STAT221Week06

Save | Save & quit | Discard & quit

last edited on May 02, 2011 08:11 PM by raazesh.sainudiin

Modular Arithmetic, Linear Congruential Generators, and Pseudo-Random Numbers

Monte Carlo Methods

What's this all about?

Question:

How can we produce realisations from $Uniform(0,1)$, the fundamental random variable?

i.e., how can we produce samples $(x_1, x_2, \ldots, x_n)$ from $X_1, X_2, \ldots, X_n \overset{IID}{\sim} Uniform(0,1)$?

What is Sage doing when we ask for `random()`?

```
random()
```

Answer:

Modular arithmetic and number theory gives us **pseudo-random number generators**.

Question:

What can we do with samples from a $Uniform(0,1)$ RV? Why bother?

Answer:

We can use them to sample or simulate from other, more complex, random variables. These simulations can be used to understand real-world phenomenon such as:

jsMath

- modelling traffic queues on land, air or sea for supply chain management
- estimate missing data in Statistics New Zealand accommodation occupancy survey to better manage NZ tourism revenues
- helping Christchurch Hospital to manage critical care for pre-term babies
- helping DOC to minimise the extinction probability of various marine organisms
- help the Government find if certain fishing boats are illegally under-reporting their catches
- find cheaper air tickets for a vacation
- various physical systems can be modeled. See http://en.wikipedia.org/wiki/Monte_Carlo_method for a bigger picture.

The starting point for all of this is modular arithmetic ...

Modular arithmetic

Modular arithmetic (arithmetic modulo m) is a central theme in number theory and is crucial for generating random numbers from a computer (in fancy-lingo, "machine-implementing objects in probability theory"). Being able to do this is essential for computational statistical experiments and Monte Carlo methods. Such computer-generated random numbers are technically called *pseudo-random numbers*.

In this worksheet we are going to learn to *add and multiply modulo* m (this part of our worksheet is adapted from William Stein's SAGE worksheet on Modular Arithmetic for the purposes of linear congruential generators). If you want a more thorough treatment see http://en.wikipedia.org/wiki/Modular_arithmetic.

Remember when we talked about the modulus operator %? The modulus operator gives the remainder after division:

```
14%12 # "14 modulo 12" or just "14 mod 12"
```

```
range(0,24,1)  # 0,1,2...,23 hours in the 24-hours clock
```

```
[x%12 for x in range(0,24,1)]  # x%12 for the 24 hours in the analog clock
```

```
set([x%12 for x in range(0,24,1)]) # unique hours 0,1,2,...,11 in analog
```

Arithmetic modulo m is like usual arithmetic, except every time you add or multiply, you also divide by m and return the remainder. For example, working modulo $m = 12$, we have:

$$8 + 6 = 14 = 2$$

since 2 is the remainder of the division of 14 by 12.

Think of this as like the hours on a regular analog clock. We already do modular addition on a regular basis when we think about time, for example when answering the question:

"If it is 8pm now then what time will it be after I have spent the 6 hours that I am supposed to dedicate this week toward this course? Answer: 2am."

Modular addition and multiplication with numbers modulo m is well defined, and has the following properties (we will just assume them here -- you'd cover them properly in a basic algebra course):

- $a + b = b + a$ (addition is commutative)
- $a \cdot b = b \cdot a$ (multiplication is commutative)
- $a \cdot (b + c) = a \cdot b + a \cdot c$ (multiplication is distributive over addition)
- If a is coprime to m (i.e., not divisible by any of the same primes), then there is a unique b (mod m) such that $a \cdot b = 1$.

jsMath

Let us make a matrix of results from addition and multiplication modulo 4 .

```
matrix(2,3,[1, 2, 3, 4, 5, 6]) # this is how you make a 2X3 matrix in Sage
```

evaluate

```
    [1 2 3]
    [4 5 6]
```

```
m=4;
# list (i,j, (i+j) mod m) as (i,j) range in [0,1,2,3]
[(i,j,(i+j)%m) for i in range(m) for j in range(m)]
```

```
    [(0, 0, 0), (0, 1, 1), (0, 2, 2), (0, 3, 3), (1, 0, 1), (1, 1, 2), (1,
    2, 3), (1, 3, 0), (2, 0, 2), (2, 1, 3), (2, 2, 0), (2, 3, 1), (3, 0, 3),
    (3, 1, 0), (3, 2, 1), (3, 3, 2)]
```

```
[(i+j)%m for i in range(m) for j in range(m)]
```

```
    [0, 1, 2, 3, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2]
```

```
#addition mod m
matrix(m,m,[(i+j)%m for i in range(m) for j in range(m)])
```

```
    [0 1 2 3]
    [1 2 3 0]
    [2 3 0 1]
    [3 0 1 2]
```

```
# multiplication mod m
matrix(m,m,[(i*j)%m for i in range(m) for j in range(m)])
```

```
    [0 0 0 0]
    [0 1 2 3]
    [0 2 0 2]
    [0 3 2 1]
```

In the following interactive image (created by William Stein) we make an addition and multiplication table modulo m, where you can control m with the slider. Try changing m and seeing what effect it has:

```
import matplotlib.cm
cmaps = ['gray'] + list(sorted(matplotlib.cm.datad.keys()))

@interact
def mult_table(m=(5,(1..200)), cmap=("Color Map",cmaps)):
    Add = matrix(m,m,[(i+j)%m for i in range(m) for j in range(m)])
    Mult = matrix(m,m,[(i*j)%m for i in range(m) for j in range(m)])
    print "Addition and multiplication table modulo %s"%m
    show(graphics_array( (plot(Add, cmap=cmap),plot(Mult,
```

Look at this for a while. Answer the following questions (which refer to the default colour map) to be sure you understand the table:

1. **Question**: Why is the top row and leftmost column of the multiplication table always black?
2. **Question**: Why is there an anti-diagonal block line in the addition table (the left-most table)?
3. **Question:** Why are the two halves of each table (the two pictures) symmetric about the diagonal?

jsMath

You can change the colour map if you want to.

