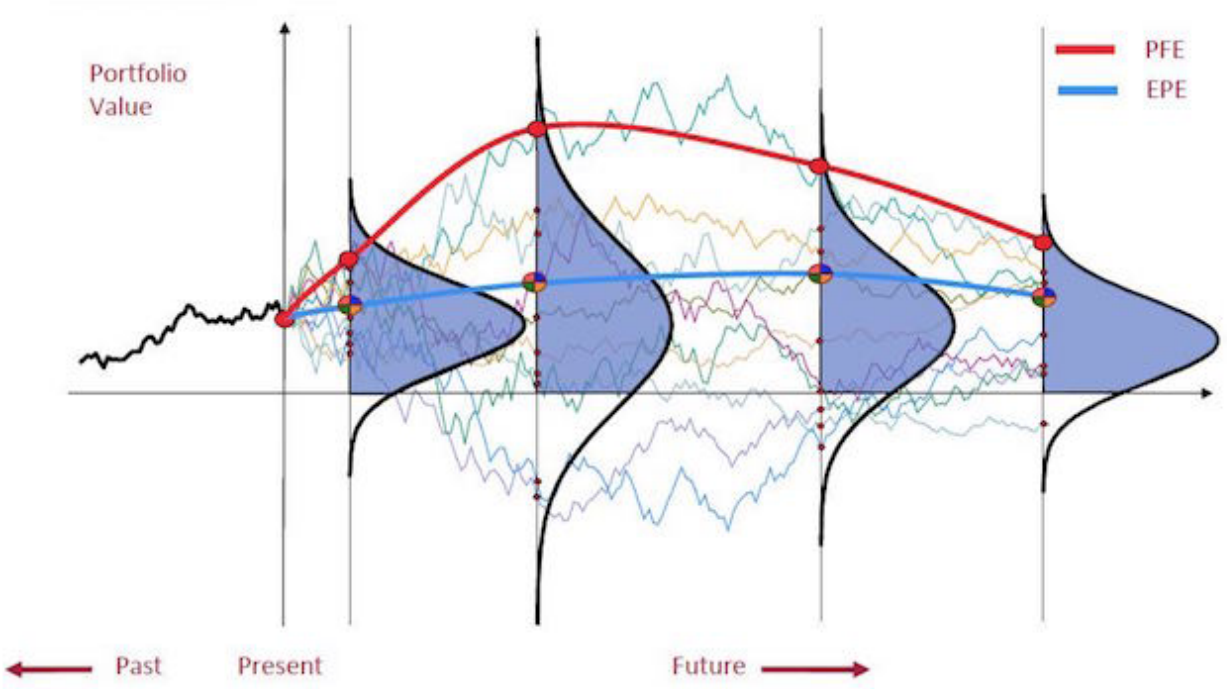
Julia as an HPC alternative for Quant finance

