Chapter 17: A case study of financial volatility and a POMP model with observations driving latent dynamics

Edward L. Ionides

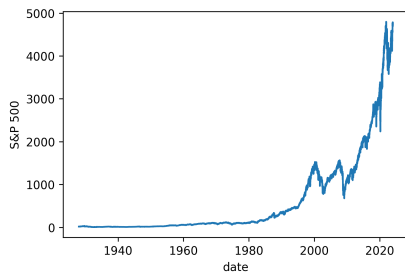
Friday, April 3, 2026

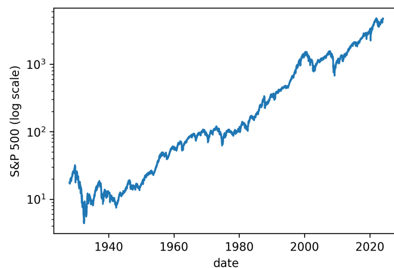
Introduction

- If investment returns are substantially correlated, investors can study their time series behavior and make money.
- If the investment is non-liquid (i.e., not reliably tradeable), or expensive to trade, then it might be hard to make money even if you can statistically predict a positive expected return.
- Otherwise, the market may notice a favorable investment opportunity. More buyers will lead to higher prices, and the opportunity will disappear.
- Consequently, **most readily traded investments (e.g., stock market indices, or stock of large companies) have close to uncorrelated returns.**
- The variability of the returns (called the volatility) can fluctuate considerably. Understanding volatility is important for quantifying and managing investment risk.

The S&P 500 data

- Recall the daily S&P 500 data that we saw earlier, in Chapter 3.





Returns, absolute returns, and autocorrelation

- We write $\{z_n, n = 1, \dots, N\}$ for the S&P 500 index value.
- We write the return, i.e., the difference of the log of the index, as

$$y_n^* = \log(z_n) - \log(z_{n-1}).$$

- We saw in Chapter 3 that $y_{2:N}^*$ has negligible sample autocorrelation.
- However, the absolute deviations from average,

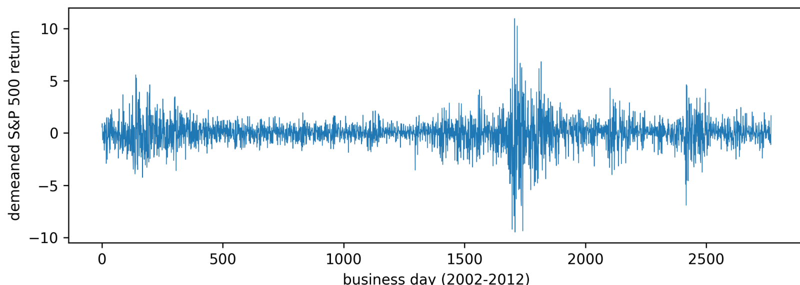
$$a_n^* = \left| y_n^* - \frac{1}{N-1} \sum_{k=2}^{N} y_k^* \right|$$

have considerable sample autocorrelation.

Demeaned daily returns for 2002–2012

- We fit models to the demeaned daily returns for the S&P 500 index for 2002–2012, to compare with Bretó (2014).

```
sp500_data = pd.read_csv(
    "sp500-2002-2012-demeaned.csv", comment="#")
y = sp500_data["x"].values
N = len(y)
```



Question. Is it appropriate to fit a stationary model to this series, or do we have evidence for violation of stationarity? Explain.

The ARCH model

- ARCH and GARCH models are widely used for financial time series modeling. Our introduction follows Cowpertwait and Metcalfe (2009); see also Section 5.4 of Shumway and Stoffer (2017).
- An order p **autoregressive conditional heteroskedasticity** model, known as ARCH(p), has the form

$$Y_n = \epsilon_n \sqrt{V_n},$$

  where $\epsilon_{1:N}$ is white noise and

$$V_n = \alpha_0 + \sum_{j=1}^{p} \alpha_j Y_{n-j}^2.$$

- If $\epsilon_{1:N}$ is Gaussian, then $Y_{1:N}$ is called a Gaussian ARCH(p). Note, however, that a Gaussian ARCH model is not a Gaussian process, just a process driven by Gaussian noise.
- If $Y_{1:N}$ is a Gaussian ARCH(p), then $Y_{1:N}^2$ is AR(p), but not Gaussian AR(p).

