Problem 1: (20 points)

Consider the problem of evaluating the function sin x, in particular, the propagated data error, which is the error in the function value due to a perturbation h in x.

(a) Estimate the the absolute and relative error in evaluating sin $\boldsymbol{x}$.
(b) Estimate the absolute and relative condition number for evaluating sin $\boldsymbol{x}$.
(c) Comment on your results from parts (a) and (b).

(a)
absolute Error =

$$|\sin(x+h)-\sin(x)| \approx h\cos(x)$$

relative Error =

$$\frac{|\sin(x+h)-\sin(x)|}{\sin(x)} \approx \frac{h}{\tan(x)}$$

(b)
absolute condition number =

$$|\frac{d(\sin x)}{dx}| = |\cos(x)|$$

relative condition number =

$$|\frac{d(sinx)}{dx}\frac{x}{\sin(x)}| = |\frac{x}{\tan(x)}|$$

(c) As we can see that the absolute error is bounded by h. However, the relative error is not bounded, and when $x->k\pi, \forall k \in \mathbf{Z}$, the error is tromendously huge.

(d) The absolute condition number is bounded by 1, however, the relative condition number is relatively large when $x->\infty$ or when $x->k\pi, \forall k \in \mathbf{Z}$, and the problem is then ill-conditioned

Problem 2: (10 points)

In a floating-point number system having an underflow level of UFL = 10^{-38}, which of the following computations will incur an underflow?

(a) $a = \sqrt{b^2 + c^2}$, with $b = 1$, $c = 10^{-25}$
(b) $a = \sqrt{b^2 + c^2}$, with $b = c = 10^{-25}$
(c) $u = (v \times w)/(y \times z)$, with $v = 10^{-15}$, $w = 10^{-30}$, $y = 10^{-20}$, and $z = 10^{-25}$
In each case where underflow occurs, is it reasonable simply to set to zero the quantity that underflows?

For (a) $c^2 = 10^{-50}$ and $b^2 = 1$, so c^2/b^2 way smaller than 10^{-35}, so underflow accurs. Using taylor approximation, it should be around $b + \frac{c^2}{2b} = 1 + 5 \times 10^{-26}$, since the error is relatively small, it is reasonable.
For (b) the underflow will ocurr with $b^2 < 10^{38}$, $c^2 < 10^{38}$, the result will directly goes to 0 whereas the expected result should be around $\sqrt{2} \times 10^{-25}$, all the infomation is lost, so it is not reasonable.
For (c) the underflow will occur with ($v \times w$) and ($y \times z$), the dividing zero will occur, so it is not reasonable.

Problem 3: (15 points)

Consider the function

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

(a) For what range of values of x is it difficult to compute $f(x)$ accurately in floating-point arithmetic and why? Give two reasons.
(b) Give a rearrangement of the terms, called $g(x)$, such that the computation is more accurate in floating-point arithmetic for the range of x given in part (a).
(c) Write code to verify your conclusions about $f(x)$ and $g(x)$. Test several different values of x, and explain your results.

(a) For the range where $\cos x$ is approximated to one, and $\sin x \in (-10^{-6}, 10^{-6})$.
First, when x->0, the denominator and numerator are both approaching zero, thus this might result in NAN which cos(x) in this case is very close to one, so $1 - \cos x$ will lose all the significant digits due to cancellation even resulting always zero.
Second, if x is small enough, $sinx \approx x$ is in this case a very small number, if x<10^{-20}, it will cause underflow.
(b) The expression is following

$$f(x) = \frac{1 - \cos x}{\sin^2 x} = \frac{(1 - \cos x)(1 + \cos x)}{(\sin^2 x)(1 + \cos x)} = \frac{1}{1 + \cos x}$$

