ARCH Models in Julia

Simon A. Broda

University of Zurich and University of Amsterdam
simon.broda@uzh.ch

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 750559).

Following Along

- These slides are available at https://github.com/s-broda/brownbag2018, in the form of a Jupyter notebook.
- Jupyter notebooks contain live code. Cells are evaluated by pressing `shift-enter`.
- You can follow along without installing Julia locally by running the notebook on Google Colab (https://colab.research.google.com/; requires a Google account).
- Julia is not officially supported on Colab yet, so we require a trick to get it to work: click on `file -> New Python 2 notebook`, paste the following into the new notebook, and execute the cell with `shift-enter`:

```
!curl -sSL "https://julialang-s3.julialang.org/bin/linux/x64/1.0/julia-1.0.1-linux-x86_64.tar.gz"\
  -o julia.tar.gz
!tar -xzf julia.tar.gz -C /usr --strip-components 1
!rm -rf julia.tar.gz*
!julia -e 'using Pkg; pkg"add IJulia; precompile"'
```

- Wait for the code to execute (~ 1min), then click on `File -> Upload notebook`, choose `Github`, paste `https://github.com/s-broda/brownbag2018` into the search field, hit enter, and click on the filename once Colab has found the notebook. Optionally, click `Copy to Drive`.
- *Important*: If you see a warning when attempting to run code, uncheck *Reset all runtimes before running* before clicking `Run anyway`, or the above procedure will need to be repeated.

Outline

- The Julia Language
- Refresher on ARCH Models
- The `ARCH` Package
    - Usage
    - Benchmarks vs. Matlab

The Julia Language

General Information

- New programming language started at MIT.
- Designed with scientific computing in mind.
- Version 1.0 released in August 2018 after 9 years of development.
- Free and open source software. Available at https://julialang.org/.
- In the words of its creators,

> *We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.*

Highlights for Scientists

- Interactive REPL (like Matlab, Python, R, etc.) allows for exploratoty analysis, rapid prototyping.
- Unlike these, Julia is JIT compiled, hence fast (typically within 2x of C)
- Syntax superficially similar to Matlab.
- Rich type system, multiple dispatch.
- Fast-growing eco-system with many state-of-the-art packages (e.g., `ForwardDiff.jl`, `DifferentialEquations.jl`, `JuMP`, ...).

A Small Taste of Julia

- Let's say we want to implement a function that sums an array:

In [2]:
```
function mysum(x)
    s = zero(eltype(x))
    for i in x
        s += i
    end
    s
end
mysum([1, 2, 3])
```

Out[2]: 6

In [3]:
```
mysum([1., 2., 3.]) # a new method is compiled every time `mysum` is called with a new type
```

Out[3]: 6.0

- Let's benchmark it:

In [4]:
```
x = randn(10^7);
```

In [5]:
```
using BenchmarkTools
@btime mysum(x); #@btime runs the code a large number of times
```

```
  11.617 ms (1 allocation: 16 bytes)
```

- For comparison, the built-in function:

In [6]:
```
@btime sum(x);
```

```
  4.395 ms (1 allocation: 16 bytes)
```

- Close, but we are about 2x slower.
- But we can do better! 2nd attempt:

In [7]:
```
function mysum(x)
    s = zero(eltype(x))
    @simd for i in x
        s += i
    end
    s
end
mysum([1., 2., 3.])
```

Out[7]:
```
6.0
```

In [8]:
```
@btime mysum(x);
```

```
  4.451 ms (1 allocation: 16 bytes)
```

- Not bad: by just adding a simple decorator, we now match the speed of the built-in function!
- This is, in fact, expected: the built-in function is implemented in Julia, like most of the standard library.
- We can even look at the code:

In [9]:
```
@which sum(x)
```

Out[9]: sum(a::**AbstractArray**) in Base at reducedim.jl:645

Refresher on ARCH Models

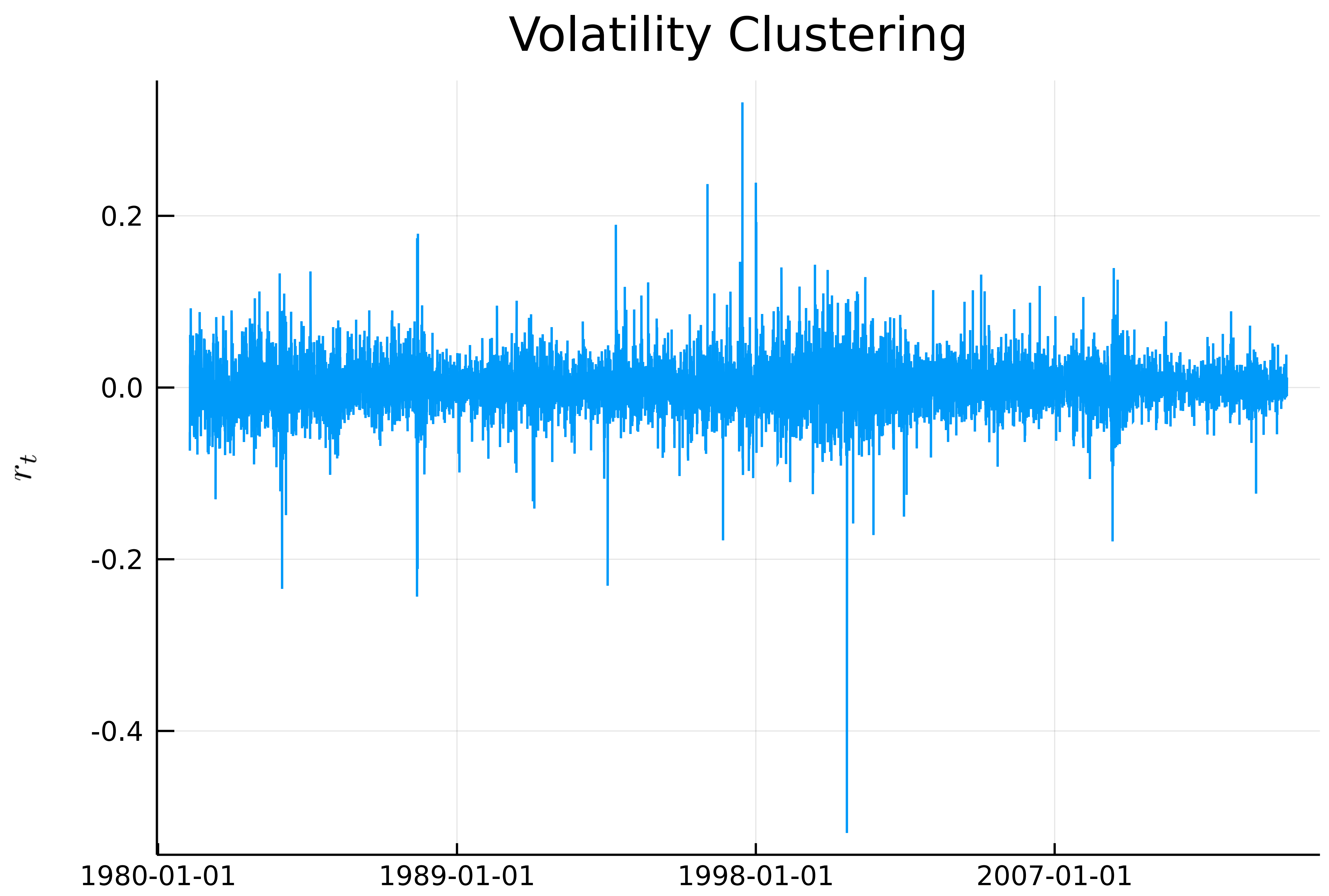
- Daily financial returns data exhibit a number of *stylized facts*:
    - Volatility clustering
    - Non-Gaussianity, fat tails
    - Leverage effects: negative returns increase future volatility
- Other types of data (e.g., changes in interest rates) exhibit similar phenomena.
- These effects are important in many areas in finance, in particular in risk management.
- [G]ARCH ([G*eneralized] *Autoregressive C*onditional *Volatility) models are the most popular for modelling them.

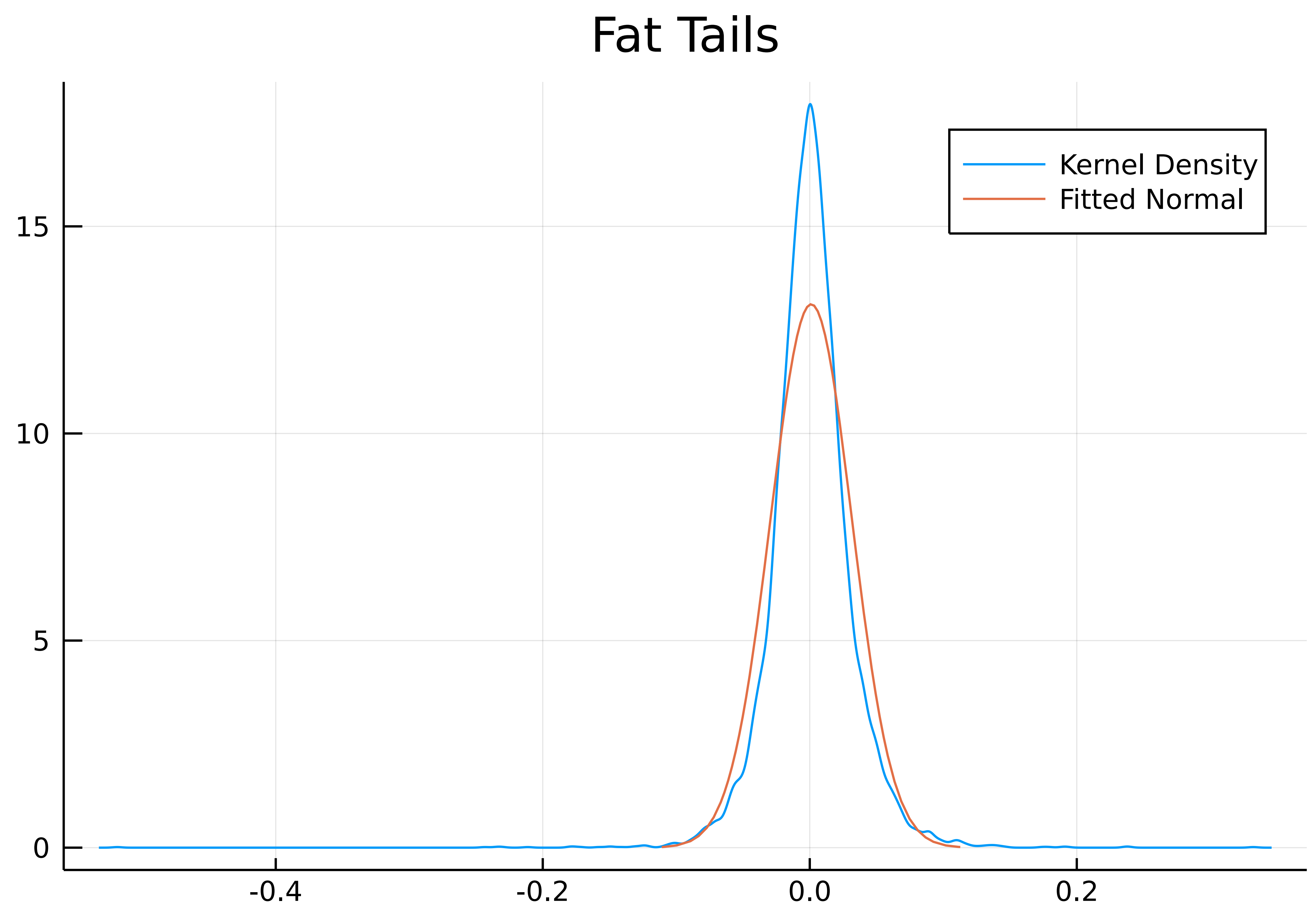
Example: volatility clustering in AAPL returns

