Estimating time series models by state space methods in Python: Statsmodels

Chad Fulton*

Abstract

This paper describes an object oriented approach to the estimation of time series models using state space methods and presents an implementation in the Python programming language. This approach at once allows for fast computation, a variety of out-of-the-box features, and easy extensibility. We show how to construct a custom state space model, retrieve filtered and smoothed estimates of the unobserved state, and perform parameter estimation using classical and Bayesian methods. The mapping from theory to implementation is presented explicitly and is illustrated at each step by the development of three example models: an ARMA(1,1) model, the local level model, and a simple real business cycle macroeconomic model. Finally, four fully implemented time series models are presented: SARIMAX, VARMAX, unobserved components, and dynamic factor models. These models can immediately be applied by users.

* I thank Josef Perktold for many helpful discussions. Financial support from the Google Summer of Code program and the University of Oregon Kleinsorge Fellowship, Department of Economics, is gratefully acknowledged.

Contents

---

1 Introduction

The class of time series models that can be represented in state space form, allowing parameter estimation and inference, is very broad. Many of the most widespread reduced form time series models

fall into this class, including autoregressive integrated moving average (ARIMA), vector autoregressions (VARs), unobserved components (UC), time-varying parameters (TVP), and dynamic factor (DFM) models. Furthermore, linear (or linearized) structural models are often amenable to representation in this form, including the important case of linearized DSGE models. This paper contributes to the literature on practical results related to the estimation of linear, Gaussian state space models and the corresponding class of time series models.

The great advantage of representing a time series as a linear, Gaussian state space model is due to existence of the celebrated Kalman filter (Kalman, 1960), which at once provides optimal contempraneous estimates of unobserved state variables and also permits evaluation of the likelihood of the model. Subsequent developments have produced a range of smoothers and computational techniques which makes feasible a estimation even in the case of large datasets and complicated models. Elegant theoretical results can be developed quite generically and applied to any of the models in the state space class.

Mirroring this theoretical conservation of effort is the possibility of a practical conservation: appropriately designed computer programs that perform estimation and inference can be written generically in terms of the state space form and then applied to any of models which fall into that class. Not only is it inefficient for each practitioner to separately implement the same features, it is unreasonable to expect that everyone devote potentially large amounts of time to produce high-performance, well-tested computer programs, particularly when their comparative advantage lies elsewhere. This paper describes a method for achieving this practical conservation of effort by making use of so-called object oriented programming, with an accompanying implementation in the Python programming language.[1]

Time series analysis by state space methods is present in nearly every statistical software package, including commercial packages like Stata and E-views, commercial compuational environments such as MATLAB, and open-source programming languages including R and gretl. A recent spe-

[1] Among others, the programming environments MATLAB and R also support object oriented programming; the implementation described here could therefore, in principle, be migrated to those languages.

cial volume of the Journal of Statistical Software was devoted to software implementations of state space models; see Commandeur et al. (2011) for the introductory article and a list of references. This is also not the first implementation of Kalman filtering and smoothing routines in Python; although many packages at various stages of development exist, one notable reference is the PySSM package presented in Strickland et al. (2014).

Relative to these libraries, this package has several important features. First, although several of the libraries mentioned above (including the Python implementation) use object-oriented techniques in their internal code, this is the first implementation to emphasize those techniques for users of the library. As described throughout the paper, this can yield substantial time saving on the part of users, by providing a unified interface to the state space model rather than a collection of disparate functions.

Second, it is the first implementation to emphasize interaction with an existing ecosystem of well-estabilished scientific libraries. Since state space estimation is a component of the larger Statsmodels package (Seabold and Perktold, 2010), users automatically have available many other econometric and statistical models and functions (in this way, Statsmodels is somewhat similar to, for example, Stata). It also has links to other packages; for example, in section 6 we describe Metropolis-Hastings posterior simulation using the Python package PyMC.

One practically important manifestation of the tighter integration of Statsmodels with the Python ecosystem is that this package is easy to install and does not require the user to compile code themselves (as does for example PySSM). Furthermore, while PySSM also uses compiled code for the performance critical filtering and smoothing operations, in this package these routines are written in a close variant of Python (see below for more details on "Cython"). This means that the underlying code is easier to understand and debug and that a tighter integration can be achieved between user-code and compiled-code.

Finally, it incorporates recent advances in state space model estimation, including the collapsed filtering approach of Jungbacker and Koopman (2014), and makes available flexible classes for

specifying and estimating four of the most popular time series models: SARIMAX, unobserved components, VARMAX, and dynamic factor models.

One note is warranted about the Python code presented in this paper. In Python, most functionality is provided by packages not necessarily loaded by default. To use these packages in your code, you must first "import" them. In all the code that follows, we will assume the following imports have already been made

