Lecture Notes 3: Sampling, Simulation, Low-Level Extensions

Sampling from Distributions

```
In [1]: import numpy,numpy.random
```

Sampling from a uniform distribution between 0 and 1

```
In [2]: print numpy.random.uniform(0,1,[10])
```

```
[ 0.72611138  0.24067817  0.71938139  0.45283454  0.70368496  0.94023131
  0.77130642  0.04720969  0.29922044  0.66994318]
```

Sampling from normal distibution of mean 10 and standard deviation 0.01

```
In [3]: print numpy.random.normal(10,0.01,[10])
```

```
[ 10.00598152  10.00735394  10.00212889   9.9986185   10.00672975
   9.98147864  10.00094005   9.995714    10.00350466  10.00550996]
```

Demo 1: Testing A Statistical Paradox (the James-Stein Paradox)

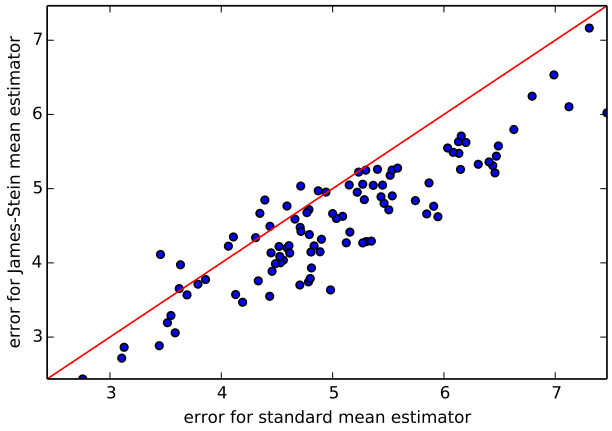
When estimating the mean of a data distribution, moving the estimator from the empirical mean, and closer to some arbitrary point (e.g. the origin) makes the estimator more accurate. Let's verify it:

```
In [6]: # Testing many times:
        errstd  = []
        errstd2 = []

        n = 10
        d = 50

        for i in range(100):

            # sample from a distribution of mean vector 1 and standard deviation 1
            m = 1.0

            X = numpy.random.normal(m,1,[n,d])

            # empirical mean
            m_emp = X.mean(axis=0)

            # some coefficient
            c = (1-(d-2)*1.0/n/((m_emp)**2).sum())

            # james-stein estimator
            m_js = c*m_emp

            # the error between the true mean and the standard estimator
            errstd  += [((m -   m_emp)**2).sum()]

            # the error between the true mean and the said better estimator
            errstd2 += [((m -   m_js)**2).sum()]
```

```
In [7]: import matplotlib
        from matplotlib import pyplot as plt
        %matplotlib inline
        from IPython.display import set_matplotlib_formats
        set_matplotlib_formats('pdf','png')
        plt.rcParams['savefig.dpi'] = 90
```

```
In [9]: plt.scatter(errstd,errstd2)

        l,h = min(errstd+errstd2),max(errstd+errstd2)
        plt.plot([l,h],[l,h],color='red')
        plt.xlabel('error for standard mean estimator')
        plt.ylabel('error for James-Stein mean estimator')
        plt.axis([l,h,l,h])
```

```
Out[9]: [2.4355343709629809,
         7.4646057017558283,
         2.4355343709629809,
         7.4646057017558283]
```



Observation: most points are below the curve (i.e. James-Stein estimator is better).

Making discrete choices

Let us suppose we have 7 fruits to choose from:

Figure 1:

```
In [10]: fruits = ['watermelon','apple','grape','grapefruit','lemon','banana','cherry']
```

We use the function random.choice to randomly choose from that list

```
In [11]: import random
         print random.choice(fruits)
```

```
grapefruit
```

```
In [12]: print random.choice(fruits)
         print random.choice(fruits)
         print random.choice(fruits)
```

```
banana
lemon
apple
```

The function `choice` of the module `numpy.random` provides some more functionalities such as choosing repeatedly

```
In [11]: print numpy.random.choice(fruits,[20])
```

```
['apple' 'lemon' 'apple' 'grape' 'banana' 'grapefruit' 'cherry' 'lemon'
 'watermelon' 'grapefruit' 'banana' 'watermelon' 'grapefruit' 'cherry'
 'cherry' 'banana' 'cherry' 'banana' 'lemon' 'banana']
```

or specify a certain distribution of fruits to choose from

```
In [12]: p = [0.05,0.70,0.05,0.05,0.05,0.05,0.05]
         print numpy.random.choice(fruits,[20],p=p)
```

```
['grapefruit' 'apple' 'apple' 'apple' 'apple' 'apple' 'apple' 'apple'
 'banana' 'apple' 'watermelon' 'cherry' 'apple' 'apple' 'apple' 'apple'
 'watermelon' 'apple' 'cherry' 'apple']
```

