A Collection of Tools Developed for

Sage's Quantitative Finance Library

Math 480A - Spring 2011

Contributions Made By: Spencer Wood, Harmony Mak, Andrea Michelle Frank, and Edwin Tsay

Table of Contents

Covariance Matrix 3

Univariate Statistics 9

Value-at-Risk Calculators 10

1. The Historical Method 10

2. Variance-Covariance Method 12

3. Monte Carlo Simulation 13

Stock Price Simulator: Monte Carlo Simulation Using Geometric Brownian Motion 16

Black-Scholes Option Pricing Model 18

Time Series Median Function 21

Bootstrapping 22

Covariance Matrix

Contributed by: Spencer Wood

For my final project I chose to add a covariance matrix class to sage. There are not a lot of things that one can do with the matrix itself but it does present the variances and covariances of a number of functions in a pleasing and practical way. As this is a class of convenience I designed it to take as many types of input as I could think of (sage/numpy lists and matrices, lists of TimeSeries, lists of strings representing stock tickers, and DiscreteRandomVariables in a list). Further, I added some additional tools which return other aspects in a visually convenient manner, including the correlations and a stats function which prints out three different types of analysis.

I have read research papers online where clever people have been able to derive certain truths from covariance matrices. So to make Sage a more attractive language for covariance analysts I have made it so that they can take out the CovMatrix as a Sage matrix to work with independently. Also, the CovMatrix is iterable and accessible in the same way as a Sage List so that one can work with the object directly for most applications. Lastly, I have seen instances where certain groups have found profound things from analysis of the eigenvalues/eigenvectors of covariance matrices, so I was sure to include funtions which would return these values and/or print them out in a convenient format.

Under the hood, the class converts all input (except random variables) to a numpy array (to ensure that all inner lists are of the same size) and then to a Sage list of TimeSeries. I chose to work with the TimeSeries class because it already has a very fast covariance algorithm, and kept me from having to rely on the slower one I wrote myself. Further, the finance.Stock class returns a TimeSeries anyway. These lists of TimeSeries are then passed on to a method which pops all the variances/covariances into a list, which is used in all the class' other methods.

The printing is governed by a set format. Labels are by default x1,x2,... but can be changed with the set_label function. When printing, labels are cut down to ten characters for the covariance table and four characters for the correlation. In regards to numbers, those with more than six digits, or that are smaller in magnitude than .001, are put into scientific notation in the covariance matrix. Otherwise, they print with three decimal points. In the correlation the numbers a floating point with eight decimal points precision since the correlation is always between zero and one. Each table prints zero as '0' to avoid ambiguity. Lastly, each of the functions which print automatically when called can be set to not print using printout=False, including the class itself.

When inputting stock tickers I made it possible to specify startdates and histperiods, but was not able to get enddates to work properly (each attempt caused Sage to crash completely). I also alloted a normalizing input, N, which if set to one will cause the covariance to be calculated using (n - 1) instead (a better fit for normally distributed data).

Type: <type ‘classobj’>

Definition: CovMatrix([noargspec])

Docstring:

Labeled table of variances and covariances of input data variables

INPUT:

- `values` Accepts data of the following types:
  - A "matrix like" object of the following types whos columns are variables and rows are observations:
    - Sage list with inner lists of all the same length
    - Sage matrix
    - Numpy array
    - Numpy matrix
  - A list of TimeSeries objects (each TimeSeries is a variable)
  - A list of strings representing ticker symbols for stocks
  - A list of DiscreteRandomVariable objects
- `N` (default = 0) if value is set to one then calculations will be made based on n - 1 degrees of freedom (a better fit for normally distributed data).
- `startdate`,``enddate``,``histperiod`` if using stock tickers then one can specify the dates to start and end data taking in the format 'Mon+d, +yyyy' and the period of the data entries takes the arguments 'daily' or 'weekly'

OUTPUT:

- A CovMatrix instance of the input data which is iterable and accessible in the same way as a sage list and is printed in a convenient manner.

AUTHORS:

- Spencer Wood (2011): initial version

EXAMPLES:

There are a number of different ways that data can be input:

Sage List.

```
sage: CovMatrix([[1,100],[1,99999],[5,10]])
 *          x1          x2
x1       3.556 -44479.556
x2 -44479.556   2.2197e9
```

Sage Matrix.

```
sage: c =
CovMatrix(matrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30
,30,30]]))
  *          x1          x2          x3
 x1     504.000     360.000     180.000
 x2     360.000     360.000           0
 x3     180.000           0     720.000
```

Numpy Array.

```
sage: a =
CovMatrix(numpy.array([[90,60,90],[90,90,30],[60,60,60],[60,60,90
],[30,30,30]]))
 *         x1         x2         x3
x1    504.000    360.000    180.000
x2    360.000    360.000          0
x3    180.000          0    720.000
```