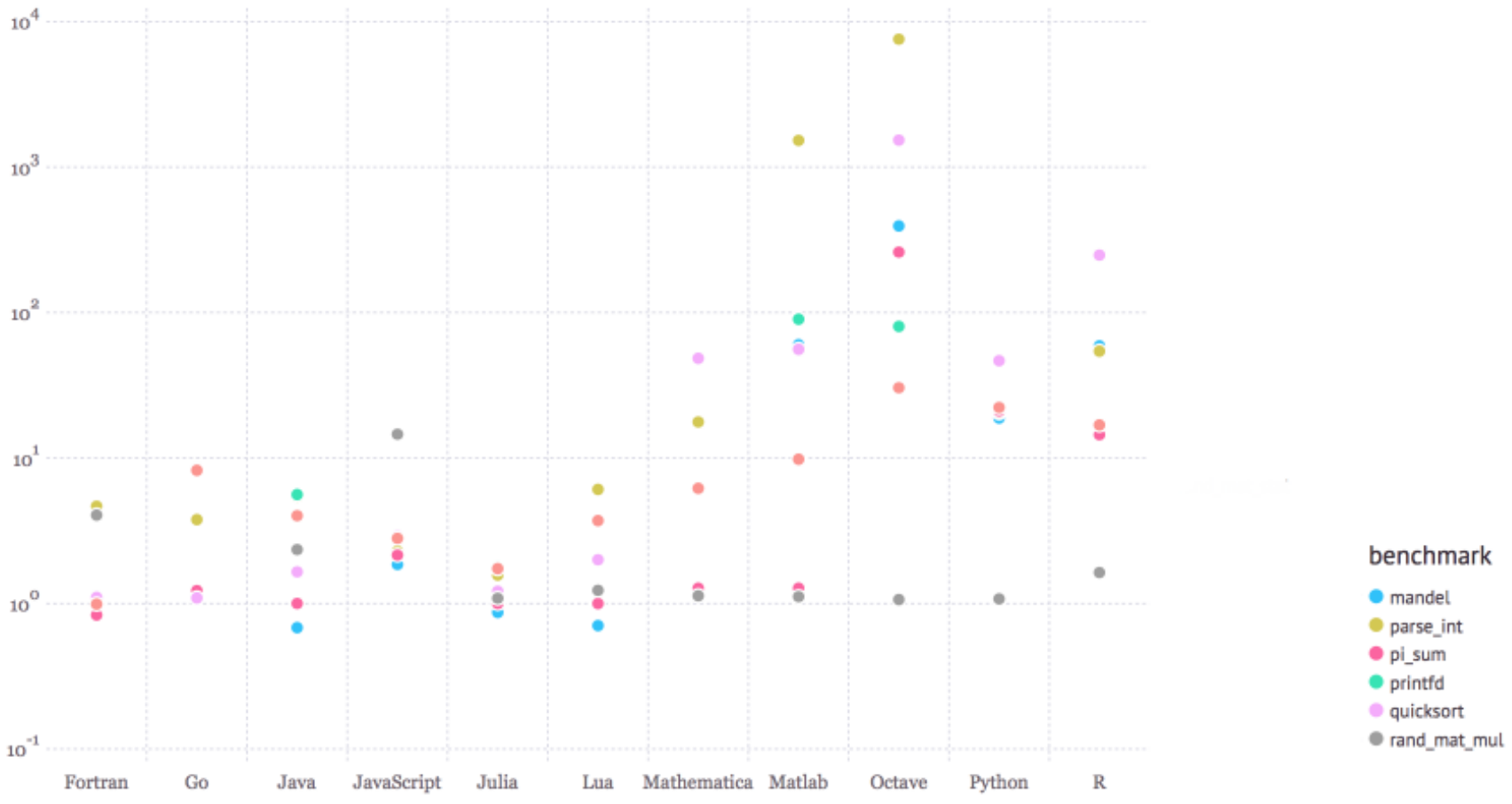


Breakdown of the talk

- Main features of Julia
- Derivative pricing
- Parallel processing
- Julia Community Groups
- High Performance Computing
- JuliaQuant
- Time series analysis
- Heteroskedasticity

What make Julia different ?

- Julia is written in Julia.
- Uses LLVM / JIT compilation, so it is fast !!!
- Code is uncluttered, basic functions are built in.
- Homoiconic design: provides runtime macros.
- Easy to execute on multicore and parallel processors.
- Can connect with modules / libraries written in other languages
- Can spawn tasks and interact with other programs



In [1]:

```
using PyPlot
T = 100;
S0 = 100;
dt = 0.01;
v = 0.2;
r = 0.05;
q = 0.0;
```

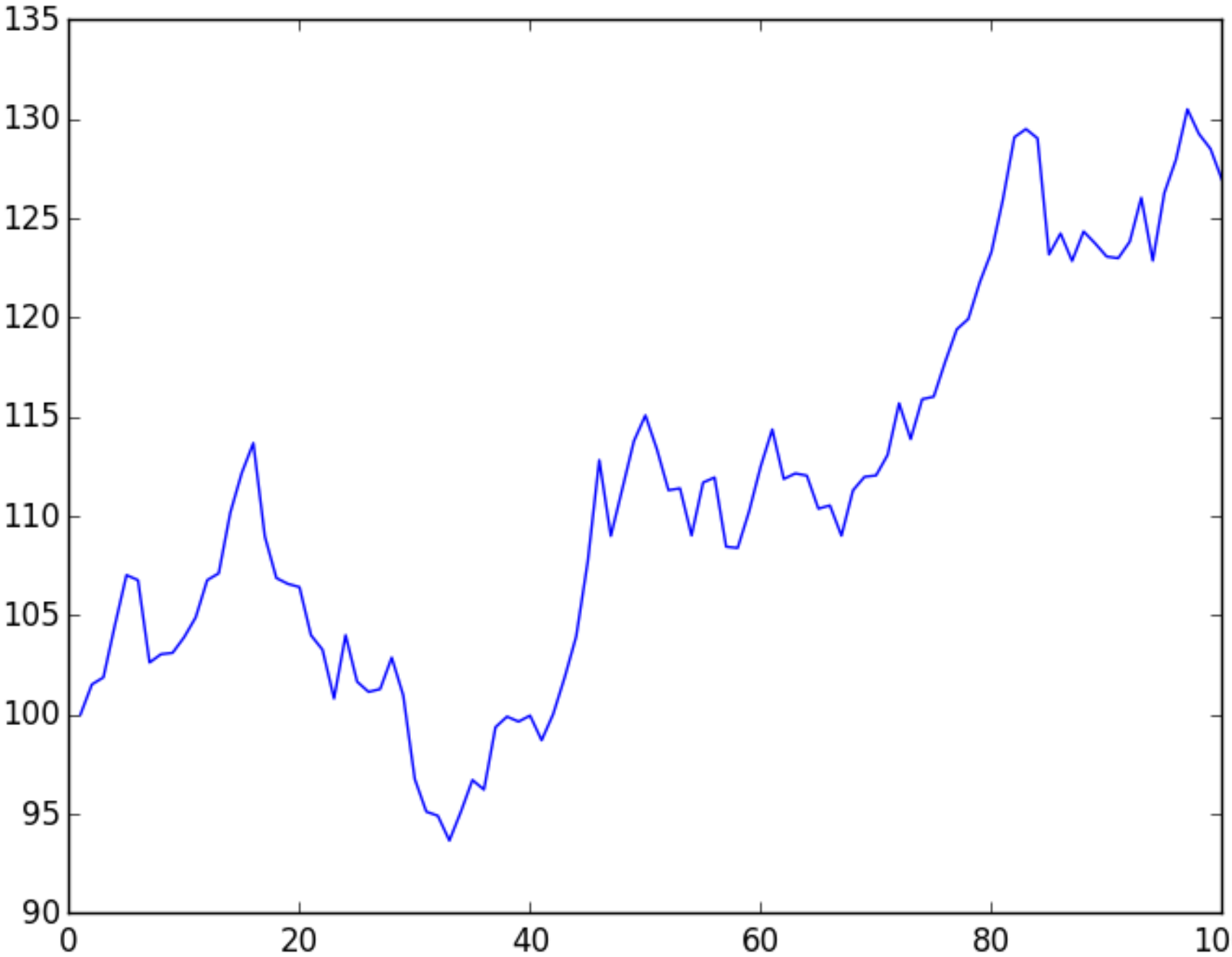
```
INFO: Loading help data...
```

In [6]:

```
S = zeros(Float64,T)
S[1] = S0;
iseed = ccall( (:clock, "libc"), Int32, ());
srand(iseed);

dW = randn(T)*sqrt(dt);
[ S[t] = S[t-1]*(1 + (r - q - 0.5*v*v)*dt + v*dW[t] + 0.5*v*v*dW[t]*dW[t]) for
t = 2:T ]

x = linspace(1,T);
plot(x,S)
```