The GARCH model

- The **generalized ARCH** model, known as GARCH(p,q), has the form
$$Y_n = \epsilon_n \sqrt{V_n},$$
where
$$V_n = \alpha_0 + \sum_{j=1}^{p} \alpha_j Y_{n-j}^2 + \sum_{k=1}^{q} \beta_k V_{n-k}$$
and $\epsilon_{1:N}$ is white noise.
- The GARCH(1,1) model is a popular choice (Cowpertwait and Metcalfe, 2009) which can be fitted using the `arch` Python package.

Fitting a GARCH model

```
from arch import arch_model
am = arch_model(y, vol="Garch", p=1, q=1, mean="Zero",
    rescale=False)
garch_fit = am.fit(disp="off")
L_garch = garch_fit.loglikelihood
print(f"GARCH(1,1) log-likelihood: {L_garch:.1f}")
```

```
GARCH(1,1) log-likelihood: -4020.5
```

- This 3-parameter model gives a maximized log-likelihood reported above.
- This is a conditional log-likelihood given the first $\max(p, q)$ values.

- We are now in a position to employ the framework of likelihood-based inference for GARCH models. In particular, profile likelihood, likelihood ratio tests, and AIC are available.
- We can readily simulate from a fitted GARCH model, if we want to investigate properties of a fitted model that we don't know how to compute analytically.
- However, GARCH is a black-box model, in the sense that the parameters don't have clear interpretation. We can develop an appropriate GARCH(p,q) model, and that may be useful for forecasting, but it won't help us understand more about how financial markets work.
- We seek models that let us entertain different hypotheses about how volatility behaves.

Stochastic volatility models

- Volatility can be modeled as a latent stochastic process, partially observed via the returns.
- A Markovian assumption for volatility leads to a POMP model.
- As usual for POMP modeling, additional dependence (on previous lags or other variables) can be included.
- These are called **stochastic volatility models**.
- The basic stochastic volatility model (Kastner, 2016) is

$$Y_n = \epsilon_n \exp\{X_n/2\} \quad (1)$$

$$X_n = \mu + \phi(X_{n-1} - \mu) + \sigma\eta_n \quad (2)$$

$$X_0 = \mu + \frac{\sigma}{\sqrt{1-\phi^2}}\eta_0 \quad (3)$$

  where ϵ_n and η_n are $\mathrm{iid}\,\mathcal{N}[0,1]$. Here, X_n is the **log volatility**.
- We can use the flexibility of the POMP framework to see if we can do better.

Volatility leverage

- It is a fairly well established empirical observation that negative shocks to a stock market index are associated with a subsequent increase in volatility.
- This phenomenon is called **leverage**.
- Here, we formally define leverage, R_n on day n as the correlation between index return on day $n-1$ and the increase in the log volatility from day $n-1$ to day n.
- Models have been proposed which incorporate leverage into the dynamics (Bretó, 2014).
- We present a `pypomp` implementation of Bretó (2014), which models R_n as a random walk on a transformed scale,

$$R_n = \frac{\exp\{2G_n\} - 1}{\exp\{2G_n\} + 1},$$

where $\{G_n\}$ is the usual, Gaussian random walk.

Time-varying parameters

- A special case of this model, with the Gaussian random walk having standard deviation zero, is a fixed leverage model.
- The POMP framework provides a general approach to time-varying parameters. Considering a parameter as a latent, unobserved random process that can progressively change its value over time (following a random walk, or some other stochastic process) leads to a POMP model.
- The resulting POMP model is usually non-Gaussian, even when the original model is Gaussian and the perturbations are Gaussian, unless the time-varying parameter enters the model additively.
- Many real-world systems are non-stationary and could be investigated using models with time-varying parameters.