Numpy Matrix.

```
sage: a =
CovMatrix(numpy.mat([[90,60,90],[90,90,30],[60,60,60],[60,60,90],
[30,30,30]]))
 *         x1         x2         x3
x1    504.000    360.000    180.000
x2    360.000    360.000          0
x3    180.000          0    720.000
```

List of TimeSeries objects.

```
sage: a = TimeSeries([50,25,37])
sage: b = TimeSeries([12,9,4])
sage: c = []
sage: c.append(a)
sage: c.append(b)
sage: d = CovMatrix(c)
 *         x1         x2         x3
x1    361.000    152.000    313.500
x2    152.000     64.000    132.000
x3    313.500    132.000    272.250
```

List of Strings Representing Stock Tickers.

```
sage: a = CovMatrix(['ge','gm','msft'])
   *         ge         gm       msft
  ge      4.569     -0.866      1.327
  gm     -0.866      6.391     -0.230
msft      1.327     -0.230      1.758
```

List of DiscreteRandomVariable objects.

```
sage: S = [ i for i in range(16) ]
sage: P = {}
sage: L = {}
sage: for i in range(15): P[i] = 2^(-i-1)
sage: for i in range(15): L[i] = 2^(-i+1)
sage: P[15] = 2^-16
sage: L[15] = 2^16
sage: X = DiscreteProbabilitySpace(S,P)
sage: R = DiscreteRandomVariable(X,P)
sage: N = DiscreteRandomVariable(X,L)
```

```
sage: l = []
sage: l.append(R)
sage: l.append(N)
sage: a = CovMatrix(l)
 *         x1         x2
x1      0.032     -0.206
x2     -0.206  65532.841
```

The printing can be suppressed with printout=False.

```
sage: a = CovMatrix([[1,100],[1,99999],[5,10]],printout=False)
(No output)
```

When using a list of strings it is possible to specify period with ‘daily’ or ‘weekly’ and starting date (default: one year prior) for the data in the format ‘Mon+d,+yyyy’.

```
sage: c =
CovMatrix(['ge','gm'],startdate='Jan+1,+2009',histperiod='daily')
 *         ge         gm
ge      8.559     -3.171
gm     -3.171      6.391
```

The class has a __repr__ function so that it prints out nicely.

```
sage: a = CovMatrix([[1,100],[1,99999],[5,10]],printout=False)
sage: print a
 *         x1         x2
x1      3.556 -44479.556
x2 -44479.556   2.2197e9
```

The class also has a __getitem__(x) function to recover individual values and can also be iterated.

```
sage: a =
CovMatrix(matrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30
,30,30]]))
 *         x1         x2         x3
x1    504.000    360.000    180.000
x2    360.000    360.000          0
x3    180.000          0    720.000
sage: a[1][1]
360.0
sage: for n in a:
sage:     print n
[504.0, 360.0, 180.0]
[360.0, 360.0, 0.0]
[180.0, 0.0, 720.0]
```

A call to corr() prints (can turn off printing with printout=False) and returns a list of correlations between the variables.

```
sage: a =
CovMatrix(matrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30
,30,30]]),printout=False)
sage: correlations = a.corr()
(  x1,  x2)(  x1,  x3)(  x2,  x3)
 0.67612340 0.23904572        0.0
sage: print correlations
[0.67612340378281333, 0.23904572186687872, 0.0]
```

A call to eigenvalues() returns a list of the CovMatrix's eigenvalues.

```
sage: a = CovMatrix(['ge','gm','msft'],printout=False)
sage: a.eigenvalues()
[1.22832367653, 4.64385624107, 6.84522680019]
```

A call to eigenvectors() prints (printout=False turns off printing) and returns a list of eigenvectors in the format [(eigenvalue,[(eigenvector)],multiplicity),...].

```
sage: a = CovMatrix(['ge','gm','msft'],printout=False)
sage: vects = a.eigenvectors()
Eigenvalue: 1.22832367653
Eigenvector: [(0.373834561268, 0.0214209528544, -0.927248005434)]
Eigenvalue: 4.64385624107
Eigenvector: [(0.823080490687, 0.453171663419, 0.34230680584)]
Eigenvalue: 6.84522680019
Eigenvector: [(-0.427535058974, 0.891165857881, -0.151780061587)]
sage: print vects
[(1.22832367653, [(0.373834561268, 0.0214209528544, -
0.927248005434)],
1), (4.64385624107, [(0.823080490687, 0.453171663419,
0.34230680584)],
1), (6.84522680019, [(-0.427535058974, 0.891165857881,
-0.151780061587)], 1)]
```