- The `MarketData` package contains several historial datasets.

In [10]:
```
using MarketData, TimeSeries
r = percentchange(MarketData.AAPL[Symbol("Adj. Close")]) # returns a TimeSeries
data = values(r) # an array containing just the plain data

if !isfile("returns.svg") || !isfile("kde.svg")
    using Plots, Distributions, KernelDensity, StatPlots
    plot(r, title="Volatility Clustering", legend=:none, ylabel="\$r_t\$")
    savefig("returns.svg")
    plot(kde(data), label="Kernel Density", title="Fat Tails")
    plot!(fit(Normal, data), label="Fitted Normal")
    savefig("kde.svg")
end
```


Volatility Clustering
0.2
0.0
-0.2
-0.4
r_t
1980-01-01
1989-01-01
1998-01-01
2007-01-01



Fat Tails
Kernel Density
Fitted Normal
15
10
5
0
-0.4
-0.2
0.0
0.2

(G)ARCH Models

- Basic setup: given a sample of financial returns $\{r_t\}_{t\in\{1,\ldots,T\}}$, decompose r_t as
$$r_t = \mu_t + \sigma_t z_t, \qquad z_t \overset{i.i.d.}{\sim} (0,1),$$
where $\mu_t \equiv \mathbb{E}[r_t \mid \mathcal{F}_{t-1}]$ and $\sigma_t^2 \equiv \mathbb{E}[(r_t - \mu_t)^2 \mid \mathcal{F}_{t-1}]$.
- Assume $\mu_t = 0$ for simplicity. Focus is on the *volatility* σ_t. G(ARCH) models make σ_t a function of *past* returns and variances. Examples:

Examples

- ARCH(q) (Engle, Econometrica 1982):

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i r_{t-i}^2$$

- GARCH(p, q) (Bollerslev, JoE 1986)

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i r_{t-i}^2$$

- EGARCH(o, p, q) (Nelson, Econometrica 1991)

$$\log(\sigma_t^2) = \omega + \sum_{i=1}^{o} \gamma_i z_{t-i} + \sum_{i=1}^{p} \beta_i \log(\sigma_{t-i}^2) + \sum_{i=1}^{q} \alpha_i (|z_t| - \mathbb{E}|z_t|)$$

Estimation

- G(ARCH) models are usually estimated by maximum likelihood: with f_z denoting the density of z_t,

$$\max \sum_t \log f(r_t \mid \mathcal{F}_{t-1}) = \max \sum_t \log f_z(r_t/\sigma_t) - \log \sigma_t.$$

- Recursive nature of σ_t means the computation cannot be "vectorized" $\Rightarrow$ loops.
- Julia is very well suited for this. Matlab (and the `rugarch` package for Python) have to implement the likelihood in C.

The `ARCH` Package

Installation

- `ARCH.jl` is available at https://github.com/s-broda/ARCH.jl.
- Extensive documentation available at https://s-broda.github.io/ARCH.jl/dev/.
- `ARCH.jl` is not a registered Julia package yet. To install it in Julia 1.0 or later, do

In [11]:
```julia
if !haskey(Pkg.installed(), "ARCH")
    Pkg.add(PackageSpec(url="https://github.com/s-broda/ARCH.jl"))
end
```

Key Features

- Supports simulating, estimating, forecasting, and backtesting ARCH models.
- Currently: ARCH, GARCH, TGARCH, and EGARCH models of arbitrary orders, with Gaussian, Student's t, and GED errors.
- Entirely written in Julia.
- Designed to be easily extensible with new models and distributions.
- Gradients and Hessians (for both numerical maximization of the likelihood and constructing standard errors) are obtained by automatic differentiation via `ForwardDiff.jl`.