Another way to make discrete choices

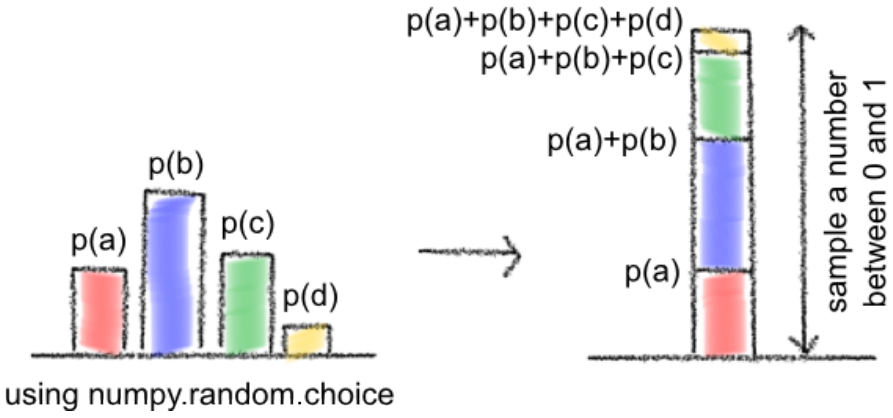


Figure 2:

```
In [13]: # Define fruits probabilities
         p = [0.05,0.70,0.05,0.05,0.05,0.05,0.05]

         # Cumulate them
```

```
        l = numpy.cumsum([0]+p[:-1]) # lower-bounds
        h = numpy.cumsum(p)          # upper-bounds

        # Draw a number between 0 and 1
        u = numpy.random.uniform(0,1)

        # Find which basket it belongs to
        s = (u>l)*(u<h)

        # retrieve the label
        fruits[numpy.argmax(s)]
```

```
Out[13]: 'apple'
```

The code can be parallelized to choose multiple fruits at the same time

```
In [14]: # generate many numbers between 0 and 1
         u = numpy.random.uniform(0,1,[20])

         # find the basket to which they belong
         na = numpy.newaxis
         s = (u[:,na]>l[na,:])*(u[:,na]<h[na,:])

         print([fruits[i] for i in numpy.argmax(s,axis=1)])
```

```
['watermelon', 'apple', 'apple', 'apple', 'apple', 'apple', 'apple', 'grapefruit', 'apple', 'banana', '
```

More importantly, the code can be parallelized to choose from multiple distributions at the same time. Let us create a matrix of probability distributions.

```
In [15]: P = [
             [0.0,0.05,0.05,0.05,0.05,0.05,0.05,0.70],
             [0.0,0.05,0.05,0.05,0.05,0.05,0.70,0.05],
             [0.0,0.05,0.05,0.70,0.05,0.05,0.05,0.05],
         ]
         # (note that we have added a zero-probability state
         #  at the beginning for ease of implementation)
```

We stack them, and compute the bounds `L` and `H`

```
In [16]: C = numpy.cumsum(P,axis=1) # lower-bounds

         L,H = C[:,:-1],C[:,1:]
```

Draw 8 fruits from each distribution, and test the bounds

```
In [19]: R = numpy.random.uniform(0,1,[3,8])

         S = (R[:,:,na] > L[:,na,:])*(R[:,:,na] < H[:,na,:])
```

```
In [20]: for s in S:
             print([fruits[i] for i in numpy.argmax(s,axis=1)])
```

```
['lemon', 'cherry', 'cherry', 'cherry', 'cherry', 'apple', 'cherry', 'cherry']
['banana', 'banana', 'banana', 'banana', 'grapefruit', 'cherry', 'grapefruit', 'banana']
['grape', 'grape', 'grape', 'grape', 'grape', 'grape', 'lemon', 'grape']
```

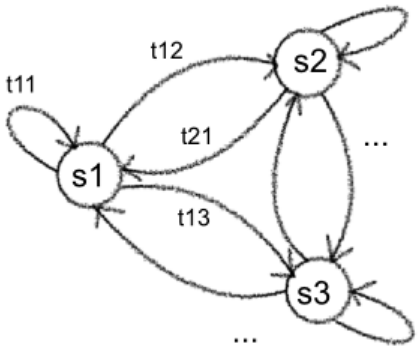


Figure 3:

Demo 2: Monte Carlo Simulation of a Markov Chain

A Markov chain transits between a set of states, where the transition between pairs of states is associated with a fixed probability. The set of probabilities can be stored in a transition matrix.

```
In [22]: # Transition matrix
         T = numpy.array([
                 [0.9,0.1,0.0], # transiting from state 1 to state 1,2,3
                 [0.0,0.9,0.1], # transiting from state 2 to state 1,2,3
                 [1.0,0.0,0.0], # transiting from state 3 to state 1,2,3
             ])
```

Thanks to our parallel implementation, we can choose from multiple probability distributions, and thus, simulate multiple markov chains in parallel, each of them being in a different state.

```
In [24]: # Add empty state to transition matrix
         P = numpy.pad(T,1,mode='constant')[1:-1,:-1]
         print(P)
```

```
[[ 0.   0.9  0.1  0. ]
 [ 0.   0.   0.9  0.1]
 [ 0.   1.   0.   0. ]]
```

```
In [25]: # Implementing transition for each particle
         def mcstep(X):
             Xp = numpy.dot(X,P)

             Xc = numpy.cumsum(Xp,axis=1)

             L,H = Xc[:,:-1],Xc[:,1:]

             R = numpy.random.uniform(0,1,[len(Xp),1])

             return (R > L)*(R < H)*1.0
```

```
In [26]: # Initialize all particles to state 1
         A = numpy.outer(numpy.ones([30]),[1.0,0,0])

         print 'initial distribution of particles states: [%.3f %.3f %.3f]'%tuple(A.mean(axis=0))
```

```
        for i in range(20):
                print '(iter %2d)'%i,numpy.argmax(A,axis=1)
                A = mcstep(A)

        # Print distribution of sampled states
        print 'final distribution of particles states: [%.3f %.3f %.3f]'%tuple(A.mean(axis=0))
```

```
initial distribution of particles states: [1.000 0.000 0.000]
(iter  0) [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
(iter  1) [0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
(iter  2) [0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0]
(iter  3) [0 1 2 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0]
(iter  4) [0 1 0 1 0 1 0 0 0 0 1 0 0 2 1 0 0 0 0 0 1 0 1 0 1 1 0 0 1 1]
(iter  5) [0 1 0 2 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 1 0 0 1 1]
(iter  6) [0 1 0 0 1 2 0 0 1 0 2 0 0 0 1 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1]
(iter  7) [0 1 0 0 1 0 0 0 2 0 0 0 0 0 1 0 1 0 0 1 1 0 2 0 1 1 0 0 1 1]
(iter  8) [0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 1 2 1 0 0 1 2]
(iter  9) [0 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0 1 0 1 0 0 1 0]
(iter 10) [0 1 0 1 2 0 0 0 0 1 0 0 0 0 1 0 1 0 0 2 1 0 0 1 0 1 0 0 1 0]
(iter 11) [0 1 0 1 0 1 0 0 0 1 0 0 0 0 1 0 2 0 0 0 1 0 0 2 0 1 0 0 1 0]
(iter 12) [0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0]
(iter 13) [0 1 0 1 1 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0]
(iter 14) [0 1 0 1 1 2 0 0 0 1 0 0 1 0 1 0 0 0 1 0 2 0 1 0 0 1 0 0 1 0]
(iter 15) [0 1 0 1 1 0 0 0 0 1 0 0 2 0 1 0 0 1 2 0 0 0 1 0 1 1 0 0 1 0]
(iter 16) [0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 1 1 0 0 1 0]
(iter 17) [0 1 0 1 1 2 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 1 2 0 0 2 1]
(iter 18) [0 1 0 2 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1]
(iter 19) [0 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1]
final distribution of particles states: [0.633 0.367 0.000]
```

Low-level extensions

Idea: Write highly optimized functions directly in C/C++, Fortran, or Cuda, and make them accessible to the Python user. Numpy is also based on this principle: making very efficient functions accessible in Python in a user-friendly manner.

Example: F2PY (Fortran to Python)

Main steps:

- Install `gfortran` (fortran compiler) and `f2py` (binding between Fortran and Python)
- Create file (e.g. `convolution.f90`) containing some optimized code for convolutions.
- Compile `convolution.f90` into a Python module using `f2py`
- Import module `convolution` directly from Python.

Applying batch convolutions

We would like to use the following Fortran code that computes a batch of convolutions in the same way as the convolutional layer of a neural network.

