Is Uncle Norm's shot going to exhibit a Weiner Process? Knowing Uncle Norm, probably, with a random drift and huge volatility.

Monte Carlo Simulations

... of stock prices – the primary model

Setting up

We sometimes regard the time-series stream of financial data that we are using as representing a continuous process, and that any data are a sample from a continuous population. More important, each observation at times "t" is completely independent (in the mathematical sense) of all prior observations except the immediately prior observation. This is sometimes called a “random number walk.”

In a financial Monte Carlo simulation, we treat each “day” as a random event, guided only by where we ended the previous day, which is a launching pad for today. Movement today is governed by a drift tendency and a weighted random selection from a standard normal distribution. For our elementary stock application, the drift tendency is our historical alpha and the distribution is, of course, our volatility measure.

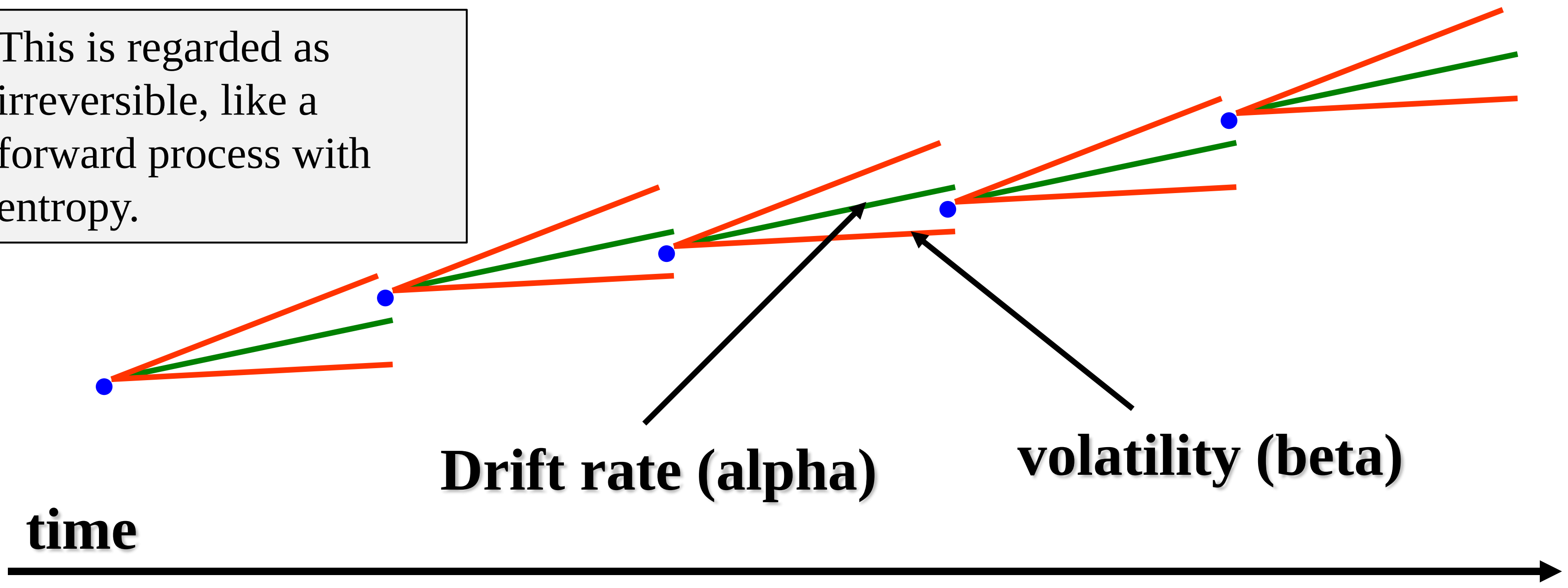
Multiple Monte Carlo simulations teach an important economic lesson: even profoundly accurate knowledge, such as a genuinely accurate estimate of a true mean and variance from a perfect Gaussian distribution, yields a future that has fundamental uncertainty.

In the Monte Carlo world, even the omniscient God really doesn’t know what is coming next. She just knows the odds.

About drift and volatility in this context ...