A call to matrix() will return a Sage Matrix of the CovMatrix.

```
sage: a =
CovMatrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30,30,30]
],printout=False)
sage: a.matrix()
[504.0 360.0 180.0]
[360.0 360.0   0.0]
[180.0   0.0 720.0]
```

One can input a list of strings to the set_labels() method to change the labels (Warning: Labels will be cut down to 10 chars for covariance and 4 chars for correlation). An empty call will reset the labels to the defaults (x1,x2,..).

```
sage: a =
numpy.matrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30,30,
30]])
sage: a = CovMatrix(a,printout=False)
sage: a.set_labels(['var1','verylongname','goober'])
```

```
sage: print a
         *       var1 verylongna     goober
      var1    504.000    360.000    180.000
verylongna    360.000    360.000          0
    goober    180.000          0    720.000

sage: b = a.corr()
(var1,very)(var1,goob)(very,goob)
 0.67612340 0.23904572        0.0

sage: a.set_labels()
sage: print a
 *         x1         x2         x3
x1    504.000    360.000    180.000
x2    360.000    360.000          0
x3    180.000          0    720.000
```

Calling stats() will neatly print out all the information CovMatrix can find out about the data.

```
sage: a =
numpy.matrix([[90,60,90],[90,90,30],[60,60,60],[60,60,90],[30,30,
30]],printout=False)
sage: a.stats()
Covariance:
 *         x1         x2         x3
x1    504.000    360.000    180.000
x2    360.000    360.000          0
x3    180.000          0    720.000

Correlations:
(  x1,  x2)(  x1,  x3)(  x2,  x3)
 0.67612340 0.23904572        0.0

Eigenvalues/Eigenvectors:
Eigenvalue: 44.8196602826
Eigenvector: [(0.648789898443, -0.741049913358, -0.172964428687)]
Eigenvalue: 910.069953041
Eigenvector: [(-0.65580225498, -0.429197796549, -0.621057689591)]
Eigenvalue: 629.110386676
Eigenvector: [(-0.385998795388, -0.516366417722, 0.764441399068)]
```

Univariate Statistics

Contributed by: Harmony Mak

The class uni_stats calculates univariate statistics commonly used in statistical analysis of stock prices. Given a times series, uni_stats converts it into a numpy array and calculates mean, variance, standard deviation, skewness, kurtosis, first/second/third quartiles of a stock. There is also a method quantile() that calculates any quantile.

Type: <type 'classobj'>

Definition: uni_stats([noargspec])

Docstring:

```
    Input: Time Series

    Output: Mean, variance, standard deviation, skewness, kurtosis,
first/second third quartiles

    Example:
    sage: v = finance.Stock('appl').close(startdate='Jan+1,+1990',
enddate='Jun+3,+2011', histperiod='weekly')
    sage: uni_stats(v)
    Mean 93.033625
    Variance 1120.05789561
    Standard deviation 33.4672660313
    Skewness 15.2115825277
    Kurtosis 91.3152704758
    First quartile 92.0
    Second quartile 92.0
    Third quartile 92.0
```

Value-at-Risk Calculators

Contributed by: Edwin Tsay – (Methods 1,2, & 3); Harmony Mak – (Method 3)

As the finance industry grows increasingly sophisticated due to the invention of complex financial instruments, the need for risk management is ever important. Companies which fail to produce daily risk estimates are at risk of overlooking potentially catastrophic losses or even insolvency. Equally important is the ability to interpret risk estimates and take action accordingly. Risk management has become the central focus of many financial institutions as they scramble to understand the pitfalls which lie before them as well as comply with new SEC regulations which require rigorous stress tests using techniques such as Monte Carlo simulations.

Beginning before the 1990s, numerous quantitative trading groups such as Banker's Trust had already begun implementing their own versions of risk management Value-at-Risk, or VaR. However, it was JP Morgan's RiskMetrics group which began extensive development of this ambiguously defined risk technique and subsequently published its methodology.

This project implements three commonly known methods of calculating VaR, which in this paper is defined as the *dollar value that an investor can lose on a specified portfolio, over a specified time period*. An example best illustrates this concept:

Using the historical method:

```
sage: hist_var(100000, "aapl", "Jan+1,+2004", "Feb+1,+2011")
sage: The 95% Value-at-Risk is: $ -3910.27
The 99% Value-at-Risk is: $ -6392.78
```

These results can simply be interpreted as follows: if an investor owned a $100,000 portfolio consisting entirely of only Apple stock (ticker: appl), he could at most lose $3910.27 with 95% confidence, over the next 24-hours (since the underlying data is the stock's daily closing price). The 99% VaR is $6392.76.

It is notable that VaR risk management is just a tool, albeit a dominant one, that risk managers use as guidelines to make informed decisions. There is no standardized definition of Var, nor a standard implementation of it. In this sense, VaR is more of an art than a science.