The stochastic leverage model

- Following the notation and model representation in equation (4) of Bretó (2014), we propose a model,

$$Y_n = \exp\{H_n/2\}\,\epsilon_n, \tag{4}$$

$$H_n = \mu_h(1-\phi) + \phi H_{n-1} + \beta_{n-1} R_n \exp\{-H_{n-1}/2\} + \omega_n, \tag{5}$$

$$G_n = G_{n-1} + \nu_n, \tag{6}$$

log volatility

leverage.

β depends on Y.

For a POMP model, we cannot include Y in the latent dynamics.

where $\beta_n = Y_n \sigma_\eta \sqrt{1-\phi^2}$, $\{\epsilon_n\}$ is an iid $N(0,1)$ sequence, $\{\nu_n\}$ is an iid $N(0,\sigma_\nu^2)$ sequence, and ω_n is $N(0,\sigma_{\omega,n}^2)$ with

$$\sigma_{\omega,n}^2 = \sigma_\eta^2(1-\phi^2)(1-R_n^2).$$

- Here, H_n is the log volatility. The latent state is $X_n = (G_n, H_n)$, noting that R_n is a function of G_n.

Building a POMP model

- A complication is that transition of the latent variables from (G_n, H_n) to (G_{n+1}, H_{n+1}) depends on the observable variable Y_n.
- This situation appears to be a violation of the POMP model structure.
- It is not so uncommon. For example, the same thing happens in a dynamic system subject to a control measure which is a function of the observed data.
- We can write out an extended model to fit this situation into the POMP structure, to provide access to methodology for POMP models.

- Formally, a POMP representation has state variable $X_n = (G_n, H_n, Y_n)$ and measurement variable Y_n being perfect observation of this component of X_n.
- When the latent state is continuous and there is no measurement error, the basic particle filter fails since all prediction particles are inconsistent with the data. We need a modification of sequential Monte Carlo (SMC).
- We write the filtered particle j at time $n-1$ as

$$X^F_{n-1,j} = (G^F_{n-1,j}, H^F_{n-1,j}, y^*_{n-1}).$$

- Now we can construct prediction particles at time n,

$$(G^P_{n,j}, H^P_{n,j}) \sim f_{G_n,H_n|G_{n-1},H_{n-1},Y_{n-1}}(g_n|G^F_{n-1,j}, H^F_{n-1,j}, y^*_{n-1})$$

with corresponding weight

$$w_{n,j} = f_{Y_n|G_n,H_n}(y^*_n|G^P_{n,j}, H^P_{n,j}).$$

- Resampling with probability proportional to these weights gives an SMC representation of the filtering distribution at time n.
- A derivation of this is given as an Appendix.

Building two versions of the model in `pypomp`

- We can coerce the basic sequential Monte Carlo algorithm, implemented as `pfilter` in `pypomp`, into carrying out this calculation by building two different `Pomp` objects, one to do filtering and another to do simulation.
- For the **filtering** version, `rproc` reads the observation Y_n from a covariate slot rather than simulating it. The measurement model `dmeas` then provides the particle weights.
- For the **simulation** version, `rproc` simulates Y_n from the model.

Model components

Here, we add type hints to the model components. This is optional, but it may help humans and AIs to read and write accurately.

```
from pypomp.types import (
    StateDict, ParamDict, CovarDict,
    TimeFloat, StepSizeFloat, RNGKey,
    ObservationDict, InitialTimeFloat,
)

statenames = ["H", "G", "Y_state"]

def rinit(theta_: ParamDict, key: RNGKey,
        covars: CovarDict, t0: InitialTimeFloat):
    G_0 = theta_["G_0"]
    H_0 = theta_["H_0"]
    Y_state = jax.random.normal(key) * jnp.exp(H_0 / 2)
    return {"H": H_0, "G": G_0, "Y_state": Y_state}
```

type hints: can be helpful for debugging.