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
```

Any additional imports will be explicitly provided in the example code. In any code with simulations we assume that the following code has been used to set the seed for the pseudo-random number generator: `np.random.seed(17429)`.

The remainder of the paper is as follows. Section 2 gives an overview of the linear, Gaussian state space model along with the Kalman filter, state smoother, disturbance smoother, and simulation smoother, and presents several examples of time series models in state space form. Section 3 describes the representation in Python of the state space model, and provides sample code for each of the example models. Sections 4 and 5 describe the estimation of unknown system parameters by maximum likelihood (MLE) and Markov chain Monte Carlo (MCMC) methods, respectively, and show the application to the example models. Up to this point, the paper has been concerned with the implementation of custom state space models. However Statsmodels also contains a number of out-of-the-box models and these are described in section 6. Section 7 concludes.[2]

[2] For instructions on the installation of this package, see *Appendix A: Installation*. Full documentation for the package is available at http://www.statsmodels.org.

Table 1: Elements of state space representation

Object	Description	Dimensions
y_t	Observed data	$p \times 1$
α_t	Unobserved state	$m \times 1$
d_t	Observation intercept	$p \times 1$
Z_t	Design matrix	$p \times m$
ε_t	Observation disturbance	$p \times 1$
H_t	Observation disturbance covariance matrix	$p \times p$
c_t	State intercept	$m \times 1$
T_t	Transition matrix	$m \times m$
R_t	Selection matrix	$m \times r$
η_t	State disturbance	$r \times 1$
Q_t	State disturbance covariance matrix	$r \times r$

2 State space models

The state space representation of a possibly time-varying linear and Gaussian time series model can be written as

$$y_t = d_t + Z_t\alpha_t + \varepsilon_t \qquad \varepsilon_t \sim N(0, H_t)$$

$$\alpha_{t+1} = c_t + T_t\alpha_t + R_t\eta_t \qquad \eta_t \sim N(0, Q_t)$$

where y_t is observed, so the first equation is called the observation or measurement equation, and α_t is unobserved. The second equation describes the transition of the unobserved state, and so is called the transition equation. The dimensions of each of the objects, as well as the name by which we will refer to them, are given in Table 1. All notation in this paper will follow that in Commandeur et al. (2011) and Durbin and Koopman (2012).

The model is called time-invariant if only y_t and α_t depend on time (so, for example, in a time-invariant model $Z_t = Z_{t+1} \equiv Z$). In the case of a time-invariant model, we will drop the time subscripts from all state space representation matrices. Many important time series models are time-invariant, including ARIMA, VAR, unobserved components, and dynamic factor models.

2.1 Kalman Filter

The Kalman filter, as applied to the state space model above, is a recursive formula running forwards through time ($t = 1, 2, \ldots, n$) providing optimal estimates of the unknown state.[3] At time t, the *predicted* quantities are the optimal estimates conditional on observations up to $t-1$, and the *filtered* quantities are the optimal estimates conditional on observations up to time t. This will be contrasted below with *smoothed* quantities, which are optimal estimates conditional on the full sample of observations.

We now define some notation that will be useful below. Define the vector of all observations up to time s as $Y_s = \{y_1, \ldots, y_s\}$. Then the distribution of the predicted state is $\alpha_t \mid Y_{t-1} \sim N(a_t, P_t)$, and the distribution of the filtered state is $\alpha_t \mid Y_t \sim N(a_{t|t}, P_{t|t})$.

As shown in, for example, Durbin and Koopman (2012), the Kalman filter applied to the model (2) above yields a recursive formulation. Given prior estimates a_t, P_t, the filter produces optimal filtered and predicted estimates ($a_{t|t}, P_{t|t}$ and a_{t+1}, P_{t+1}, respectively) as follows

$$
\begin{aligned}
v_t &= y_t - Z_t a_t - d_t & F_t &= Z_t P_t Z_t' + H_t \\
a_{t|t} &= a_t + P_t Z_t' F_t^{-1} v_t & P_{t|t} &= P_t - P_t Z_t' F_t^{-1} Z_t P_t \\
a_{t+1} &= T_t a_{t|t} + c_t & P_{t+1} &= T_t P_{t|t} T_t' + R_t Q_t R_t'
\end{aligned}
$$

An important byproduct of the Kalman filter iterations is evaluation of the loglikelihood of the observed data due to the so-called "prediction error decomposition".

The dimensions of each of the objects, as well as the name by which we will refer to them, are given in Table 2. Also included in the table is the Kalman gain, which is defined as $K_t = T_t P_t Z_t' F_t^{-1}$.

[3] In this paper, "optimal" can be interpreted in the sense of minimizing the mean-squared error of estimation. In chapter 4, Durbin and Koopman (2012) show three other senses in which optimal can be defined for this same model.

Table 2: Elements of Kalman filter recursions

Object	Description	Dimensions
a_t	Prior state mean	$m \times 1$
P_t	Prior state covariance	$m \times m$
v_t	Forecast error	$p \times 1$
F_t	Forecast error covariance matrix	$p \times p$
$a_{t\|t}$	Filtered state mean	$m \times 1$
$P_{t\|t}$	Filtered state covariance matrix	$m \times m$
a_{t+1}	Predicted state mean	$m \times 1$
P_{t+1}	Predicted state covariance matrix	$m \times m$
$\log L(Y_n)$	Loglikelihood	scalar
K_t	Kalman gain	$m \times p$

2.2 Initialization

Notice that since the Kalman filter is a recursion, for $t = 2, \ldots, n$ the prior state mean and prior state covariance matrix are given as the output of the previous recursion. For $t = 1$, however, no previous recursion has been yet applied, and so the mean a_1 and covariance P_1 of the distribution of the initial state $\alpha_1 \sim N(a_1, P_1)$ must be specified. The specification of the distribution of the initial state is referred to as *initialization*.

There are four methods typically used to initialize the Kalman filter: (1) if the distribution is known or is otherwise specified, initialize with the known values; (2) initialize with the unconditional distribution of the process (this is only applicable to the case of stationary state processes); (3) initialize with a diffuse (i.e. infinite variance) distribution; (4) initialize with an approximate diffuse distribution, i.e. $a_1 = 0$ and $P_1 = \kappa I$ where κ is some large constant (for example $\kappa = 10^6$). When the state has multiple elements, a mixture of these four approaches can be used, as appropriate.

Of course, if options (1) or (2) are available, they are preferred. In the case that there are non-stationary components with unknown initial distribution, either (3) or (4) must be employed. While (4) is simple to use, Durbin and Koopman (2012) note that "while the device can be useful for approximate exploratory work, it is not recommended for general use since it can lead to large rounding errors;" for that reason they recommend using exact diffuse initialiation. For more about initialization, see Koopman and Durbin (2003) and chapter 5 of Durbin and Koopman (2012).

Note that when diffuse initialization is applied, a number of initial loglikelihood values are excluded ("burned") when calculating the joint loglikelihood, as they are considered under the influence of the diffuse states. In exact diffuse initialization the number of burned periods is determined in initialization, but in the approximate case it must be specified. In this case, it is typically set equal to the dimension of the state vector; it turns out that this often coincides with the value in the exact case.

2.3 State and disturbance smoothers

The state and disturbance smoothers, as applied to the state space model above, are recursive formulas running backwards through time ($t = n, n-1, \ldots, 1$) providing optimal estimates of the unknown state and disturbance vectors based on the full sample of observations.

As developed in Koopman (1993) and Chapter 4 of Durbin and Koopman (2012), following an application of the Kalman filter (yielding the predicted and filtered estimates of the state) the smoothing recursions can be written as (where $L_t = T_t - K_t Z_t$)

$$\begin{aligned}
\hat{\alpha}_t &= a_t + P_t r_{t-1} & V_t &= P_t - P_t N_{t-1} P_t \\
\hat{\varepsilon}_t &= H_t u_t & Var(\varepsilon_t \mid Y_n) &= H_t - H_t(F_t^{-1} + K_t' N_t K_t) H_t \\
\hat{\eta}_t &= Q_t R_t' r_t & Var(\eta_t \mid Y_n) &= Q_t - Q_t R_t' N_t R_t Q_t \\
u_t &= F_t^{-1} v_t - K_t' r_t & & \\
r_{t-1} &= Z_t' u_t + T_t' r_t & N_{t-1} &= Z_t' F_t^{-1} Z_t + L_t' N_t L_t
\end{aligned}$$

The dimensions of each of the objects, as well as the name by which we will refer to them, are given in Table 3.

Table 3: Elements of state and disturbance smoother recursions

Object	Description	Dimensions
$\hat{\alpha}_t$	Smoothed state mean	$m \times 1$
V_t	Smoothed state covariance matrix	$m \times m$
$\hat{\varepsilon}_t$	Smoothed observation disturbance mean	$p \times 1$
$Var(\varepsilon_t \mid Y_n)$	Smoothed observation disturbance covariance matrix	$p \times p$
$\hat{\eta}_t$	Smoothed state disturbance mean	$m \times 1$
$Var(\eta_t \mid Y_n)$	Smoothed state disturbance covariance matrix	$m \times m$
u_t	Smoothing error	$p \times 1$
r_{t-1}	Scaled smoothed estimator	$m \times 1$
N_{t-1}	Scaled smoothed estimator covariance matrix	$m \times m$

Table 4: Output of the simulation smoother

Object	Description	Dimensions
$\tilde{\alpha}_t$	Simulated state	$m \times 1$
$\tilde{\varepsilon}_t$	Simulated observation disturbance	$p \times 1$
$\tilde{\eta}_t$	Simulated state disturbance	$m \times 1$

2.4 Simulation smoother

The simulation smoother, developed in Durbin and Koopman (2002) and Chapter 4 of Durbin and Koopman (2012), allows drawing samples from the distributions of the full state and disturbance vectors, conditional on the full sample of observations. It is an example of a "forwards filtering, backwards sampling" algorithm because one application of the simulation smoother requires one application each of the Kalman filter and state / disturbance smoother. An often-used alternative forwards filtering, backwards sampling algorithm is that of Carter and Kohn (1994).

The output of the simulation smoother is the drawn samples; the dimensions of each of the objects, as well as the name by which we will refer to them, are given in Table 4.

2.5 Practical considerations

There are a number of important practical considerations associated with the implementation of the Kalman filter and smoothers in computer code. Two of the most important are numerical stability and computational speed; these issues are briefly described below, but will be revisited when the

Python implementation is discussed.

In the context of the Kalman filter, numerical stability usually refers to the possibility that the recursive calculations will not maintain the positive definiteness or symmetry of the various covariance matrices. Numerically stable routines can be used to mitigate these concerns, for example using linear solvers rather than matrix inversion. In extreme cases a numerically stable Kalman filter, the so-called square-root Kalman filter, can be used (see Morf and Kailath (1975) or chapter 6.3 of Durbin and Koopman (2012)).

Performance can be an issue because the Kalman filter largely consists of iterations (loops) and matrix operations, and it is well known that loops perform poorly in interpreted languages like MATLAB and Python.[4] Furthermore, regardless of the high-level programming language used, matrix operations are usually ultimately performed by the highly optimized BLAS and LAPACK libraries written in Fortran. For performant code, compiled languages are preferred, and code should directly call the BLAS and LAPACK libraries directly when possible, rather than through intermediate functions (for details on the BLAS and LAPACK libraries, see Anderson et al. (1999)).

2.6 Additional remarks

Several additional remarks are merited about the Kalman filter. First, under certain conditions, for example a time-invariant model, the Kalman filter will converge, meaning that the predicted state covariance matrix, the forecast error covariance matrix, and the Kalman gain matrix will all reach steady-state values after some number of iterations. This can be exploited to improve performance.

The second remark has to do with missing data. In the case of completely or partially missing observations, not only can the Kalman filter proceed with making optimal estimates of the state vector, it can provide optimal estimates of the missing data.

Third, the state space approach can be used to obtain optimal forecasts and to explore impulse

[4] The availability of a just-in-time (JIT) compiler can help with loop performance in interpreted languages; one is integrated into MATLAB, and the Numba project introduces one into Python.

response functions.

Finally, the state space approach can be used for parameter estimation, either through classical methods (for example maximum likelihood estimation) or Bayesian methods (for example posterior simulation via Markov chain Monte Carlo). This will be described in detail in sections 4 and 5, below.

2.7 Example models

As mentioned above, many important time series models can be represented in state space form. We present three models in detail to use as examples: an autoregressive moving average (ARMA) model, the local level model, and a simple real business cycle (RBC) dynamic stochastic general equilibrium (DSGE) model.

In fact, general versions of several time series models have already been implemented in Statsmodels and are available for use (see *Out-of-the-box models* for details). However, since the goal here is to provide information sufficient for users to specify and estimate their own custom models, we emphasize the translation of a model from state space formulation to Python code. Below we present state space representations mathematically, and in subsequent sections we describe their representation in Python code.

ARMA(1, 1) model

Autoregressive moving average models are widespread in the time series literature, so we will assume the reader is familiar with their basic motivation and theory. Suffice it to say, these models are often successfully applied to obtain reduced form estimates of the dynamics exhibited by time series and to produce forecasts. For more details, see any introductory time series text.

An ARMA(1,1) process (where we suppose y_t has already been demeaned) can be written as

$$y_t = \phi y_{t-1} + \varepsilon_t + \theta_1 \varepsilon_{t-1}, \qquad \varepsilon_t \sim N(0, \sigma^2)$$

It is well known that any autoregressive moving average model can be represented in state-space form, and furthermore that there are many equivalent representations. Below we present one possible representation based on Hamilton (1994), with the corresponding notation from (2) given below each matrix.

$$y_t = \underbrace{\begin{bmatrix} 1 & \theta_1 \end{bmatrix}}_{Z} \underbrace{\begin{bmatrix} \alpha_{1,t} \\ \alpha_{2,t} \end{bmatrix}}_{\alpha_t}$$

$$\begin{bmatrix} \alpha_{1,t+1} \\ \alpha_{2,t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} \phi & 0 \\ 1 & 0 \end{bmatrix}}_{T} \begin{bmatrix} \alpha_{1,t} \\ \alpha_{2,t} \end{bmatrix} + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{R} \underbrace{\varepsilon_{t+1}}_{\eta_t}$$

One feature of ARMA(p,q) models generally is that if the assumption of stationarity holds, the Kalman filter can be initialized with the unconditional distribution of the time series.

As an application of this model, in what follows we will consider applying an ARMA(1,1) model to inflation (first difference of the logged US consumer price index). This data can be obtained from the Federal Reserve Economic Database (FRED) produced by the Federal Reserve Bank of St. Louis. In particular, this data can be easily obtained using the Python package pandas-datareader.[5]