(c) If we do Taylor expansion on $f(x)$, it is approximately

$$\frac{1}{2} + \frac{x^2}{8} + O(x^4)$$

Thus, when x is small, it should approach 2 like a quadratic function.
(d) Here is the code

```
from numpy import sin, cos
from numpy import arange

def f(x):
    return (1-cos(x))/(sin(x)*sin(x))
def g(x):
    return 1/(1+cos(x))
def approx(x):
    return 1/2.0+x*x/8.0

h= arange(0.0, 10.0, 1.0)
x= 10**(-h)

for line in zip(h, f(x), g(x), approx(x)):
    print "x: 10^-%d, f:%f, g:%f, approx:%f"%line
```

result

```
x: 10^-0, f:0.649223, g:0.649223, approx:0.625000
x: 10^-1, f:0.501252, g:0.501252, approx:0.501250
x: 10^-2, f:0.500013, g:0.500013, approx:0.500012
x: 10^-3, f:0.500000, g:0.500000, approx:0.500000
x: 10^-4, f:0.500000, g:0.500000, approx:0.500000
x: 10^-5, f:0.500000, g:0.500000, approx:0.500000
```

```
x: 10^-6, f:0.500044, g:0.500000, approx:0.500000
x: 10^-7, f:0.499600, g:0.500000, approx:0.500000
x: 10^-8, f:0.000000, g:0.500000, approx:0.500000
x: 10^-9, f:0.000000, g:0.500000, approx:0.500000
```

As we can see from the above calculation, due to cancellation, when h is below the magnitute of 10^{-6}, data started to fluctuate, and it completely disappear when it reaches below 10^{-8}

Problem 4: (10 points)

> (a) Compute the unit roundoff ε_{mach} and the underflow level (UFL) based on the IEEE double precision standard.
>
> (b) Write code to determine the approximate values of the ε_{mach} and UFL. Comment on your approach and provide your numerical results.

(a) For percision p, the unit roundoff ϵ_{mach} can computed as the smallest number ϵ such that $fl(1+\epsilon) > 1$. It is usually $\epsilon_{mach} = \frac{1}{2}\beta^{1-p}$ and for here $\beta = 2$ and $p = 53$, so $\epsilon_{mach} = 2^{-53}$
The underflow level should be the smallest number that can be represended by floating point. UFL=2^{-1074}

(b) The code is as below

```python
import numpy as np

def machine_percision_test(p, b, data_type):
    one = data_type(1) #test against
    return one+(b**(-p))!=1 # fl(1+epsilon)!=fl(1)

def machine_undeflow_test(p, b, data_type):
    one = data_type(1) # test against
    return b**(-p)!=0

def epsilon_mach(data_type):
    p = 1
    b = data_type(2) # assume base of 2
    while(machine_percision_test(p, b, data_type)):
        p += 1
    return p

def UFL(data_type):
    p = 1
    b = data_type(2) # assume base of 2
    while(machine_undeflow_test(p, b, data_type)):
        p += 1
    return p-1


print "The epsilon_machine for float 64 is 2^{-%d}, the UFL is 2^{-
%d}"%(epsilon_mach(np.float64), UFL(np.float64))
print "The epsilon_machine for float 128 is 2^{-%d}, the UFL is
2^{-%d}"%(epsilon_mach(np.float128), UFL(np.float128))
```

result:

The epsilon_machine for float 64 is 2^{-53}, the UFL is 2^{-1074}
The epsilon_machine for float 128 is 2^{-64}, the UFL is 2^{-16445}

The code above is the way I am examing the ϵ_{mach}. Assume the base of two, and the code below is trying to calculate the percision p. It examines the float64 and float128 respectively. Using the fact that

$$fl(1+\epsilon) - 1! = 0$$

Problem 5: (15 points)

Suppose you need to generate $n+1$ equally spaced points on the interval $[a, b]$, with spacing $h = (b-a)/n$.
(a) In floating-point arithmetic, which of the following methods,

$$x_0 = a, \quad x_k = x_{k-1} + h, \quad k = 1, \dots, n$$

or

$$x_k = a + kh, \quad k = 0, \dots, n,$$

is better, and why?
(b) Write a program implementing both methods and find an example, say with $a = 0$ and $b = 1000$, that illustrates the difference between them. You should find an example such that $n < 10^9$. Explain why your example illustrates the behavior. You should find a way to clearly and concisely illustrate your example. Printing out a long list of numbers is not an acceptable choice.