You try

You should be able to understand a bit of what is happening here. See that there are two list comprehensions in there. Take the line "`matrix(m,m,[(i+j)%m for i in range(m) for j in range(m)])`". The list comprehension part is "`[(i+j)%m for i in range(m) for j in range(m)]`". Let's pull it out and have a look at it. We have set the modulo `m` to 6 but you could change it if you want to:

```
m = 6
listFormAdd = [(i+j)%m for i in range(m) for j in range(m)]
listFormAdd
```

This list comprehension doing double duty: remember that a list comprehension is like a short form for a for loop to create a list? This one is like a short form for one for-loop nested within another for-loop. We could re-create what is going on here by making the same list with two for-loops. This one uses modulo `m = 6` again.

```
m = 6
listFormAddTheHardWay = []
for i in range(m):
    for j in range(m):
        listFormAddTheHardWay.append((i+j)%m)
listFormAddTheHardWay
```

Notice that the last statement in the list comprehension, "`for j in range(m)`", is the *inner* loop in the nested for-loop.

The next step that Stein's interactive image is to make a matrix out of the list.   We won't be doing matrices in detail in this course (we decided to concentrate on arrays instead), but if you are interested, this statement uses the **`matrix`** function to make a matrix out of the list `listFormAdd`. The dimensions of the matrix are given by the first two arguments to the matrix function, `m, m`.

```
matrixForm = matrix(m,m, listFormAdd)
matrixForm
```

Optionally, you can find out more about matrices from the documentation.

```
help(matrix) # start with this help link to figure out how the interact
above works as an OPTIONAL exercise
```

    docs-0.html

Try recreating the matrix for multiplication, just as we have just recreated the one for addition.

(end of You Try)

jsMath

Modular arithmetic in Sage

The simplest way to create a number modulo m in Sage is to use the `Mod(a,m)` command. We illustrate this below.

```
Mod(8, 12)
```

Let's assign it to a variable so that we can explore it further:

```
myModInt = Mod(8, 12)
myModInt
```

```
type(myModInt)
```

```
parent(myModInt)
```

We will compare `myModInt` to a 'normal' Sage integer:

```
myInt = 8
myInt
```

```
type(myInt)
```

```
parent(myInt)
```

We can see that `myModInt` and `myInt` are different types, but what does this mean? How do they behave?

Try addition:

```
myModInt + 6
```

```
myInt + 6
```

Was this what you already expected?

What about multiplication?

```
myModInt * 6
```

```
myInt * 6
```

What's going on here? As we said above, arithmetic modulo m is like usual arithmetic, except every time you add or multiply, you also divide by m and return the remainder. 8 x 6 is 48, and the

jsMath

remainder of 48 divided by 12 (12 is our modulo) is 0.

What about this one? What's happening here?

```
Mod(-37,2010) # Raaz was born in
```

```
Mod(-1,12)  # what's an hour before mid-night
```

You can create the number giving your year of birth (± 1 year) in a similar way. For example, if you are 19 years old now then find the number -19 modulo 2010.

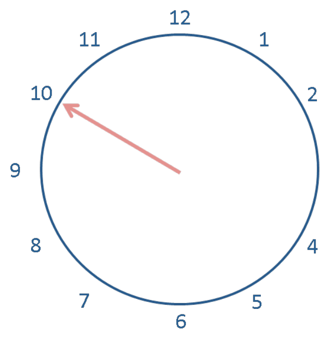
```

```

```

```

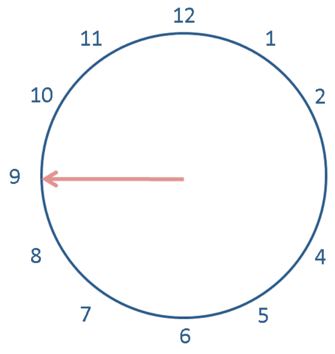
You try

Let us assign 10 modulo 12 to a variable `now`

```
now = Mod(10, 12)
now
```



And add -1 to `now`

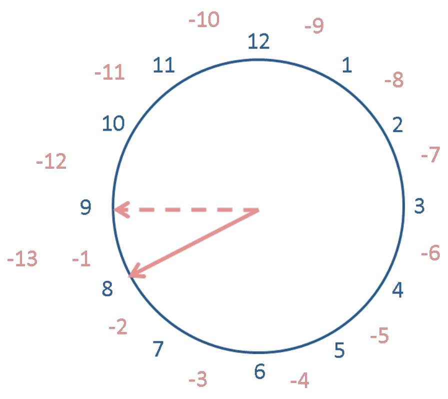


Put in the expression to do this in Sage into the cell below

```

```

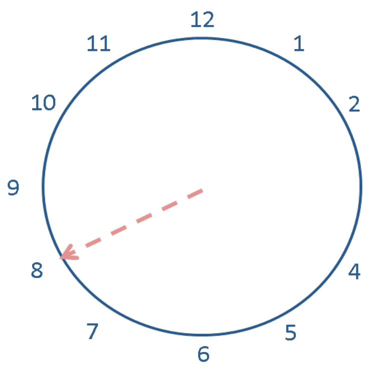
jsMath

And subtract 13 from `the previous expression.`



Sage expression is

Also try adding 14 to `the previous expression` -- the new postition is not shown on this clock but you should be able to see what it should be.



And multiplying 2 by `now` (or `now` by 2)

Try making and using some modular integers of your own. Make sure that you can predict the results of simple addition and multiplication operations on them: this will confirm that you understand what's happening.

What happens if you try arithmetic with two modular integers? Does it matter if the moduli of both operands `a` and b in say a+b are the same if a and b are both modular integers? Does this make sense to you?

(**end of You Try**)

Linear Congruential Generators

A **linear congruential generator (LCG)** is a simple pseudo-random number generator - a simple way of imitating the $Uniform(0, 1)$. "Pseudo-random" means that the numbers are not really random. We will look at what we mean by that as we find out about linear congruential generators.

The theory behind LCGs is easy to understand, and they are easily implemented and fast.

To make a LCG we need:

1. a modulus m $(m > 0)$
2. an integer multiplier a $(0 \leq a < m)$
3. an integer increment c $(0 \leq c < m)$
4. an integer seed x_0 $(0 \leq x_0 < m)$
5. an integer sequence length n

Using these ***inputs***, the LCG generates numbers $x_1, x_2, \ldots x_{n-1}$ where x_{i+1} is calculated from x_i as defined by the following ***recurrence relation***:

$$x_{i+1} \leftarrow mod\left(ax_i + c, m\right)$$

$x_0, x_1, \ldots, x_{n-1}$ is the sequence of pseudo-random numbers called the linear congruential sequence.

We can define a function parameterised by (m, a, c, x_0, n) to give us a linear congruential sequence in the form of a list.