```
from pandas_datareader.data import DataReader
cpi = DataReader('CPIAUCNS', 'fred', start='1971-01', end='2016-12')
cpi.index = pd.DatetimeIndex(cpi.index, freq='MS')
inf = np.log(cpi).resample('QS').mean().diff()[1:] * 400
```

[5] This is for illustration purposes only, since an ARMA(1, 1) model with mean zero is not a good model for quarterly CPI inflation.

Fig. 1 shows the resulting time series.

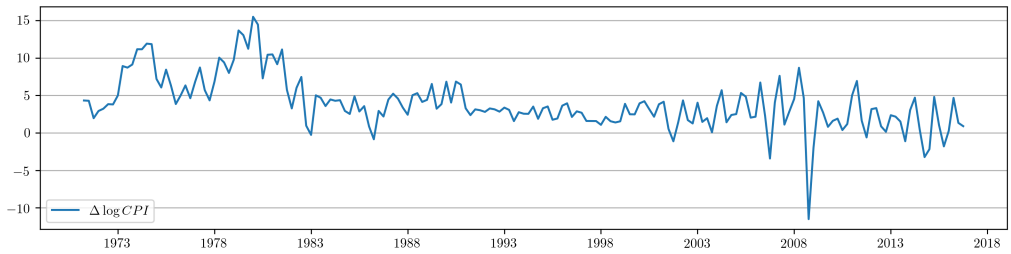

Fig. 1: Time path of US CPI inflation from 1971:Q1 - 2016:Q4.

Local level model

The local level model generalizes the concept of intercept (i.e. "level") in a linear regression to be time-varying. Much has been written about this model, and the second chapter of Durbin and Koopman (2012) is devoted to it. It can be written as

$$y_t = \mu_t + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma_\varepsilon^2)$$
$$\mu_{t+1} = \mu_t + \eta_t, \qquad \eta_t \sim N(0, \sigma_\eta^2)$$

This is already in state space form, with $Z = T = R = 1$. This model is not stationary (the unobserved level follows a random walk), and so stationary initialization of the Kalman filter is impossible. Diffuse initialization, either approximate or exact, is required.

As an application of this model, in what follows we will consider applying the local level model to the annual flow volume of the Nile river between 1871 and 1970. This data is freely available from many sources, and is included in many econometrics analysis packages. Here, we use the data from the Python package Statsmodels.