Hint: You may want to adjust the precision of your output to aid in finding an example. For this, you may find the following commands helpful:

+ `format longe` (MATLAB)
+ `np.set printoptions(formatter={'float': '{: 0.16f}'.format})` (Python)

(a) The second method is better since every floating point operation will leads to protential percision lost. For the first method, To compute x_k, there are k oporation that can lead to percision lost and for the second method, there are only two oporation
(b) The code is as below

```
from numpy import arange, float16
import numpy as np
a = float16(0.0)
b = float16(1000.0)
n = float16(1e4)
np.set_printoptions(formatter={'float': '{: 0.16f}'.format})
def add_method(a, b, n):
    h = (b-a)/n
    x = a
    xs = list()
    for k in range(int(n)+1):
        xs.append(x)
        x = x+h
    return xs

def mult_method(a, b, n):
    h = (b-a)/n
    xs = list()
    for k in range(int(n)+1):
        x = a+k*h
        xs.append(x)
    return xs

print "The x(1000) for addition method is: %f"%add_method(a, b, n)
[-1]
print "The x(1000) for multiplication is: %f"%mult_method(a, b, n)
[-1]
```

The result:
The x(1000) for addition method is: 256.000000
The x(1000) for multiplication is: 999.755859

The example above clearly illustrated the point that the addition has more damaging percision lose due to multiple floating point calculation. The actual x_{10000} value should be 1000. As we can see, if we have very low percision, even 1×10^4 iterations can leads to davastating error.

Problem 6: (20 points)

(a) Write a program to compute an approximate value for the derivative of a function using the
finite-difference formula

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Test your program using the function $\cos(x)$ for $x = 1$.
Determine the error by comparing with the true derivative, $-\sin(x)$.
Plot the magnitude of the error as a function of h, for $h = 10^{-k}$, $k = 0, \ldots, 16$. You should use log scale for h and for the magnitude of the error.
Is there a minimum value for the magnitude of the error?
How does the corresponding value for h compare with the rule of thumb $h \approx \sqrt{\varepsilon_{mach}}$ (derived in Example 1.3)?
Your program should use double precision.
(b) Now, consider the 4th order central finite-difference formula for the first derivative

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

The truncation error for the finite difference approximation is bounded by $\frac{Mh^4}{30}$, where M is a bound on $|f^{(5)}(t)|$ for t near x. The rounding error for evaluating the finite difference formula is bounded by 3ε. The total computational error is therefore bounded by the sum of the two
functions

$$\frac{Mh^4}{30} + \frac{3\varepsilon}{2h}.$$

Determine when the bound on the total computational error is minimized. Your answer should be an expression for h.
(c) Repeat part (a) with the central difference formula given in part (b) this time comparing with the rule of thumb determined in part (b).
(d) What would you expect to see occur if you performed the same process with the 2nd order centered finite difference formula for first derivative? Explain. You do not need to

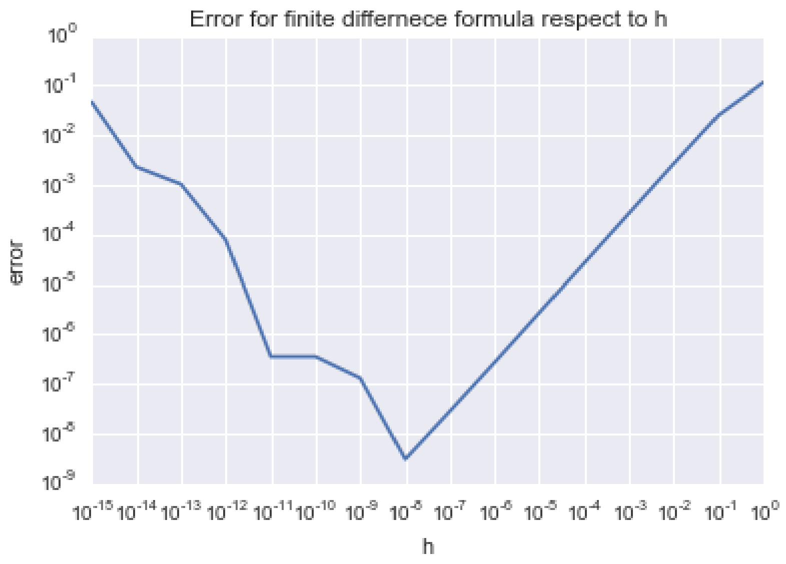
> actually implement this one, just give a general description of what you would expect to see.