(Remember about function ***parameters***? The function parameters here are `m, a, c, x0`, and `n`. Note that the ***return value*** of the function is a list.

jsMath

```
%auto
def linConGen(m, a, c, x0, n):
    '''A linear congruential sequence generator.

    Param m is the integer modulus to use in the generator.
    Param a is the integer multiplier.
    Param c is the integer increment.
    Param x0 is the integer seed.
    Param n is the integer number of desired pseudo-random numbers.

    Returns a list of n pseudo-random integer modulo m numbers.'''

    x = x0 # the seed
    retValue = [Mod(x,m)]  # start the list with x=x0
    for i in range(2, n+1, 1):
        x = mod(a * x + c, m) # the generator, using modular arithmetic
        retValue.append(x) # append the new x to the list
    return retValue
```

You **don't** need to be able write this function, but you **should** be able to fully understand what it is doing. The function is merely implementing the pseudocode or algorithm of the linear congruential generator using a for-loop and **modular arithmetic**: note that the generator produces integers modulo m.

Linear Congruential Generators: the Good, the Bad, and the Ugly

Are all linear congruential generators as good as each other? What makes a good LCG?

One desirable property of a LCG is to have the longest possible **period**. The period is the length of sequence we can get before we get a repeat. The longest possible period a LCG can have is m. Lets look at an example.

```
help(mod)
```

docs-0.html

```
# first assign values to some variables to pass as arguments to the function
m  = 17  # set modulus to 17
a  = 2   # set multiplier to 2
c  = 7   # set increment to 7
x0 = 1   # set seed to 1
n  = 18  # set length of sequence to 18 = 1 + maximal period
L1 = linConGen(m, a, c, x0, n) # use our linConGren function to make the
sequence
L1                    # this sequence repeats itself with period 8
```

You should be able to see the repeating pattern 9, 8, 6, 2, 11, 12, 14. If you can't you can see that the sequence actually contains 8 unique numbers by making a set out of it:

```
Set(L1)                          # make a Sage Set out of the sequence in the
list L1
```

Changing the seed x_0 will, at best, make the sequence repeat itself over other numbers but with the same period:

jsMath

```
x0 = 3             # leave m, a, c as it is but just change the seed from 1
to 3 here
L2 = linConGen(m, a, c, x0, n)
L2                 # changing the seed makes this sequence repeat itself
over other numbers but also with period 8
```

```
Set(L2)                       # make a Sage Set out of the sequence in the
list L2
```

At worst, a different seed might make the sequence get stuck immediately:

```
x0 = 10            # leave m, a, c as it is but just change the seed to 10
L3 = linConGen(m, a, c, x0, n)
L3
```

```
Set(L3)                       # make a Sage Set out of the sequence in the
list L3
```

```
Set(L3)+Set(L2)+Set(L1) # + on Sage Sets gives the union of the three Sets
and it is the set of integers modulo 17
```

What about changing the multiplier a?

```
m  = 17 # same as before
a  = 3  # change the multiplier from 2 to 3 here
c  = 7  # same as before
x0 = 1  # set seed at 1
n  = 18 # set length of sequence to 18 = 1 + maximal period
L4 = linConGen(m, a, c, x0, n)
L4                            # now we get a longer period of 16 but with
the number 5 missing
```

```
Set(L4)
```

```
x0 = 5  # just change the seed to 5
linConGen(m, a, c, x0, n)
```

We want an LCG with a **full period** of m so that we can use it with any seed and not get stuck at fixed points or short periodic sequences. This is a minimal requirement for simulation purposes that we will employ such sequences for. Is there anyway to know what makes a LCG with a full period? It turns out that an LCG will have a full period if and only if:

1. c and m are **relatively prime** or coprime. i.e. the **greatest common devisor** (gcd) of c and m is 1; and
2. $a-1$ is divisible by all prime factors of m; and
3. $a-1$ is a multiple of 4 if m is a multiple of 4.

(Different conditions apply when $c=0$. The Proposition and Proof for this are in Knuth, *The Art of Computer Programming*, vol. 2, section 3.3).

jsMath

Sage has a function `gcd` which can calculate the **greatest common divisor** of two numbers:

```
gcd(7,17)
```

Sage can also help us to calculate **prime factors** with the `prime_factors` function:

```
prime_factors(m)
```

```

```

$(m, a, c, x0, n) = (256, 137, 123, 13, 256)$ gives a linear congruential sequence with the longest possible period of 256. Let us see how these parameters satisfy the above three requirements while those earlier with $m = 17$, $a = 2$, and $c = 7$ do not.

```
(m,a,c,x0,n)=(256, 137, 123, 13, 256) # faster way to assign a bunch of
parameters
```

```
gcd(c,m)         # checking if the greatest common divisor of c and m is
indeed 1
```

```
prime_factors(m)   # it is easy to get a list of all the prime factors of m
```

```
(a-1) % 2            # checking if a-1=136 is divisible by 2, the only
prime factors of m
```

```
[ (a-1)%x for x in prime_factors(m) ]   # a list comprehension check for an
m with more prime factors
```

```
m % 4          # m is a multiple of 4 check
```

```
(a-1) % 4       # if m is a multiple of 4 then a-1 is also a multiple of 4
```

```
(m, a, c, x0, n)  # therefore these parameter values satisfy all conditions
to have maximum period length m
```

Thus, the parameters $(m,\ a,\ c,\ x_0,\ n) = (256, 137, 123, 13, 256)$ do indeed satisfy the three conditions to guarantee the longest possible period of 256. In contrast, for the LCG example earlier with $m = 17$, $a = 2$, and $c = 7$, although m and c are relatively prime, i.e., $gcd(m, c) = 1$, we have the violation that $a - 1 = 2 - 1 = 1$ is not divisible by the only prime factor 17 of $m = 17$. Thus, we cannot get a period of maximal length 17 in that example.

```
[ (2-1)%x for x in prime_factors(17) ]
```

jsMath

Let us see if the parameters $(m, a, c, x_0, n) = (256, 137, 123, 13, 256)$ that satisfy the three conditions to guarantee the longest possible period of 256 do indeed produce such a sequence:

```
(m,a,c,x0,n)=(256, 137, 123, 13, 256) # faster way to assign a bunch of
parameters
ourLcg = linConGen(m,a,c,x0,n)
ourLcg
```

```
S = Set(ourLcg)  # sort it in a set to see if it indeed has maximal period
of 256
S
```

`ourLcg` is an example of a **good** LCG.

The next example will demonstrate what can go wrong.