```
nile = sm.datasets.nile.load_pandas().data['volume']
nile.index = pd.date_range('1871', '1970', freq='AS')
```

Fig. 2 shows the resulting time series.

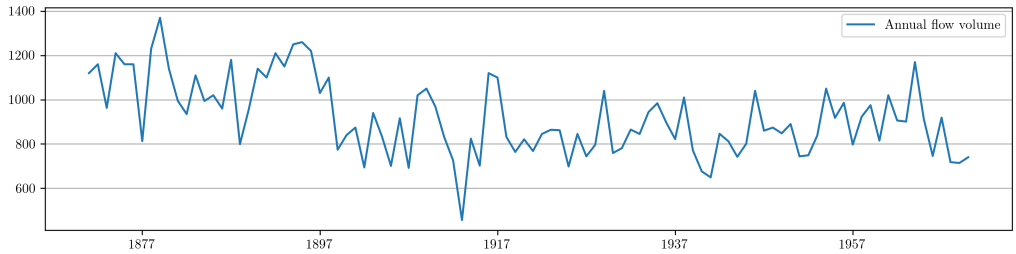

Fig. 2: Annual flow volume of the Nile river 1871 - 1970.

Real business cycle model

Linearized models can often be placed into state space form and evaluated using the Kalman filter. A very simple real business cycle model can be represented as[6]

$$\begin{bmatrix} y_t \\ n_t \\ c_t \end{bmatrix} = \underbrace{\begin{bmatrix} \phi_{yk} & \phi_{yz} \\ \phi_{nk} & \phi_{nz} \\ \phi_{ck} & \phi_{cz} \end{bmatrix}}_{Z} \underbrace{\begin{bmatrix} k_t \\ z_t \end{bmatrix}}_{\alpha_t} + \underbrace{\begin{bmatrix} \varepsilon_{y,t} \\ \varepsilon_{n,t} \\ \varepsilon_{c,t} \end{bmatrix}}_{\varepsilon_t}, \qquad \varepsilon_t \sim N\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \sigma_y^2 & 0 & 0 \\ 0 & \sigma_n^2 & 0 \\ 0 & 0 & \sigma_c^2 \end{bmatrix} \right)$$

$$\begin{bmatrix} k_{t+1} \\ z_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} T_{kk} & T_{kz} \\ 0 & \rho \end{bmatrix}}_{T} \begin{bmatrix} k_t \\ z_t \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{R} \eta_t, \qquad \eta_t \sim N(0, \sigma_z^2)$$

where y_t is output, n_t is hours worked, c_t is consumption, k_t is capital, and z_t is a technology shock process. In this formulation, output, hours worked, and consumption are observable whereas the capital stock and technology process are unobserved. This model can be developed as the

[6] Note that this simple RBC model is presented for illustration purposes and so we aim for brevity and clarity of exposition rather than a state-of-the-art description of the economy.

linearized output of a fully microfounded DSGE model, see for example Ruge-Murcia (2007) or DeJong and Dave (2011). In the theoretical model, the variables are assumed to be stationary.

There are six structural parameters of this RBC model: the discount rate, the marginal disutility of labor, the depreciation rate, the capital-share of output, the technology shock persistence, and the technology shock innovation variance. It is important to note that the reduced form parameters of the state space representation (for example ϕ_{yk}) are complicated and non-linear functions of these underlying structural parameters.

The raw observable data can be obtained from FRED, although it must be transformed to be consistent with the model (for example to induce stationarity). For an explanation of the datasets used and the transformations, see either of the two references above.

```
from pandas_datareader.data import DataReader
start = '1984-01'
end = '2016-09'
labor = DataReader('HOANBS', 'fred',start=start, end=end).resample('QS').first()
cons = DataReader('PCECC96', 'fred', start=start, end=end).resample('QS').first()
inv = DataReader('GPDIC1', 'fred', start=start, end=end).resample('QS').first()
pop = DataReader('CNP16OV', 'fred', start=start, end=end)
pop = pop.resample('QS').mean()  # Convert pop from monthly to quarterly observations
recessions = DataReader('USRECQ', 'fred', start=start, end=end)
recessions = recessions.resample('QS').last()['USRECQ'].iloc[1:]

# Get in per-capita terms
N = labor['HOANBS'] * 6e4 / pop['CNP16OV']
C = (cons['PCECC96'] * 1e6 / pop['CNP16OV']) / 4
I = (inv['GPDIC1'] * 1e6 / pop['CNP16OV']) / 4
Y = C + I