Out[6]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x11c26e2d0>
```

In [7]:

```
function asianOpt(N::Int64, T::Int64; S0=100.0, K=100.0, r=0.05, q=0.0, v=0.1,
tma=0.25)

# European Asian option.
# Euler and Milstein discretization for Black-Scholes.

  @assert N > 0;
  @assert T > 0;

  dt = tma/T;
  S = zeros(Float64,T);
  A = zeros(Float64,N);

  for n = 1:N
    S[1] = S0
    dW = randn(T)*sqrt(dt);
    for t = 2:T
      z0 = (r - q - 0.5*v*v)*S[t-1]*dt;
      z1 = v*S[t-1]*dW[t];
      z2 = 0.5*v*v*S[t-1]*dW[t]*dW[t];
      S[t] = S[t-1] + z0 + z1 + z2;
    end
    A[n] = mean(S);
  end

# Define the payoff and calculate price

  P = zeros(Float64,N);
  [ P[n] = max(A[n] - K, 0) for n = 1:N ];
  return exp(-r*tma)*mean(P);

end
```

Out[7]:

```
asianOpt (generic function with 1 method)
```

In [8]:

```
price = asianOpt(100000,100; K=102.0,v=0.2);
@printf "Option Price: %10.4f\n\n" price;
```

```
Option Price:     1.6638
```

In [9]:

```
@elapsed asianOpt(100000,100;K=102.0,v=0.2)
```

Out[9]:

```
0.374801638
```

Comparison with Python, R, Matlab etc.

`	C	Python (v3)	R	Octave	Julia	Java	Javascript
Timings	1	32.67	154.3	789.3	1.41	2.97	4.22
Price	1.681	1.671	1.646	1.632	1.680	1.663	1.664

In [10]:

```
# global CPU_CORES  (sysinfo.jl : line 22 )
#
ncores = ccall(:jl_cpu_cores, Int32, ())
```

Out[10]:

```
4
```

In [11]:

```
addprocs(3);
```

In [12]:

```
n = nprocs()
```

Out[12]:

```
4
```

In [13]:

```
;cat Exotics.jl
```

```
module Exotics

using Yeppp
yp = Yeppp

function asianOpt(N::Int64, T::Int64; S0=100.0, K=100.0, r=0.05, q=0.0, v=0.1,
tma=0.25)

# European Asian option.
# Euler and Milstein discretization for Black-Scholes.

  @assert N > 0;
  @assert T > 0;

  dt = tma/T;
  S = zeros(Float64,T);
  A = zeros(Float64,N);

  for n = 1:N
```

```
    S[1] = S0
    dW = randn(T)*sqrt(dt);
    for t = 2:T
      z0 = (r - q - 0.5*v*v)*S[t-1]*dt;
      z1 = v*S[t-1]*dW[t];
      z2 = 0.5*v*v*S[t-1]*dW[t]*dW[t];
      S[t] = S[t-1] + z0 + z1 + z2;
    end
    A[n] = mean(S);
  end

# Define the payoff and calculate price

  P = zeros(Float64,N);
  [ P[n] = max(A[n] - K, 0) for n = 1:N ];
  return exp(-r*tma)*mean(P);

end

function asianOptYep(N::Int64, T::Int64; S0=100.0, K=100.0, r=0.05, q=0.0, v=0.
1, tma=0.25)

# European Asian option.
# Euler and Milstein discretization for Black-Scholes.

  @assert N > 0;
  @assert T > 0;

  dt = tma/T;
  S = zeros(Float64,T);
  A = zeros(Float64,N);
  W = Array(Float64,N*T);

  k = [1.0 + (r - q - 0.5*v*v)*dt, v, 0.5*v*v]
  dW = randn(N*T)*sqrt(dt);
  yp.evalpoly!(W,k,dW);

  for n = 1:N
    S[1] = S0
    for t = 2:T
      i = (n-1)*T + t;
      S[t] = S[t-1]*W[i];
    end
    A[n] = mean(S);
  end

# Define the payoff and calculate price

  P = zeros(Float64,N);
  [ P[n] = max(A[n] - K, 0) for n = 1:N ];
  return exp(-r*tma)*mean(P);

end

end
```

In [14]:

```
import Exotics
@time Exotics.asianOpt(1000000,100; K=102.0,v=0.2)
```

```
elapsed time: 3.708321446 seconds (1818867656 bytes allocated, 41.92% gc time)
```

Out[14]:

```
1.675951151108044
```

In [15]:

```
@everywhere using Exotics
```

In [16]:

```
tic();
@everywhere price = Exotics.asianOpt(250000,100; K=102.0,v=0.2);
totprice=0.0;
for i = 1:n
    totprice += remotecall_fetch(i,()->price)
end
totprice = totprice/n;
toc()

@printf "Asian price is %7.4f\n" totprice
```

```
elapsed time: 1.952977148 seconds
Asian price is  1.6728
```

Julia Community Groups:

- JuliaStats
- JuliaOpt
- JuliaWeb

- JuliaQuant
- JuliaFinMetriX
- QuantEcon

- JuliaParallel
- JuliaGPU

JuliaStats group :

- StatsBase
- Distributions
- MultivariateStats
- KernelDensity / KernelEstimator

- Dataframes
- RDatasets

- GLM
- Lora / MCMC
- Clustering

JuliaParallel group :

- ClusterManagers
- Elly, HDFS
- MPI
- Yeppp

In [17]:

```
using Yeppp
@time Exotics.asianOptYep(1000000,100; K=102.0,v=0.2)
```

```
elapsed time: 2.732427 seconds (2426614384 bytes allocated, 4.00% gc time)
```

Out[17]:

```
1.676402559571753
```

JuliaGPU group :

- CUDA, CUDArt, CUBLAS, CUFFT
- OpenCL, CLBLAS, CLFFT

Kepler GK110

Architecture

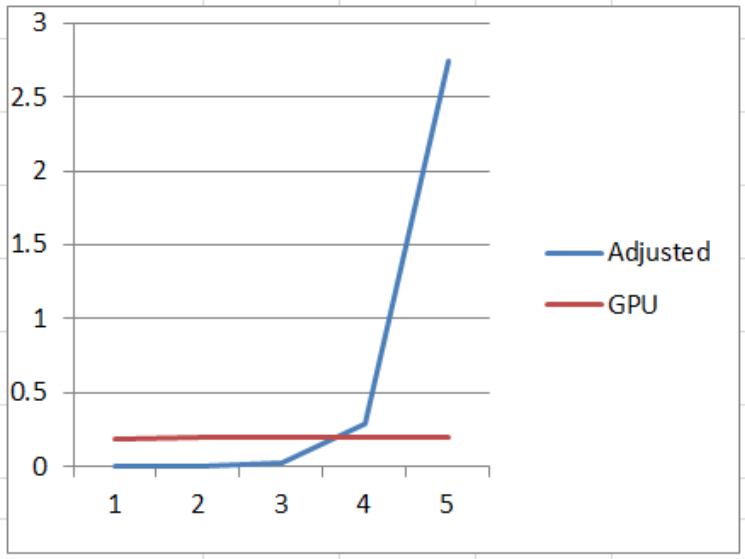
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

In [ ]:

```
;cat OCL/saxpy.jl
```

OpenCL on Lenovo i7 / NVIDIA GeForce 820M

Runs	Julia	Adjusted	GPU
1000	0.0178	0.0002	0.192
10000	0.0201	0.0025	0.196
100000	0.0439	0.0263	0.195
1000000	0.3061	0.2885	0.197
10000000	2.7565	2.7389	0.196



In [18]:

```
;cat OCL/vaxbyz.jl
```

```
import OpenCL
cl = OpenCL

const vaxbyz_kernel = "
__kernel void vaxbyz(
                    __global const float *a,
                    __global const float *b,
                    __global const float *x,
                    __global const float *y,
                    __global const float *z,
                    __global float *s)
    {
      int gid = get_global_id(0);
      s[gid] = a[gid]*x[gid]*x[gid] + b[gid]*y[gid] + z[gid];
    }
"

for n in (1, 1000, 50_000, 1_000_000, 5_000_000, 25_000_000, 50_000_000)

  a = rand(Float32, n);
  b = rand(Float32, n);
  x = rand(Float32, n);
  y = rand(Float32, n);
  z = rand(Float32, n);
  s = Array(Float32, n);

  device, ctx, queue = cl.create_compute_context();

  a_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=a);
  b_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=b);
  x_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=x);
  y_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=y);
  z_buff = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=z);
  s_buff = cl.Buffer(Float32, ctx, :w, n)

  p = cl.Program(ctx, source=vaxbyz_kernel) |> cl.build!
  k = cl.Kernel(p, "vaxbyz");

  cl.call(queue, k, size(x), nothing, a_buff, b_buff, x_buff, y_buff, z_buff, s
_buff);
  t0 = @elapsed s0 = cl.read(queue, s_buff);

  s1 = zeros(n);
  t1 = @elapsed begin
    for i = 1:length(x)
      s1[i] = a[i]*x[i]*x[i] + b[i]*y[i] + z[i]
    end
  end
  if n > 1
    @printf "%10d  : %8.5f  :  %8.5f\n" n t0 t1;
  else
    @printf "     Loops  :      GPU  :    Native\n";
  end
end
println();
```

In [19]:

```
run(`./vaxbyz.sh`)
```

```
    100000 : 0.21780 :  0.05390
    500000 : 0.21689 :  0.31798
   1000000 : 0.21892 :  0.52458
   5000000 : 0.27383 :  2.73475
  10000000 : 0.34168 :  5.90696
  25000000 : 0.53296 : 15.33412
```

JuliaQuant group :

- TimeSeries
- MarketData
- MarketTechnicals

- Quandl
- Ito
- GARCH

- FinancialAssets
- Grist
- TradeModels
- PortfolioModels

In [20]:

```
## FinancialSeries.jl api demo
using MarketData, PyPlot
```

In [21]:

```
#using FinancialSeries
include("./FS/FinancialSeries.jl")
```

In [22]:

```
# example stock from MarketData
AAPL.colnames
```

Out[22]:

```
12-element Array{UTF8String,1}:
 "Open"
 "High"
 "Low"
 "Close"
 "Volume"
 "Ex-Dividend"
 "Split Ratio"
 "Adj. Open"
 "Adj. High"
 "Adj. Low"
 "Adj. Close"
 "Adj. Volume"
```

In [23]:

```
# construct financial time series
Apple = TimeArray(AAPL.timestamp, AAPL.values, AAPL.colnames, FinancialSeries.
Stock(FinancialSeries.Ticker("AAPL")))
```

Out[23]:

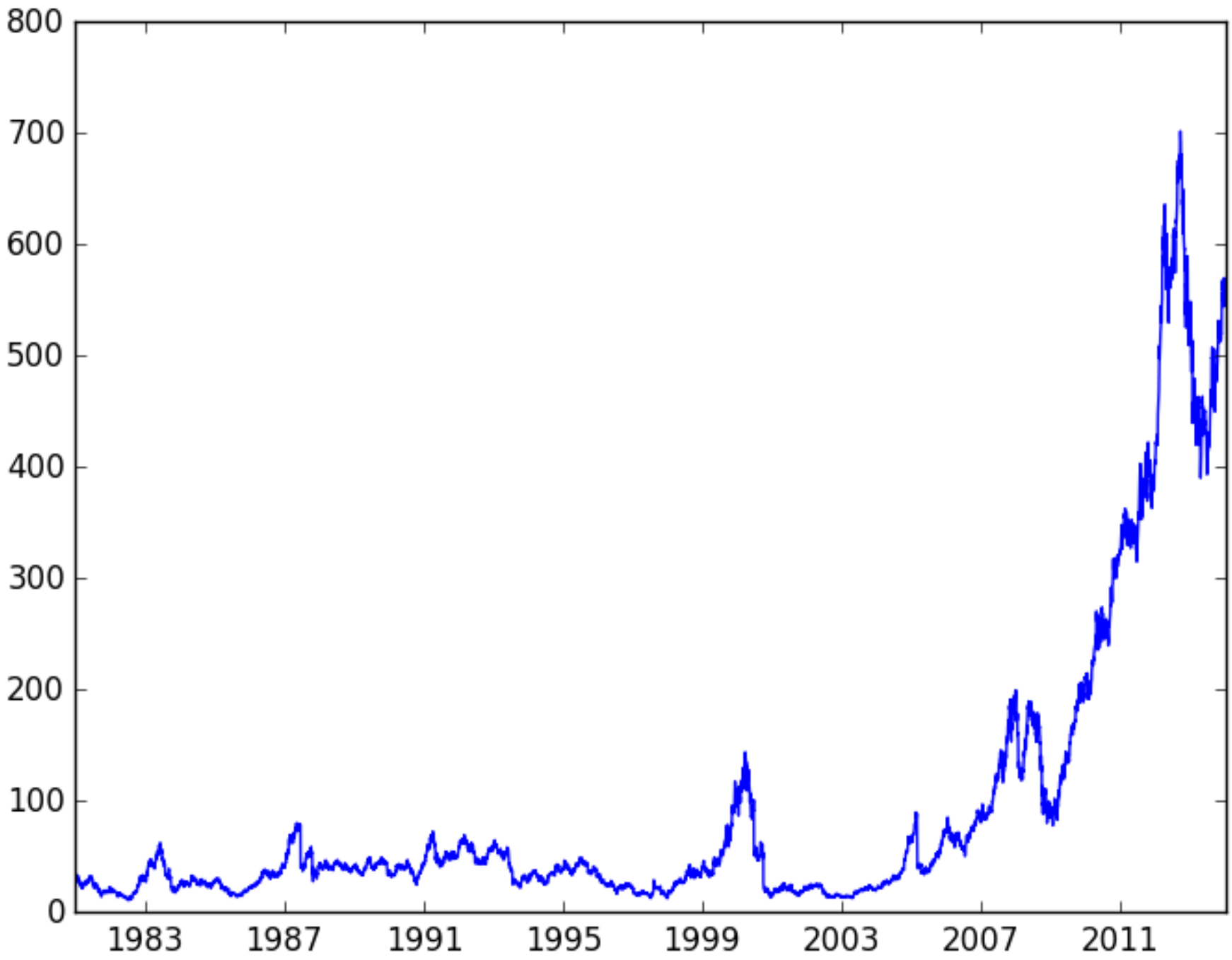
```
8336x12 FinancialTimeSeries for Stock, 1980-12-12 to 2013-12-31

ticker:          AAPL
currency:        USD
tick:            0.01
multiplier:      1.0

              Open    High    Low     Close   Volume        Ex-Dividend  Split R
atio  Adj. Open  Adj. High  Adj. Low  Adj. Close  Adj. Volume
1980-12-12 | 28.75   28.88   28.75   28.75   2093900       0.0          1
3.38       3.39       3.38       3.38       16751200
1980-12-15 | 27.38   27.38   27.25   27.25   785200        0.0          1
3.22       3.22       3.2        3.2        6281600
1980-12-16 | 25.38   25.38   25.25   25.25   472000        0.0          1
2.98       2.98       2.97       2.97       3776000
1980-12-17 | 25.88   26.0    25.88   25.88   385900        0.0          1
3.04       3.05       3.04       3.04       3087200
⋮
2013-12-26 | 568.1   569.5   563.38  563.9   7286000       0.0          1
564.74     566.13     560.05     560.56     7286000
2013-12-27 | 563.82  564.41  559.5   560.09  8067300       0.0          1
560.48     561.07     556.19     556.78     8067300
2013-12-30 | 557.46  560.09  552.32  554.52  9058200       0.0          1
554.16     556.78     549.05     551.24     9058200
2013-12-31 | 554.17  561.28  554.0   561.02  7967300       0.0          1
550.89     557.96     550.72     557.7      7967300
```

In [24]:

```
ac = Apple["Close"].values;
tm = timestamp(Apple);
plot(tm,ac)
```

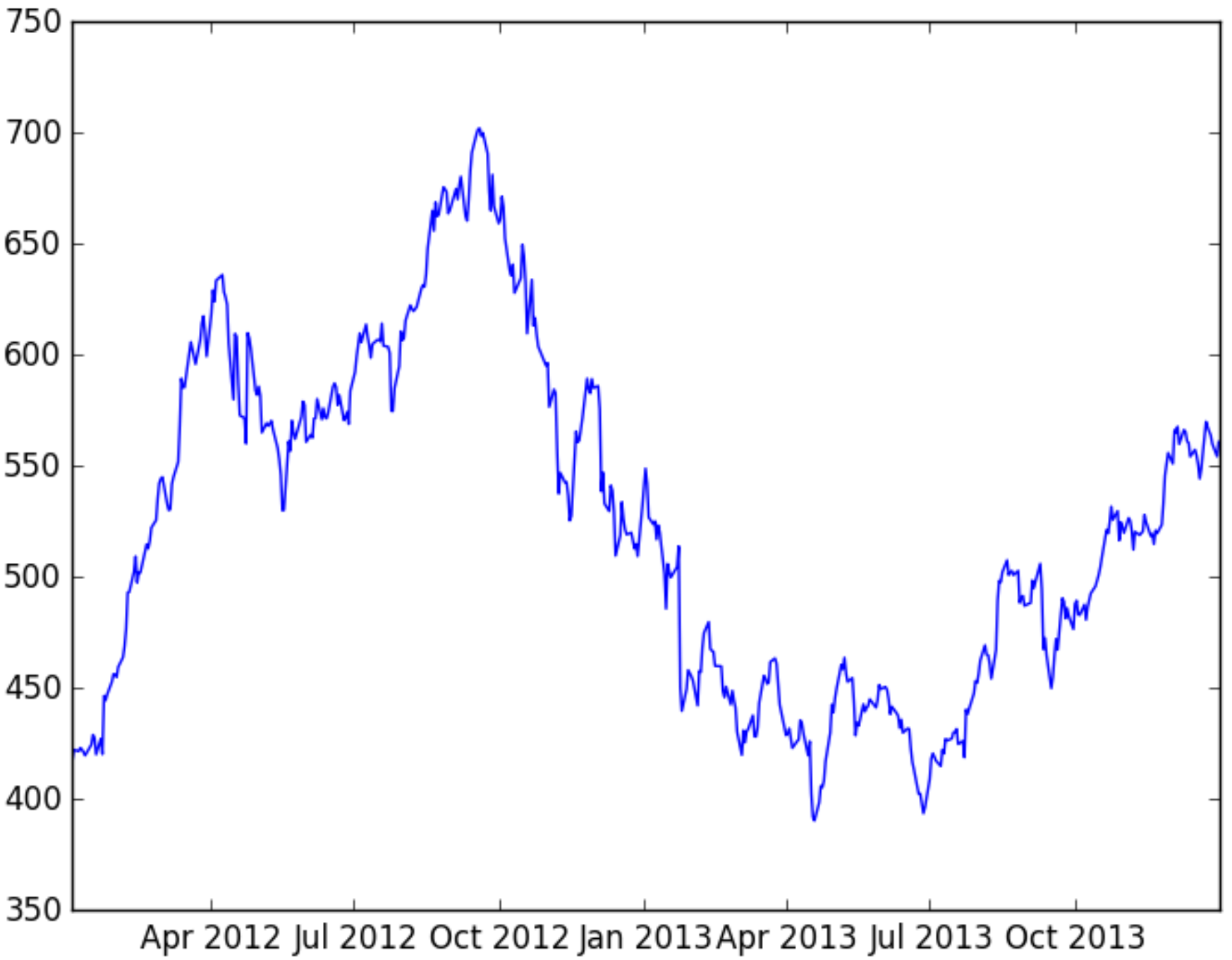


Out[24]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x1aee8b250>
```

