lab7sol

October 13, 2014

1 Virtual Lab 7 Solution: Polynomials, Secret Sharing, and Histograms

1.0.1 EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, October 20th, 2014 at 12pm **Name**: EECS 70
Login: cs70-ta
Instructions:

- **Please fill out your name and login above.**
- Complete this lab by filling in all of the required functions, marked with `"YOUR CODE HERE"`.
- If you plan to use Python, make sure to go over **Tutorial 1A: Introduction to Python and IPython** before attempting the lab.
- Make sure you run every code cell one after another, i.e. don't skip any cell. A shortcut for doing this in the notebook is Shift+Enter. When you finish, choose 'Cell > Run All' to test your code one last time all at once.
- Most of the solution requires no more than a few lines each.
- Please do not hardcode the result or change any function without the `"YOUR CODE HERE"` mark.
- Questions? Bring them to our Office Hour and/or ask on Piazza.
- Good luck, and have fun!

1.1 Table of Contents

The number inside parentheses is the number of functions or code blocks you are required to fill out for each question. Always make sure to double check before you submit.

```
In [2]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [3]: import random
```

## Introduction
In this week's Virtual Lab, we will dive deeper into Polynomials and their applications. We will start by generalizing our implementation of degree 2 polynomial's Lagrange Interpolation for any degree d polynomial in a finite field. We will then implement two simple Secret Sharing and Erasure Error clients using Lagrange Interpolation and Polynomial properties studied in lecture.

Below, you will find our sample implementation of a `Polynomial` class. You don't need to understand the details of the implementation, but please read the docstring of each function to understand the general idea. Recall VL3, the Person class, and the Propose-and-Reject algorithm for example.

```
In [4]: # Here's our favorite algorithm again

        def egcd(x, y):
            """
            Extended Euclid's Algorithm. It takes a pair of natural numbers x >= y,
            and returns a triple of integers (d, a, b) such that d = gcd(x, y) = ax + by.
            """

            if y == 0:
                return (x, 1, 0)
            (d, a, b) = egcd(y, x % y)
            return (d, b, a - (x // y) * b)
```

```
In [5]: def inverse(x, m):
            """
            Returns the positive multiplicative inverse of x mod m
            """

            triple = egcd(x, m)
            assert triple[0] == 1, "There doesn't exist an inverse when gcd != 1"
            inverse = triple[1]
            if inverse < 0:
                inverse += m
            return inverse
```

```
In [6]: class Polynomial:
            def __init__(self, coef, n=None):
                """
                Creates the polynomial a0 + a1x + a2x^2 + ... + anx^n
                coef = [a0, a1, a2, ..., an]
                n is the modulus
                """

                self.coef = coef
                self.degree = len(coef)-1
                self.n = n
                self.reduce_coefs()

            def reduce_coefs(self):
                """
                Helper function which keeps the coefficients low
                if we know the modulus
                """

                if not self.n:
                    return
```