Usage

- The first step in building an ARCH model is usually to test for the presence of volatility clustering.
- `ARCH.jl` provides Engle's (Econometrica 1982) ARCH-LM test for this.
- The null is that $\gamma_i = 0$ in the auxiliary regression
$$r_t^2 = \alpha + \gamma_1 r_{t-1}^2 + \gamma_2 r_{t-2}^2 + \cdots + \gamma_p r_{t-p}^2 + \epsilon_t.$$

In [12]:
```
using ARCH
ARCHLMTest(data, 4) # p=4 lags
```

Out[12]:
```
ARCH LM test for conditional heteroskedasticity
-----------------------------------------------
Population details:
    parameter of interest:   T⋅R² in auxiliary regression of rₜ² on an intercept and its own lags
    value under h_0:         0
    point estimate:          102.70507547678012

Test summary:
    outcome with 95% confidence: reject h_0
    p-value:                     <1e-20

Details:
    sample size:                    8335
    number of lags:                 4
    LM statistic:                   102.70507547678012
```

- Unsurprisingly, the test rejects.

- Fitting a GARCH model is as simple as

In [13]:
```
model = fit(GARCH{1, 1}, data)
```

Out[13]:
```
GARCH{1,1} model with Gaussian errors, T=8335.


Mean equation parameters:

       Estimate   Std.Error z value Pr(>|z|)
μ    0.00188447 0.000310897 6.06138    <1e-8

Volatility parameters:

       Estimate  Std.Error  z value Pr(>|z|)
ω    9.75917e-6 7.20954e-6 1.35365   0.1758
β₁     0.929442  0.0309706 30.0104   <1e-99
α₁    0.0636943  0.0268269 2.37427   0.0176
```

- Alternatively, the (multithreaded!) `selectmodel` method does automatic model selection (i.e., chooses p and q by minimizing the AIC/AICC/BICC).

In [14]:
```
model2 = selectmodel(GARCH, data; maxlags=3, criterion=bic) # optional keyword arguments
```

Out[14]:
```
GARCH{2,1} model with Gaussian errors, T=8335.


Mean equation parameters:

       Estimate   Std.Error z value Pr(>|z|)
μ    0.00184854 0.000305871 6.04354    <1e-8

Volatility parameters:

       Estimate  Std.Error  z value Pr(>|z|)
ω    1.38392e-5 9.53439e-6 1.45151   0.1466
β₁     0.379911   0.106348 3.57232   0.0004
β₂     0.516406   0.103062 5.01062    <1e-6
α₁    0.0940149  0.0359409 2.61582   0.0089
```

- The return value is of type `ARCHModel`:

In [15]:
```
typeof(model)
```
Out[15]:
```
ARCHModel{Float64,GARCH{1,1,Float64},StdNormal{Float64},Intercept{Float64}}
```

- `ARCHModel` supports many useful methods, such as `confint`, `aic`, `bic`, `aicc`, `informationmatrix`, `score`, `vcov`:

In [16]:
```
aic(model)
```
Out[16]:
```
-35866.1305913175
```

- An ARCHModel essentially consists of a volatility specification (of type `VolatilitySpec`), an error distribution (of type `StandardizedDistribution`), and a mean specification (of type `MeanSpec`).
- The following are currently implemented:

In [17]:
```
print.(string.(subtypes(VolatilitySpec)) .* " ");
```
```
EGARCH GARCH TGARCH
```

In [18]:
```
print.(string.(subtypes(StandardizedDistribution))[2:end] .* " ");
```
```
StdGED StdNormal StdT
```

In [19]:
```
print.(string.(subtypes(MeanSpec)).* " ");
```
```
Intercept NoIntercept
```

- We can use these to construct an ARCHModel manually:

In [20]:
```
T = 10^4
mydata = zeros(T)
volaspec = EGARCH{1, 1, 1}([0.02, .09, .83, .01])
using Random; Random.seed!(1)
mymodel = ARCHModel(volaspec, mydata; dist=StdT(4.)) # dist is an optional keyword argument
```

Out[20]:
```
EGARCH{1,1,1} model with Student's t errors, T=10000.


                              ω    γ₁    β₁    α₁
Volatility parameters:     0.02 0.09 0.83 0.01

                             ν
Distribution parameters:   4.0
```

- With an `ARCHModel` in hand, we can do things like

In [21]:
```julia
simulate!(mymodel) # by convention, the bang indicates that the method modifies its argument
```

Out[21]:
```
EGARCH{1,1,1} model with Student's t errors, T=10000.


                              ω    γ₁   β₁   α₁
Volatility parameters:     0.02 0.09 0.83 0.01

                             ν
Distribution parameters:   4.0
```

In [22]:
```julia
fit!(mymodel)
```

Out[22]:
```
EGARCH{1,1,1} model with Student's t errors, T=10000.


Volatility parameters:

        Estimate  Std.Error  z value Pr(>|z|)
ω      0.0274054 0.00733817  3.73464   0.0002
γ₁      0.113959  0.0131327  8.67753   <1e-17
β₁      0.811765  0.0244774  33.1638   <1e-99
α₁   -0.00637894  0.0153358 -0.41595   0.6774

Distribution parameters:

     Estimate Std.Error z value Pr(>|z|)
ν     3.83768  0.152659  25.139   <1e-99
```