# Log, detrend
y = np.log(Y).diff()[1:]
c = np.log(C).diff()[1:]
n = np.log(N).diff()[1:]
i = np.log(I).diff()[1:]
rbc_data = pd.concat((y, n, c), axis=1)
rbc_data.columns = ['output', 'labor', 'consumption']
```

Fig. 3 shows the resulting time series.

2.8 Parameter estimation

In order to accomodate parameter estimation, we need to introduce a couple of new ideas, since the generic state space model described above considers matrices with known values. In particular

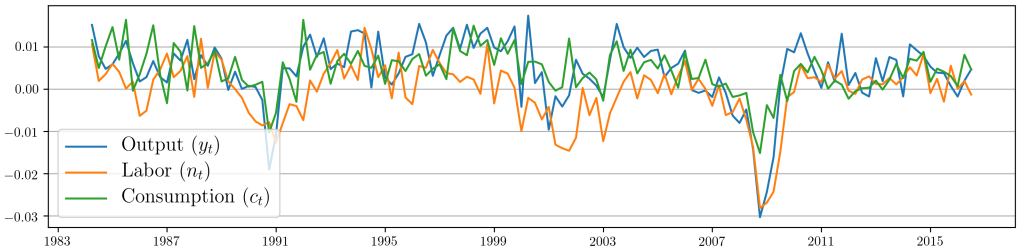

Fig. 3: US output, labor, and consumption time series 1984:Q1 - 2016:Q4.

(following the notation in Chapter 7 of Durbin and Koopman (2012)), suppose that the unknown parameters are collected into a vector ψ. Then each of the state space representation matrices can be considered as, and written as, a function of the parameters. For example, to take into account the dependence on the unknown parameters, we write the design matrix as $Z_t(\psi)$.

The three methods for parameter estimation considered in this paper perform filtering, smoothing, and / or simulation smoothing iteratively, where each iteration has a (generally) different set of parameter values. Given this iterative approach, it is clear that in order to perform parameter estimation we will need two new elements: first, we must have the mappings from parameter values to fully specified system matrices; second, the iterations must begin with some initial parameter values and these must be specified.

The first element has already been introduced in the three examples above, since the state space matrices were written with known values (such as 1 and 0) as well as with unknown parameters (for example ϕ in the ARMA(1,1) model). The second element will be described separately for each of parameter estimation methods, below.

3 Representation in Python

The basic guiding principle for us in translating state space models into Python is to allow users to focus on the specification aspect of their model rather than on the machinery of efficient and accurate filtering and smoothing computation. To do this, we apply the programmatic technique of

object oriented programming (OOP). While a full description and motivation of OOP is beyond the scope of this paper, one of the primary benefits for our purposes is that it facilitates organization and prevents the writing and rewriting of the same or similar code. This feature is quite attractive in general, but as will be shown below, state space models fit particularly well into - and reap substantial benefits from - the object oriented paradigm. For state space models, filtering, smoothing, a large part of parameter estimation, and some postestimation results are standard; they depend only on the generic form of the model given in (2) rather than the specializations found in, for example, (2.7), (2.7), and (2.7)).

The Python programming language is general-purpose, interpreted, dynamically typed, and high-level. Relative to other programming languages commonly used for statistical computation, it has both strengths and weaknesses. It lacks the breadth of available statistical routines present in the R programming language, but instead features a core stack of well-developed scientific libraries. Since it began life as a general purpose programming language, it lacks the native understanding of matrix algebra which makes MATLAB so easy to begin working with (these features are available, but are provided by the the Numeric Python (NumPy) and Scientific Python (SciPy) libraries) but it has more built-in features for working with text, files, web sites, and more. All of Python, R, and MATLAB feature excellent graphing and plotting features and the ability to integrate compiled code for faster performance.

Of course, anything that can be done in one language can in principle be done in many others, so familiarity, style, and tradition play a substantial role in determining which language is used in which discipline. There is much to recommend R, MATLAB, Stata, Julia, and other languages. Nonetheless, it is hoped that this paper will not only show how state space models can be specified and estimated in Python, but also introduce some of the powerful and elegent features of Python that make it a strong candidate for consideration in a wide variety of statistical computing projects.

3.1 Object oriented programming

What follows is a brief description of the concepts of object oriented programming. The content follows Wegner (1990), which may be consulted for more detail. The Python Language Reference may be consulted for details on the implementation and syntax of object oriented programming specific to Python.

Objects are "collections of operations that share a state" (Wegner, 1990). Another way to put it is that objects are collections of data (the "state") along with functions that manipulate or otherwise make use of that data (the "operations"). In Python, the data held by an object are called its *attributes* and the operations are called its *methods*. An example of an object is a point in the Cartesian plane, where we define the "state" of the point as its coordinates in the plane and define two methods, one to change its x-coordinate to $x + dx$, and one to change the y-coordinate to $y + dy$.

Classes are "templates from which objects can be created ... whereas the [attributes of an] object represent *actual* variables, class [attributes] are *potential*, being instantiated only when an object is created" (*Ibid.*). The point object described above could be written in Python code as follows. First, a `Point` class is defined, providing the template for all actual points that will later be represented.

```
# This is the class definition. Object oriented programming has the concept
# of inheritance, whereby classes may be "children" of other classes. The
# parent class is specified in the parentheses. When defining a class with
# no parent, the base class `object` is specified instead.
class Point(object):

    # The __init__ function is a special method that is run whenever an
    # object is created. In this case, the initial coordinates are set to
    # the origin. `self` is a variable which refers to the object instance
    # itself.
    def __init__(self):
        self.x = 0
        self.y = 0

    def change_x(self, dx):
        self.x = self.x + dx

    def change_y(self, dy):
        self.y = self.y + dy
```

With the template defined, we can create as many `Point` objects (instantiations of the `Point`

template), with actual data, as we like. Below, `point_object` holds an actual instance of a point with coordinates first at $(0, 0)$ and then at $(-2, 0)$.

```
# An object of class Point is created
point_object = Point()

# The object exposes it's attributes
print(point_object.x)  # 0

# And we can call the object's methods
# Notice that although `self` is the first argument of the class method,
# it is automatically populated, and we need only specify the other
# argument, `dx`.
point_object.change_x(-2)
print(point_object.x)  # -2
```

Object oriented programming allows code to be organized hierarchically through the concept of class inheritance, whereby a class can be defined as an extension to an existing class. The existing class is called the *parent* and the new class is called the *child*. Wegner (1990) writes "inheritance allows us to reuse the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new [attributes]".

Through the mechanism of inheritance, a parent class can be defined with a set of generic functionality, and then many child classes can subsequently be defined with specializations. Each child thus contains both the generic functionality of the parent class as well as its own specific functionality. Of course the child classes may have children of their own, and so on.

As an example, consider creating a new class describing vectors in $\mathbb{R}^2$. Since a vector can be described as an ordered pair of coordinates, the `Point` class defined above could also be used to describe vectors and allow users to modify the vector using the `change_x` and `change_y` methods. Suppose that we wanted to add a method to calculate the length of the vector. It would not make sense to add a length method to the `Point` class, since a point does not have a length, but we can create a new `Vector` class extending the `Point` class with the new method. In the code below, we also introduce arguments into the class constructor (the `__init__` method).

