Homework 2

Louis Carnec
15204934
September 29, 2015

Exercise 0.1

Below is the code used in demonstrating counter-examples to ring theory using Python's floating point numbers.

```
from numpy import *

# Associativity of +
#If the result of an arithmetic operation is an integer outside this range then
# an integer overflow occurs.
k = (np.float32(0.2) + np.float32(0.2)) + np.float32(1.0)
j = np.float32(0.2) + (np.float32(0.2) + np.float32(1.0))
print k,j,k==j
print "%.32f %.32f" %(k,j)

#Commutativity of +
#a+b = b+a
x = np.float64(0.1) + np.float64(0.2) + np.float64(0.3)
y =  np.float64(0.3) + np.float64(0.2) + np.float64(0.1)
print x,y,x==y
print "%.60f %.60f" % (x,y)

#-a is the additive inverse of a
#a + (a) = 0
x = 1.0/3.0
y = 1/3
xminusy = x - y
print xminusy
print x,y,x==y
print "%.60f %.60f" % (x,y)
```

```
#a + (a) = 0
x = 1.0/3.0
xminusx = round(x,0) + (-x)
print xminusx

# Associativity of Multiplication
k = (np.float32(math.pi) * np.float32(1.0)) * np.float32(1.0)
j = np.float64(math.pi) * (np.float64(1.0) * np.float64(1.0))
print k,j,k==j
print "%.64f %.64f" %(k,j)

# Left Distributivity
x = np.float16(1.0) * (np.float32(0.1) + np.float64(0.2))
y = (1.0 * 0.1) + (1.0 * 0.2)
print x,y,x==y
print "%.60f %.60f" % (x,y)

# Right Distributivity
x = (np.float32(0.1) + np.float64(0.2)) * np.float16(1.0)
y = (1.0 * 0.1) + (1.0 * 0.2)
print x,y,x==y
print "%.60f %.60f" % (x,y)
```

Exercise 0.2

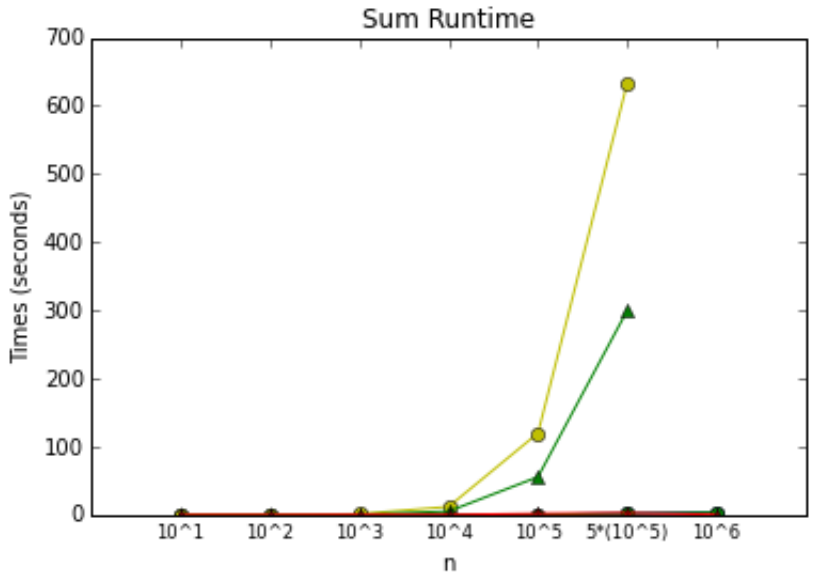
In this exercise, we sum n numbers in a number of numerical representation (integers, floating point, decimal, fraction and symbolic numbers). Using a *python* function called *timeit*, the runtime for each function is measured in seconds.
The code below demonstrates how the sum was calculated for a range of numbers from 1 to 10, to, 1 to 1,000,000. I inputted each of the numbers (x) to be summed from 1 to x manually into each timeit function. The result after timing each sum was then manually inputted into a list for each numerical representation (I tried to use a code which would automate this process but was unsuccessful unfortunately). The increments in the numbers I used for summing were of 10^n, where x is 2 to 6. Due to the fact that the summations of some numbers were taking too long to compile, I included the summation of $5 * 10^5$.
I started timing with 100 repetitions but it was taking too long to compile so I changed to 10 repetitions.
Looking at plot 1 and 2, we can see that up to the summation of 10^2, the run time is the same for all numerical representations.
Plot 1 demonstrates that decimals (yellow circle) and fractions (green pyramid)

take longest to compile, where decimals take longer to compile at an increasing rate relative to fractions.
The rest of the numerical representations appear to take the same amount of time relative to decimals and fractions but by excluding these we can observe the way integers, floating points and symbolic numbers act.
Both integers and floating points are relatively similar in compiling time, however this could be due to the fact that only one zero was inputted for floating points. What is interesting is that the symbolic number representation $\sqrt{n}$ does not follow a positive relationship. This makes sense, some numbers will require the calculation of fewer decimals than other when taking their square root.