In [23]:
```
ARCHLMTest(mymodel) # tests the standardized residuals
```

Out[23]:
```
ARCH LM test for conditional heteroskedasticity
-----------------------------------------------
Population details:
    parameter of interest:   T⋅R² in auxiliary regression of rₜ² on an intercept and its own lags
    value under h_0:         0
    point estimate:          0.08739963833992448

Test summary:
    outcome with 95% confidence: fail to reject h_0
    p-value:                     0.7675

Details:
    sample size:                    10000
    number of lags:                 1
    LM statistic:                   0.08739963833992448
```

In [24]:
```
predict(mymodel, :volatility) # (:volatility | :variance | :return | :VaR)
```

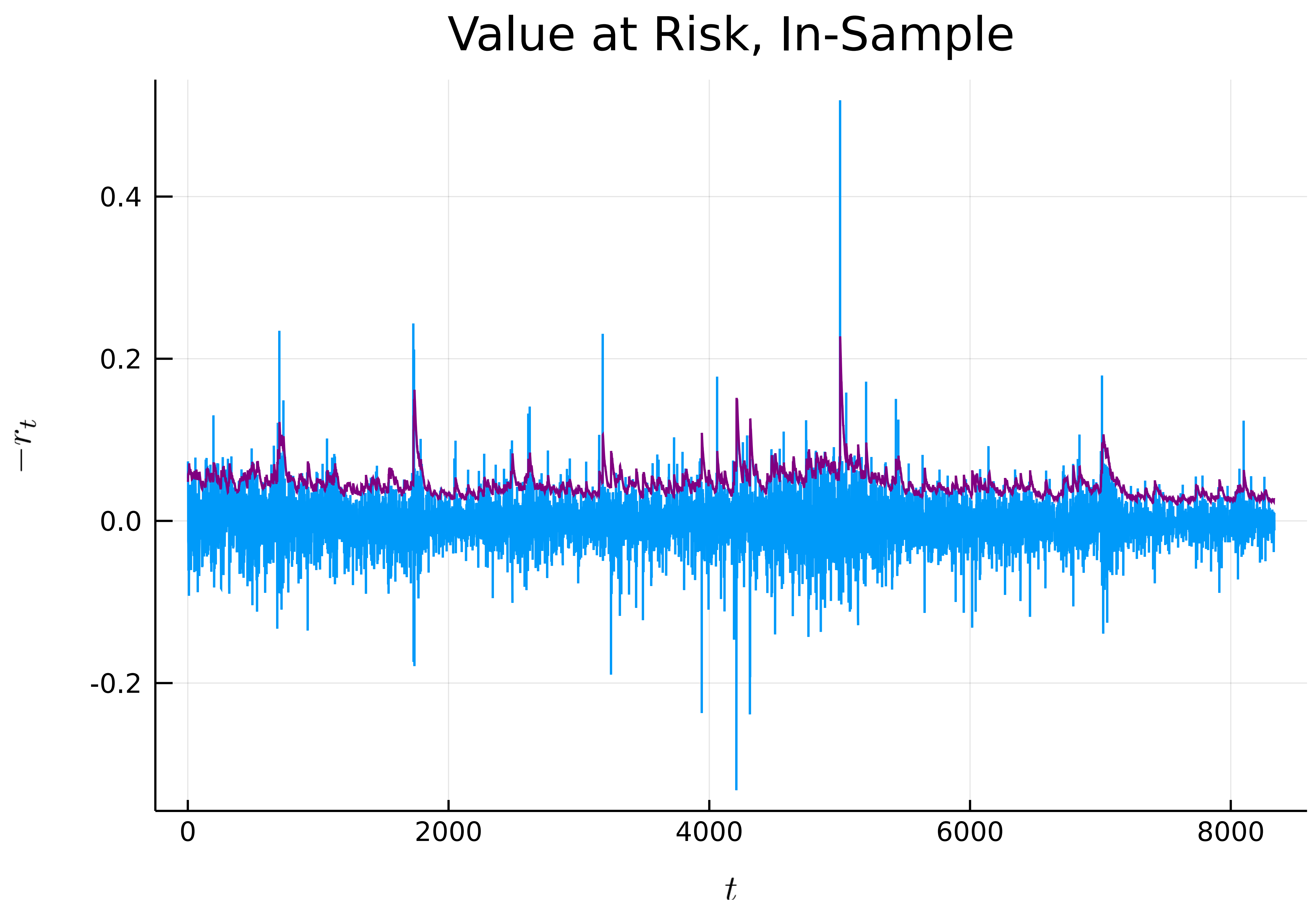
Out[24]:
```
0.958298864614992
```

Value at Risk

- In-sample Value-at-Risk estimates are available via the `VaRs` function

In [25]:
```
model = fit(GARCH{1, 1}, data)
vars = VaRs(model, 0.05)

if !isfile("VaRplot.svg")
    using Plots
    plot(-data, legend=:none, xlabel="\$t\$", ylabel="\$-r_t\$", title="Value at Risk, In-Sample")
    plot!(vars, color=:purple)
    savefig("VaRplot.svg");
end
```