1. The Historical Method

This method is also commonly known as the non-parametric method, meaning nothing about the underlying distribution is assumed. Historical VaR estimates are calculated using empirical data.

File: /tmp/tmpZ4VYmx/___code___.py

Type: <type 'function'>

Definition: hist_var(w, ticker, start, end)

Docstring:

This function calculates VaR using empirical data. This method of calculating VaR is also known as the historical method, or the non-parametric method.

INPUT:

- `w` - float, the amount of 'wealth' (dollar amount) the investor wants to invest
- `ticker` - string, the ticker of the stock
- `start` - string, first day of the financial time series
- `end` - string, last day of the financial time series

OUTPUT:

float - the 95% and 99% Value-at-Risk of the given amount 'w' plot - a histogram of the returns

EXAMPLES:

This example illustrates basic use of this function assuming the investor wants to invest `100,000 into Apple (aapl), with data beginning from Jan 1st, 2004 and ending February 1st, 2011.

```
sage: hist_var(100000, "aapl", "Jan+1,+2004", "Feb+1,+2011")
sage: The 95% Value-at-Risk is: $ -3910.27
The 99% Value-at-Risk is: $ -6392.79
```

One may also input a float amount for wealth

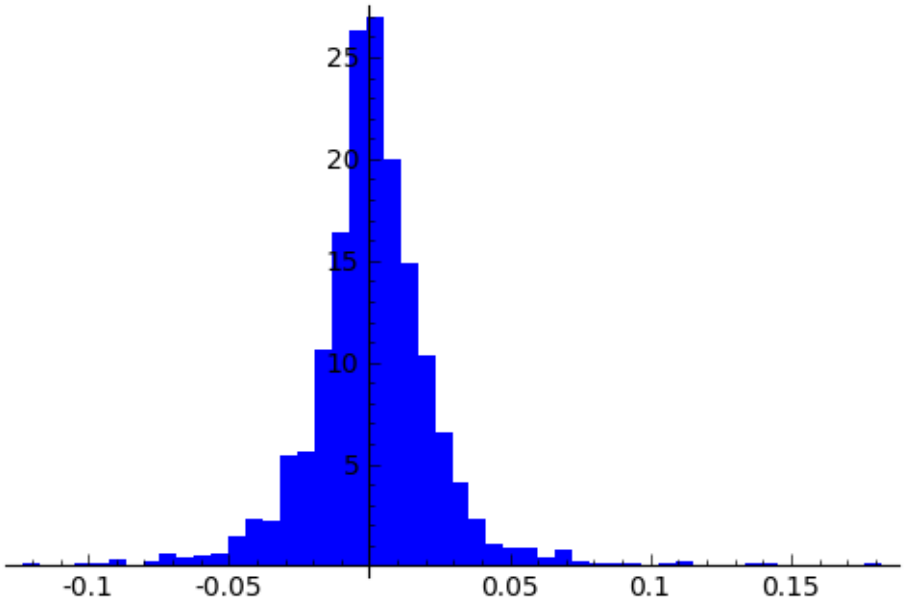
```
sage: hist_var(38242.24, "aapl", "Jan+1,+2004", "Feb+1,+2011")
sage: The 95% Value-at-Risk is: $ -1495.38
The 99% Value-at-Risk is: $ -2444.74
```

NOTES:

The non-parametric method uses specified empirical data to create the VaR figures. In this function, we model returns using the difference of the logs which is standard practice in the finance industry.

AUTHORS:

- Edwin Tsay (2011-05-31)



2. Variance-Covariance Method

This method is commonly known as the parametric method, meaning is assumes that the underlying stock price data follows a normal distribution. As this method converts stock price data into a lognormal distribution (common practice in industry), this is a valid assumption.

File: /tmp/tmpiIcrzX/___code___.py

Type: <type 'function'>

Definition: varcov_var(w, ticker, start, end)

Docstring:

This function calculates VaR using the variance-covariance (or parametric) method.

INPUT:

- `w` - float, the amount of 'wealth' (dollar amount) the investor wants to invest
- `ticker` - string, the ticker of the stock
- `start` - string, first day of the financial time series
- `end` - string, last day of the financial time series

OUTPUT:

float - the 95% and 99% Value-at-Risk of the given amount 'w' plot - a histogram of the returns

EXAMPLES:

This example illustrates basic use of this function assuming the investor wants to invest `100,000 into Apple (aapl), with data beginning from Jan 1st, 2004 and ending February 1st, 2011.

```
sage: varcov_var(100000, "goog", "Jan+1,+2004", "Feb+1,+2011")
sage: The 95% Value-at-Risk is: $ -3575.20
The 99% Value-at-Risk is: $ -5060.95
```