Consider $m = 256$, $a = 136$, $c = 3$, $x_0 = 0$, $n = 15$.

```
m,a,c,x0,n = 256, 136, 3, 0, 15
gcd(m,c)
```

```
prime_factors(m)
```

But, since $a - 1$ =135 is not divisible by 2, the only prime factor of m =256, we get into the fixed point 91 no matter where we start from. Try changing the **seed** x_0. Does it make a difference?

```
linConGen(m, a, c, x0, n)
```

We can look at the linear congruential sequence generated by $(m, a, c, x_0, n) = (256, 137, 0, 123, 256)$ from Knuth's classic [The Art of Computer Programming, vol. 2, Sec. 3.3.4, Table 1, Line 5] and compare it with `ourLcg`.

```
m, a, c, x0, n = 256, 137, 0, 123, 256
lcgKnuth334T1L5 = linConGen(m, a, c, x0, n)
lcgKnuth334T1L5
```

```
Set(lcgKnuth334T1L5)
```

Note that although `ourLcg` has maximal period of 256, `lcgKnuth334T1L5` has period of 32. Let us look at them as points.

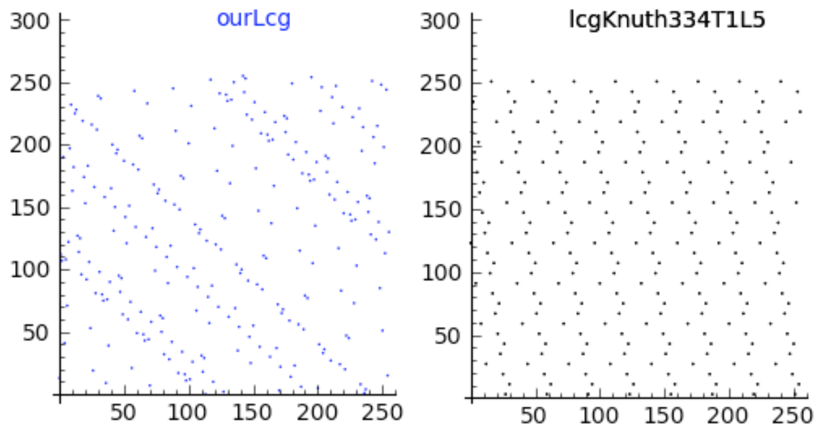
We can plot each number in the sequence, say `ourLcg`, against its index in the sequence -- i.e. plot the first num jsMath

13 against 0 as the tuple (0, 13), the second number against 1 as the tuple (1, 112), etc.   To do this, we need to make a list of the index values, which is simple using `range(256)` which as you know will give us a list of numbers from 0 to 255 going up in steps of 1.   Then we can zip this list with the sequence itself to make the list of tuples .

```
ourPointsToPlot = zip(range(256), ourLcg)
knuPointsToPlot = zip(range(256), lcgKnuth334T1L5)
```

Then we plot the points for `ourLcg` and for `lcgKnuth334T1L5`

```
p1 = points(ourPointsToPlot, pointsize='1')
t1 = text('ourLcg', (150,300), rgbcolor='blue',fontsize=10)
p2 = points(knuPointsToPlot, pointsize='1',color='black')
t2 = text(' lcgKnuth334T1L5', (150,300), rgbcolor='black',fontsize=10)
show(graphics_array((p1+t1,p2+t2)),figsize=[6,3])
```



We can see that in section 3.3.4, Table 1, line 5 in *The Art of Computer Programming,* Knuth is giving an example of a particularly **bad** LCG.

When we introducted LCGs, we said that using an LCG was a simple way to imiate the $Uniform(0, 1)$, but clearly so far we have been generating sequences of integers. How does that help?

To get a simple pseudo Uniform(0,1) generator, we scale the linear congruential sequence over [0, 1]. We can do this by dividing each element by the largest number in the sequence (256 in the case of `ourLcg`).

Important note: The numbers in the list returned by our linConGen function are integers modulo m.

```
type(ourLcg[0])
```

You will need to convert these to Sage integers or Sage multi-precision floating point numbers using `int()` or `RR()` command before dividing by m. Otherwise you will be doing modular arithmetic when you really want the usual division. This may come up in assignments so make sure you have noted it!

```
type(RR(ourLcg[0]))
```

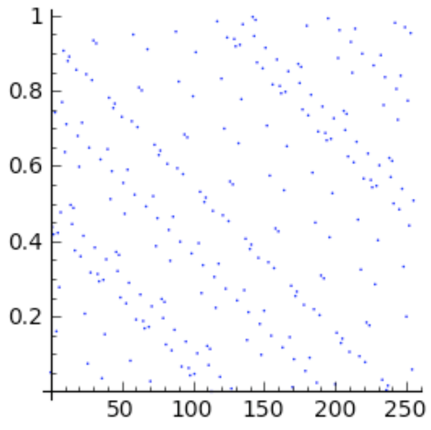
Having sorted that out, we can use a list comprehension as a nice neat way to do our scaling:

jsMath

```
ourLcgScaled = [RR(x)/256 for x in ourLcg]    # convert to mpfr real
numbers before division
#ourLcgScaled = [int(x)/256 for x in ourLcg]  # or convert x to usual
integer before division
#ourLcgScaled = [x/256 for x in ourLcg]       # a very bad idea
ourLcgScaled
```

This is more like it! We could have a look at this on a plot. Again again want tuples (index, element in scaled sequence at index position), which we can get using `range(256)` (to get the indexes 0, .., 255) and `zip`:

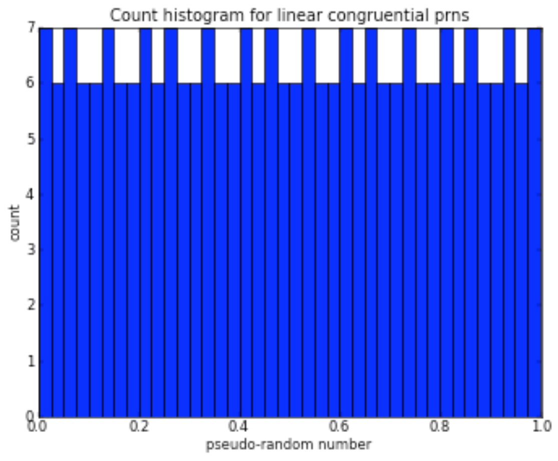
```
ourPointsToPlotScaled = zip(range(256), ourLcgScaled)
p = points(ourPointsToPlotScaled, pointsize='1')
show(p, figsize = (3,3))
```