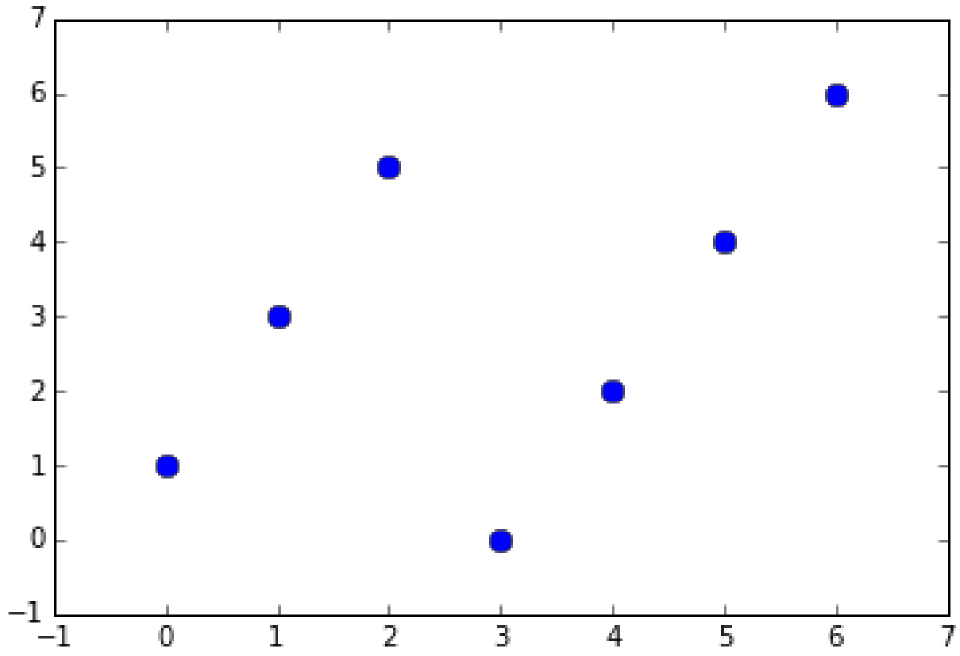
```
        self.coef = [c%self.n for c in self.coef]

    def __call__(self, x):
        """
        Returns the value of the polynomial at point x
        """

        total = 0
        for i in range(self.degree+1):
            total += self.coef[i] * (x**i)
            if self.n:
                total %= self.n
        return total

    def __mul__(self, val):
        """
        Multiplication either for real values or for other polynomials
        """

        if type(val) == int or type(val) == float:
            new_poly = Polynomial([val * c for c in self.coef], self.n)
            return new_poly
        elif val.__class__.__name__ == 'Polynomial':
            assert self.n == val.n, "inconsistent moduli"
            new_degree = self.degree + val.degree
            new_coefs = [0] * (new_degree + 1)
            for i in range(self.degree + 1):
                for j in range(val.degree + 1):
                    new_coefs[i+j] += self.coef[i] * val.coef[j]
            return Polynomial(new_coefs, self.n)
        else:
            raise Exception("invalid multiplication of polynomials")

    def __add__(self, val):
        """
        Addition for polynomials (not defined for reals)
        """

        assert self.n == val.n, "inconsistent moduli"
        new_degree = max(self.degree, val.degree)
        new_coef = [0] * (new_degree+1)
        for i in range(min(self.degree, val.degree)+1):
            new_coef[i] = self.coef[i] + val.coef[i]
        i += 1
        if self.degree > val.degree:
            new_coef[i:] = self.coef[i:]
        else:
            new_coef[i:] = val.coef[i:]
        return Polynomial(new_coef, self.n)

    def __repr__(self):
        """
        String representation of a Polynomial object
        """
        return_str = ""
        first = True

        for i in reversed(xrange(1, len(self.coef))):
            if self.coef[i] < 0 and not first:
                return_str += '-'
            elif self.coef[i] == 0:
                continue
            elif self.coef[i] > 0 and not first:
                return_str += '+'
            first = False
            return_str += str(abs(self.coef[i])) + "(x^" + str(i) + ")"

        if self.coef[0] > 0:
            return_str += '+' + str(self.coef[0])
        elif self.coef[0] < 0:
            return_str += '-' + str(self.coef[0])

        if self.n != None:
            return_str += ' mod ' + str(self.n)
        return return_str

    def __div__(self, val):
        """
        Under a modulus, multiplies by the multiplicative inverse of val
        Otherwise, divides by the real value val
        """

        if not self.n:
            return self.__mul__(1.0 / val)
        val = val % self.n
        val_inverse = inverse(val, self.n)
        return self.__mul__(val_inverse)

    def __eq__(self, other):
        """
        Checks if this polynomial is "equal" to the
        other polynomial.

        We define two polynomials "equal" if they have the same
        coefficients, degrees, and moduli.
        """

        return (isinstance(other, self.__class__)
            and self.coef == other.coef
            and self.degree == other.degree
            and self.n == other.n)

    def __ne__(self, other):
        """
        Checks if the this polynomial is not "equal"
        to the other polynomial
        """
```

```
            return not self.__eq__(other)

        def plot(self, num_points=10, lim=None, reals=False, style='o', size=8):
            """
            Plots the polynomial.

            num_points -- number of points to plot (for the non-mod case)
            lim        -- the range of the points as a list
            reals      -- are we plotting in the reals, default to False
            style      -- what marker to use
            size       -- the size of the marker

            Make sure you play with this function and understand what each parameter does.
            """

            if not self.n:
                if not lim:
                    x = np.linspace(-5, 10, num_points)
                else:
                    x = np.linspace(lim[0], lim[1], num_points)
                y = [self(xval) for xval in x]
                return plt.plot(x, y)
            else:
                if not lim:
                    x = xrange(0, self.n)
                else:
                    if reals:
                        x = np.linspace(lim[0], lim[1], num_points)
                    else:  # plot discrete integer values
                        x = xrange(lim[0], lim[1])
                y = [self(xval) for xval in x]
                if reals:
                    polynomial_plot = plt.plot(x, y)
                else:
                    polynomial_plot = plt.plot(x, y, style, markersize=size)
                plt.xlim([-1, max(x)+1])
                plt.ylim([-1, max(y)+1])
                return polynomial_plot

        __rmul__ = __mul__
        __radd__ = __add__
```

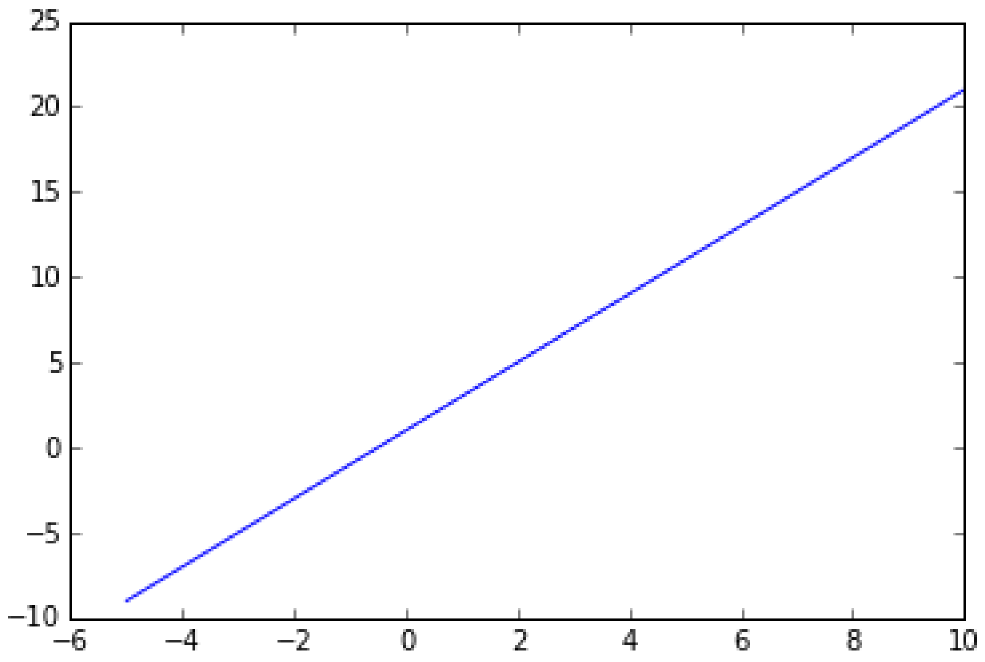
## Part (a): Polynomials Basic
Here are a few examples to help you understand the basics of the `Polynomial` class

```
In [7]: f = Polynomial([1, 2], 7)  # f(x) = 2x + 1 (mod 7)
        f

Out[7]: 2(x^1)+1 mod 7

In [8]: f(5)  # (1 + 2x5) mod 7 = 4

Out[8]: 4
```

```
In [9]: g = Polynomial([1, 2], 7) # g(x) = 2x + 1 (mod 7)
        h = f*g  # h(x) = 4x^2 + 4x + 1
        h

Out[9]: 4(x^2)+4(x^1)+1 mod 7

In [10]: j = f + g  # j(x) = 4x + 2 (mod 7)
         j

Out[10]: 4(x^1)+2 mod 7

In [11]: k = f * 3  # k(x) = 6x + 3 (mod 7)
         k

Out[11]: 6(x^1)+3 mod 7

In [12]: l = f / 3  # l(x) = 3x + 5 (mod 7)
         l

Out[12]: 3(x^1)+5 mod 7

In [13]: f.plot()  # x=0 then y=1, x=1 then y=3, and so on

Out[13]: [<matplotlib.lines.Line2D at 0x7f77f19bbb90>]
```



```
In [14]: # Here's a normal polynomial in the reals

         k = Polynomial([1, 2])
         k.plot()

Out[14]: [<matplotlib.lines.Line2D at 0x7f77f1a16390>]
```