We are going to regard the path of stock prices as a process with actual price behavior over time reflecting ***drift*** and ***volatility***, where the latter is represented by a Gaussian distribution. The resulting pattern will reflect randomness with a trend. We also suspect the pattern will be non-repeating.

This is regarded as irreversible, like a forward process with entropy.



Where we are going with this ...

From our original assumption that this is geometric Brownian motion:

$$P_t = P_0 e^{\left[\left(\mu - \sigma^2/2\right)t + \sigma\varepsilon_t\right]}$$

We derive a slight alteration:

$$P_{t+1} = P_t e^{\left[\left(\mu - \sigma^2/2\right) + \sigma\varepsilon\right]}$$

The adjusted drift term

The volatility term

This is our gambling game: We have a special die. It has a Gaussian distribution with a mean μ and a standard deviation σ. At step "t" in our world, we role the die. Then we take the result of our roll, multiply it times sigma, add it to our adjusted mean, make that the power of an exponential and then multiply that times the value of P (price) at time t (now). Then we do it again, and again.

The gamble itself is represented by the expression σε.

ε refers to a random selection from a standard normal probability distribution (mean of zero, variance of 1) and that is multiplied times our standard deviation.

Using various Python random number generators ...

https://realpython.com/python-random/

I would like students in this class to at least scan-read this and understand the contents ... mostly because I want you to see how powerful numpy.random is.

… useful extracts from these pages

from the article on the previous page …

Python random Module	NumPy Counterpart	Use
random()	rand()	Random float in [0.0, 1.0)
randint(a, b)	random_integers()	Random integer in [a, b]
randrange(a, b[, step])	randint()	Random integer in [a, b)
uniform(a, b)	uniform()	Random float in [a, b]
choice(seq)	choice()	Random element from seq
choices(seq, k=1)	choice()	Random k elements from seq with replacement
sample(population, k)	choice() with replace=False	Random k elements from seq without replacement
shuffle(x[, random])	shuffle()	Shuffle the sequence x in place
normalvariate(mu, sigma) or gauss(mu, sigma)	normal()	Sample from a normal distribution with mean mu and standard deviation sigma

Note: NumPy is specialized for building and manipulating large, multidimensional arrays. If you just need a single value, random will suffice and will probably be faster as well. For small sequences, random may even be faster too, because NumPy does come with some overhead.

You should use numpy because of this!

… and you are **assigned to look at this page** to see what is there:
https://docs.scipy.org/doc/numpy/reference/routines.random.html

These include:
normal([mean,sigma,size])
standard_normal([size])
lognormal([mean,sigma,size])
laplace([loc,sigma,size])
poisson([lamda,size])
standard_cauchy([size])

The term "size" here refers to the size of the array that you want to build.

np.random.standard_normal([100]) will build an array of 100 rolls from a SN distribution, which is how we want to do it.

```
import numpy as np
draw = np.random.standard_normal([100])
```

Note: We are still using a psuedo-random number generator (PRNG), but we can build a crytographically secure random generator in Linux (and Windows I guess) if we want.

... more on the drift adjustment

monte_carlo_stock_price_v1_3

This is a version of the Monte Carlo simulator that is consistent with the modeling contained in the **Monte Carlo Simulations** lecture in Economics 136, assuming **Geometric Brownian Motion**. Prepared by Professor Evans on March 3, 2019, modified in April 2019 and January 6, 2020 (V3). This calculates Ito-adjusted-drift.

In [419]:
```
%matplotlib inline
```

In [420]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Set assumptions, including simulation length and the number of simulations. Note that if sims is increased more than 12, the color palettes below must be expanded.

In [421]:
```
days = 18                # default 18
sims = 1000              # default 1000
stock_sym = "HMC"
stock_pr = 100.00        # default 100.0
drift = 0.00041          # our mean, and we could call it that, but it is drift in our model default = 0.00041
sigma = 0.0180           # default 0.0180
call_strike = 110.0      # default 110.0
call_price = 0.84        # default 0.84
call_be = call_strike + call_price
# put_strike = 90
# put_price = 0.40
# put_be = put_strike - put_price
```

Set up the numpy arrays for efficiency. We are going to take our random draws for each step in all simulations before we do anything else. Numpy arrays must be typed (often a default is assumed) and the arrays of fixed size, and arrays must be initialized, just like the glory days of Fortran. Order equals 'C' is actually default and unnecessary but it is there to remind you that 'F' is an option.