Now we have points on the real line. We could use a histogram to look at their distribution. If we are hoping that our LCG, once we have scaled the results, is imitating the $Uniform(0,1)$, what kind of shape would we want our histogram to be?

```
import pylab
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(ourLcgScaled, 40) # make the histogram (don't
have to have n, bins, patches = ...)
pylab.xlabel('pseudo-random number') # use pyplot methods to set labels,
titles etc similar to as in matlab
pylab.ylabel('count')
pylab.title('Count histogram for linear congruential prns')
pylab.savefig('myHist', dpi=(50)) # save figure (dpi) to display the figure
pylab.show() # and finally show it
```

jsMath

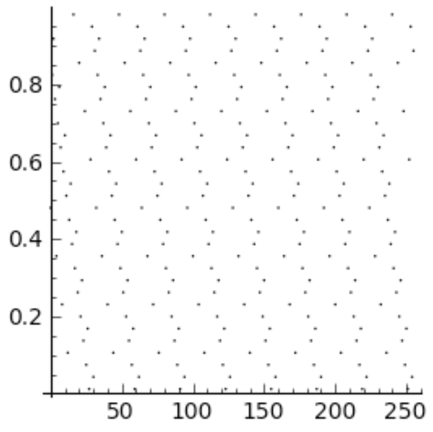


Is this roughly what you expected?

(Please note: we are using the histogram plots to help you to see the data, but you are not expected to be able to make one yourself.)

We could repeat this for the Knuth bad LCG example:

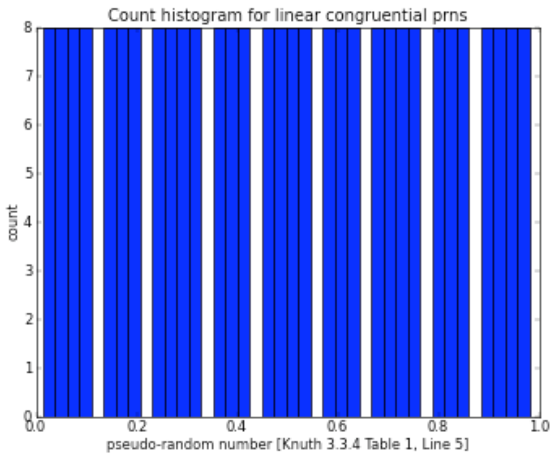
```
knuLcgScaled = [RR(x)/256 for x in lcgKnuth334T1L5]
knuPointsToPlotScaled = zip(range(256), knuLcgScaled)
p = points(knuPointsToPlotScaled, pointsize='1', color='black')
show(p, figsize = (3,3))
```



And show it as a histogram. Given the pattern above, what would you expect the histogram to look like?

```
import pylab
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(knuLcgScaled, 40) # make the histogram (don't have to have n, bins, patches = ...)
pylab.xlabel('pseudo-random number [Knuth 3.3.4 Table 1, Line 5]') # use pyplot methods to set labels, titles etc similar to as in matlab
pylab.ylabel('count')
pylab.title('Count histogram for linear congruential prns')
pylab.savefig('myHist', dpi=(50)) # save figure (dpi) to display the figure
pylab.show() # and finally show it
```

jsMath



Larger LCGs

The above generators are cute but not useful for simulating Lotto draws with 40 outcomes. Minimally, we need to increase the period length with a larger modulus m.

But remember that the quality of the pseudo-random numbers obtained from a LCG is extremely sensitive to the choice of m, a, and c.

To illustrate that having a large m alone is not enough we next look at RANDU, an infamous LCG which generates sequences with strong correlations between 3 consecutive points, which can been seen if we manipulate the sequence to make 3-dimensional tuples out of groups of 3 consecutive points.

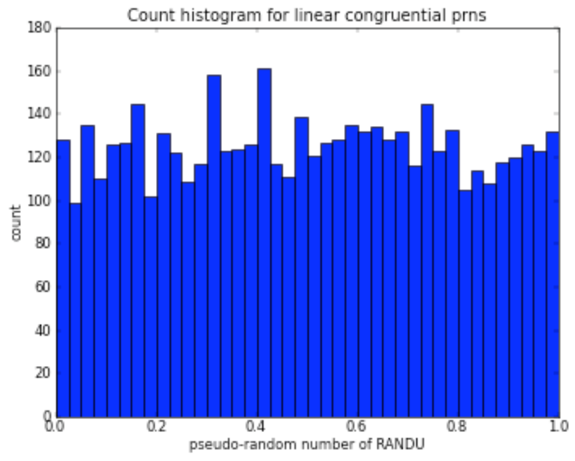
In the cell below we make our scaled sequence in one step, using a list comprehension which contains the expression to generate the LCG and the scaling.

```
m, a, c, x0, n = 2147483648, 65539, 0, 1, 5010
RANDU = [RR(x)/m for x in linConGen(m, a, c, x0, n)]
```

Have a look at the results as a histogram:

```
import pylab
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(RANDU, 40) # make the histogram (don't have
to have n, bins, patches = ...)
pylab.xlabel('pseudo-random number of RANDU') # use pyplot methods to set
labels, titles etc similar to as in matlab
pylab.ylabel('count')
pylab.title('Count histogram for linear congruential prns')
pylab.savefig('myHist', dpi=(50)) # save figure (dpi) to display the figure
pylab.show() # and finally show it
```

jsMath



Now we are going to use some of the array techniques we have learned about to resize the sequence from the RANDU LCG to an array with two columns. We can then zip the two columns together to make tuples (just pairs or two-tuples).

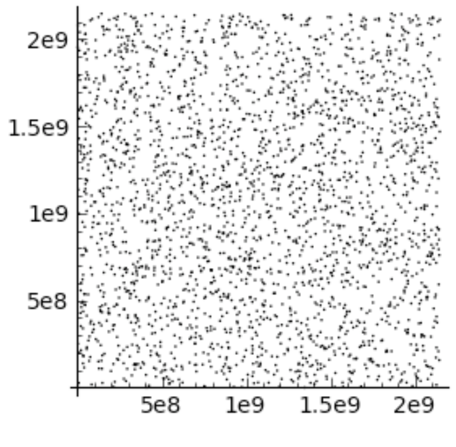
```
import pylab
m, a, c, x0, n = 2147483648, 65539, 0, 1, 5010
randu = pylab.array(linConGen(m, a, c, x0, n))
```

```
pylab.shape(randu)
```

```
randu.resize(5010/2, 2) # resize the randu array to 2 columns
```

```
pylab.shape(randu)
```