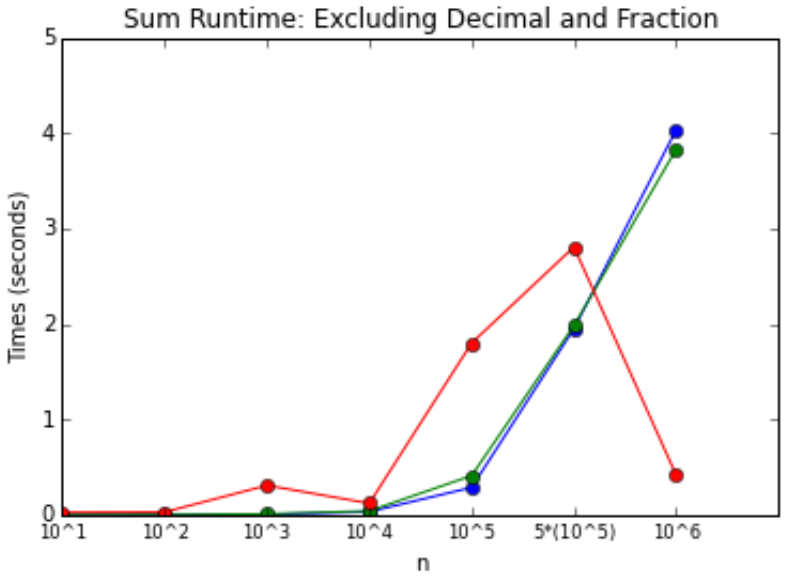
```
import timeit
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from decimal import Decimal as D
from fractions import Fraction as F
from sympy import *

def theSum(init=0, end=0):
    result=0
    while init <= end:
        result = result + init
        init += 1
    return result

#run = timeit.timeit(lambda: theSum(0,10), number=10)
#run1 = timeit.timeit(lambda: theSum(0,10), number=100)
IntegerList = [3.98473243877e-05,0.000419604400577,0.00200987489188,0.031300677042
, 1.95032789518,4.03597066271]
IntegerList1 = [0.000323608573694, 0.00221997896546,0.0251116630704,
               0.224708722096,2.70572147203,18.2572223275,38.0584542136]

#run = timeit.timeit(lambda: theSum(0.0,10.0), number=10)
#run1 = timeit.timeit(lambda: theSum(0.0,10.0), number=100)

FloatingPointList = [0.000124975699237,0.000724496806924,0.00746412835178,0.039585
, 0.409503707612, 1.98173845424, 3.83093263266]
FloatingPointList1 = [0.00105655784319,0.00373659228103,0.0494106822223,0.374828082
,3.68944323628,17.8895178595,35.9459682356]
```

```
#run = timeit.timeit(lambda: theSum(D(0),D(500000)), number=10)
#run1 = timeit.timeit(lambda: theSum(D(0),D(10)), number=100)
DecimalList = [0.0309988033728,0.142998764733, 1.15926613303,11.7853503123,119.7614
DecimalList1 = [0.145675780434,1.19396288886,11.8938720853,123.059856719,1433.3360

#run = timeit.timeit(lambda: theSum(F(0,1),F(1000000,1)), number=10)
#run1 = timeit.timeit(lambda: theSum(F(0,1),F(500000,1)), number=100)
FractionList = [0.0065663560581,0.0842408662083,0.515851990132,5.01664410664,55.554
FractionList1 = [0.0588273294686, 0.514672871579,5.14962066555,54.5055224769, 534.8

#run = timeit.timeit(lambda: theSum(sqrt(1),sqrt(1000000)), number=10)
SymbolicList = [0.0248013369383,0.0276836267349,0.307705265039,0.119238288207,1.798
,2.80614578566,0.413982909122]

print IntegerList
print FloatingPointList
print DecimalList
print FractionList
print SymbolicList

#print run
#print run1

fig = plt.figure()
plt.plot(IntegerList,'bo',FloatingPointList, 'go',DecimalList, 'yo',FractionList,'g
plt.axis([-1,7,0,700])
plt.xticks(arange(7), ('10^1','10^2','10^3','10^4','10^5','5*(10^5)','10^6'), size=
plt.xlabel('n')
plt.ylabel('Times (seconds)')
plt.title('Sum Runtime')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.show()

#plt.plot(IntegerList, 'ro',FloatingPointList,'g^',DecimalList,'yo',FractionList,
#plt.axis([0,6,0,1500])
#plt.show()

plt.plot(IntegerList,'bo',FloatingPointList, 'go',SymbolicList, 'ro',linestyle='-')
plt.axis([0,7,0,5])
plt.xticks(arange(7), ('10^1','10^2','10^3','10^4','10^5','5*(10^5)','10^6'), size=
```

```
plt.xlabel('n')
plt.ylabel('Times (seconds)')
plt.title('Sum Runtime: Excluding Decimal and Fraction')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.show()
```