In [422]:
```
setup = np.arange(sims*days)   #note here that we are creating one very long 1D array
setup = setup.reshape((sims,days),order='C')  #for contiguous columns, order = 'F'
draw = np.zeros_like(setup, dtype="float32")
price = np.zeros_like(setup, dtype="float32")
```

Set the random seed value if you want each simulation to be the same (while debugging or when asking students to submit simulations that must be identical for grading). To make it more "random," remove the seed command.

In [423]:
```
np.random.seed(742)
draw = np.random.standard_normal([sims,days])
```

Let's do the mean adjustment for the Ito method separately so that we remember that it is necessary:

In [424]:
```
ito_adj_drift = drift - ((sigma**2)/2)
"{:.7f}".format(ito_adj_drift)
```

Out[424]: '0.0002480'

$$\left(\mu - \sigma^2/_2\right)$$

... is necessary because the following two conditions are true:

Even though $EV(\varepsilon) = 0$ and $e^0 = 1$ because we have a skewed log-normal transformation, $EV(e^{\varepsilon}) > 1$

mu can be set to zero in this context, and often is, but not in our HW

Issues about Python/NumPy efficiency and latency ...
Contiguous arrays and speed

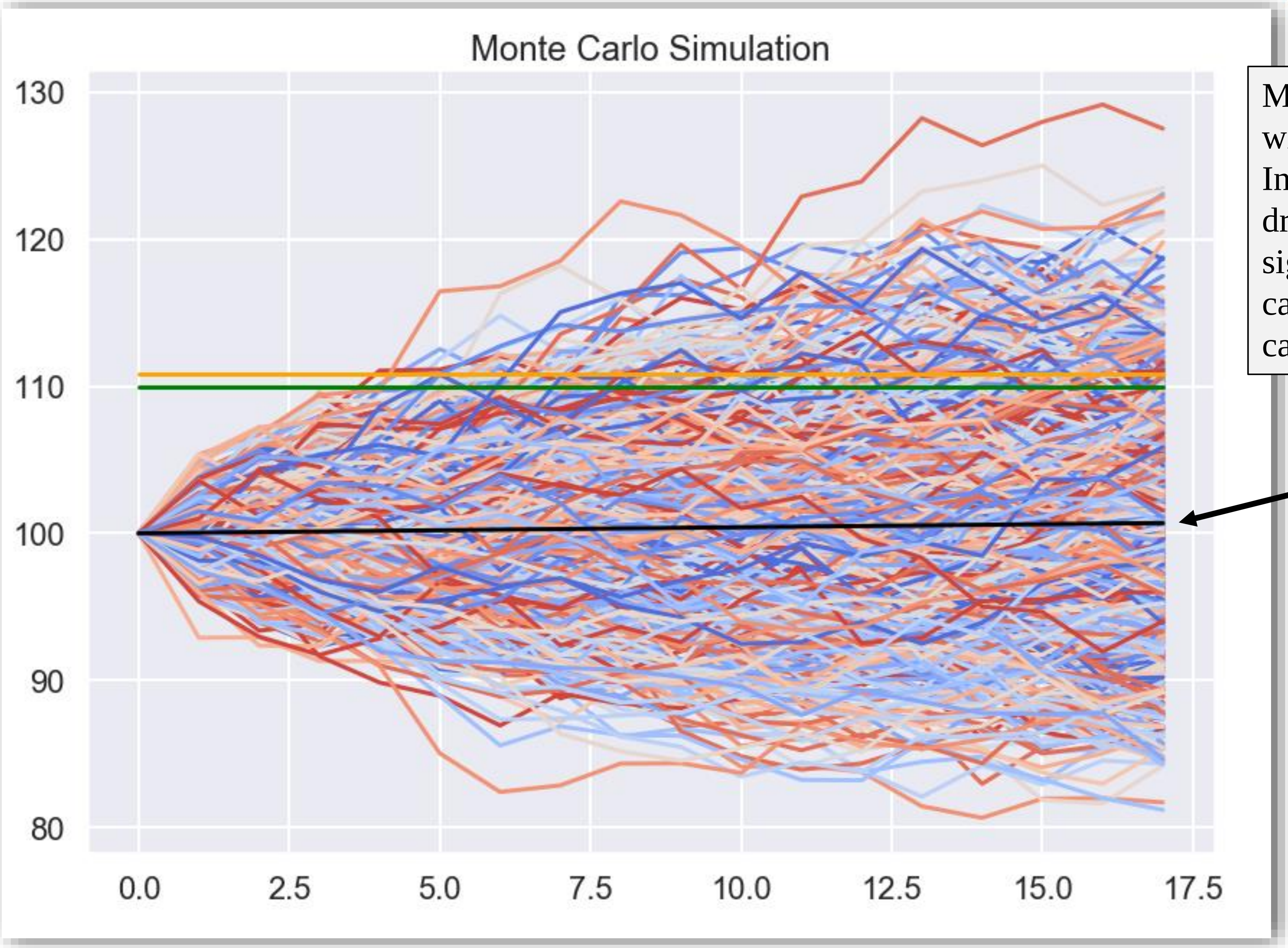
From our Monte-Carlo python assignment:

Set up the numpy arrays for efficiency. We are going to take our random draws for each step in all simulations before we do anything else. Numpy arrays must be typed (often a default is assumed) and the arrays of fixed size, and arrays must be initialized, just like the glory days of Fortran. Order equals 'C' is actually default and unnecessary but it is there to remind you that 'F' is an option.

In [422]:
```
setup = np.arange(sims*days)    #note here that we are creating one very long 1D array
setup = setup.reshape((sims,days),order='C')   #for contiguous columns, order = 'F'
draw = np.zeros_like(setup, dtype="float32")
price = np.zeros_like(setup, dtype="float32")
```

If you want speed from Numpy,

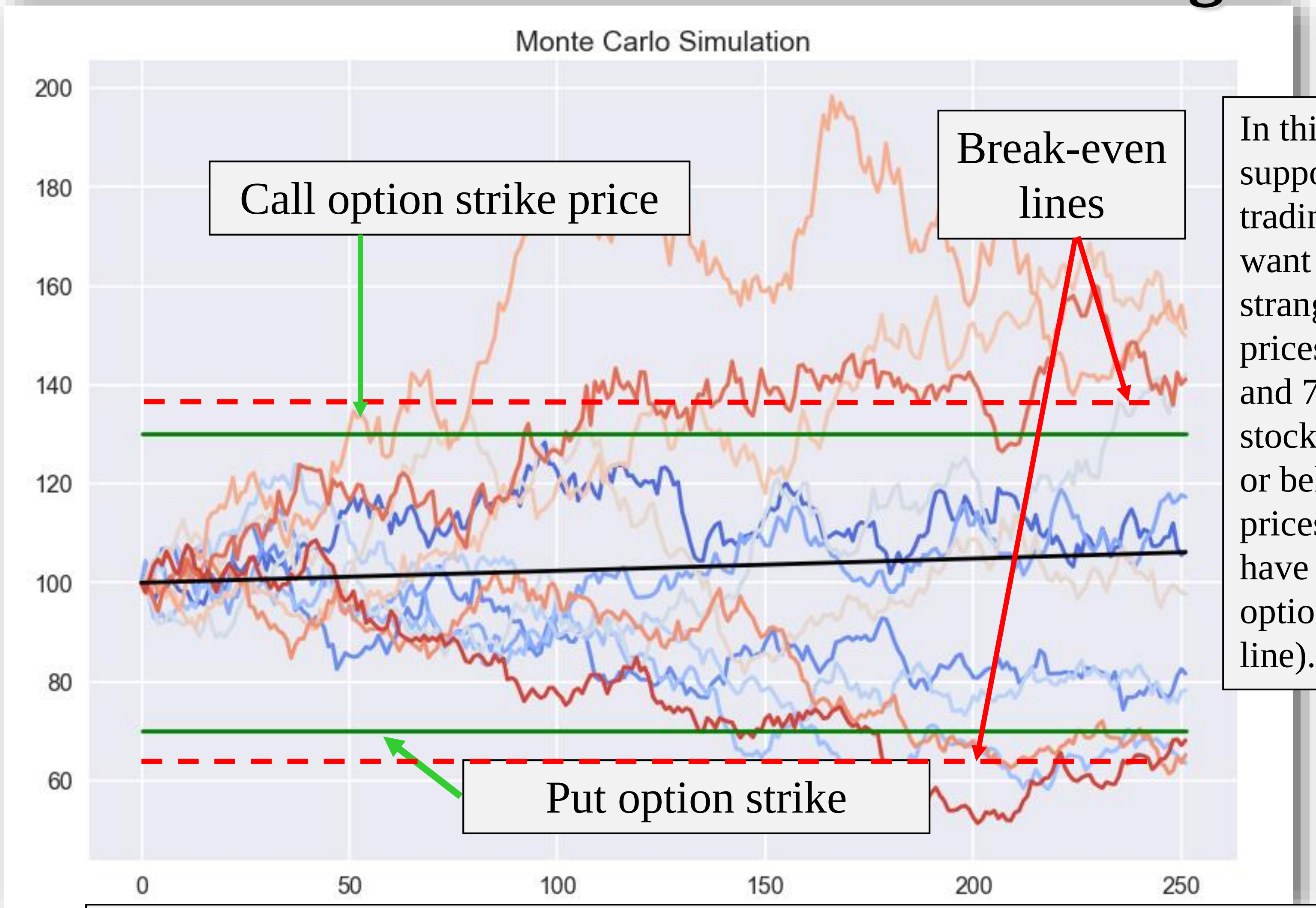
1. You should start n-dimensional arrays as defined 1-D arrays as shown in step 32.
2. You should reshape them into the n-dimensions that you want to use as shown in step 33. Numpy does not actually reshape them – this is a "view" feature of this language. In memory Numpy arrays are always 1D and sequential, BUT ...
3. You control the contiguous order with the "order" kwarg; C represents C-congruency, F represents Fortan-congruency (see documentation for "A"). C is default
4. You should then initialize matrix values as shown in step 34.