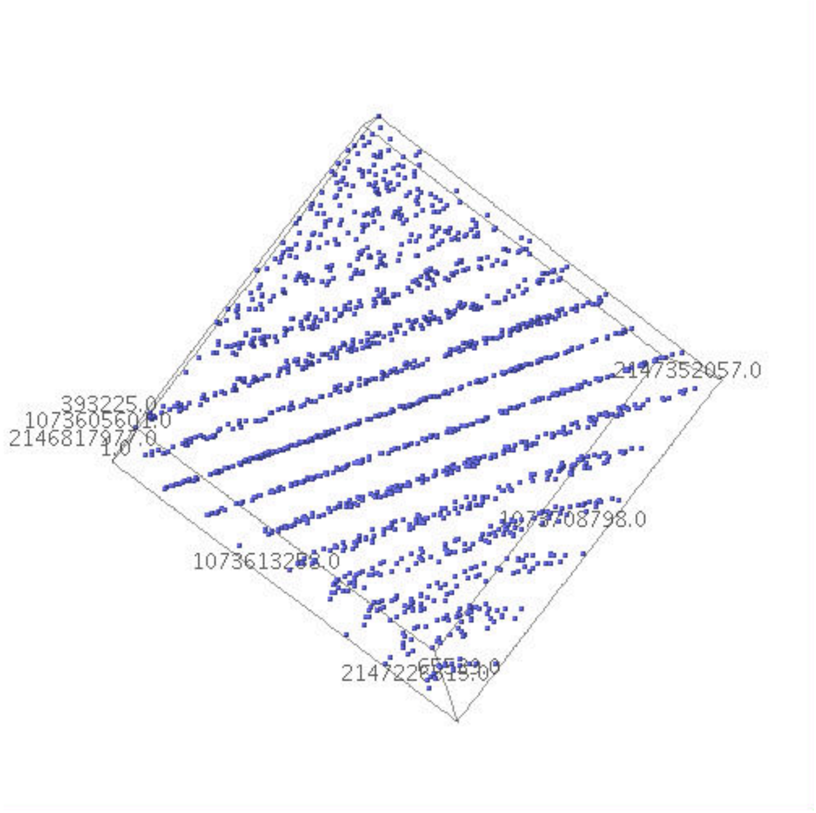
```
seqs = zip(randu[:, 0], randu[:, 1]) # zip to make tuples from columns
p = points(seqs, pointsize='1', color='black')
show(p, figsize = (3,3))
```



Let us resize the LCG to an array with three columns. We can then zip the three columns together to make tuples. The effect will be that if our original sequence was $x_0, x_1, x_2, x_3, \ldots, x_{n-3}, x_{n-2}, x_{n-1}$, we will get a list of triplets, tuples of length three, $(x_0, x_3, x_6), (x_1, x_4, x_7), \ldots, (x_{n-1-6}, x_{n-1-3}, x_{n-1})$. Unlike the pairs in 2D which seem well-scattered and random, triplets from the RANDU LCG are not very random at all! They all lie on parallel planes in 3D.

jsMath

```
import pylab
m, a, c, x0, n = 2147483648, 65539, 0, 1, 5010
randu = pylab.array(linConGen(m, a, c, x0, n))
randu.resize(5010/3, 3) # resize the randu array to 3 columns
seqs = zip(randu[:, 0], randu[:, 1], randu[:, 2]) # zip to make tuples from
columns
point3d(seqs, size=3)
```



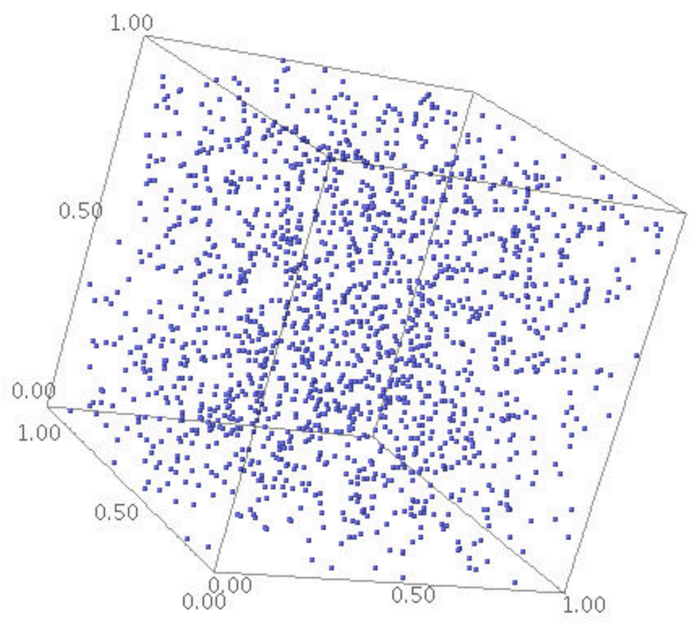
```

```

You can alter your perspective on this image using the mouse. From a particular perspective you can see that something has gone horribly wrong ... RANDU is a really **ugly** LCG.

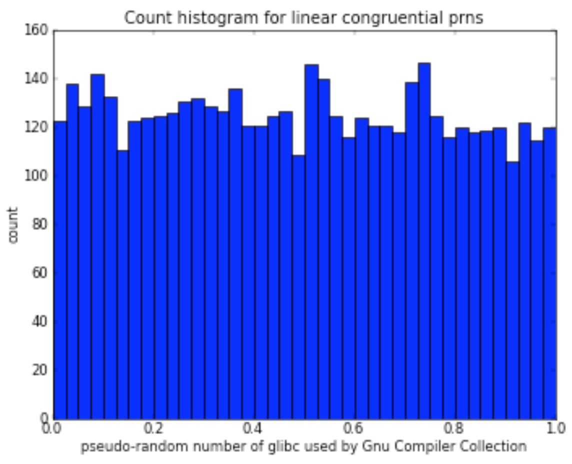
The above generators are of low quality for producing pseudo-random numbers to drive statistical simulations. We end with a positive note with a LCG that is in use in the Gnu Compiler Collection. It does not have obvious problems as in small periods or as high a correlation as RANDU.

```
#jmol error JmolInitCheck is not defined
import pylab
glibCGCCLcg = pylab.array([RR(x)/(2^32) for x in linConGen(2^32,
1103515245,12345,13,5010)])
glibCGCCLcg.resize(1670, 3) # resize the randu array to 3 columns
seqs = zip(glibCGCCLcg[:, 0], glibCGCCLcg[:, 1], glibCGCCLcg[:, 2]) # zip
to make tuples from columns
point3d(seqs, size=3)
```

jsMath



```
import pylab
glibCGCCLcg = pylab.array([RR(x)/(2^32) for x in linConGen(2^32,
1103515245,12345,13,5010)])
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(glibCGCCLcg, 40) # make the histogram (don't
have to have n, bins, patches = ...)
pylab.xlabel('pseudo-random number of glibc used by Gnu Compiler
Collection') # use pyplot methods to set labels, titles etc similar to as
in matlab
pylab.ylabel('count')
pylab.title('Count histogram for linear congruential prns')
pylab.savefig('myHist',dpi=(50)) # seem to need to have this to be able to
actually display the figure
```