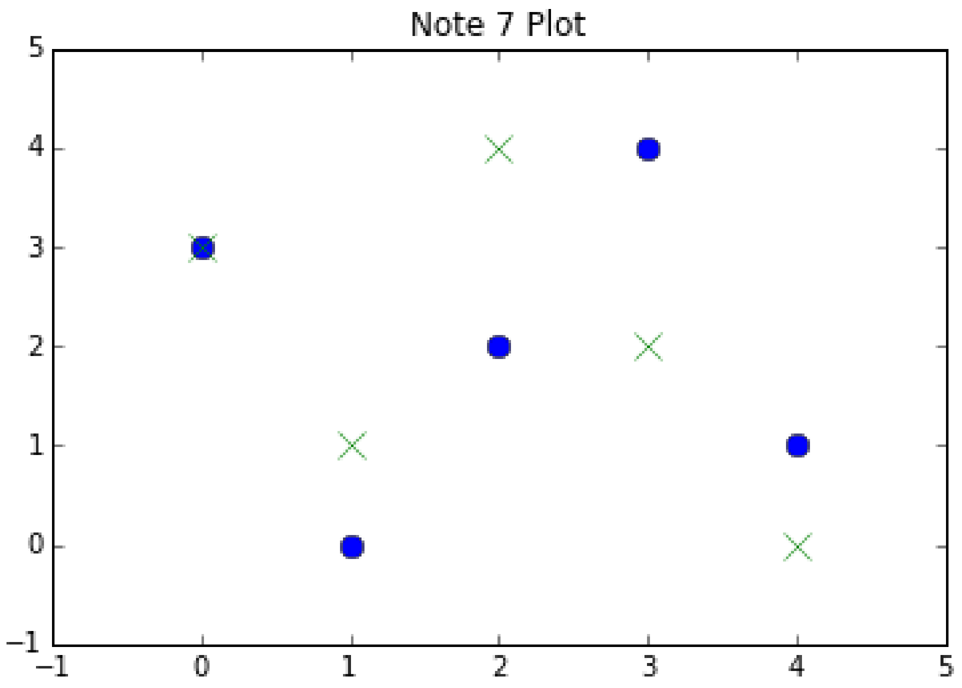


Make sure you understand the Polynomial class and spend time playing with the examples above; otherwise you will be very confused during the next few parts.

## Part (b): Plot Polynomials in Finite Field

Make and plot the polynomials $p(x) = 2x + 3$ and $q(x) = 3x - 2$ with all numbers reduced mod 5. Make sure you use the correct symbol ('o' or 'x') to represent the points so that your result looks similar to the figure on page 5 of Note 7 (there's no need to display the legend on the side).