Monte Carlo Simulation of Stock with a call option and breakeven:
Initial Price: 100
drift mean: 0.00041
sigma: 0.0180
call strike: 110.0
call price: 0.84

Note that the drift line is rising slightly.

Monte Carlo Simulation of a Strangle



In this simulation suppose the stock is trading at 100 and we want to do a 1-year strangle at strike prices of 130 (call) and 70 (put). The stock has to go above or below these strike prices but we also have to cover our option costs (green line).

You wouldn’t use a graph to do this. Using a reliable random number generator, you would simulate 1,000+ simulations of a shorter period (maybe a few days) and count the number of times that the simulation is profitable at expiry, and perhaps the number of times the option goes to profitability (depending upon the strategy). I would like your model to be able to do this.

Again, this simulation does not include a Poisson (or equivalent) distribution, but perhaps we should. Here, though, we don't have to wait until expiration and normally wouldn't. If we did, two of these make money, one has value but we lose money, and two expire worthless. What clearly matters? Volatility.

... adding a Poisson distribution:

If we saw a daily distribution of 3 times our estimated normal distribution 2.7 times every 252 days, what it the probability of this abnormality not happening (and happening one, two, or three times in the next 30 days)?

0:	0.725
1:	0.233
2:	0.037
3:	0.004

Then you have to go back and have a random number generator spin a Boolean event and an activity level.