Even good LCG are not suited for realistic statistical simulation problems. This is because of the strong correlation between successive numbers in the sequence. For instance, if an LCG is used to choose points in an n-dimensional space, the points will lie on, at most, $m^{1/n}$ hyper-planes. There are various statistical tests that one can use to test the quality of a pseudo-random number generator. For example, the spectral test checks if the points are not on a few hyper-planes. Of course, the Sample Mean, Sample Variance, etc. should be as expected. Let us check those quickly:

Recall that the population mean for a $Uniform(0,1)$ RV is $\frac{1}{2}$ and the population variance is $\frac{1}{12}$.

```
glibCGCCLcg.mean()  # check that the mean is close to the population mean
of 0.5 for Uniform(0,1) RV
```

```
1/12.0
```

```
glibCGCCLcg.var()   # how about the variance
```

To go into this topic in detail is clearly beyond the scope of this course. You should just remember that using computers to generate random numbers is not a trivial problem and use care when employing them especially in higher dimensional or less smooth problems.

More Sophisticated Pseudo-Random Number Generators

We will use a pseudo-random number generator (PRNG) called the Mersenne Twister for simulation purposes in this course. It is based on more sophisticated theory than that of LCG but the basic principles of recurrence relations are the same.

(The Mersenne Twister is a variant of the recursive relation known as a twisted generalised feedback register. See [Makato Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modelling and Computer Simulation, vol. 8, no. 1, Jan. 1998, pp. 3-20.], or the Wikipedia page.)

The Mersenne Twister has a period of $2^{19937}-1 \approx 10^{6000}$ (which is essentially a Very Big Number) and is currently widely used by researchers interested in statistical simulation.

```

```

You try

Accumulating sequences with pylab.cumsum

Our example using prngs is going to use a very useful function from the pylab module called cumsum. This calculates a cumulative sum from a sequence of numeric values. What do we mean by a cumulative sum? If we have a sequence $x_0, x_1, x_2, x_3, \dots, x_n$, then we can derive a sequence that is the cumulative sum,

$$x_0,\, x_0 + x_1,\, x_0 + x_1 + x_2,\, \dots, \sum_{i=0}^{n} x_i$$

Try evaluating the next cell to get the cumulative sum of the sequence 0, 1, 2, ..., 9

```
from pylab import cumsum, array # make sure we have the pylab stuff we need
cumsum(range(10))
```

You will see that the result is in a `pylab.array`. This can be useful, but all we need is a list, we can convert this to a list with the array's `tolist` method:

```
cumsum(range(10)).tolist()
```

Using the `list` function to make a list will do the same thing:

```
list(cumsum(range(10)))
```

Try some lists and cumsums for yourself. Please note that `cumsum` and `tolist (or list)` may come up in assessments. This is your chance to become familiar with them.

Now, say we have some probabilities in a list and we want to get the cumulative sum of them. Wecan use cumsum as we did above. We are going to start with just two probabilties which (for reasons that will shortly become clear), wel will assign as values to variables pLeft and pRight.

```
pLeft, pRight = 0.25, 0.25 # assign values to variables
cumulativeProbs = cumsum((pLeft, pRight)).tolist() # cumsum works on a
numeric tuple
cumulativeProbs
```

Note that `cumsum` works on a tuple as well as a list, providing that the elements of the tuple are some kind of number.

Simulating a Drunkard's Walk

You are now going to use some simulated $Uniform(0,1)$ samples to simulate the famous **Drunkard's Walk**. The idea of the Drunkard's Walk is that the Drunkard has no idea where he is going: at each decision point he makes a random decision about which way to go. We simulate this random decison using our $Uniform(0,1)$ samples.

We are going to have quite a limited version of this. At each decision point the Drunkard can either go one unit left, right, up or down. Effectively, he is moving around on a (x, y) coordinate grid. The points on the grid will be tuples. Each tuple will have two elements and will represent a point in 2-d space. For example, (0,0) is a tuple which we could use as a starting point.

First, we look at a useful feature of tuples: we can **unpack** a tuple and assign its elements as values to specified variables:

```
some_point = (3,2) # make a tuple
some_point_x, some_point_y = some_point # unpack the tuple and assign
values to variables
some_point_x # disclose one of the variables
```

```
some_point_y
```

We use this useful feature in the functions we have defined below. You now know what is happening in each ste jsMath these functions and should be able to understand what is happening.

```
def makeLeftTurnPoint(listOfPoints):
    '''Function to make a point representing the destination of a left turn
from the current path.

    Param listOfPoints is a list of tuples representing the path so far.
    Returns a new point to the immediate left of the last point in
listOfPoints.

    Tuples in the list represent points in 2d space.
    The destination of a left turn is a tuple representing a point which on
a 2d axis would
       be to the immediate left of the last point in the list representing
the current path.'''

    newPoint = (0,0)  # a default return value
    if len(listOfPoints) > 0:  # check there is at least one point in the
list
        lastPoint_x, lastPoint_y = listOfPoints[len(listOfPoints)-1]  #
unpack the last point in the list
        new_x = lastPoint_x - 1  # a new x coordinate, one unit to the left
of the last one in the list
```

```
def makeRightTurnPoint(listOfPoints):
    '''Function to make a point representing the destination of a right
turn from the current path.

    Param listOfPoints is a list of tuples representing the path so far.
    Returns a new point to the immediate right of the last point in
listOfPoints.

    Tuples in the list represent points in 2d space.
    The destination of a right turn is a tuple representing a point which
on a 2d axis would
       be to the immediate right of the last point in the list representing
the current path.'''

    newPoint = (0,0)  # a default return value
    if len(listOfPoints) > 0:  # check there is a least one point in the
list
        lastPoint_x, lastPoint_y = listOfPoints[len(listOfPoints)-1]  # the
last point in the list
        new_x = lastPoint_x + 1  # a new x coordinate one unit to the right
of the last one in the list
        newPoint = (new_x, lastPoint_y) # a new point one unit to the right
```

You should be thinking "Hey that is a lot of duplicated code: I thought that functions should help us not to duplicate code!". You are completely right, but this example will be easier to understand if we just live with some bad programming for the sake of clarity.