In [25]:

```
plot(tm[end-500:end],ac[end-500:end])
```

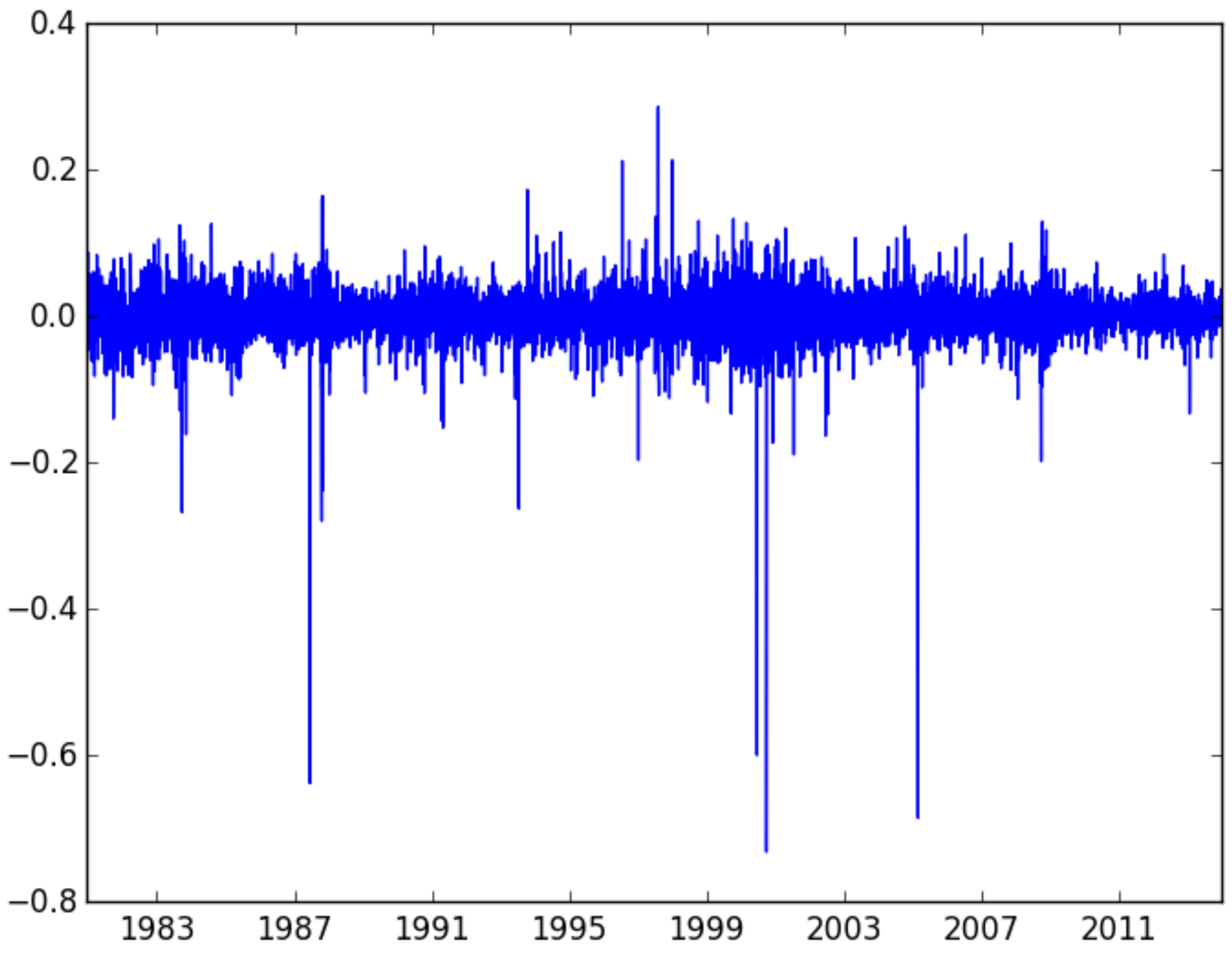


Out[25]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x1af19dad0>
```

In [26]:

```
pc = percentchange(Apple["Close"],method="log").values;
tm = timestamp(Apple)[2:end];
plot(tm,pc)
```



Out[26]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x1af4d9d10>
```

Generalized AutoRegressive Conditional Heteroskedasticity (GARCH) Process

Developed in 1982 to describe an approach to estimate volatility in financial markets.

The GARCH process is often preferred in financial modeling because it gives a more real-world context than other forms when predicting the prices and rates of financial instruments.

The general process for a GARCH model involves three steps:

- Estimate a best-fitting autoregressive model
- Compute autocorrelations of the error term
- Test for significance

In [2]:

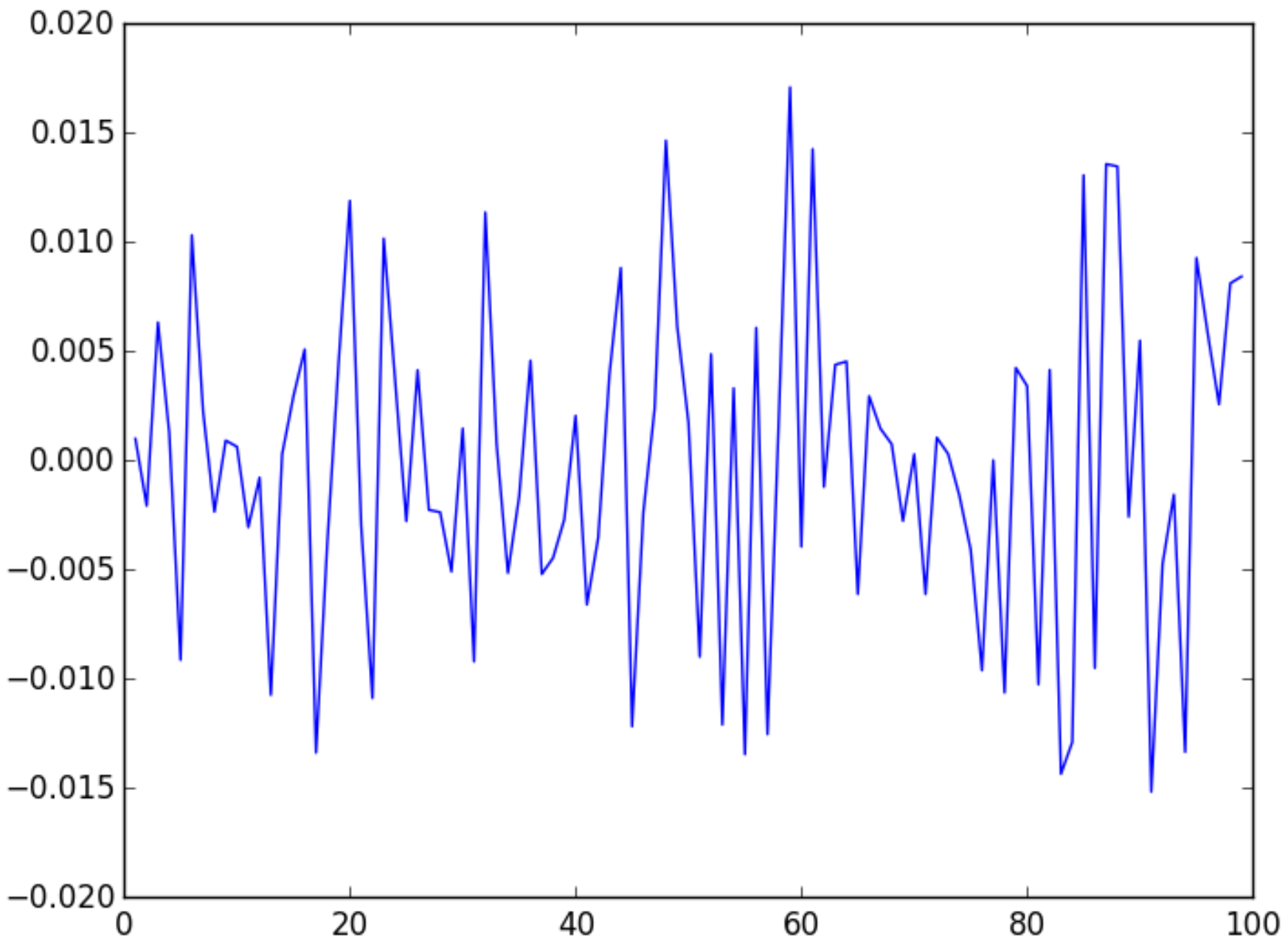
```
using Quandl
quotes = quandl("YAHOO/INDEX_GSPC", format="DataFrame")
```