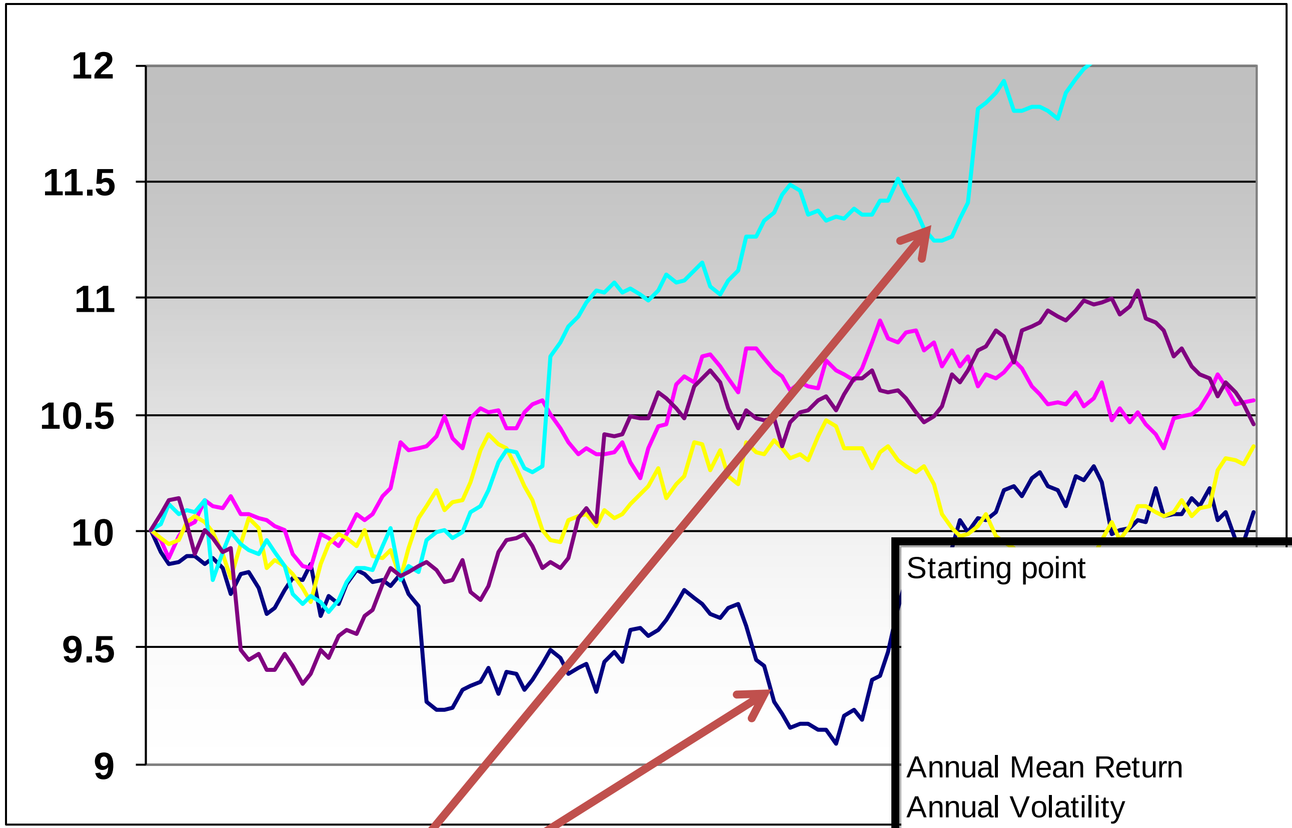
```
#
# Calculating probabilities with a Poisson distribution
# poisson.py in PyCo
#
import math

def poisson(p_lambda,k):
    prob = ((p_lambda**k)*math.exp(-p_lambda))/math.factorial(k)
    return prob

sample_days = 252
interval_days = 30
freq_252 = 2.7
p_lambda = freq_252*(interval_days/sample_days)
print (" Lambda:", p_lambda)
for x in range(4):
    probability = (poisson(p_lambda,x))
    print (" Number of bad events:", x, " Probability:",probability)
```

Why do we do this? Because we are trying to simulate kurtosis.

Jonathan Litz MC with Poisson Distribution



Courtesy Jonathan Litz, 2007.

Starting point		10	10
		9.910411	9.960015
		9.857999	9.883132
		9.862884	9.974233
		9.892259	10.01679
Annual Mean Return	1.08	9.890498	10.0397
Annual Volatility	0.10	9.854755	10.13563
One day	0.004000	9.884486	10.10473
SQRT One day	0.063246	9.841271	10.09925
Lambda	1.4	9.72816	10.14517
Lamdba / Day	0.0112	9.81413	10.06823
k	1	9.822809	10.07021
Prob of 1 Event on 1 Day	0.011075	9.754197	10.05046
Number Years	0.5	9.645041	10.04242

... you can model the random six-sigma event with this addition to the MCS.