Now, experience the flexibility of Python: We can have lists of functions!

```
movementFunctions = [makeLeftTurnPoint, makeRightTurnPoint]
type(movementFunctions[0])
```

jsMath

To demonstrate how we can use our list of functions, take some path which is represented by a list of points (tuples):

```
somePath = [(0,0), (1,0), (2,0)]
# find the point that is the rightTurn from the last point in the path
```

If you want to, see if you can find the tuple which is the last point in the path so far using `len(somePath)` and the indexing operator [ ].

We put the functions into the list in order, left to right, so we know that the first function in the list (index 0) is `makeLeftTurnPoint`, and the second function in the list (index 1) is `makeRightTurnPoint`. We can now call `makeLeftTurnPoint` by calling `movementFunctions[0]`, and passing it the arguement `somePath` which is our example path so far. We should get back a new point (tuple) which is the point to the immediate left of the last point in the list `somePath`:

```
movementFunctions[0](somePath)
```

This has not added the new point to the list, but we can do this as well:

```
newPoint = movementFunctions[0](somePath) # make the new point
somePath.append(newPoint) # add it to the list
somePath # disclose the list
```

Try doing a similar thing to find same to now find the next **right** point.

That's all very well, but what about some random moves? What we are now going to do is to use random( ) to make our decisions for us. We know that the number generated by random will be in the interval [0,1). If all directions (up, down, left, right) are equally probable, then each has probability 0.25. All directions are independent. The cumulative probabiltiies can be thought of as representing the probabilities of (up, up or down, up or down or left, up or down or left or right).

```
from pylab import cumsum, array # make sure we have the pylab stuff we need
probs = [0.25 for i in range(4)]
cumProbs = cumsum(probs).tolist()
cumProbs
```

Using these accumulated probabilities we can simulate a random decision: if a realisation of a $Uniform(0,1)$, u, is such that $0 \leq u < 0.25$ then we go left. If $0.25 \leq u < 0.50$ we go right, if $0.50 \leq u < 0.75$, we go up, and if $0.75 \leq u < 1$ we go down.

We can demonstrate this:

jsMath

```
probs = [0.25 for i in range(2)] # only doing left and right here
cumProbs = cumsum(probs).tolist() # cumulative probabilities
n = 6  # number of simulated moves
prns = [random() for i in range(n)] # make a list fo random uniform(0,1)
samples
for u in prns: # for each u in turn
    if u < cumProbs[0]:
        print "u was", u, "so go left"
    elif u < cumProbs[1]:
```

You will see that we have only dealt with the cases for left and right. You may well not get n lines of output. You can have a go at improving this in very soon. First, one more thing ...

We can tidy up the if statement a bit by using another for loop and the break statement. The break statement can be used to break out a loop. In this case we want to compare u to each cumulative probability until we find a 'match' (u < cumulative probability) and then break out of the for loop.

```
probs = [0.25 for i in range(2)]  # only doing left and right here
cumProbs = cumsum(probs).tolist()  # cumulative probabilities
n = 6 # number of simulated moves
directions = ['left', 'right', 'up', 'down']  # all the directions even
though we only use left, right here
prns = [random() for i in range(n)] # make a list of random uniform(0,1)
samples
for u in prns: # for each u in turn
    for i in range(len(cumProbs)): # nest for loop
        if u < cumProbs[i]:
            print "u was", u, "so go", directions[i]
            break # break out of the nested for-loop, back into the outer
for-loop
        else:
```

Now, try adding the cases to deal with up and down.

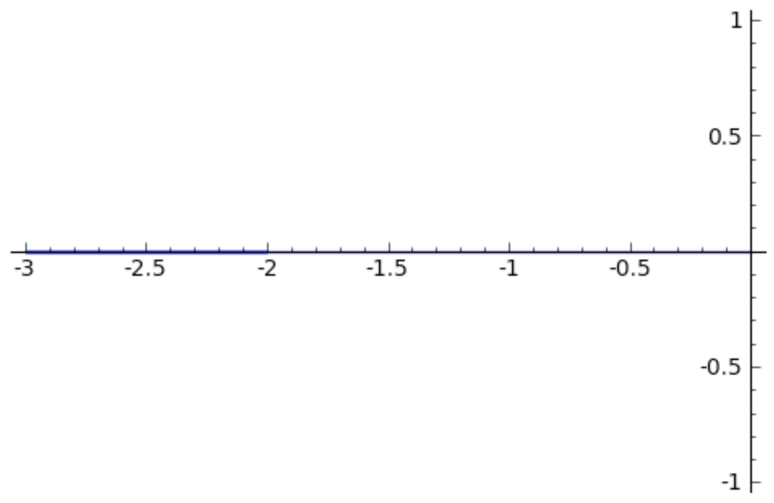
Now we can combine all this together to make a simulated Drunkard's Walk: First, a little helper function to plot a list of points as lines.

```
def linePlotter(listOfPoints):
    '''Function to plot a list of points as a lines between the points.

    Param listOfPoints is the list of points to plot lines between.'''

    l = line(listOfPoints)
    show(l)
```

Now the real stuff:

jsMath

```
from pylab import cumsum, array
startingPoint = (0,0)
drunkardsPath = [startingPoint] # start list with starting point tuple
n = 10
pLeft, pRight = 0.25, 0.25 # assign some probabilities to left and right
probs = [pLeft, pRight] # list of probabilities left and right only so far
movementFunctions = [makeLeftTurnPoint, makeRightTurnPoint] # list of
corresponding movement functions
cumProbs = cumsum(probs).tolist() # cumulative probabilities
prns = [random() for i in range(n)] # pseudo-random Uniform(0,1) samples
for u in prns:                        # for each pseudo-random u
    for i in range(len(cumProbs)):    # for each cumulative direction
probability
        if (u < cumProbs[i]):         # check if u is less than this
direction cumulative probability
            pointToAdd = movementFunctions[i](drunkardsPath)  # if so, find
new point to go to
            drunkardsPath.append(pointToAdd)                  # add it to
the path
            break    # the break statement breaks out of a loop, in the
case out of both loops
# out of both loops, have a path, so plot it
linePlotter(drunkardsPath)
```



A bit boring? A bit one-dimensional? See if you can add up and down to the Drunkard's repetoire. You will need to:

- Start by making adding some functions to make an up turn and a down turn, exactly as we have for making a left turn and a right turn.
- Add probabilities for up and down into the code.
- Remember to add your functions for up and down into the function list.

Try it and see!

jsMath

jsMath