Process model for filtering

The filtering version sets `Y_state` from the covariate (i.e., the observed data). This should implement the equations.

```
def rproc_filt(X_: StateDict, theta_: ParamDict,
        key: RNGKey, covars: CovarDict,
        t: TimeFloat, dt: StepSizeFloat):
    H = X_["H"]
    G = X_["G"]
    Y_state = X_["Y_state"]
    sigma_nu = theta_["sigma_nu"]
    mu_h = theta_["mu_h"]
    phi = theta_["phi"]
    sigma_eta = theta_["sigma_eta"]
    key1, key2 = jax.random.split(key)
    omega = jax.random.normal(key1) * (
        sigma_eta * jnp.sqrt(1 - phi**2)
        * jnp.sqrt(1 - jnp.tanh(G)**2))
    nu = jax.random.normal(key2) * sigma_nu
    G = G + nu
    beta = Y_state * sigma_eta * jnp.sqrt(1 - phi**2)
    H = (mu_h * (1 - phi) + phi * H
        + beta * jnp.tanh(G) * jnp.exp(-H / 2)
        + omega)
    Y_state = covars["covaryt"]
    return {"H": H, "G": G, "Y_state": Y_state}
```

Process model for simulation

The simulation version simulates `Y_state` from the model.

```
def rproc_sim(X_: StateDict, theta_: ParamDict,
        key: RNGKey, covars: CovarDict,
        t: TimeFloat, dt: StepSizeFloat):
    H = X_["H"]
    G = X_["G"]
    Y_state = X_["Y_state"]
    sigma_nu = theta_["sigma_nu"]
    mu_h = theta_["mu_h"]
    phi = theta_["phi"]
    sigma_eta = theta_["sigma_eta"]
    key1, key2, key3 = jax.random.split(key, 3)
    omega = jax.random.normal(key1) * (
        sigma_eta * jnp.sqrt(1 - phi**2)
        * jnp.sqrt(1 - jnp.tanh(G)**2))
    nu = jax.random.normal(key2) * sigma_nu
    G = G + nu
    beta = Y_state * sigma_eta * jnp.sqrt(1 - phi**2)
    H = (mu_h * (1 - phi) + phi * H
        + beta * jnp.tanh(G) * jnp.exp(-H / 2)
        + omega)
    Y_state = jax.random.normal(key3) * jnp.exp(H / 2)
    return {"H": H, "G": G, "Y_state": Y_state}
```

Measurement model

```
def dmeas(Y_: ObservationDict, X_: StateDict,
        theta_: ParamDict, covars: CovarDict,
        t: TimeFloat):
    H = X_["H"]
    y = Y_["y"]
    return jax.scipy.stats.norm.logpdf(
        y, 0, jnp.exp(H / 2))

def rmeas(X_: StateDict, theta_: ParamDict,
        key: RNGKey, covars: CovarDict,
        t: TimeFloat):
    H = X_["H"]
    return jnp.array(
        [jax.random.normal(key) * jnp.exp(H / 2)])
```

Parameter transformations

For optimization procedures such as iterated filtering, it is convenient to transform parameters to be defined on the whole real line. We therefore transform σ_η, σ_ν and ϕ.

```
def to_est(theta):
    return {
        "sigma_nu": jnp.log(theta["sigma_nu"]),
        "mu_h": theta["mu_h"],
        "phi": jnp.log(
            (1 + theta["phi"]) / (1 - theta["phi"])),
        "sigma_eta": jnp.log(theta["sigma_eta"]),
        "G_0": theta["G_0"],
        "H_0": theta["H_0"],
    }

def from_est(theta):
    return {
        "sigma_nu": jnp.exp(theta["sigma_nu"]),
        "mu_h": theta["mu_h"],
        "phi": (jnp.exp(theta["phi"]) - 1)
            / (jnp.exp(theta["phi"]) + 1),
        "sigma_eta": jnp.exp(theta["sigma_eta"]),
        "G_0": theta["G_0"],
        "H_0": theta["H_0"],
    }
```

Building the pypomp objects

Note that the data are also placed in a covariate slot. This allows the state process transition to depend on the data. In a POMP model, the latent process transition depends only on the current latent state.