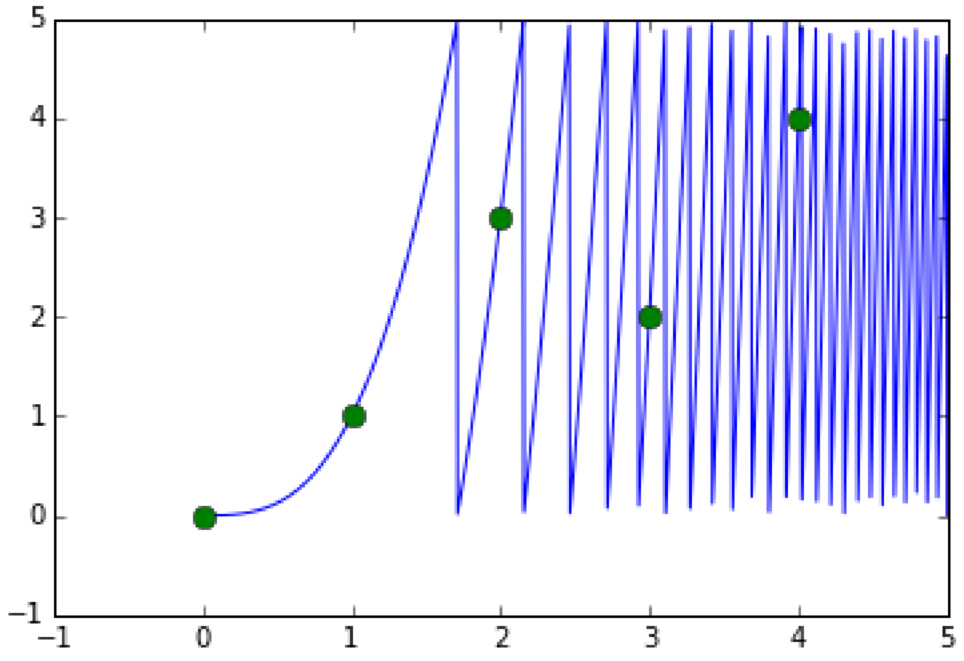
```
In [15]: # YOUR CODE HERE

         p = Polynomial([3, 2], 5)
         q = Polynomial([-2, 3], 5)
         p.plot(style='o')
         q.plot(style='x', size=10)
         plt.title("Note 7 Plot")

Out[15]: <matplotlib.text.Text at 0x7f77f183a310>
```



## Part (c): Plot & Compare Polynomials

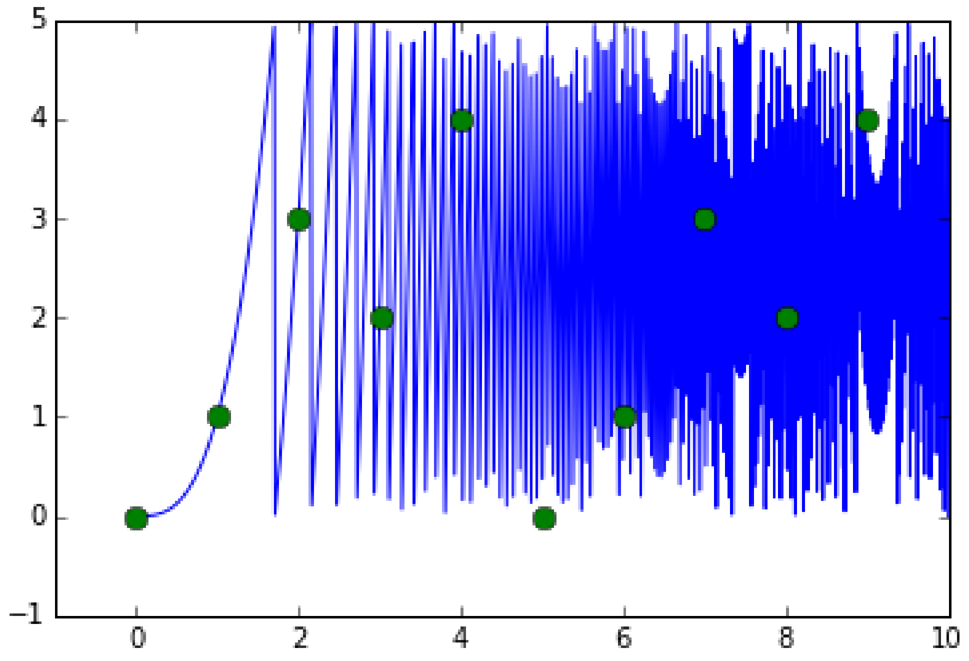
Plot the polynomials x^3 and x^3 mod 5 in the range $[0, 5]$. What do you observe?

```
In [16]: # YOUR CODE HERE
         # You can specify num_points to be 1000 if you're plotting in the reals

         h = Polynomial([0, 0, 0, 1], 5)
         h.plot(num_points=1000, lim=[0, 5], reals=True)
         h.plot(lim=[0, 5])

Out[16]: [<matplotlib.lines.Line2D at 0x7f77f18e3550>]
```



## Part (d): Plot & Compare Polynomials, Part 2
Now repeat (c) for $x \in [0, 10]$ and $x = 0, 1, \ldots, 10$. What do you observe? What happens to each plot as x gets larger? Why do you think this is happening?

```
In [17]: h.plot(num_points=1000, lim=[0, 10], reals=True)
         h.plot(lim=[0, 10])

Out[17]: [<matplotlib.lines.Line2D at 0x7f77f18447d0>]
```



## Part (e): Lagrange Interpolation

Last week, we implemented Lagrange Interpolation for a degree 2 polynomial. This week, we will do so again for any degree d polynomial in a finite field. Implement the function `interpolate`, which takes a list of points and a modulus n, and returns a polynomial that passes through the points in $GF(n)$.

```
In [18]: def interpolate(pts, n):
             """
             Takes a list of points and a modulus n,
             and returns a polynomial that passes through the points in GF(n).

             Fill in the three blanks marked with YOUR CODE HERE,
             using what you know about Lagrange Interpolation.
             """

             final_p = Polynomial([0], n)
             for pt in pts:
                 delta_i = Polynomial([1], n)
                 for pt2 in pts:
                     if pt2 == pt:
                         continue
                     delta_i *= Polynomial([-pt2[0], 1], n)
                     delta_i /= (pt[0] - pt2[0])
                 final_p += pt[1] * delta_i
             return final_p
```

Test your implementation below. Both tests should print True if your implementation is correct.

```
In [19]: g = interpolate([(1, 1), (2, 2), (3, 4)], 5)
         print g
         g == Polynomial([1, 2, 3], 5)

3(x^2)+2(x^1)+1 mod 5

Out[19]: True

In [20]: h = interpolate([(1, 1), (2, 2), (3, 3), (4, 4)], 7)
         print h
         h == Polynomial([0, 1, 0, 0], 7)

1(x^1) mod 7

Out[20]: True
```

Notice how in the example above, we give it four points, but we only get back the line $y = x$ because all the points belong to the same degree 1 polynomial.