One may also input a float amount for wealth

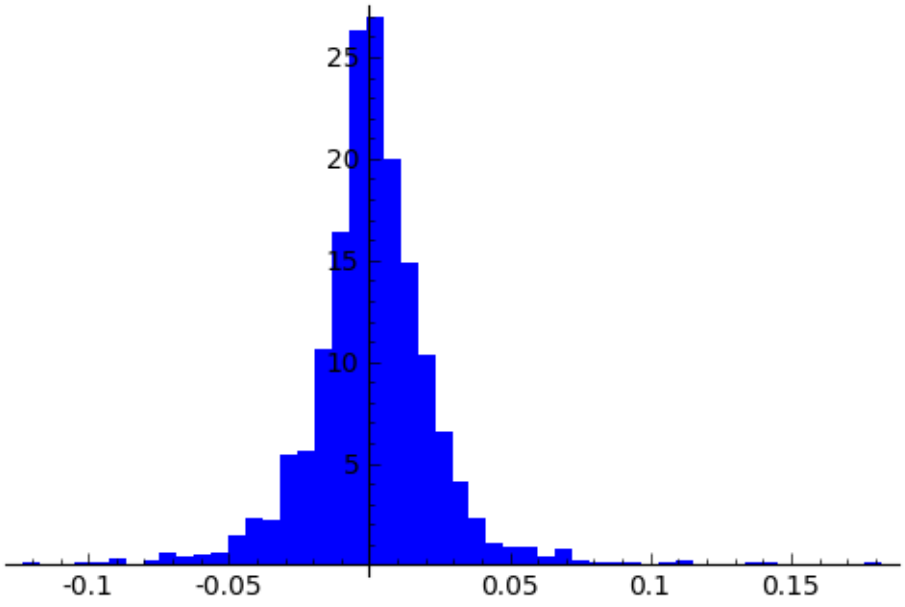
```
sage: varcov_var(38242.24, "aapl", "Jan+1,+2004", "Feb+1,+2011")
sage: The 95% Value-at-Risk is: $ -1478.29
The 99% Value-at-Risk is: $ -2102.42
```

NOTES:

The parametric method assumes a normal distribution. In this function, we model returns using the difference of the logs which is standard practice in the finance industry.

AUTHORS:

- Edwin Tsay (2011-05-31)



3. Monte Carlo Simulation

The final method explored in this project used to calculate VaR is the Monte Carlo simulation. The underlying model is geometric Brownian motion, a basic building block in the world of finance.

File: /tmp/tmpWCjUC_/___code___.py

Type: <type 'function'>

Definition: montecarlo_var(w, ticker, start, end, trials)

Docstring:

This function calculates VaR by running a Monte Carlo simulation, using geometric brownian motion, a specified number of trials.

INPUT:

- `w` - float, the amount of 'wealth' (dollar amount) the investor wants to invest
- `ticker` - string, the ticker of the stock
- `start` - string, first day of the financial time series
- `end` - string, last day of the financial time series
- `trials` - integer, number of desired trials

OUTPUT:

float - the 95% and 99% Value-at-Risk of the given amount 'w' plot - a histogram of the returns

EXAMPLES:

This example illustrates basic use of this function assuming the investor wants to invest `100,000 into Apple (aapl), with data beginning from Jan 1st, 2004 and ending February 1st, 2011, and wants to run a total of 10,000 trials.

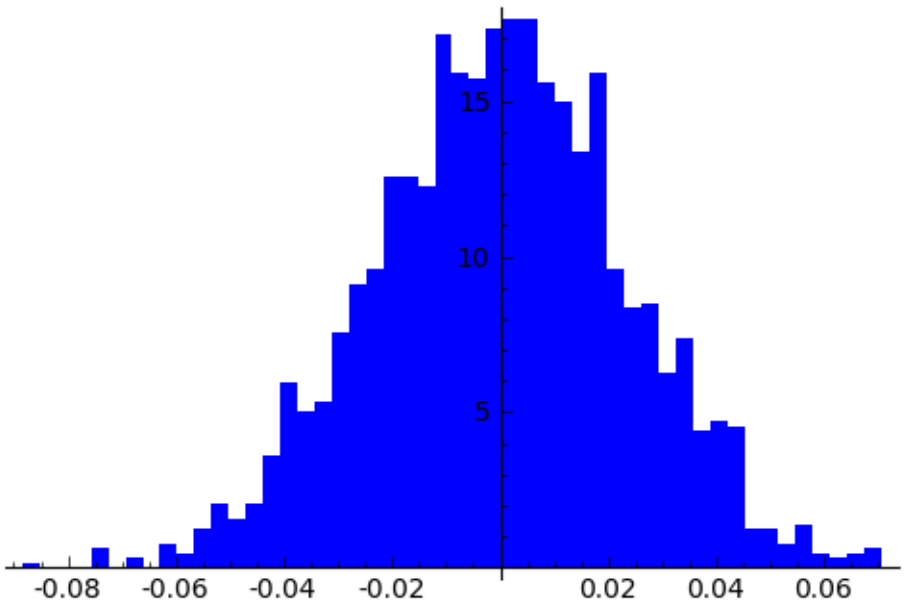
```
sage: montecarlo_var(100000, "aapl", "Jan+1,+2004", "Feb+1,+2011", 10000)
sage: The 95% Value-at-Risk is: $ -3627.70
The 99% Value-at-Risk is: $ -5013.99
```