We run the command

`f2py -c convolution.f90 -m convolution`

that compiles the code into a module file `convolution.so`, that we can now access in Python.

```
In [27]: import convolution
         convolution,convolution.conv
```

```
Out[27]: (<module 'convolution' from 'convolution.so'>, <fortran object>)
```

```
! -*- f90 -*-
      subroutine conv(x,wx,hx,y,w,ww,hw,nx,ny)
      integer wx,hx,nx,wy,hy,ny,ww,hw
      real x(wx,hx,nx),y(wx-ww+1,hx-hw+1,ny),w(ww,hw,nx,ny)
!f2py intent(out) :: y
      y = 0
      wy = wx-ww+1
      hy = hx-hw+1
      do i=1,nx
      do m=1,hw
      do k=1,ww
      forall(l=1:wy,j=1:ny,n=1:hy) y(l,n,j)=y(l,n,j)+x(l+k-1,n+m-1,i)*w(k,m,i,j)
      end do
      end do
      end do
      end
```

Figure 4: file: convolution.f90

Demo 3: Filtering images to detect interesting features

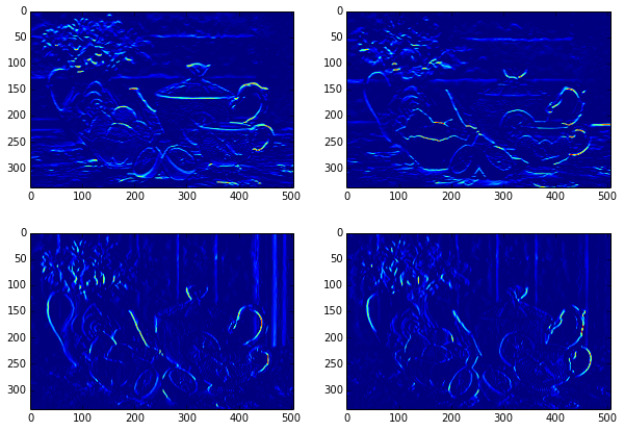
```
In [33]: import numpy,cv2
         import matplotlib
         from matplotlib import pyplot as plt
         %matplotlib inline

In [34]: X = numpy.array(cv2.imread('tea.jpg'))[:,:,::-1]*1
         plt.imshow(X)

Out[34]: <matplotlib.image.AxesImage at 0x7f9d76c85810>
```


```
In [35]: # filter size x
         # filter size y
         # filter size z
         # filter size

         W = numpy.zeros([4,4,3,5])

         # horizontal edge detector
         W[:2,:,:,0] =  1.0
         W[2:,:,:,0] = -1.0
```

```
        # horizontal edge detector in reverse direction
        W[:2,:,:,1] = -1.0
        W[2:,:,:,1] =  1.0

        # vertical edge detector
        W[:,:2,:,2] =  1.0
        W[:,2:,:,2] = -1.0

        # vertical edge detector in reverse direction
        W[:,:2,:,3] = -1.0
        W[:,2:,:,3] =  1.0

        # more red than yellow detector
        W[:,:,0,4] =  1.0
        W[:,:,1,4] = -1.0
```

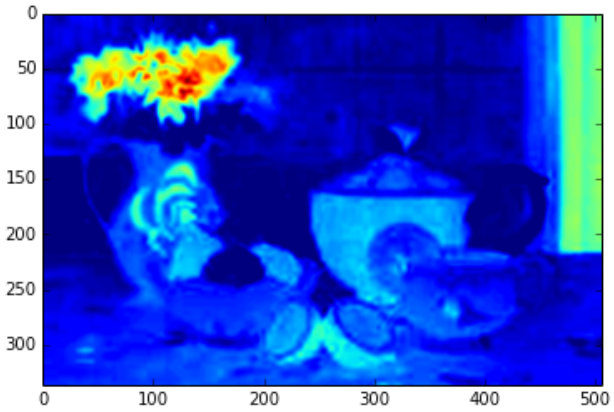
In [36]:
```
Z = convolution.conv(X,W)
Y = numpy.maximum(0,Z)
```

In [37]:
```
f = plt.figure(figsize=(10,7))
for i in range(4):
    p = f.add_subplot(2,2,i+1)
    p.imshow(Y[:,:,i])
```



In [38]:
```
plt.imshow(Y[:,:,4])
```

Out[38]:
```
<matplotlib.image.AxesImage at 0x7f9d8e9ef690>
```


0
50
100
150
200
250
300
0
100
200
300
400
500