(That said, one of the drawbacks of this toy implementation is that we still need to create a "degree 3" polynomial to do the comparison, as seen above, even though what we really want is to compare it with just `Polynomial([0, 1], 7)`. **Can you think of a way to implement this behavior for Extra Credit? Post on Piazza your implementation of a new and improved `Polynomial` class, but only after you have carefully tested your implementation.**

## Part (f): Simple Secret Sharing Scheme

Below is an incomplete implementation of a simple Secret Sharing class. First, implement the method `add_share`, which simply adds a share (a tuple, similar to the Polynomial class) to the list of shares.

After adding enough shares (information), you can then reconstruct the Polynomial with Lagrange Interpolation (part (b)), and evaluate the Polynomial at 0 to find the secret. Implement the method `find_secret`, which basically conveys the idea described above.

```
In [21]: class ShamirSecret:
             """
             A class that manages the client of a Shamir Secret Sharing scheme
             """

             def __init__(self, n, k):
                 """
                 Initializes a Secret Sharing client with zero share,
                 a modulus n, and the least number of people k to recover the secret.
                 """

                 self.shares = []
                 self.n = n
                 self.k = k

             def add_share(self, x):
                 """
                 Adds a share (a tuple) to the list of shares.

                 YOUR CODE HERE
                 """

                 self.shares.append(x)

             def find_secret(self):
                 """
                 Reconstructs the original polynomial with Lagrange Interpolation,
                 then returns the secret.

                 You can assume the secret is evaluated at x = 0.

                 YOUR CODE HERE
                 """

                 assert len(self.shares) >= self.k, "Not enough shares to recover secret"
                 return interpolate(self.shares, self.n)(0)
```

Test your implementation by carrying out the example on page 7/8 of Note 7. Use the same shares in the example (assuming officials 3, 4, and 5 get together to recover the secret).

The secret should be consistent with the example in the Note, and the test should print True.

```
In [22]: # You should first instantiate a new ‘ShamirSecret‘ object,
         # then call ‘add_share‘ three times, and finally ‘find_secret‘.
         # Assign the secret to the variable s for the final comparison.
         # YOUR CODE HERE

         example = ShamirSecret(7, 3)
         example.add_share((3, 1))
         example.add_share((4, 6))
         example.add_share((5, 3))
         s = example.find_secret()
         s == 1

Out[22]: True
```

Now, choose a different set of 3 shares and recover the secret. What do you get? Is it consistent with the above part? Why is that the case?

```
In [23]: # YOUR CODE HERE

         example2 = ShamirSecret(7, 3)
         example2.add_share((1, 2))
         example2.add_share((2, 2))
         example2.add_share((4, 6))
         s = example2.find_secret()
         s == 1

Out[23]: True
```

## Part (g): Erasure Errors
Below is a simple implementation of an Erasure Error client. It has the ability to add packets and drop deterministic or random packet(s).

```
In [24]: class ErasureError:
             """
             A class that manages the client of an Erasure Error problem
             """

             def __init__(self, n, k, m):
                 """
                 Initializes an Erasure Error client with n (empty) packets, k potential
                 lost packets, and modulus m (i.e. working in GF(m)).
                 """

                 self.n = n
                 self.k = k
                 self.m = m
                 self.packets = []

             def add_packet(self, x):
                 """
                 Adds a packet to the list of packets
                 """

                 self.packets.append(x)

             def drop(self, x):
                 """
                 Drops the packet x by removing it from the list of packets
                 Notices that x is a packet, not an index
                 """

                 assert x in self.packets, "The packet to be removed is not in the list of packets!"
                 self.packets.remove(x)

             def random_drop(self):
                 """
                 Drops a packet from the list of packets at random
                 """
```

```
                drop_index = random.randint(0, len(self.packets)-1)
                self.packets.pop(drop_index)
```

Your task is to carry out the example between Alice and Bob on page 2 and 3 of Note 8. There are seven (7) code blocks that you need to fill in below; however, each one should be relatively short and straightforward.

