Homework #1, 10/03/17

Ramaseshan Parthasarathy
Akhil Velagapudi
Tarun Sreenathan

Problem 1

The code to compute absolute and relative error given approximation is as follows:

```
import numpy as np
import math as math

def computeError(value, approx):
        absolute = np.float32(math.fabs(np.float32(value - approx)))
        relative = np.float32(math.fabs(np.float32(value - approx))/np.float32(math.fabs(value)))

        print("Absolute Error: %s" % (absolute))
        print("Relative Error: %s\n" % (relative))


print("a.")
computeError(np.float32(math.pi), np.float32(3))

print("b.")
computeError(np.float32(math.pi), np.float32(3.14))

print("c.")
computeError(np.float32(math.pi), np.float32(22/7.0))
```

The above code generated the following errors for pi given three different approximations:

```
Using single precision:

a.
Absolute error: 0.141593
Relative error: 0.0450704
b.
Absolute error: 0.00159264
Relative error: 0.000506952
c.
Absolute error: 0.00126433
Relative error: 0.00040245
```

Problem 2

Machine epislon is stated to be the smallest ε such that $1 - \varepsilon < 1 < 1 + \varepsilon$. It is the *smallest* quantity representable by the computer and provides spacing between machine representable integers. In a single precision, it can store apprimxately 7 digits ($\approx 2^{-23}$) while in double precision it can store up to 15 digits ($\approx 2^{-56}$). During floating-point computations, the machine epsilon forms an upper bound on relative error. Thus, the inequality $1 + \varepsilon \neq 1$ holds true.

Problem 3

The code to compute the sterling approximation is as follows:

```
import numpy as np
import math as math

def power32(a, b):
        return np.float32(a ** b)

# compute the sterling approximation for
# double and single precision
def approximate_sterling(count):
        #initialize variabales
        fact = 1
        approx = None
        abserr = None
        relerr = None

        print("DOUBLE PRECISION: ")
        for n in range (1, count):
                fact  = fact * n
                root = math.sqrt(2.0 * math.pi * n)
                expo = math.exp(-n)
                power = math.pow(n, n)
                approx = root * expo * power
                abserr = math.fabs(fact-approx)
                relerr = abserr/fact
                print("n = %s, sterling approximation = %s, absolute error = %s, relative error = %s"
                          % (n, approx, abserr, relerr))

        #re-initialize variables
        fact = 1
        approx = None
        abserr = None
        relerr = None

        #single precision
        print("SINGLE PRECISION: ")

        for n in range (1, count):
                fact  = fact * n
                root = np.float32(math.sqrt(np.float32(np.float32(2.0)*np.float32(math.pi))*np.float32(n)))
                expo = power32(np.float32(math.e), np.float32(-n))
                power = power32(np.float32(n), np.float32(n))
                approx = np.float32(np.float32(root * expo)) * power
                abserr = math.fabs(fact - approx)
                relerr = abserr/fact
                print("n = %s, sterling approximation = %s, absolute error = %s, relative error = %s"
                          % (n, approx, abserr, relerr))

#execute method for n = 1,2,...,10
approximate_sterling(11)
```

Running the above code would generate an output like so:

```
rpartha@rpartha-HP-Notebook:~/Documents/Numerical_Analysis$ python3 sterling_approx.py
DOUBLE PRECISION:
n = 1, sterling approximation = 0.9221370088957891, absolute error = 0.07786299110421091, relative error = 0.07786299110421091
n = 2, sterling approximation = 1.9190043514889832, absolute error = 0.08099564851101682, relative error = 0.04049782425550841
n = 3, sterling approximation = 5.836209591345864, absolute error = 0.16379040865413597, relative error = 0.027298401442355995
n = 4, sterling approximation = 23.50617513289329, absolute error = 0.49382486710671003, relative error = 0.02057603612944625
n = 5, sterling approximation = 118.01916795759008, absolute error = 1.980832042409915, relative error = 0.01650693368674929
n = 6, sterling approximation = 710.0781846421849, absolute error = 9.921815357815149, relative error = 0.013780299108076596
n = 7, sterling approximation = 4980.395831612461, absolute error = 59.604168387539175, relative error = 0.011826223886416503
n = 8, sterling approximation = 39902.3954526567, absolute error = 417.6045473432969, relative error = 0.010357255638474625
n = 9, sterling approximation = 359536.8728419483, absolute error = 3343.127158051706, relative error = 0.00921276223008076
n = 10, sterling approximation = 3598695.6187410364, absolute error = 30104.381258963607, relative error = 0.008295960443938384
SINGLE PRECISION:
n = 1, sterling approximation = 0.922137, absolute error = 0.077863, relative error = 0.077863
n = 2, sterling approximation = 1.919, absolute error = 0.0809957, relative error = 0.0404978
n = 3, sterling approximation = 5.83621, absolute error = 0.16379, relative error = 0.0272984
n = 4, sterling approximation = 23.5062, absolute error = 0.493824, relative error = 0.020576
n = 5, sterling approximation = 118.019, absolute error = 1.98083, relative error = 0.0165069
n = 6, sterling approximation = 710.078, absolute error = 9.92181, relative error = 0.0137803
n = 7, sterling approximation = 4980.4, absolute error = 59.604, relative error = 0.0118262
n = 8, sterling approximation = 39902.4, absolute error = 417.605, relative error = 0.0103573
n = 9, sterling approximation = 359537.0, absolute error = 3343.12, relative error = 0.00921276
n = 10, sterling approximation = 3.5987e+06, absolute error = 30104.2, relative error = 0.00829592
```

From the output, we can see that for both single and double precision, as *n* increases, the absolute error **increases** but the relative error **decreases**. It can be seen that, for some odd reason, that the single precision that uses float32 is more precise than double precision (default to Python3) but remains unchanged.