Plot 1: Plot of runtime for integers, floating point, decimal, fraction and symbolic.



Plot 2: Plot of runtime for integers, floating point and symbolic. (Excluding decimals and fractions).

Exercise 0.3

This code implements a sequence:

$$x_n = 108 - \frac{(815 - \frac{1500}{x_{n-2}})}{x_{n-2}},$$

where $x_0 = 4$ and $x_1 = 4.25$.

Lists are created containing 30 iterations of the sequence for both floating-point numbers and decimals of increaing precision. The resulting lists are then plotted using the matplot library in python.
Looking at the list output, the rounded result for floating-point numbers is 99.672. The result for decimals of increasing precision is Decimal('1E+2') (or 100) for precisions of 10 to 100 decimals.
Plot 3 of floating point numbers undershoots the value, but unlike decimals then overshoots the value. Decimals (see plot 4) do not overshoot the final result but at first estimate an even lower number (around 200) expect decimals of precision 1.

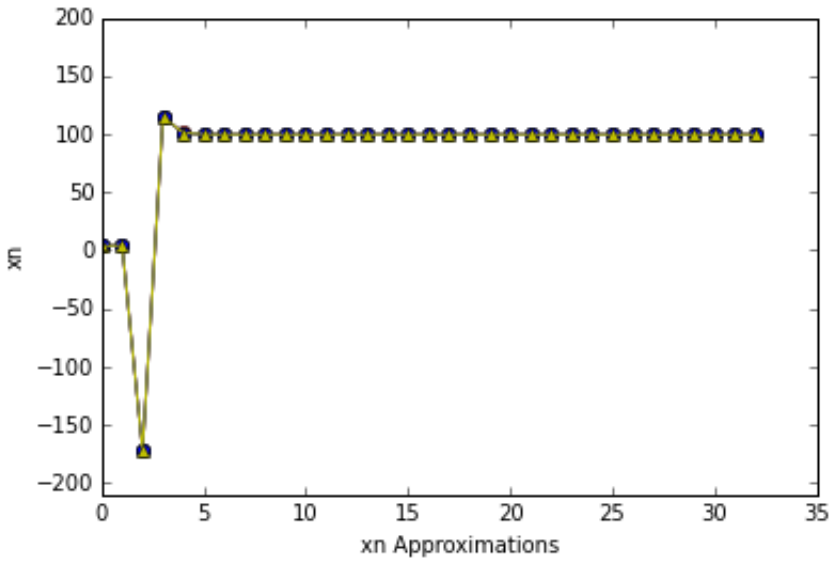
```
from decimal import Decimal as D
from decimal import *
```

```
def formula(stop, z = ""):
    mylist = [4.0,4.25]
    mylist1 = [4.0,4.25]
    mylist2 = [4.0,4.25]
    mylist3 = [4.0,4.25]

    x0 = 4.0
    x1 = 4.25
    i = 0
    if z == "%.1f":
        while i <= stop:
            xn = 108.0 - (815.0/x1)-(1500.0/(x0*x1))
            i = i+1
            x0=x1
            x1 = xn
            xn = ("%.1f" % xn)
            mylist.append(xn)
        return mylist
    if z == "%.5f":
        while i <= stop:
            xn = 108.0 - (815.0/x1)-(1500.0/(x0*x1))
            i = i+1
            x0=x1
            x1 = xn
            xn = ("%.5f" % xn)
            mylist1.append(xn)
        return mylist1
    if z == "%.10f":
        while i <= stop:
            xn = 108.0 - (815.0/x1)-(1500.0/(x0*x1))
            i = i+1
            x0=x1
            x1 = xn
            xn = ("%.10f" % xn)
            mylist2.append(xn)
        return mylist2
    if z == "%.100f":
        while i <= stop:
            xn = 108.0 - (815.0/x1)-(1500.0/(x0*x1))
            i = i+1
```