Important: do NOT run any code cell in this problem more than once. If you do, the list of packets might be modified and your result will be inconsistent. If you happen to run a code cell more than once, you have to go to the Step 1 cell and rerun everything again (shortcut: Shift+Enter).

```
In [25]: # Step 1: Instantiate a new ErasureError client. Then, add the four packets
         # that Alice wants to send to Bob from page 2 of Note 8.

         client = ErasureError(4, 2, 7)

         # YOUR CODE HERE
         client.add_packet((1, 3))
         client.add_packet((2, 1))
         client.add_packet((3, 5))
         client.add_packet((4, 0))

         print client.packets
         len(client.packets) == client.n

[(1, 3), (2, 1), (3, 5), (4, 0)]

Out[25]: True

In [26]: # Step 2: Construct a Polynomial based on the packets with Lagrange Interpolation
         # Make sure you use client.m for the modulus. You should NOT use 7 (or
         # any other number in the problem) directly.
         # YOUR CODE HERE

         p = interpolate(client.packets, client.m)
         print p

1(x^3)+4(x^2)+5 mod 7

In [27]: # Step 3: Evaluates the Polynomial at two extra points, as described in the Note.
         # Adds these points to the list of shares
         # YOUR CODE HERE

         client.add_packet((5, p(5)))
         client.add_packet((6, p(6)))

         print client.packets
         len(client.packets) == client.n + client.k

[(1, 3), (2, 1), (3, 5), (4, 0), (5, 6), (6, 1)]

Out[27]: True

In [28]: # Step 4: Drops two packets by calling `self.drop()`
         # We will do random drops later
         # YOUR CODE HERE
         client.drop((2, 1))
         client.drop((6, 1))

         print client.packets
         len(client.packets) == client.n

[(1, 3), (3, 5), (4, 0), (5, 6)]

Out[28]: True

In [29]: # Step 5: Reconstructs the original polynomial using Lagrange Interpolation
         # Make sure you save this polynomial to a different variable than the original one
         # YOUR CODE HERE

         p_reconstruct = interpolate(client.packets, client.m)
         p == p_reconstruct

Out[29]: True
```

Now how about we drop the packets at random? Let's add the two dropped packets back to our list of packets first.

```
In [30]: client.add_packet((2, 1))
         client.add_packet((6, 1))

In [31]: # Step 4b: Drops two random packets
         # YOUR CODE HERE

         client.random_drop()
         client.random_drop()
         print client.packets
         len(client.packets) == client.n

[(3, 5), (4, 0), (5, 6), (6, 1)]

Out[31]: True

In [32]: # Step 5b: Reconstructs the original polynomial using Lagrange Interpolation
         # Make sure you save this polynomial to a different variable than the original one
         # Simply copy your code from Step 5 here...
         # YOUR CODE HERE

         p_random_reconstruct = interpolate(client.packets, client.m)
         p == p_random_reconstruct

Out[32]: True
```

## Part (h): Gaussian Histogram

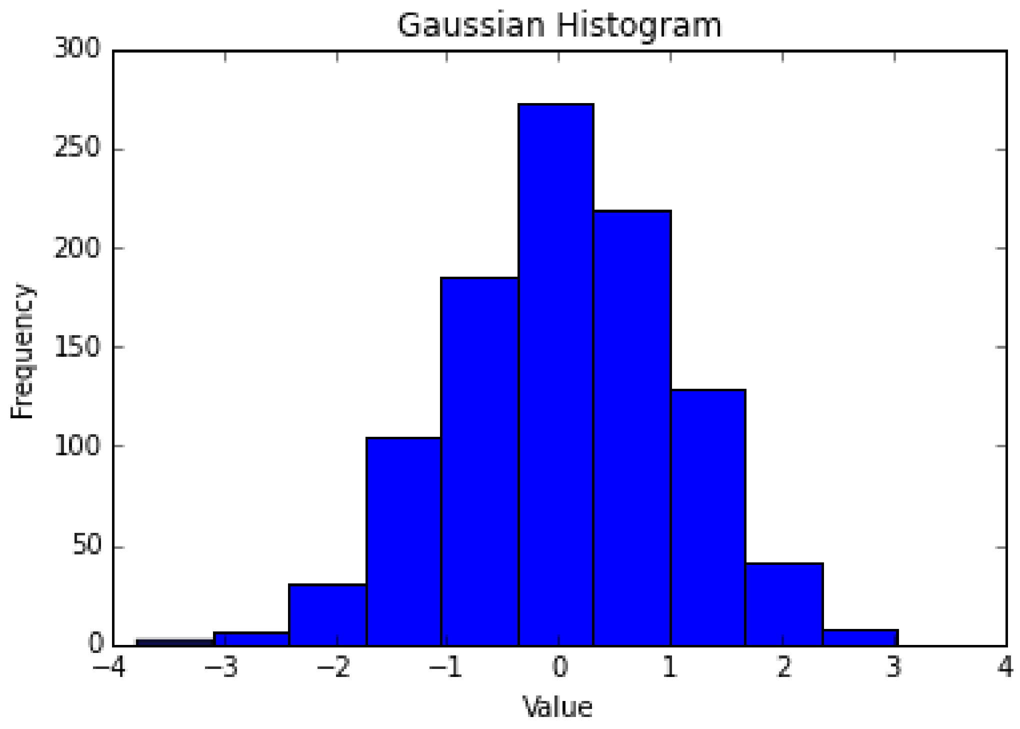
In the last couple of labs, we learned how to plot a simple curve and a bar chart. To wrap up our Matplotlib intro series, we will learn another major plot today: the histogram.

Before we begin, we want to stress that histograms are quite different than bar charts, even though both use bars. Histograms are used to show distributions of variables while bar charts are used to compare variables. Histograms plot quantitative data with ranges of the data grouped into bins or intervals while bar charts plot categorical data.

For more information, check out this Forbes article. http://www.forbes.com/sites/naomirobbins/2012/01/04/a-histogram-is-not-a-bar-chart/

What we will use for our data is 1000 random numbers, drawn from a Gaussian distribution. This is the common "normal" distribution, or the "bell curve" that occurs so frequently in nature. We will use a Gaussian centered about zero, with a standard deviation of 1.0 (this is the default for `np.random.normal`).

```
In [33]: gaussian_numbers = np.random.normal(size=1000)
         plt.hist(gaussian_numbers)
         plt.title("Gaussian Histogram")
         plt.xlabel("Value")
         plt.ylabel("Frequency")

Out[33]: <matplotlib.text.Text at 0x7f77f15ac3d0>
```



Again, that is pretty straightforward! We also learn a few new commands this week.