One may also input a float amount for wealth

```
sage: montecarlo_var(38242.24, "aapl", "Jan+1,+2004", "Feb+1,+2011", 10000)
sage: The 95% Value-at-Risk is: $ -1385.54
The 99% Value-at-Risk is: $ -1938.043
```

AUTHORS:

- Edwin Tsay (2011-05-31)
- Harmony Mak (2011-05-31)


15
10
5
-0.08
-0.06
-0.04
-0.02
0.02
0.04
0.06

Stock Price Simulator: Monte Carlo Simulation Using Geometric Brownian Motion

Contributed by: Edwin Tsay, Harmony Mak

The most basic and widespread model used to predict stock prices is the Monte Carlo Simulation using geometric Brownian motion.

File: /tmp/tmpP_d4Oq/___code___.py

Type: <type 'function'>

Definition: monte_carlo_gbm(ticker, start, end, n)

Docstring:

This method predicts the price path of a specified stock where each trial represents the price of the stock the next day.

INPUT:

- `ticker` - string, the ticker of the stock
- `start` - string, first day of the financial time series
- `end` - string, last day of the financial time series
- `n` - integer, the number of days out the investor wants to predict

OUTPUT:

plot - a histogram of the stock prices plot - the price path of the stock over the next 'n' days

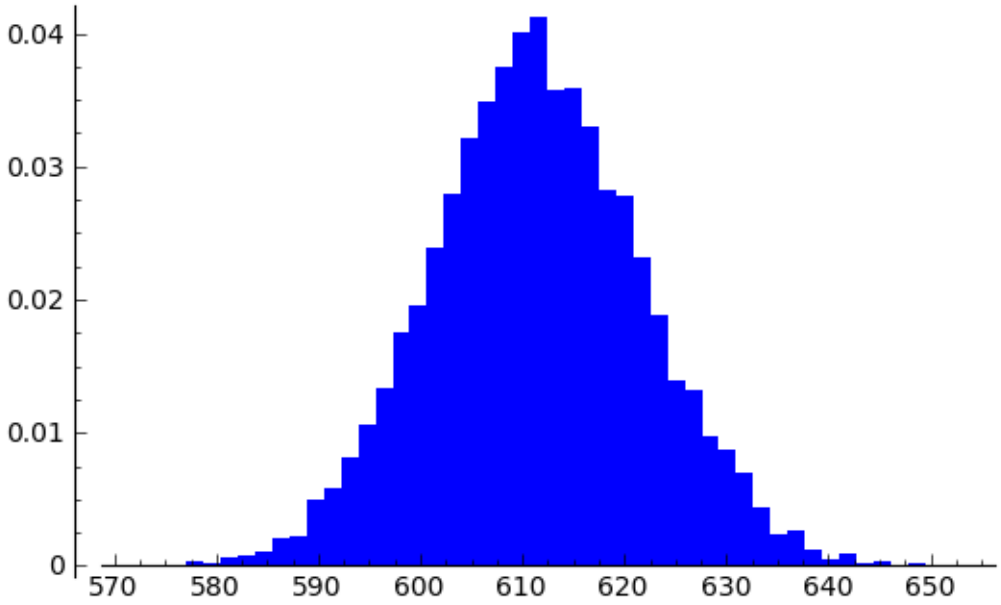
EXAMPLES:

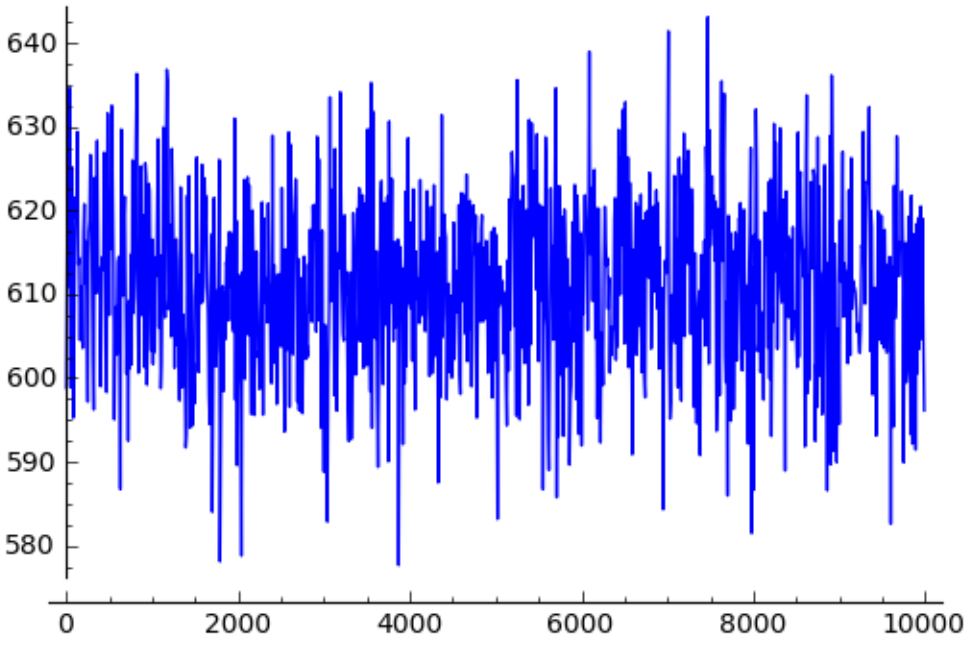
This example illustrates basic use of this function assuming the investor wants to invest `100,000 into Apple (aapl), with data beginning from Jan 1st, 2004 and ending February 1st, 2011, and wants to run a total of 10000 trials.

```
sage: monte_carlo_gbm("goog", "Jan+1,+2004", "Feb+1,+2011", 10000)
```

AUTHORS:

- Edwin Tsay (2011-05-31)
- Harmony Mak (2011-05-31)


0.04
0.03
0.02
0.01
0
570
580
590
600
610
620
630
640
650


640
630
620
610
600
590
580
0
2000
4000
6000
8000
10000

Black-Scholes Option Pricing Model

Contributed by: Eddie Tsay

The following can be found on http://trac.sagemath.org/sage_trac/ticket/4083. My contributions included resolving all these issues, which did not take an insignificant amount of time as working through the math behind Black-Scholes and swaptions is not trivial. I am happy to write that I have resolved all the following issues to my satisfaction and hope to resubmit this for review in the near future.

- **summary** changed from *[with patch, needs review] sage.finance - Options pricing implementation* to *[with patch, needs work] sage.finance - Options pricing implementation*

REFEREE:

This code is *really* good, and I definitely want it in Sage. There are a bunch of minor issues that need to be fixed.

1. A bunch of doctests fail when run on OSX 10.5 32-bit, which is maybe numerical noise:

```
teragon-2:finance wstein$ sage -t black_scholes.py
sage -t  devel/sage-main/sage/finance/black_scholes.py
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
43:
    sage: tenyr_swap.Black(5, 1.2, 0.25, 'n')
Expected:
    0.0679829347644
Got:
    0.067982934764359987
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
333:
    sage: opt.Black(5, 1.2, 0.25, 'ln')
Expected:
    1.38685477149
Got:
    1.3868547714858503
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
335:
    sage: opt._bs_ln()
Expected:
    1.38685477149
Got:
    1.3868547714858503
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
272:
    sage: aapl_200c.Black(175, 0.4, 0.5, 'ln')
Expected:
```

```
    10.8744664878
Got:
    10.874466487776381
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
318:
    sage: opt.Black(5, 1.2, 0.25, 'n')
Expected:
    0.567982934764
Got:
    0.5679829347643599
**********************************************************************
File "/Users/wstein/sage/devel/sage-main/sage/finance/black_scholes.py", line
320:
    sage: opt._bs_n()
Expected:
    0.567982934764
Got:
    0.5679829347643599
**********************************************************************
4 items had failures:
   1 of   7 in __main__.example_1
   2 of   5 in __main__.example_10
   1 of   4 in __main__.example_8
   2 of   5 in __main__.example_9
***Test Failed*** 6 failures.
For whitespace errors, see the file
/Users/wstein/sage/tmp/.doctest_black_scholes.py
        [13.8 s]
exit code: 1024

----------------------------------------------------------------------
The following tests failed:


        sage -t  devel/sage-main/sage/finance/black_scholes.py
Total time for all tests: 13.8 seconds
teragon-2:finance wstein$
```

2. There are three convenience functions with no doctests, which breaks the 100% coverage rule:

```
def n(x):       return scipy.stats.norm.pdf(float(x))
def N(x):       return scipy.stats.norm.cdf(float(x))
def invNorm(x): return scipy.stats.norm.ppf(float(x), loc=0, scale=1)
```

In fact the coverage score isn't very good:

```
teragon-2:finance wstein$ sage -coverage black_scholes.py
----------------------------------------------------------------------
black_scholes.py
ERROR: Please define a s == loads(dumps(s)) doctest.
SCORE black_scholes.py: 50% (9 of 18)

Missing documentation:
        * n(x):
```

```
* N(x):
* invNorm(x):
* _fullalpha(self, cps):
* _alpha(self):
* _beta(self):
* _ratio(self):
* _d1(self):
* _d2(self):
```

3. There is an empty TESTS: block at the top of the file. (Just delete it.)

4. There are several instances of % in docstrings, which will confuse latex. I'm not sure if this matters, since we're switching to Sphinx.