A POMP model allows the basic components to depend on arbitrary covariates. In `pypomp`, this means `rproc` has access to a covariate slot.

```
ys = pd.DataFrame({"y": y}, index=np.arange(1, N + 1).astype(float))

covars = pd.DataFrame(
    {"covaryt": np.concatenate([[0.0], y])},
    index=np.arange(0, N + 1).astype(float))
```

The version for filtering uses `rproc_filt`

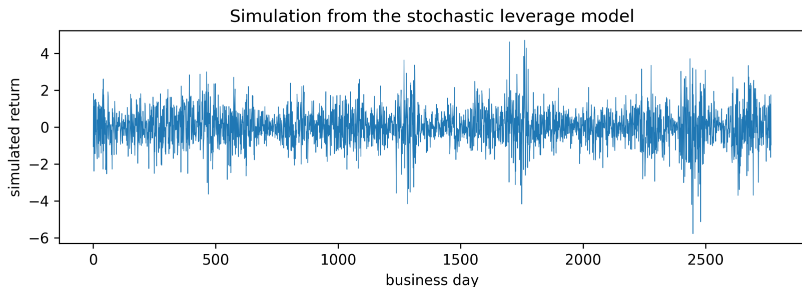
```
sp500_filt = pp.Pomp(
    ys=ys,
    theta=params_test,
    statenames=statenames,
    t0=0.0,
    nstep=1,
    rinit=rinit,
    rproc=rproc_filt,
    dmeas=dmeas,
    rmeas=rmeas,
    covars=covars,
    par_trans=par_trans,
    ydim=1,
)
```

The version for simulating uses `rproc_sim`

```
sp500_sim = pp.Pomp(
    ys=ys,
    theta=params_test,
    statenames=statenames,
    t0=0.0,
    nstep=1,
    rinit=rinit,
    rproc=rproc_sim,
    dmeas=dmeas,
    rmeas=rmeas,
    covars=covars,
    par_trans=par_trans,
    ydim=1,
)
```

Simulating from the model

```
key = jax.random.key(1)
X_sim, Y_sim = sp500_sim.simulate(key=key, nsim=1)
```



Building a filtering object from simulated data

To test the filtering and parameter estimation code on simulated data, we need to copy the simulated data into the covariate slot and use the filtering version of `rproc`.

```
ys_sim = pd.DataFrame(
    {"y": sim_y},
    index=np.arange(1, len(sim_y) + 1).astype(float))
covars_sim = pd.DataFrame(
    {"covaryt": np.concatenate([[0.0], sim_y])},
    index=np.arange(0, len(sim_y) + 1).astype(float))

sim1_filt = pp.Pomp(
    ys=ys_sim,
    theta=params_test,
    statenames=statenames,
    t0=0.0,
    nstep=1,
    rinit=rinit,
    rproc=rproc_filt,
    dmeas=dmeas,
    rmeas=rmeas,
    covars=covars_sim,
    par_trans=par_trans,
    ydim=1,
)
```

Filtering on simulated data

- We check that we can indeed filter and re-estimate parameters successfully for this simulated data.
- We set up code to switch between different levels of computational intensity:

```
sp500_Np =            [50, 500, 2000][RL]
sp500_Nmif =          [5,  20,  200][RL]
sp500_Nreps_eval =    [2,  5,    20][RL]
sp500_Nreps_local =   [2,  5,    20][RL]
sp500_Nreps_global =  [2,  5,   100][RL]
```