- `plt.hist` plots a histogram, as you might suspect. There are a lot of options which you can pass to this function, and some of which will be further explored in this question.
- `plt.xlabel` gives the x-axis a meaningful label.
- `plt.ylabel` gives the y-axis a meaningful label.
- `np.random.normal` generates random samples from a normal distribution.

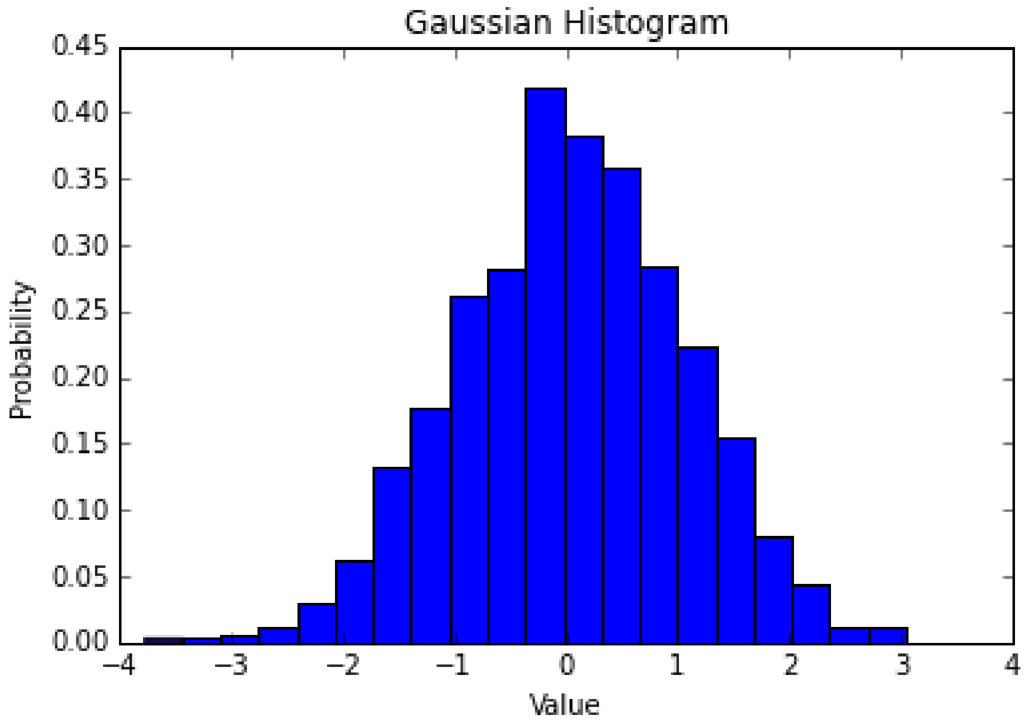
We now want to plot a probability distribution instead of just frequency counts. In other words, we want to scale the values appropriately so that rather than showing how many numbers in each bin, we instead have a probability of finding a number in that bin.

We also want to increase the number of bins. `plt.hist` defaults to 10 bins, and since we have 1000 points, that seems a bit too small. **Change the number of bins to** 20.

You need to look into the documentation of `plt.hist` and find the argument(s) that help you accomplish the tasks above.

```
In [34]: # YOUR CODE HERE

         plt.hist(gaussian_numbers, bins=20, normed=True)
         plt.title("Gaussian Histogram")
         plt.xlabel("Value")
         plt.ylabel("Probability")
```

```
Out[34]: <matplotlib.text.Text at 0x7f77f14f7cd0>
```



```
In [35]: # This might be helpful

         #help(plt.hist)
```

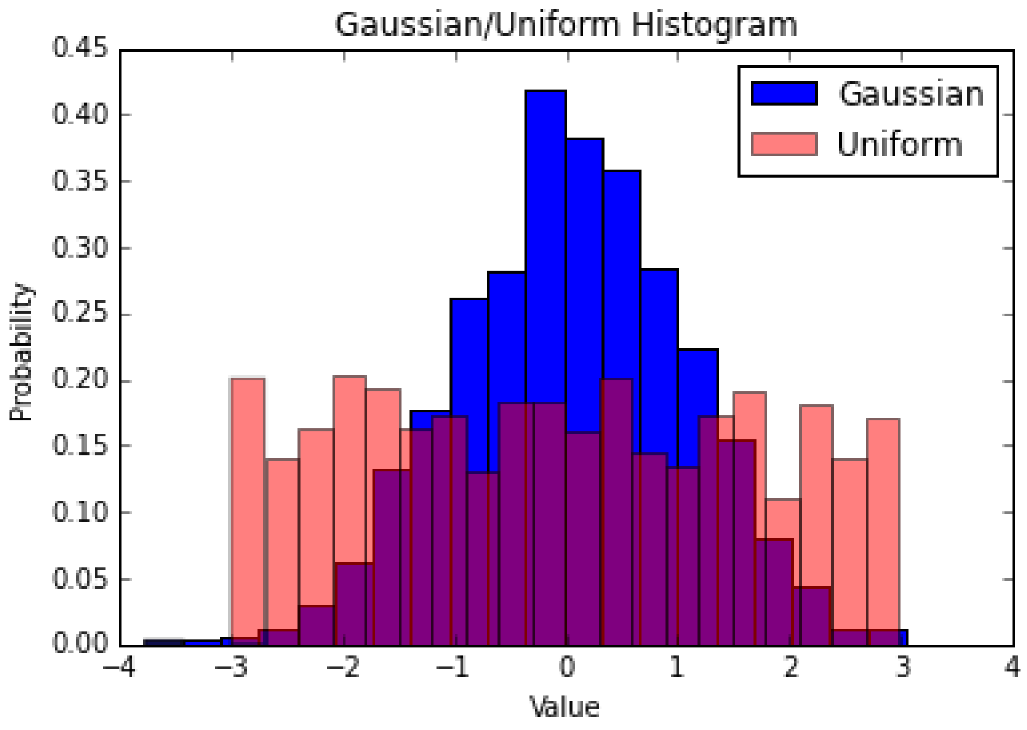
## Part (i): Uniform Histogram

Now, we want to add a uniform distribution to the above plot. If you simply add a second histogram, the plots will obscure each other. We can fix this problem by using Matplotlib's ability to handle alpha transparency.