5. I see the text "Funciton call is inherent:" in the init method. It has typos and makes no sense.

6. Change things like this in docstrings

```
# optional -- requires internet and random
```

to

```
# random; optional -- internet
```

This will work using the new -only_optional doctesting framework, which allows us to test only particular components (e.g., those that require the internet).

Time Series Median Function

Contributed by: Andrea Michelle Frank

Sage did not include the median functionality for a TimeSeries object so we thought that including this would provide users with a practical function that can be one of many tools used to calculate statistical information on a set of data. Specifically, the TimeSeries median function has many uses within the Sage quantitative finance library one of which includes the ability to determine the median price of a stock based on its price history which also corresponds to the 2nd quartile of the distribution of the historical stock price.

Bootstrapping

Contributed by: Andrea Michelle Frank, Edwin Tsay

Resampling methods are useful in determining standard error, confidence intervals and significance tests. Using the bootstrap resampling method is one of the best ways to determine these values because this technique provides an understanding of the sampling distribution of a certain statistic, like a mean stock price. Understanding this distribution is necessary to obtain the values for standard error and other figures. Bootstrapping allows one to examine the accuracy of a sample statistic by taking various samples from the population with replacement. For each sample, this method finds the necessary sample statistic from each sample, which reveals the sampling distribution, and then one can approximate the population statistic from the various sample values. This process describes the Monte Carlo algorithm for resampling which produces an empirical bootstrap distribution of a sample statistic.

The bootstrapping method can be beneficial when the sampling distribution is unknown because sample statistics are determined without knowing the underlying distribution. The assumptions about the large sample size or a normal distribution are not necessary, making the calculations for each statistic consistent and independent of the distribution. Although bootstrapping doesn't require assumptions of the sampling distribution, by sampling with replacement one assumes that the population is independent and identically distributed. Bootstrapping is of great use in determining standard errors because at times bootstrapping methods can provide a more accurate answer than conventional difference in means procedures. Also, the same resampling procedure is performed no matter what statistic is being calculated, making it useful to calculate standard error for a wide range of distribution parameters.

As such, one can imagine the importance and convenience of bootstrapping as applied to finance. One of the most common models utilizing the bootstrap method is that of constructing a zero-coupon, fixed-income yield curve from empirical data such as prices from a set of coupon-bearing bonds. This is important because the yield curve is one of the most highly scrutinized graphs in all of finance. It is used to derive swap rates and any financial instrument requiring bond interest rates.

As applied to VaR, bootstrapping is used to compute estimated standard errors and confidence intervals for our n% VaR. These results inform the investor of the accuracy of the VaR estimates, which helps the investor make better risk-taking decisions.

File: /tmp/tmpT0haEB/___code___.py

Type: <type 'function'>

Definition: bootstrap(sample, samplesize=None, nsamples=1000, statfunc=<function mean at 0x433e500>)

Docstring:

Performs resampling from sample with replacement, gathers statistic in a list computed by statfunc on each generated sample.

INPUT:

- `sample` - time series, tuple or list; input sample of values
- `nsamples` - int; number of samples to generate
- `samplesize` - int; sample size of each generated sample
- `statfunc` - string; statistical function to apply to each generated sample

OUTPUT:

plot - a histogram of the stock prices plot - the price path of the stock over the next 'n' days

EXAMPLES:

This example illustrates basic use of this function assuming the investor wants to invest `100,000 into Apple (aapl), with data beginning from Jan 1st, 2004 and ending February 1st, 2011.

```
sage: goog_price_ld =
finance.Stock('goog').close(startdate="May+1,+2011",enddate="May+11,+2011").l
og().diffs()
sage: bootstrap(goog_price_ld, None, 1000, mean)
sage:   Number of Trials: 1000
```

Summary Statistics

Observed Mean Bias Standard Error

mean -0.00081734 -0.00067141 0.00014593 0.00255143 std 0.00735860 0.00720795 -0.00015065 0.00159275

AUTHORS:

- Edwin Tsay (2011-05-31)
- Andrea Michelle Frank (2011-05-31)

Reference: D.S. Moore, G.P. McCabe, W.M. Duckworth, S.L. Sclove, The Practice of Business Statistics: Using Data for Decisions, W.H. Freeman, New York (2002).