Test the particle filter on simulated data

```
cache_file = cache_dir + "/pf1.pkl"
if os.path.exists(cache_file):
    with open(cache_file, 'rb') as f:
        pf1_result = pickle.load(f)
else:
    key = jax.random.key(34118892)
    t_start = time.time()
    sim1_filt.pfilter(
        key=key, J=sp500_Np,
        reps=sp500_Nreps_eval)
    t_pf1 = time.time() - t_start
    pf1_result = sim1_filt.results_history.last()
    pf1_result.time_elapsed = t_pf1
    with open(cache_file, 'wb') as f:
        pickle.dump(pf1_result, f)
```

```
# Average using logmeanexp since the likelihood estimate
# is unbiased on the natural scale but not the log scale.
logLiks = np.array(pf1_result.logLiks).flatten()
L_pf1 = pp.logmeanexp(logLiks)
L_pf1_se = pp.logmeanexp_se(logLiks)
print(f"Log-likelihood estimate: {L_pf1:.2f}"
    f" (SE: {L_pf1_se:.2f})")
```

```
Log-likelihood estimate: -93061.00 (SE: 13.65)
```

Fitting the stochastic leverage model to S&P 500 data

- We are now ready to try out iterated filtering on the S&P 500 data. We will use the IF2 algorithm of Ionides et al. (2015), implemented by `mif` in `pypomp`.

```
sp500_rw_sd_rp = 0.02
sp500_rw_sd_ivp = 0.1
sp500_cooling_fraction_50 = 0.5

rw_sd = pp.RWSigma(
    sigmas={
        "sigma_nu":  sp500_rw_sd_rp,
        "mu_h":      sp500_rw_sd_rp,
        "phi":       sp500_rw_sd_rp,
        "sigma_eta": sp500_rw_sd_rp,
        "G_0":       sp500_rw_sd_ivp,
        "H_0":       sp500_rw_sd_ivp,
    },
    init_names=["G_0", "H_0"],
)
```

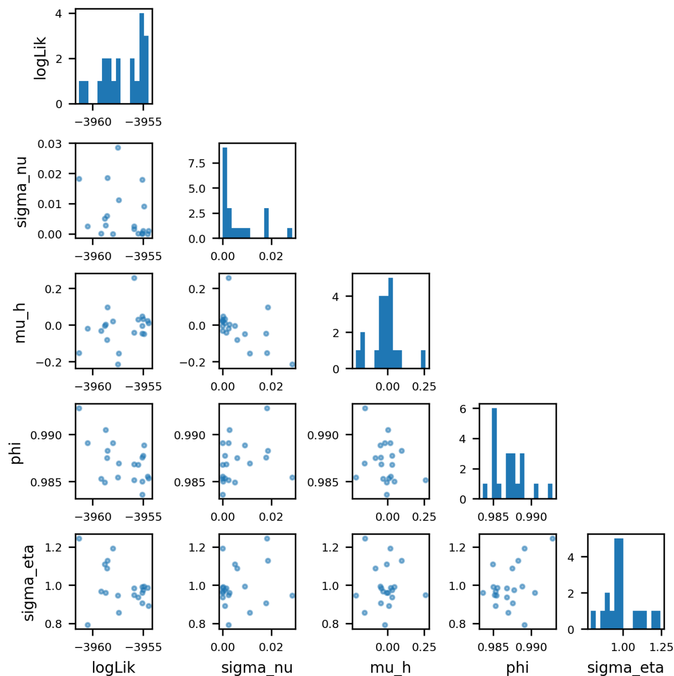
Local search

1. Set up the cache for the results
2. Conduct multiple searches from a single initial condition
3. Conduct multiple log-likelihood evaluations

```
r_if1 = local_results
print(f"Best log-likelihood: "
    f"{r_if1['logLik'].max():.1f}")
```

```
Best log-likelihood: -3954.5
```

- We proceed to look at convergence diagnostics...



logLik
sigma_nu
mu_h
phi
sigma_eta
−3960
−3955
0.00
0.02
0.00
0.25
0.985
0.990
1.00
1.25

Likelihood maximization using randomized starting values

but not too large!
test on simulated data to assess the capabilities for searching this model.