Add a histogram of 1000 **uniformly distributed random numbers from -3 to 3 in red with 50% transparency over top the blue Gaussian from the first task**.

Also, **give each of your histogram a label**. Then, at the very end, you can call `plt.legend()`, which will display the legends nicely. Check the homework handout to see what your final plot should look like.

```
In [36]: # YOUR CODE HERE

         uniform_numbers = np.random.uniform(low=-3, high=3, size=1000)
         plt.hist(gaussian_numbers, bins=20, normed=True, color='b', label='Gaussian')
         plt.hist(uniform_numbers, bins=20, normed=True, color='r', alpha=0.5, label='Uniform')
         plt.title("Gaussian/Uniform Histogram")
         plt.xlabel("Value")
         plt.ylabel("Probability")
         plt.legend()

Out[36]: <matplotlib.legend.Legend at 0x7f77f140a550>
```



```
In [37]: # Beside the help for plt.hist, this might also be helpful
         # to look for a function that generates data from a uniform distribution.

         #help(np.random)
```

## Part (j): Fair Coin Toss
This last question is meant to warm you up for next week's lab on randomness. Imagine you are tossing a fair coin k times, and you would like to count the numer of heads. Implement the function `count_heads`, which takes in k, the number of tosses, and returns the number of heads in k tosses.

Do 1000 coin tosses. Plot a bar chart of how many heads you got v.s. how many tails. What do you observe? Does it match your expectation?

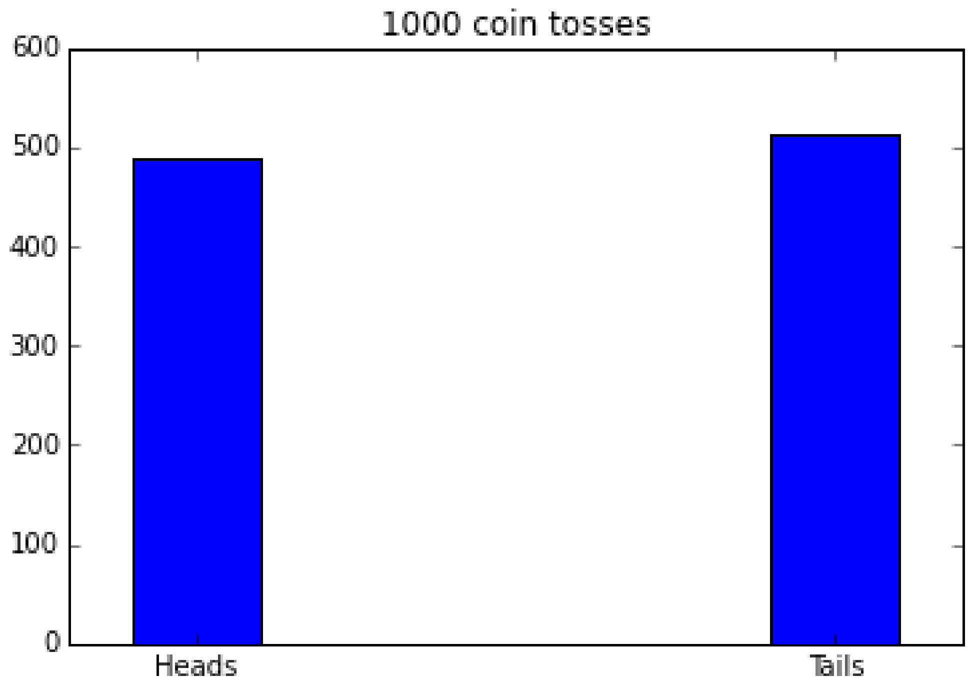
Hint: In Python, `random.randint(a, b)` returns a random integer x such that $a \leq x \leq b$. Can you think of a way to use this function to simulate the fair coin tosses? You don't necessarily have to use `randint`, but you will definitely need a function from the `random` module.

```
In [38]: def count_heads(k):
             """
             Counts the number of heads in k tosses

             YOUR CODE HERE
             """

             return sum([round(random.random()) for _ in range(k)])
```

```
In [39]: # YOUR CODE HERE

         heads = count_heads(1000)
         plt.bar([0, 1], [heads, 1000 - heads], width=0.2, align="center")
```

```
        plt.xticks([0, 1], ["Heads", "Tails"])
        plt.title("1000 coin tosses")
```

```
Out[39]: <matplotlib.text.Text at 0x7f77f11ca5d0>
```



Congratulations! You are done with Virtual Lab 7.

Don't forget to convert this notebook to a pdf document, merge it with your written homework, and submit both the pdf and the code (as a zip file) on glookup.

Reminder: late submissions are NOT accepted. If you have any technical difficulty, resolve it early on or use the provided VM.