... adding "six-sigma" to our placid s-normal distribution:

Price path = GBM random draw + extreme value* distribution draw

(1) Poisson distribution (what is the probability that this event will happen in the next interval given that it has happened with **λ** frequency in past intervals)?:

$$\mathrm{P}(k = x) = \frac{\lambda^k e^{-\lambda}}{k!}$$

(2) Gumbel distribution (used to model maximum levels from a sample of maximum values)

$$pdf = \frac{1}{\beta} e^{-(x+e^{-z})} \qquad z = \frac{x-\mu}{\beta}$$

μ is the mode,
B >0 is assigned

*drawn from "extreme value theory," (look this up in *Wikipedia*).

... adding a known high-sigma event (earnings) at the right time

NFLX	earnings reactions						
	Volume	Adj Close	ln	DCGR	Norm DCGR		
7/15/2015	30,898,600	98.13	4.59				
7/16/2015	63,461,000	115.81	4.75	0.17	5.11		
10/14/2015	33,231,500	110.23	4.70	0.00	0.14		
10/15/2015	48,484,300	101.09	4.62	-0.09	-2.67		
1/19/2016	33,283,700	107.89	4.68	0.04	1.12		
1/20/2016	52,926,300	107.74	4.68	0.00	-0.04		
4/18/2016	27,001,500	108.40	4.69	-0.03	-0.87		
4/19/2016	55,623,900	94.34	4.55	-0.14	-4.28		
7/18/2016	28,669,700	98.81	4.59	0.00	0.13		
7/19/2016	55,681,200	85.84	4.45	-0.14	-6.03		
10/17/2016	26,589,500	99.80	4.60	-0.02	-0.75		
10/18/2016	42,168,200	118.79	4.78	0.17	7.35		
1/18/2017	14,666,800	133.26	4.89	0.00	0.07		
1/19/2017	23,163,700	138.41	4.93	0.04	1.56		
	ADJ Close	Vol					
4/17/2017	147.25	16,364,700	4.99213	0.02985	1.32408		
4/18/2017	143.36	19,671,000	4.96536	-0.02677	-1.34312		
Date	Julian	Day	Close	Volume	cgr	XSigma2yr	XSigma1yr
2017-10-16 00:00:00	17289	Monday	202.68	18103086	0.015864	0.640317723	0.779835
2017-10-17 00:00:00	17290	Tuesday	199.48	23819054	-0.01591	-0.770017732	-1.02633
date	**open**	**close**	**volume**	**cgr**	**cgrnorm**	**XSigma2yr**	**XSigma1yr**
2018-01-22 00:00:00	222	227.58	17703293	0.031786	1.299796	1.299796107	1.201524
2018-01-23 00:00:00	255.05	250.29	27705332	0.095118	4.153723	4.153722962	3.93768
2018-04-16 00:00:00	315.99	307.78	20307921	-0.0125	-0.69561	-0.695612417	-0.71154
2018-04-17 00:00:00	329.66	336.06	33866456	0.087904	3.828646	3.828645663	3.626018

Simple ... use our software to

1. use hviexksmaster to pull 5 years (override) of earnings data
2. take the mean Xsigma values as our new sigma for only that date
3. use earn_calendar to figure out when the next earnings date will be
4. adjust your Monte Carlo to take a draw from XSigma Epsilon on that date

Old slide but still relevant ...

Portfolio Volatility and Monte Carlo Diversification Simulations

The slides that follow demonstrate the benefits of diversification using Vanguard's S&P500 Index fund **VFINX** and Vanguard's Intermediate Term U.S. Treasury Bond Fund, **VFITX**.