- As for our other case studies, carrying out searches starting randomly throughout a large box can lead to reasonable evidence for successful global maximization.
- For our volatility model, a box containing plausible parameter values might be

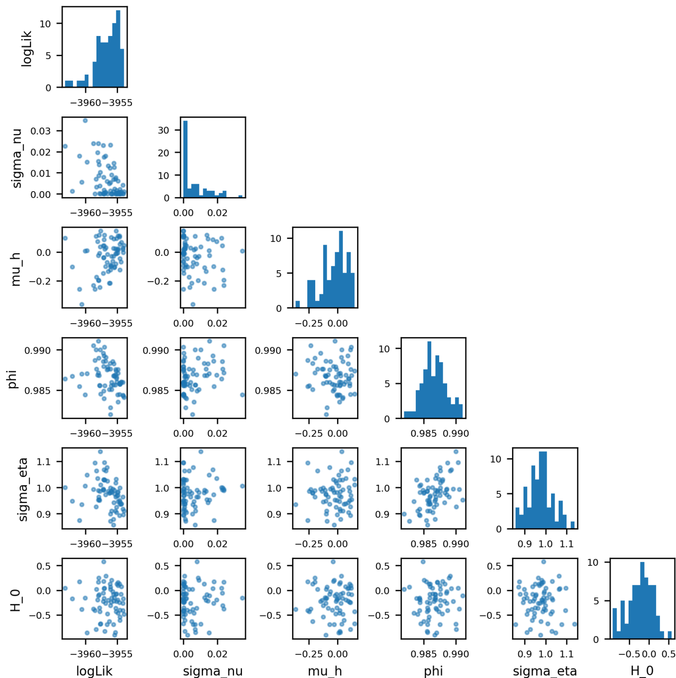
```
sp500_box = {
    "sigma_nu":  (0.005, 0.05),
    "mu_h":      (-1.0, 0.0),
    "phi":       (0.95, 0.99),
    "sigma_eta": (0.5, 1.0),
    "G_0":       (-2.0, 2.0),
    "H_0":       (-1.0, 1.0),
}
```

Global search

- More searches, from diverse starting points.

```
count      100.000000
mean     -3966.559410
std         15.502613
min      -3997.697044
25%      -3986.828089
50%      -3957.659646
75%      -3955.568455
max      -3953.909832
Name: logLik, dtype: float64

Best log-likelihood: -3953.9
Best parameters:
  sigma_nu: 0.0010
  mu_h: 0.0467
  phi: 0.9841
  sigma_eta: 0.9109
  G_0: -1.0043
  H_0: -0.4879
```


logLik
sigma_nu
mu_h
phi
sigma_eta
H_0
−3960
−3955
0.00
0.02
−0.25
0.00
0.985
0.990
0.9
1.0
1.1
−0.5
0.0
0.5

- This preliminary analysis does not show clear evidence for the hypothesis that $\sigma_\nu > 0$.
- That is likely because we are studying only a subset of the 1988 to 2012 dataset analyzed by Bretó (2014).
- Also, it might help to refine our inference by computing a likelihood profile over σ_ν.

Benchmark likelihoods for alternative models

To assess the overall success of the model, it is helpful to put the log likelihoods in the context of simpler models, called **benchmarks.**

Benchmarks provide a complementary approach to residual analysis and the investigation of simulations from the fitted model.

```
L_pomp = r_box["logLik"].max()
print(f"GARCH(1,1) log-likelihood: {L_garch:.1f}"
    f" (3 parameters)")
print(f"Stochastic leverage log-likelihood: "
    f"{L_pomp:.1f} (6 parameters)")
aic_garch = -2 * L_garch + 2 * 3
aic_pomp = -2 * L_pomp + 2 * 6
```

```
GARCH(1,1) log-likelihood: -4020.5 (3 parameters)
Stochastic leverage log-likelihood: -3953.9 (6 parameters)
```

Interpretation of benchmark results

```
AIC comparison:
  GARCH(1,1): 8047.0
  Stochastic leverage: 7919.8
```

- A model which both fits better and has meaningful interpretation has clear advantages over a simple statistical model.
- The disadvantage of the sophisticated modeling and inference is the extra effort required.
- You should choose the best-fitting benchmarks that you can think of to rigorously test your mechanistic model.