```
# This is the new class definition. Here, the parent class, `Point`, is in
# the parentheses.
class Vector(Point):

    def __init__(self, x, y):
        # Call the `Point.__init__` method to initialize the coordinates
        # to the origin
        super(Vector, self).__init__()

        # Now change to coordinates to those provided as arguments, using
        # the methods defined in the parent class.
        self.change_x(x)
        self.change_y(y)

    def length(self):
        # Notice that in Python the exponentiation operator is a double
        # asterisk, "**"
        return (self.x**2 + self.y**2)**0.5

# An object of class Vector is created
vector_object = Vector(1, 1)
print(vector_object.length())  # 1.41421356237
```

Returning to state space models and Kalman filtering and smoothing, the object oriented approach allows for separation of concerns and prevents duplication of effort. The base classes contain the functionality common to all state space models, in particular Kalman filtering and smoothing routines, and child classes fill in model-specific parameters into the state space representation matrices. In this way, users need only specify the parts that are absolutely necessary and yet the classes they define contain full state space operations. In fact, many additional features beyond filtering and smoothing are available through the base classes, including methods for estimation of unknown parameters, summary tables, prediction and forecasting, model diagnostics, simulation, and impulse response functions.

3.2 Representation

In this section we present a prototypical example in which we create a subclass specifying a particular model. That subclass then inherits state space functionality from its parent class. Tables detailing the attributes and methods that are available through inheritance of the parent class are provided in *Appendix B: Inherited attributes and methods*.

The parent class is `sm.tsa.statespace.MLEModel` (referred to as simply `MLEModel` in what

follows), and it provides an interface to the state space functionality described above. Subclasses are required to specify the state space matrices of the model they implement (i.e. the elements from Table 1) and in return they receive a number of built-in functions that can be called by users. The most important of these are `update`, `loglike`, `filter`, `smooth`, and `simulation_smoother`.

The first, `update`, accepts as arguments parameters of the model (for example the ϕ autoregressive parameter of the ARMA(1, 1) model) and updates the underlying state space system matrices with those parameters. Note that the second, third, and fourth methods, described just below, implicitly call `update` as part of their operation.

The second, `loglike`, performs the Kalman filter recursions and returns the joint loglikelihood of the sample. The third, `filter`, performs the Kalman filter recursions and returns an object holding the full output of the filter (see Table 2), as well as the state space representation (see Table 1). The fourth, `smooth`, performs Kalman filtering and smoothing recursions and returns an object holding the full output of the smoother (see Table 3) as well as the filtering output and the state space representation. The last, `simulation_smoother`, creates a new object that can be used to create an arbitrary number of simulated state and disturbance series (see Table 4).

The first four methods - `update`, `loglike`, `filter`, and `smooth` - require as their first argument a parameter vector at which to perform the operation. They all first update the state space system matrices, and then the latter three perform the appropriate additional operation. The `simulation_smoother` method does not require the parameter vector as an argument, since it performs simulations based on whatever parameter values have been most recently set, either by one of the other three methods or by the `update` method.

As an example of the use of this class, consider the following code, which constructs a local level model for the Nile data with known parameter values (the next section will consider parameter estimation) and then applies the above methods. Recall that to fully specify a state space model, all of the elements from Table 1 must be set and the Kalman filter must be initialized. For subclasses of `MLEModel`, all state space elements are created as zero matrices of the appropriate shapes; often

only the non-zero elements need be specified.[7]

```
# Create a new class with parent sm.tsa.statespace.MLEModel
class LocalLevel(sm.tsa.statespace.MLEModel):

    # Define the initial parameter vector; see update() below for a note
    # on the required order of parameter values in the vector
    start_params = [1.0, 1.0]

    # Recall that the constructor (the __init__ method) is
    # always evaluated at the point of object instantiation
    # Here we require a single instantiation argument, the
    # observed dataset, called `endog` here.
    def __init__(self, endog):
        super(LocalLevel, self).__init__(endog, k_states=1)

        # Specify the fixed elements of the state space matrices
        self['design', 0, 0] = 1.0
        self['transition', 0, 0] = 1.0
        self['selection', 0, 0] = 1.0

        # Initialize as approximate diffuse, and "burn" the first
        # loglikelihood value
        self.initialize_approximate_diffuse()
        self.loglikelihood_burn = 1

    # Here we define how to update the state space matrices with the
    # parameters. Note that we must include the **kwargs argument
    def update(self, params, **kwargs):
        # Using the parameters in a specific order in the update method
        # implicitly defines the required order of parameters
        self['obs_cov', 0, 0] = params[0]
        self['state_cov', 0, 0] = params[1]

# Instantiate a new object
nile_model_1 = LocalLevel(nile)
```

Three elements of the above code merit discussion. First, we have included a class attribute `start_params`, which will later be used by the model when performing maximum likelihood estimation.[8] Second, note that the signature of the `update` method includes `**kwargs` as an argument. This allows it to accept an arbitrary set of keyword arguments, and this is required to allow handling of parameter transformations (discussed below). It is important to remember that in all subclasses of `MLEModel`, the `update` method signature must include `**kwargs`.

Second, the state space representation matrices are set using so-called "slice notation", such as

[7] More specifically, potentially time-varying matrices are created as zero matrices of the appropriate non-time-varying shape. If a time-varying matrix is required, the whole matrix must be re-created in the appropriate time-varying shape before individual elements may be modified.

[8] It may seem restrictive to require the initial parameter value to be a a class attribute, which is set to a specific value. In practice, the attribute can be replaced with a class *property*, allowing dynamic creation of the attribute's value. In this way the initial parameter vector for an ARMA(p,q) model could, for example, be generated using ordinary least squares.

`self['design']`, rather than the so-called "dot notation" that is usually used for attribute and method access, such as `self.loglikelihood_burn`. Although it is possible to access and set state space matrices and their elements using dot notation, slice notation is strongly recommended for technical reasons.[9] Note that only the state space matrices can be set using slice notation (see Table 9 for the list of attributes that can be set with slice notation).

This class `LocalLevel` fully specifies the local level state space model. At our disposal now are the methods provided by the parent `MLEModel` class. They can be applied as follows.

First, the `loglike` method returns a single number, and can be evaluated at various sets of parameters.

```
# Compute the loglikelihood at values specific to the nile model
print(nile_model_1.loglike([15099.0, 1469.1]))  # -632.537695048

# Try computing the loglikelihood with a different set of values; notice that it is different
print(nile_model_1.loglike([10000.0, 1.0]))  # -687.5456216
```

The `filter` method returns an object from which filter output can be retrieved.

```
# Retrieve filtering output
nile_filtered_1 = nile_model_1.filter([15099.0, 1469.1])
# print the filtered estimate of the unobserved level
print(nile_filtered_1.filtered_state[0])          # [ 1103.34065938  ... 798.37029261 ]
print(nile_filtered_1.filtered_state_cov[0, 0])  # [ 14874.41126432  ... 4032.15794181 ]
```

The `smooth` method returns an object from which smoother output can be retrieved.