Value at Risk, In-Sample
0.4
0.2
0.0
-0.2
$-r_t$
0
2000
4000
6000
8000
t

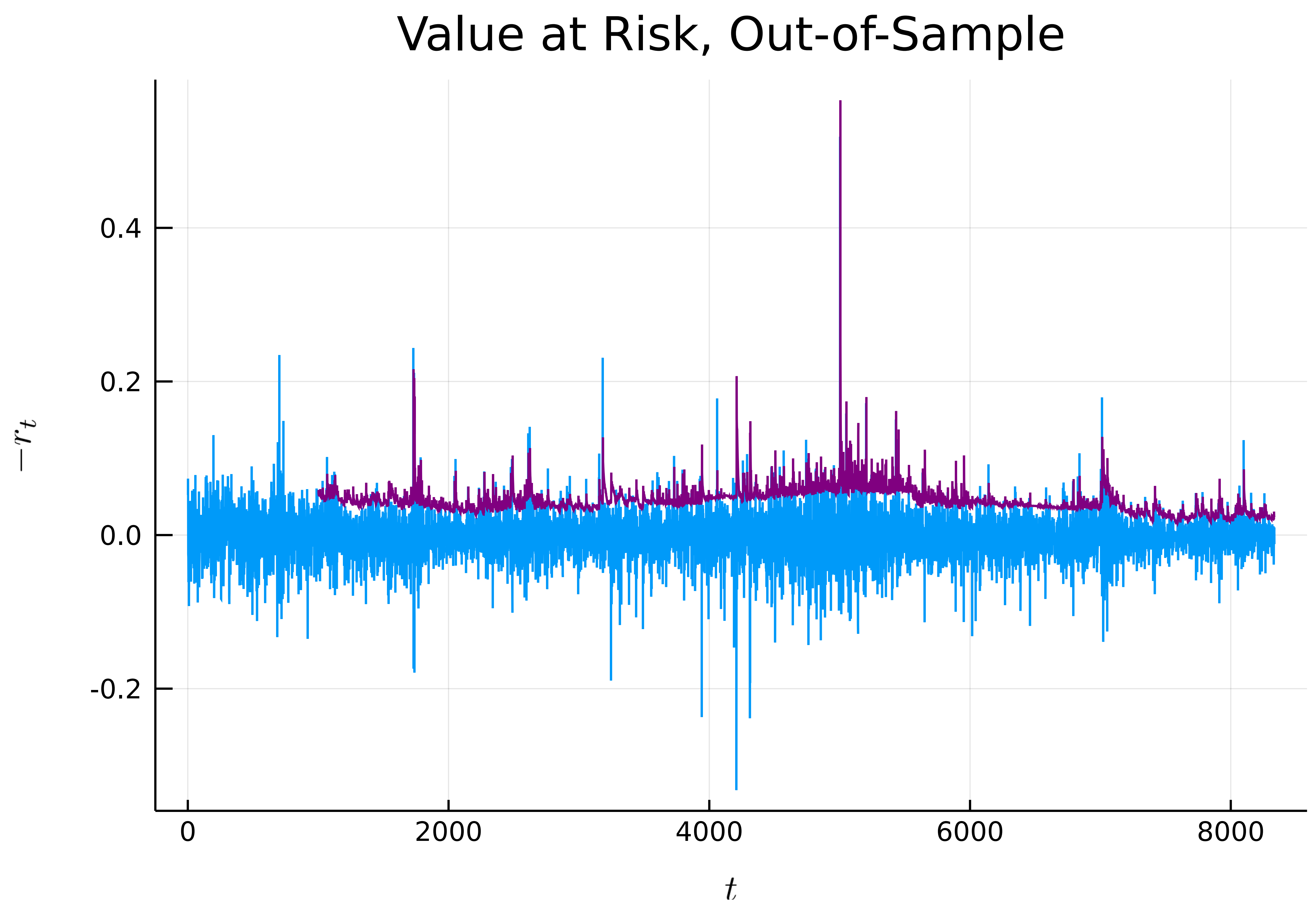
- How about out-of-sample (backtesting)?
- Requires re-estimating the model at each time step.
- Not built in, but easy to do:

In [26]:
```
using ProgressMeter

T = length(data)
windowsize = 1000
vars = similar(data); fill!(vars, NaN)
@showprogress "Fitting $(T-1-windowsize) GARCH models: " for t = windowsize+1:T-1
    m = fit(GARCH{1, 1}, data[t-windowsize:t])
    vars[t+1] = predict(m, :VaR; level=0.05)
end

if !isfile("VaRplot_oos.svg")
    using Plots
    plot(-data, legend=:none, xlabel="\$t\$", ylabel="\$-r_t\$", title="Value at Risk, Out-of-Sample")
    plot!(vars, color=:purple)
    savefig("VaRplot_oos.svg");
end
```

```
Fitting 7334 GARCH models: 100%|██████████████████████████| Time: 0:00:20
```