Out[2]:

	Date	Open	High	Low	Close	Volume	Adjusted_Close
1	2015-06-02	2110.40991	2117.59009	2099.13989	2109.6001	3.04935e9	2109.6001
2	2015-06-01	2108.63989	2119.1499	2102.54004	2111.72998	3.01171e9	2111.72998
3	2015-05-29	2120.65991	2120.65991	2104.88989	2107.38989	3.92739e9	2107.38989
4	2015-05-28	2122.27002	2122.27002	2112.86011	2120.79004	2.98035e9	2120.79004
5	2015-05-27	2105.12988	2126.21997	2105.12988	2123.47998	3.12796e9	2123.47998
6	2015-05-26	2125.34009	2125.34009	2099.17993	2104.19995	3.34213e9	2104.19995
7	2015-05-22	2130.36011	2132.1499	2126.06006	2126.06006	2.57186e9	2126.06006
8	2015-05-21	2125.55005	2134.28003	2122.94995	2130.82007	3.07046e9	2130.82007
9	2015-05-20	2127.79004	2134.71997	2122.59009	2125.8501	3.02588e9	2125.8501
10	2015-05-19	2129.44995	2133.02002	2124.5	2127.83008	3.29603e9	2127.83008
11	2015-05-18	2121.30005	2131.78003	2120.01001	2129.19995	2.88819e9	2129.19995
12	2015-05-15	2122.07007	2123.88989	2116.81006	2122.72998	3.09208e9	2122.72998
13	2015-05-14	2100.42993	2121.44995	2100.42993	2121.1001	3.22574e9	2121.1001
14	2015-05-13	2099.62012	2110.18994	2096.04004	2098.47998	3.37426e9	2098.47998
15	2015-05-12	2102.87012	2105.06006	2085.57007	2099.12012	0.0	2099.12012
16	2015-05-11	2115.56006	2117.68994	2104.58008	2105.33008	2.99267e9	2105.33008
17	2015-05-08	2092.12988	2117.65991	2092.12988	2116.1001	3.39944e9	2116.1001
18	2015-05-07	2079.95996	2092.8999	2074.98999	2088.0	3.67664e9	2088.0
19	2015-05-06	2091.26001	2098.41992	2067.92993	2080.1499	3.79221e9	2080.1499
20	2015-05-05	2112.62988	2115.23999	2088.45996	2089.45996	3.79395e9	2089.45996
21	2015-05-04	2110.22998	2120.94995	2110.22998	2114.48999	3.09158e9	2114.48999
22	2015-05-01	2087.37988	2108.40991	2087.37988	2108.29004	3.37939e9	2108.29004
23	2015-04-30	2105.52002	2105.52002	2077.59009	2085.51001	4.50968e9	2085.51001
24	2015-04-29	2112.48999	2113.6499	2097.40991	2106.8501	4.07497e9	2106.8501
25	2015-04-28	2108.3501	2116.04004	2094.88989	2114.76001	3.54627e9	2114.76001
26	2015-04-27	2119.29004	2125.91992	2107.04004	2108.91992	3.43875e9	2108.91992
27	2015-04-24	2112.80005	2120.91992	2112.80005	2117.68994	3.37578e9	2117.68994
28	2015-04-23	2107.20996	2120.48999	2103.18994	2112.92993	3.63667e9	2112.92993
29	2015-04-22	2098.27002	2109.97998	2091.05005	2107.95996	3.34848e9	2107.95996
30	2015-04-21	2102.82007	2109.63989	2094.37988	2097.29004	3.24341e9	2097.29004
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [3]:

```
using PyPlot
qac = diff(log(array(quotes[:Adjusted_Close])))
plot(1:length(qac),qac)
```



Out[3]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x11a0c3c90>
```

In [4]:

```
using GARCH
garchFit(qac)
```

Out[4]:

```
Fitted garch model
 * Coefficient(s):      omega           alpha           beta
                        0.000000        0.014263        0.990000
 * Log Likelihood: 347.223248
 * Converged: false
 * Solver status: XTOL_REACHED

 * Standardised Residuals Tests:
                                Statistic       p-Value
   Jarque-Bera Test     χ²      0.839373        0.657253

 * Error Analysis:
              Estimate        Std.Error       t value         Pr(>|t|)
   omega      0.000000        0.000003        0.005231        0.995826
   alpha      0.014263        0.012117        1.177084        0.239162
   beta       0.990000        0.079091        12.517164       0.000000
```

In Summary

- Coding in Julia is simple
- At has (even now) impressive functionality
- This can be extended by using C/Fortran/Python/R/Java modules and libraries
- Julia as implemented data frames, similar to those in R and Python/Pandas
- Community groups exist and are active in statistics, finance, econometrics etc.
- JuliaQuant already covers many of the principal financial requirements