```
# Retrieve smoothing output
nile_smoothed_1 = nile_model_1.smooth([15099.0, 1469.1])
# print the smoothed estimate of the unobserved level
print(nile_smoothed_1.smoothed_state[0])          # [ 1107.20389814 ... 798.37029261 ]
print(nile_smoothed_1.smoothed_state_cov[0, 0])  # [ 4015.96493689  ... 4032.15794181 ]
```

Finally the `simulation_smoother` method returns an object that can be used to simulate state

[9] The difference between `self['design', 0, 0] = 1` and `self.design[0,0] = 1` lies in the order of operations. With dot notation (the latter example) first the `self.design` matrix is accessed and then the [0,0] element of that matrix is accessed. With slice notation, a class method (`__setitem__`) is given the matrix name and the [0,0] element simultaneously. Usually there is no difference between the two approaches, but, for example, if the matrix in question has a floating point datatype and the new value is a complex number, then only the real component of that new value will be set in the matrix and a warning will be issued. This problem does not occur with the slice notation.

or disturbance vectors via the `simulate` method.

```
# Retrieve a simulation smoothing object
nile_simsmoother_1 = nile_model_1.simulation_smoother()

# Perform first set of simulation smoothing recursions
nile_simsmoother_1.simulate()
print(nile_simsmoother_1.simulated_state[0, :-1])  # [ 1000.09720165 ... 882.30604412 ]

# Perform second set of simulation smoothing recursions
nile_simsmoother_1.simulate()
print(nile_simsmoother_1.simulated_state[0, :-1])  # [ 1153.62271051 ... 808.43895425 ]
```

Fig. 4 plots the observed data, filtered series, smoothed series, and the simulated level from ten simulations, generated from the above model.

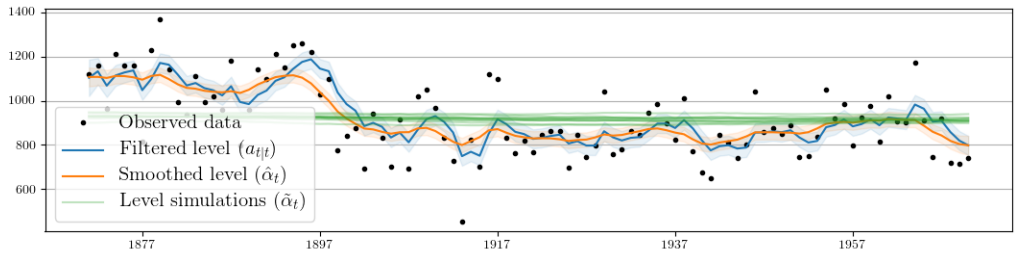

Fig. 4: Filtered and smoothed estimates and simulatations of unobserved level for Nile data.

3.3 Additional remarks

Once a subclass has been created, it has access to a variety of features from the base (parent) classes. A few remarks about available features are merited.

First, if the model is time-invariant, then a check for convergence will be used at each step of the Kalman filter iterations. Once convergence has been achieved, the converged state disturbance covariance matrix, Kalman gain, and forecast error covariance matrix are used at all remaining iterations, reducing the computational burden. The tolerance for determining convergence is controlled by the `tolerance` attribute, which is initially set to 10^{-19} but can be changed by the user. For example, to disable the use of converged values in the model above one could use the code `nile_model_3.tolerance = 0.`

Second, two recent innovations in Kalman filtering are available to handle large-dimensional observations. These include the univariate filtering approach of Koopman and Durbin (2000) and the collapsed approach of Jungbacker and Koopman (2014). The use of these approaches are controlled by the `set_filter_method` method. For example, to enable both of these approaches in the Nile model, one could use the code `nile_model_3.set_filter_method(filter_univariate=True, filter_collapsed=True)` (this is just for illustration, since of course there is only a single variable in that model so that these options would have no practical effect).

Next, options to enable conservation of computer memory (RAM) are available and are controllable via the `set_conserve_memory` method. It should be noted that the usefulness of these options depends on the analysis required by the user because smoothing requires all filtering values and simulation smoothing requires all smoothing and filtering values. However, in maximum likelihood estimation or Metropolis-Hastings posterior simulation, all that is required is the joint likelihood value. One might enable memory conservation until optimal parameters have been found and then disable it so as to calculate any filtered and smoothed values of interest. In Gibbs sampling MCMC approaches, memory conservation is not available because the simulation smoother is required.

Fourth, predictions and impulse response functions are immediately available for any state space model through the filter results object (obtained as the returned value from a `filter` call), through the `predict` and `impulse_responses` methods. These will be demonstrated below.

Fifth, the Kalman filter (and smoothers) are fully equipped to handle missing observation data; no special code is required.

Finally, before moving on to specific parameter estimation methods it is important to note that the simulation smoother object created via the `simulation_smoother` method generates simulations based on the state space matrices as they are defined *when the simulation is performed* and not when the `simulate` method was called. This will be important when implementing Gibbs

sampling MCMC parameter estimation methods. As an illustration, consider the following code:

```
# BEFORE: Perform some simulations with the original parameters
nile_simsmoother_1 = nile_model_1.simulation_smoother()
nile_model_1.update([15099.0, 1469.1])
nile_simsmoother_1.simulate()
# ...