Value at Risk, Out-of-Sample
$-r_t$
0.4
0.2
0.0
-0.2
0
2000
4000
6000
8000
t

- We can backtest these out-of-sample VaR predictions using Engle and Manganelli's (2004) dynamic quantile test.
- The test is based on a *hit series*

$$I_t \equiv \begin{cases} 1, r_t < -VaR_t \\ 0, \text{otherwise.} \end{cases}$$

- The null is that $\alpha = \beta = \gamma = 0$ in the auxiliary regression

$$I_t - 0.05 = \alpha + \beta I_{t-1} + \gamma VaR_t + \epsilon_t.$$

In [27]:
```
DQTest(data[windowsize+1:end], vars[windowsize+1:end], 0.05)
```

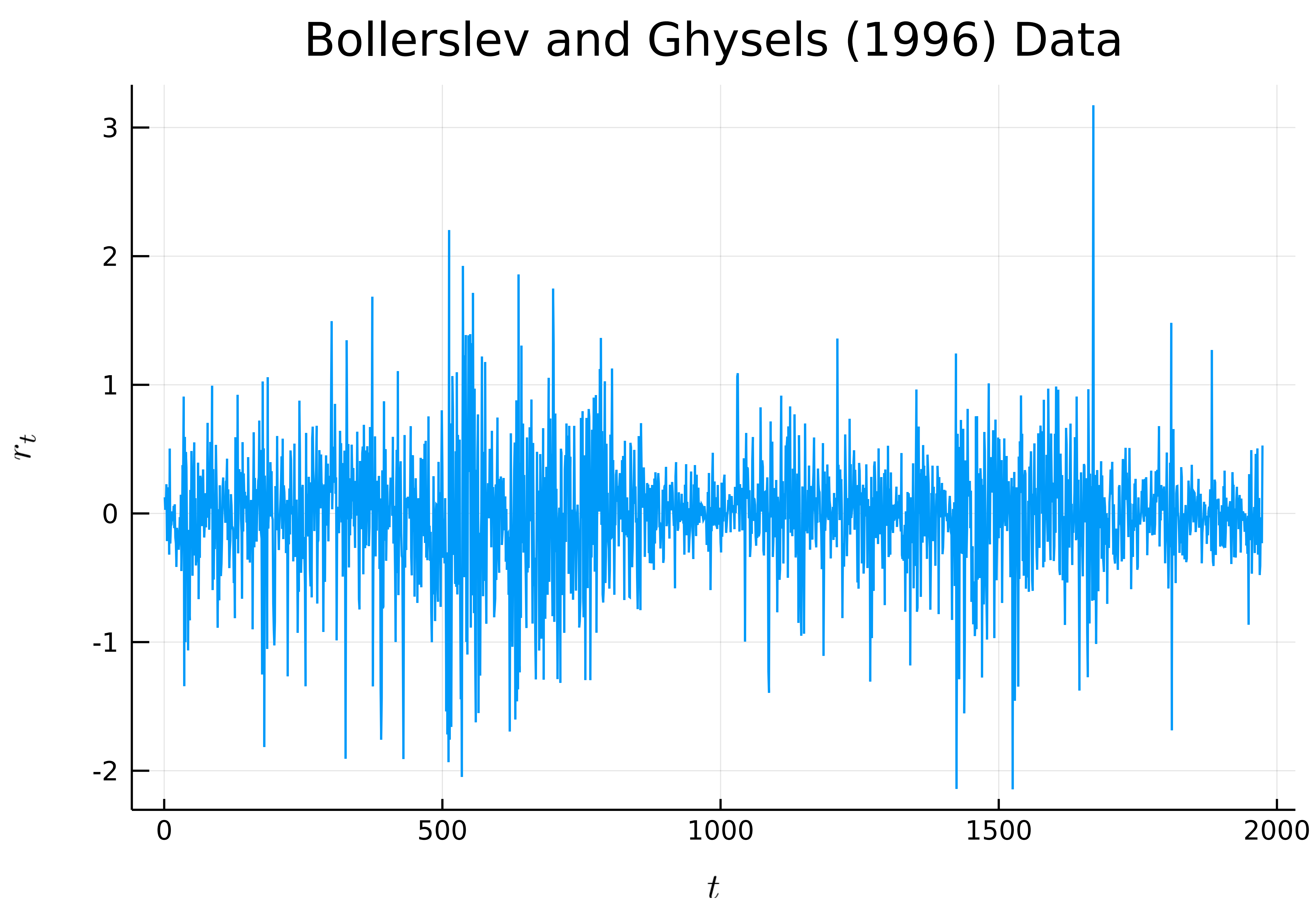
Out[27]:
```
Engle and Manganelli's (2004) DQ test (out of sample)
-----------------------------------------------------
Population details:
    parameter of interest:   Wald statistic in auxiliary regression
    value under h_0:         0
    point estimate:          53.71639796837963

Test summary:
    outcome with 95% confidence: reject h_0
    p-value:                     <1e-10

Details:
    sample size:                    7335
    number of lags:                 1
    VaR level:                      0.05
    DQ statistic:                   53.71639796837963
```

Benchmarks

- Bollerslev and Ghysels (JBES 1996) data is de facto standard in comparing implementations of GARCH models.
- Data consist of daily German mark/British pound exchange rates (1974 observations).
- Available in `ARCH.jl` as the constant `BG96`.