Can a mechanistic model be helpful if it loses to a non-mechanistic alternative?

- Sometimes, the mechanistic model does not beat simple benchmark models. That does not necessarily mean the mechanistic model is entirely useless.
- We may be able to learn about the system under investigation from what a scientifically interpretable model fails to explain.
- We may be able to use preliminary results to improve the model, and subsequently beat the benchmarks.
- If the mechanistic model fits disastrously compared to the benchmark, our model is probably missing something important. We must reconsider the model, based on clues we might obtain by carrying out residual analysis and looking at simulations from the fitted model.

Appendix: Proper weighting for a partially plug-and-play algorithm with a perfectly observed state space component

- Suppose a POMP model with $X_n = (U_n, V_n)$ and measurement model $f_{Y_n|X_n}(y_n \mid u_n, v_n) = f_{Y_n|V_n}(y_n|v_n)$, depending only on v_n.
- The proper weight for an SMC proposal density $q_n(x_n|x_{n-1})$ is

$$w_n(x_n|x_{n-1}) = \frac{f_{Y_n|X_n}(y_n^*|x_n) f_{X_n|X_{n-1}}(x_n|x_{n-1})}{q_n(x_n|x_{n-1})}.$$

- Consider the proposal $q_n(u_n, v_n|x_{n-1}) = f_{U_n|X_{n-1}}(u_n|x_{n-1}) g_n(v_n)$. This is partially plug-and-play, in the sense that the U_n part of the proposal is drawn from a simulator of the dynamic system.

- Computing the weights, we see that the transition density for the U_n component cancels out and does not have to be computed, i.e.,

$$w_n(x_n|x_{n-1}) = \frac{f_{Y_n|V_n}(y_n^*|v_n) f_{U_n|X_{n-1}}(u_n|x_{n-1}) f_{V_n|U_n,X_{n-1}}(v_n|u_n,x_{n-1})}{f_{U_n|X_{n-1}}(u_n|x_{n-1}) g_n(v_n)}$$

$$= \frac{f_{Y_n|V_n}(y_n^*|v_n) f_{V_n|U_n,X_{n-1}}(v_n|u_n,x_{n-1})}{g_n(v_n)}.$$

- Now consider the case where the V_n component of the state space is perfectly observed, i.e., $Y_n = V_n$. In this case,

$$f_{Y_n|V_n}(y_n|v_n) = \delta(y_n - v_n),$$

  interpreted as a point mass at v_n in the discrete case and a singular density at v_n in the continuous case.

- We can choose $g_n(v_n)$ to depend on the data, and a natural choice is

$$g_n(v_n) = \delta(y_n^* - v_n),$$

  for which the proper weight is

$$w_n(x_n|x_{n-1}) = f_{Y_n|U_n,X_{n-1}}(y_n^*|u_n, x_{n-1}).$$

- This is the situation in the context of our case study, with $U_n = (G_n, H_n)$ and $V_n = Y_n$.

References and Acknowledgements

- Licensed under the Creative Commons Attribution-NonCommercial license. Please share and remix non-commercially, mentioning its origin.
- We acknowledge previous versions of this course.
- Translated from a previous R version with assistance from Claude Code opus 4.6.

Bretó, C. (2014). On idiosyncratic stochasticity of financial leverage effects. *Statistics & Probability Letters*, 91:20–26.

Cowpertwait, P. S. and Metcalfe, A. V. (2009). *Introductory time series with R*. Springer Science & Business Media.

Ionides, E. L., Nguyen, D., Atchadé, Y., Stoev, S., and King, A. A. (2015). Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proceedings of the National Academy of Sciences of the U.S.A.*, 112(3):719–724.

Kastner, G. (2016). Dealing with stochastic volatility in time series using the R package stochvol. *Journal of Statistical Software*, 69.

Shumway, R. H. and Stoffer, D. S. (2017). *Time Series Analysis and its Applications: With R Examples*. Springer, 4th edition.