(a) code as below (used in ipython notebook)

```
from numpy import abs
def finiteDiff(f, x, h):
    return (f(x+h)-f(x))/h

def error(f, df, x, h):
    return abs(finiteDiff(f, x, h)-df(x))

%matplotlib inline
from numpy import sin, cos, minimum, arange
from matplotlib import pyplot as plt
import seaborn

dcos = lambda x: -sin(x)

sin_error = lambda h: error(cos, dcos, 1, h)

k = arange(0, 16, 1) # the testing value
h = 10.0**(-k)

sin_errors = sin_error(h)
plt.loglog(h, sin_errors)
plt.title("Error for finite differnece formula respect to h")
plt.xlabel("h")
plt.ylabel("error")

print "The smallest error is %e"%min(sin_errors)
```

The result as below

The smallest error is 3.025119e-09



(b)Clearly we can see that the smallest value is 3.025119×10^{-9}, and it is clearly near $\sqrt{\varepsilon_{mach}} \approx 1.05 \times 10^{-8}$, they are pretty close to each other
Since $\varepsilon = 2^{-53}$ and M is bounded by $|\sin(1)| = 0.841$

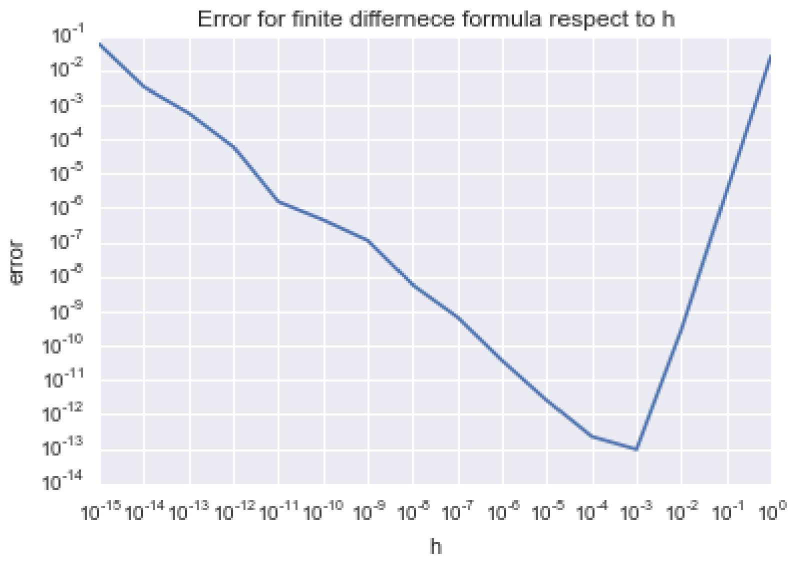
$$\frac{Mh^4}{30} + \frac{3\varepsilon}{2h} = \frac{\sin(1) \times h^4}{30} + \frac{3 \times 2^{-53}}{2h}$$

$\leq 2.26816669290816132025462052888066821382804975 6 \times 10^{-13} when\ h \approx 0.00108291$

(c)Code is as below (using an ipython notebook)

```
def finiteDiff4(f, x, h):
    return (-f(x+2*h)+8*f(x+h)-8*f(x-h)+f(x-2*h))/(12*h)



k = arange(0, 16, 1) # the testing value
h = 10.0**(-k)

sin_error4 = lambda h: abs(finiteDiff4(cos, 1, h)-dcos(1))

sin_errors4 = sin_error4(h)
plt.loglog(h, sin_errors4)
plt.title("Error for finite differnece formula respect to h")
plt.xlabel("h")
plt.ylabel("error")

print "The smallest error is %e"%min(sin_errors4)
```

The result is a below



(d) The error should not be between the fourth difference equation and the first difference equation. plot of h versus error should have the same trend as the fourth order and the first order, but with a different minimum between the first and the fourth.

Problem 7: (10 points)

> Write a program to compute the absolute and relative errors in Stirling's approximation
>
> $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$
>
> for $n = 1, \ldots, 10$. Does the magnitude of the absolute error grow or shrink as n increases? Does the magnitude of the relative error grow or shrink as n increases?
>
> **Hint**: When plotting errors, it is often most appropriate to use a either a log-log or semi-log plot, depending on the problem. For this problem you may find `plt.semilogy (Python)` or `semilogy (MATLAB)` helpful.