We take advantage of the sum of weighted variances:

$$V(aX+bY)=a^2V(X)+b^2V(Y)+2abCOV(X,Y)$$

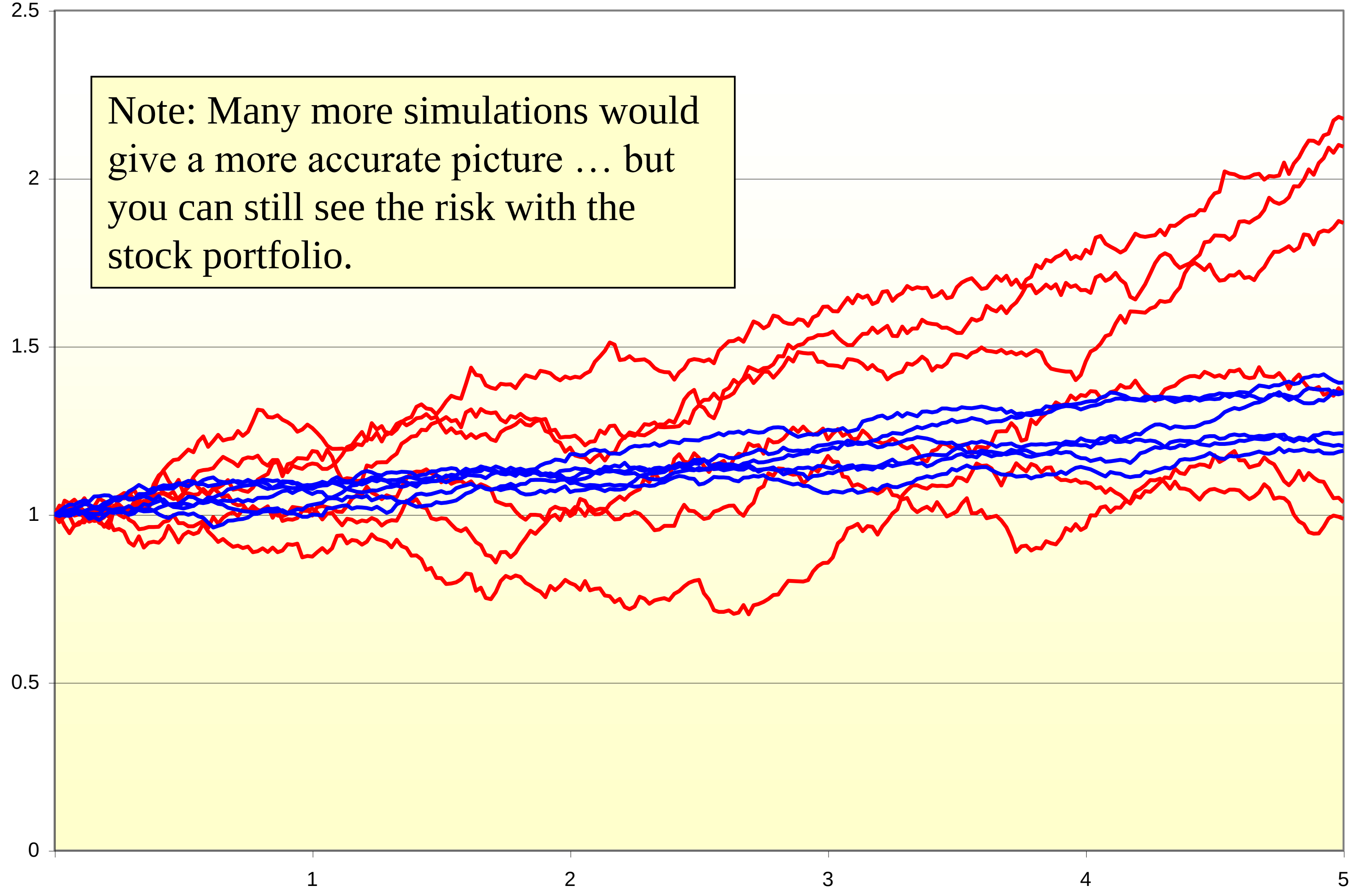
Remembering that statistically covariance is defined to be equal to the correlation coefficient of X and Y times the product of their standard deviations:

$$COV(X,Y)=COR(X,Y)SD(X)SD(Y)$$

we will achieve diversification only if X and Y are largely ***independent***!

Old slide but still relevant …

VFITX & VFINX Together



Old slide but still relevant …

An 80/20 vs. 50/50 Portfolio