.5 0. 0.5 1. ]
```

```python
print yData
[ 0.5 0.8 1. 0.8 0.5]
```

Next, build a function to return the Newton polynomial coefficients \( a_0, a_1, ..., a_N \).

```python
def NewtonCoef(xData, yData):
    N = len(xData) - 1
    a = yData.copy()  # initialize coefficients to ordinate values
    for i in range(1,N+1):
        for j in range(i,N+1):
            a[j] = (a[j] - a[i-1])/(xData[j]-xData[i-1])
    return a
```

```python
aCoef = NewtonCoef(xData, yData)
print aCoef
[ 0.5 0.6 -0.2 -0.4 0.4]
```

Now, build a function that evaluates the Newton polynomial.

```python
def NewtonPoly(a, xData, xEval):
    N = len(xData)-1
    p = a[N]  # Horner scheme
    for i in range(1,N+1):
        p = a[N-i] + (xEval - xData[N-i])*p
    return p
```

A function to compute the error is also useful.

```python
def err(x):
    return NewtonPoly(aCoef, xData, x) - f(x)
```
The error at the data points should be zero

```
for i in range(5):
    print err(xData[i])
    0.0
0.0
0.0
0.0
0.0
```

Next, compare the value predicted by the interpolation with the exact function \( f \) for other points in \([-1, 1]\). We generate data on the fine partition of \([-1, 1]\) and compare with the interpolation.

```
xEval,yExact=GenData(100,f,-1.,1.)
yInterp=NewtonPoly(aCoef,xData,xEval)
x = open('xy.dat','w')
for j in range(101):
    xy.write(str(j) + ' ' + str(yEval[j]) + ' ' + str(yInterp[j]) + '
')
xy.close()
```

The values in the `xy.dat` file can be plotted with GNUplot.

```
plot "xy.dat" using 1:2 w l, "xy.dat" using 1:3 w p
```
Instead of the function and interpolation we can plot the errors directly.

```python
xEval, yEval = GenData(100, f, -1., 1.)
xy = open('xy.dat', 'w')
for j in range(101):
    errj = abs(err(xEval[j]))  # compute absolute value of error
    ej = log10(max(1.e-17, errj))  # avoid log(0.)
    xy.write(str(xEval[j]) + ' ' + str(ej) + '
')
xy.close()
```

The values in the `xy.dat` file can be plotted with GNUplot.

```
plot "xy.dat" w lp
```

Repeat for \( n = 8 \) subintervals
Python] xData, yData = GenData(8, f, -1., 1.)
Python] aCoef = NewtonCoef(xData, yData)
Python] xEval, yExact = GenData(100, f, -1., 1.)
Python] yInterp = NewtonPoly(aCoef, xData, xEval)
Python] xy = open('xy.dat', 'w')
Python] for j in range(101):
  xy.write(str(xEval[j]) + ' ' + str(yEval[j]) + ' ' + str(yInterp[j]) + '
')
Python] xy.close()


Repeat for \( n = 16 \) subintervals

Python] xData, yData = GenData(16, f, -1., 1.)
Python] aCoef = NewtonCoef(xData, yData)
Python] xEval, yExact = GenData(100, f, -1., 1.)
Python] yInterp = NewtonPoly(aCoef, xData, xEval)
Python] xy = open('xy.dat', 'w')
Python] for j in range(101):
  xy.write(str(xEval[j]) + ' ' + str(yEval[j]) + ' ' + str(yInterp[j]) + '
')
Python] xy.close()
For $f(x) = 1/(1 + x^2)$ the interpolation exhibits convergent behavior. If we however try to interpolate $g(x) = 1/(1 + 25x^2)$ on $[-1, 1]$ (or, equivalently, $f(x)$ on $[-0.2, 0.2]$)

```python
def f(x):
    return 1./(1.+25.*x**2)
```

we observe

```python
xData,yData = GenData(16,f,-1.,1.)
aCoef=NewtonCoef(xData,yData)
xEval,yExact=GenData(100,f,-1.,1.)
yInterp=NewtonPoly(aCoef,xData,xEval)
xy = open('xy.dat','w')
for j in range(101):
    xy.write(str(xEval[j]) + ', ' + str(yEval[j]) + ', ' + str(yInterp[j]) + '
')
xy.close()
```

```gnu
plot "xy.dat" using 1:2 w l, "xy.dat" using 1:3 w p
```
The interpolation in this case diverges.