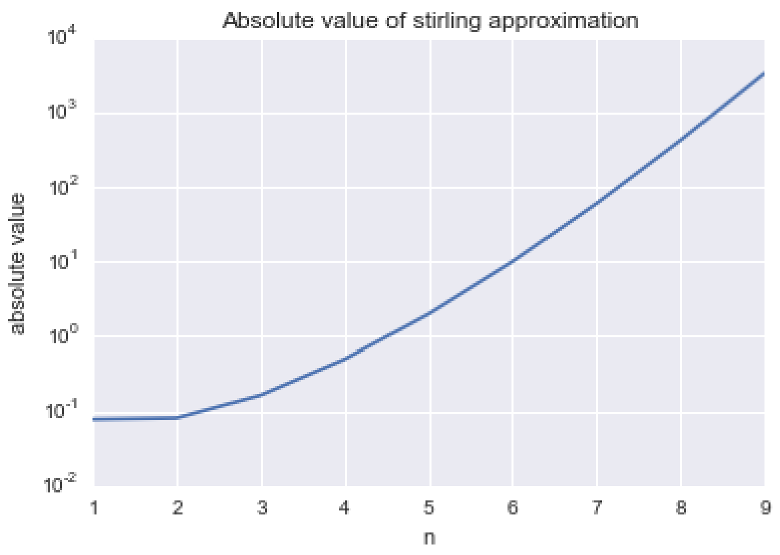
The code is as below:

```
from numpy import sqrt,pi,e,abs,arange

import matplotlib.pyplot as plt
def sterling(n):
    return sqrt(2*pi*n)*(n/e)**n

def fact(n):
    accum = 1
    for x in range(1,n+1):
        accum *= x
    return accum

def absolute_error(real, approx, x):
    return abs(real(x)-approx(x))

def relative_error(real, approx, x):
    return abs(real(x)-approx(x))/real(x)

fact_abs_error = lambda x: absolute_error(fact, sterling, x)
fact_rel_error = lambda x: relative_error(fact, sterling, x)
```

The absolute error code as follow:

```
%matplotlib inline
import seaborn

x = np.arange(1, 10, 1) # the testing value

plt.semilogy(x, map(fact_abs_error, x))
plt.title("Absolute value of stirling approximation")
plt.xlabel("n")
plt.ylabel("absolute value")
```

The result as below

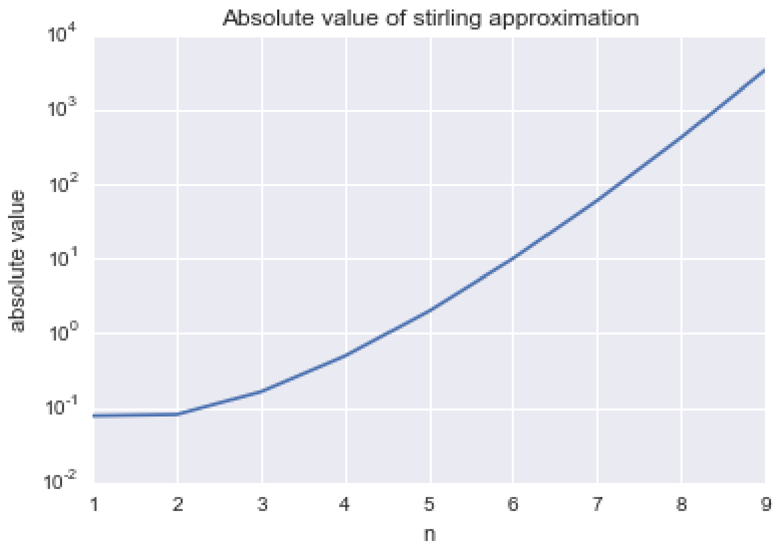


The relative error code as follow:

```
plt.semilogy(x, map(fact_rel_error, x))
```

```
    plt.title("relative value of stirling approximation")
    plt.xlabel("n")
    plt.ylabel("absolute value")
```

The result as below



As a result, the absolute error of stirling approximation increases as n goes up
the relative error of stirling approximation decreases as n goes up