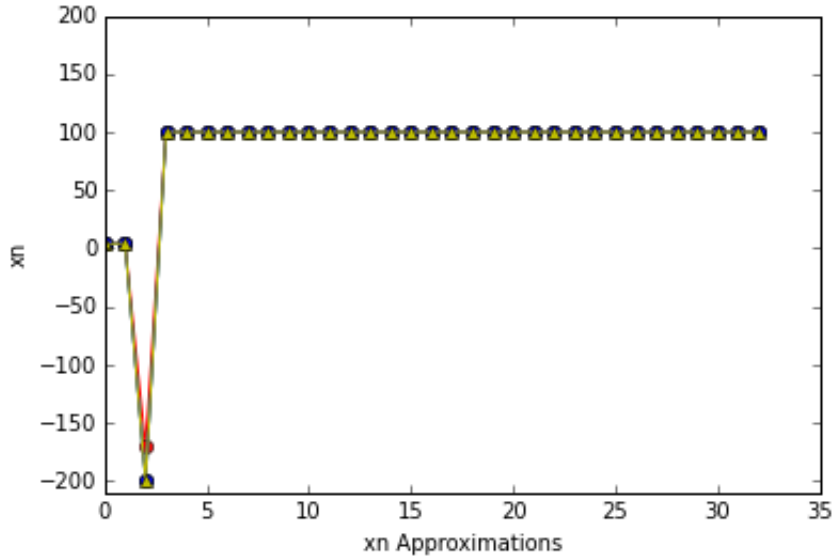
```
            x0=x1
            x1 = xn
            xn = ("%.100f" % xn)
            mylist3.append(xn)
        return mylist3


def formula1(stop, z = "1"):
    myList = [4,4.25]
    myList1 = [D(4),D(4.25)]
    myList2 = [D(4),D(4.25)]
    myList3 = [D(4),D(4.25)]
    x0 = D(4.0)
    x1 = D(4.25)
    i=0
    if "1":
        while i <= stop:
            xn = D(108) - (D(815.0)/x1)-(D(1500.0)/(x0*x1))
            i = i+1
            getcontext().prec = 1
            x0=x1
            x1 = xn
            myList.append(xn)
        return myList
    if "5":
        while i <= stop:
            xn = D(108.0) - (D(815.0)/x1)-(D(1500.0)/(x0*x1))
            i = i+1
            getcontext().prec = 5
            x0=x1
            x1 = xn
            myList1.append(xn)
        return myList1
    if "10":
        while i <= stop:
            xn = D(108.0) - (D(815.0)/x1)-(D(1500.0)/(x0*x1))
            i = i+1
            getcontext().prec = 10
            x0=x1
            x1 = xn
```

```
            myList2.append(xn)
        return myList2
    if "100":
        while i <= stop:
            xn = D(108.0) - (D(815.0)/x1)-(D(1500.0)/(x0*x1))
            i = i+1
            getcontext().prec = 10
            x0=x1
            x1 = xn
            myList3.append(xn)
        return myList3

a = formula(30,"%.1f")
a1 = formula(30,"%.5f")
a2 = formula(30,"%.10f")
a3 = formula(30,"%.100f")


print a
print a1
print a2
print a3

b = formula1(30,"1")
b1 = formula1(30,"5")
b2 = formula1(30,"10")
b3 = formula1(30,"100")

print b
print b1
print b2
print b3

fig = plt.figure()
plt.plot(a, 'ro', a1, 'go', a2, 'bo',a3, 'y^',linestyle='-')
plt.axis([0,35,-210,200])
plt.xlabel('xn Approximations')
plt.ylabel('xn')
plt.title('')
plt.show()
```

```
fig = plt.figure()
plt.plot(b, 'ro', b1, 'go', b2, 'bo',b3,'y^',linestyle='-')
plt.axis([0,35,-210,200])
plt.xlabel('xn Approximations')
plt.ylabel('xn')
plt.title('')
plt.show()

fig = plt.figure()
plt.plot(a, 'ro', a1, 'go', a2, 'bo',a3, 'y^',b, 'r^', b1, 'g^', b2, 'b^',b3, 'yo'
plt.axis([0,35,-210,200])
plt.xlabel('xn Approximations')
plt.ylabel('xn')
plt.title('')
plt.show()
```



Plot 3: Plot of xn for n iterations using floating points of increasing precision



Plot 4: Plot of xn for n iterations using decimal numbers of increasing precision