Problem 4

$$\|x\|_\infty \leq \|x\|_1 \leq n \cdot \|x\|_\infty$$
$$\|x\|_2 \leq \|x\|_1 \leq \sqrt{n} \cdot \|x\|_2$$

$$\|x\|_1 \leq \sqrt{n} \cdot \|x\|_2$$
$$\frac{\|x\|_1}{\sqrt{n}} \leq \|x\|_2, n > 0$$

$$\left.\begin{matrix} \frac{\|x\|_1}{\sqrt{n}} \leq \|x\|_2 \leq \|x\|_1 \\ \|x\|_\infty \leq \|x\|_1 \end{matrix}\right\} \Rightarrow \left.\begin{matrix} \frac{\|x\|_\infty}{\sqrt{n}} \leq \|x\|_2 \leq \|x\|_1 \\ \|x\|_1 \leq n \cdot \|x\|_\infty \end{matrix}\right\} \Rightarrow \frac{\|x\|_\infty}{\sqrt{n}} \leq \|x\|_2 \leq n \cdot \|x\|_\infty$$

Two vector norms $\|x\|_a$ and $\|x\|_b$ are considered equivalent if there exist real numbers c,d > 0 such that:
$c\|x\|_a \leq \|x\|_b \leq d\|x\|_a$

$\therefore \|x\|_2 \equiv \|x\|_\infty$

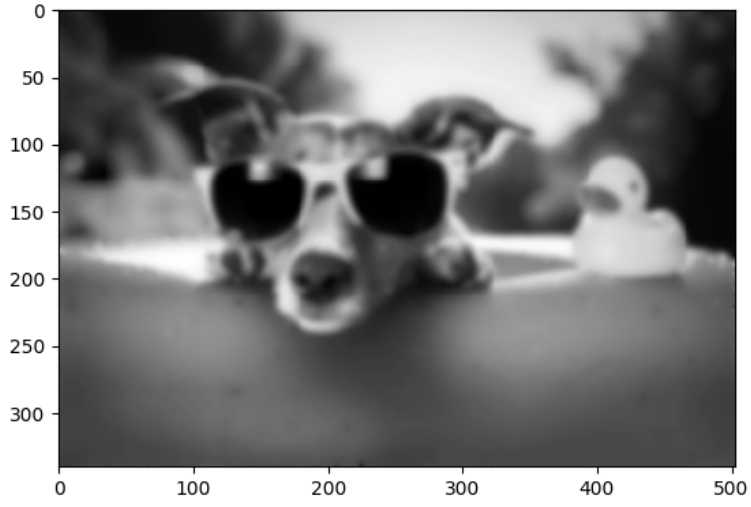
Problem 5

The modified code is as follows:

```
# blur.py
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.sparse import lil_matrix

# read image file
fname = 'chill.jpg'
image = Image.open(fname).convert("L")
arr = np.asarray(image)
arr.setflags(write = 1)

# initialize blurring matrix
m = arr.shape[0]
n = arr.shape[1]
dofs = m*n
```

```
A = lil_matrix((dofs,dofs))
A.setdiag(np.ones(dofs))
for i in range(1,m-1):
    for j in range(1,n-1):
        A[n*i+j,n*i+j] = 8./16.
        A[n*i+j,n*(i-1)+j] = 1./16.
        A[n*i+j,n*(i-1)+(j-1)] = 1./16.
        A[n*i+j,n*(i-1)+(j+1)] = 1./16.
        A[n*i+j,n*i+j-1] = 1./16.
        A[n*i+j,n*i+j+1] = 1./16.
        A[n*i+j,n*(i+1)+j] = 1./16.
        A[n*i+j,n*(i+1)+(j-1)] = 1./16.
        A[n*i+j,n*(i+1)+(j+1)] = 1./16.
A = A.tocsr()

# Blurring function - converts image to a vector, multiplies by
# the blurring matrix, and copies the result back into the image
def blur():
    x = np.zeros(shape=(dofs,1))
    for i in range(0,m):
        for j in range(0,n):
            x[n*i+j] = arr[i,j]

    y = A.dot(x)
    for i in range(0,m):
        for j in range(0,n):
            arr[i,j] = y[n*i+j]

# Execute the blurring function 20 times
for i in range(0,20):
    blur()

# Display the blurred image
plt.imshow(arr,cmap='gray')
plt.show()
```

This code results in the following output image:



As opposed to the original code, which blurred the background but focused on the dog/duck, this code blurs the entire

picture. It works in the same way in that the operation performed on the pixel replaces the grayscale value at every pixel by the weighted average of it's neighbors.