Bollerslev and Ghysels (1996) Data
r_t
t
3
2
1
0
-1
-2
0
500
1000
1500
2000

GARCH

- Fitting in Julia:

In [29]:
```
@btime fit(GARCH{1, 1}, $BG96, meanspec=NoIntercept)  # Matlab doesn't use an intercept, so let's not, either
```

```
  2.736 ms (1720 allocations: 344.88 KiB)
```

Out[29]:
```
GARCH{1,1} model with Gaussian errors, T=1974.


Volatility parameters:

      Estimate  Std.Error z value Pr(>|z|)
ω    0.0108661 0.00657449 1.65277   0.0984
β₁    0.804431  0.0730395 11.0136   <1e-27
α₁    0.154597  0.0539319 2.86651   0.0042
```

- Now Matlab:

In [30]:
```
using MATLAB
mat"version"
```

Out[30]:
```
"9.4.0.813654 (R2018a)"
```

In [50]:
```
# run this cell a few times to give Matlab a fair chance
mat"tic; estimate(garch(1, 1), $BG96); toc; 0";
```

```
    GARCH(1,1) Conditional Variance Model (Gaussian Distribution):

                  Value      StandardError    TStatistic      PValue
                 ________    _____________    __________    __________

    Constant     0.010868      0.0012972        8.3779      5.3896e-17
    GARCH{1}      0.80452       0.016038        50.162               0
    ARCH{1}       0.15433       0.013852        11.141      7.9448e-29

Elapsed time is 0.070031 seconds.
```

- ARCH.jl is faster by a factor of about 20-30, depending on the machine, despite Matlab calling into compiled C code.
- Estimates are quite similar, but standard errors and t-statistics differ.
- So which standard errors are correct? Let's compare with the results from Brooks et. al. (Int. J. Fcst. 2001).

- Brooks et. al. compare implementations of the GARCH(1, 1) model. They use a model with intercept, so let’s re-estimate in Julia (Matlab doesn't seem to allow this):

In [32]:
```
fit(GARCH{1, 1}, BG96)
```

Out[32]:
```
GARCH{1,1} model with Gaussian errors, T=1974.


Mean equation parameters:

        Estimate  Std.Error   z value Pr(>|z|)
μ    -0.00616637 0.00920163 -0.670139   0.5028

Volatility parameters:

      Estimate  Std.Error z value Pr(>|z|)
ω    0.0107606 0.00649493 1.65677   0.0976
β₁    0.805875  0.0725003 11.1155   <1e-27
α₁    0.153411  0.0536586 2.85903   0.0042
```

- Brooks et. al. give the estimates (t-**stats**) $\mu = -0.00619\,(\mathbf{-0.67})$, $\omega = 0.0108\,(\mathbf{1.66})$, $\beta_1 = 0.806\,(\mathbf{11.11})$, $\alpha_1 = 0.153\,(\mathbf{2.86})$. Dead on!

EGARCH

- Julia:

In [33]:
```
@btime fit(EGARCH{1, 1, 1}, $BG96, meanspec=NoIntercept)
```

```
  4.670 ms (2030 allocations: 414.27 KiB)
```

Out[33]:
```
EGARCH{1,1,1} model with Gaussian errors, T=1974.


Volatility parameters:

      Estimate Std.Error  z value Pr(>|z|)
ω    -0.128026 0.0518431 -2.46948   0.0135
γ₁   -0.032216 0.0255372 -1.26153   0.2071
β₁    0.911947 0.0331381  27.5196   <1e-99
α₁    0.333243  0.070109  4.75321    <1e-5
```

- Matlab:

```
In [54]: mat"tic; estimate(egarch(1, 1), $BG96); toc; 0";  # Matlab sets o=q
```

```
    EGARCH(1,1) Conditional Variance Model (Gaussian Distribution):

                     Value      StandardError    TStatistic      PValue
                   _________    _____________    __________    __________

    Constant         -0.1283       0.015788       -8.1267      4.4118e-16
    GARCH{1}         0.91186      0.0084535        107.87               0
    ARCH{1}          0.33317       0.021769        15.305      7.1324e-53
    Leverage{1}    -0.032252       0.012564        -2.567        0.010258

Elapsed time is 0.109377 seconds.
```

- Brooks et. al. give no benchmark results. But again, Julia is faster by a factor of about 20.

TODO

- More distributions (NIG, α-Stable, ...)
- More GARCH models (APARCH, RiskMetrics, IGARCH, ...)
- Multivariate GARCH

References

- Bollerslev, T (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics* **31**, 307–327.
- Bollerslev, T. & Ghysels, E. (1996). Periodic Autoregressive Conditional Heteroscedasticity. *Journal of Business & Economic Statistics* **14**, 139-151. https://doi.org/10.1080/07350015.1996.10524640.
- Brooks, C., Burke, S. P., & Persand, G. (2001). Benchmarks and the accuracy of GARCH model estimation. *International Journal of Forecasting* **17**, 45-56. https://doi.org/10.1016/S0169-2070(00)00070-4.
- Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica* **50**, 987-1007. https://doi.org/10.2307/1912773.
- Engle, R. F., and Manganelli, S. (2004). CAViaR: Conditional Autoregressive Value at Risk by Regression Quantiles. *Journal of Business & Economic Statistics* **22**, 367-381. https://doi.org/10.1198/073500104000000370
- Nelson, D.B. (1991). Conditional Heteroskedasticity in Asset Returns: A New Approach. *Econometrica* **59**, 347--370. https://doi.org/10.2307/2938260.