# AFTER: Perform some new simulations with new parameters
nile_model_1.update([10000.0, 1.0])
nile_simsmoother_1.simulate()
# ...
```

Fig. 5 plots ten simulations generated during the BEFORE period, and ten simulations from the AFTER period. It is clear that they are simulating different series, reflecting the different parameters values in place at the time of simulation.

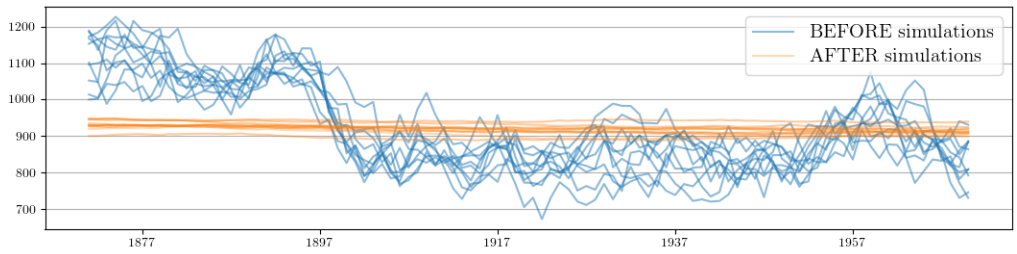

Fig. 5: Simulations of the unobserved level for Nile data under two different parameter sets.

3.4 Practical considerations

As described before, two practical considerations with the Kalman filter are numerical stability and performance. Briefly discussed were the availability of a square-root filter and the use of compiled computer code. In practice, the square-root filter is rarely required, and this Python implementation does not use it. One good reason for this is that "the amount of computation required is substantially larger" (Durbin and Koopman, 2012), and acceptable numerical stability for most models is usually achieved via enforced symmetry of the state covariance matrix (see Grewal and Andrews, 2014, for example).

High performance is achieved primarily through the use of Cython (Behnel et al., 2011). Cython

allows suitably modified Python code to be compiled to C, in some cases (such as the current one) dramatically improving performance. Note that compiled code for performance-critical computation is also available in several of the other Kalman filtering implementations mentioned in the introduction. Other performance-related features, such as the recent advances in filtering with large-dimensional observations described in the preceding section, are also available.

An additional practical consideration whenever computer code is at issue is the possibility of programming errors ("bugs"). McCullough and Vinod (1999) emphasize the need for tests ensuring *accurate* results, as well as good documentation and the availability of source code so that checking for bugs is possible. The source code for this implementation is available, with reasonably extensive inline comments describing procedures. Furthermore, even though the spectre of bugs cannot be fully exorcised, over a thousand "unit tests" have been written, and are available for users to run themselves, comparing output to known results from a variety of outside sources. These tests are run continuously with the software's development to prevent errors from creeping in.

At this point, we once again draw attention to the separation of concerns made possible by the implementation approach pursued here. Although writing the code for a conventional Kalman filter is relatively trivial, writing the code for a Kalman filter, smoother, and simulation smoother using the univariate and collapsed approaches, properly allowing for missing data, and in a compiled language to achieve acceptable performance is not. And yet, for models in state space form, the solution, once created, is entirely generic. The use of an object oriented approach here is what allows users to have the best of both worlds: classes can be custom designed using only Python and yet they contain methods (`loglike`, `filter`, etc.) which have been written and compiled for high performance and tested for accuracy.

3.5 Example models

In this section, we provide code describing the example models in the previous sections. This code is provided to illustrate the above principles in specific models, and it is not necessarily the best

way to develop these models. For example, it is more efficient to develop a single class to handle all ARMA(p,q) models at once rather than separate classes for different orders.[10]

ARMA(1, 1) model

The following code is a straightforward translation of (2.7). Notice that here the state dimension is 2 but the dimension of the state disturbance is only 1; this is represented in the code by setting `k_states=2` but `k_posdef=1`.[11] Also demonstrated is the possibility of specifying the Kalman filter initialization in the class construction call with the argument `initialization='stationary'`.[12]

```
class ARMA11(sm.tsa.statespace.MLEModel):

    start_params = [0, 0, 1]

    def __init__(self, endog):
        super(ARMA11, self).__init__(
            endog, k_states=2, k_posdef=1, initialization='stationary')

        self['design', 0, 0] = 1.
        self['transition', 1, 0] = 1.
        self['selection', 0, 0] = 1.

    def update(self, params, **kwargs):
        self['design', 0, 1] = params[1]
        self['transition', 0, 0] = params[0]
        self['state_cov', 0, 0] = params[2]

# Example of instantiating a new object, updating the parameters to the
# starting parameters, and evaluating the loglikelihood
inf_model = ARMA11(inf)
print(inf_model.loglike(inf_model.start_params))  # -2682.72563702
```

[10] See the `SARIMAX` class described in *Out-of-the-box models* for a fully featured class built-in to Statsmodels that allows estimating a large set of models, including ARMA(p, q).

[11] The dimension of the state disturbance is named `k_posdef` because the selected state disturbance vector is given not by η_t but by $R_t \eta_t$. The dimension of the selected state disturbance vector is always equal to the dimension of the state, but the selected state disturbance covariance matrix will be have `k_states - k_posdef` zero-eigenvalues. Thus the dimension of the state disturbance gives the dimension of the subset of the selected state disturbance for which a positive definite covariance matrix; hence the name `k_posdef`.

[12] Of course the assumption of stationarity would be violated for certain parameter values, for example if $\phi = 1$. This has important implications for parameter estimation where we typically want to only allow parameters inducing a stationary model. This is discussed in the specific sections on parameter estimation.

Local level model

The class for the local level model was defined in the previous section.

Real business cycle model

The real business cycle model is specified by the equations (2.7). It again has a state dimension of 2 and a state disturbance dimension of 1, and again the process is assumed to be stationary. Unlike the previous examples, here the (structural) parameters of the model do not map directly to elements of the system matrices. As described in the definition of the RBC model, the thirteen reduced form parameters found in the state space matrices are non-linear functions of the six structural parameters. We want to set up the model in terms of the structural parameters and use the `update` method to perform the appropriate transformations to retrieve the reduced form parameters. This is important because the theory does not allow the reduced form parameters to vary arbitrarily; in particular, only certain combinations of the reduced form parameters are consistent with generation from the underlying structural parameters through the model.

Solving the structural model for the reduced form parameters in terms of the structural parameters requires the solution of a linear rational expectations model, and a full description of this process is beyond the scope of this paper. This particular RBC model can be solved using the method of Blanchard and Kahn (1980); more general solution methods exist for more general models (see for example Klein (2000) and Sims (2002)).

Regardless of the method used, for many linear (or linearized) models the solution will be in state space form and so the state space matrices can be updated with the reduced form parameters. For expositional purposes, the following code snippet is not complete, but shows the general formulation in Python. A complete version of the class is found in *Appendix C: Real business cycle model code*.

```
class SimpleRBC(sm.tsa.statespace.MLEModel):

    start_params = [...]

    def __init__(self, endog):
        super(SimpleRBC, self).__init__(
            endog, k_states=2, k_posdef=1, initialization='stationary')

        # Initialize RBC-specific variables, parameters, etc.
        # ...

        # Setup fixed elements of the statespace matrices
        self['selection', 1, 0] = 1

    def solve(self, structural_params):
        # Solve the RBC model
        # ...

    def update(self, params, **kwargs):
        params = super(SimpleRBC, self).update(params, **kwargs)

        # Reconstruct the full parameter vector from the
        # estimated and calibrated parameters
        structural_params = ...
        measurement_variances = ...

        # Solve the model
        design, transition = self.solve(structural_params)

        # Update the statespace representation
        self['design'] = design
        self['obs_cov', 0, 0] = measurement_variances[0]
        self['obs_cov', 1, 1] = measurement_variances[1]
        self['obs_cov', 2, 2] = measurement_variances[2]
        self['transition'] = transition
        self['state_cov', 0, 0] = structural_params[...]
```

4 Maximum Likelihood Estimation

Classical estimation of parameters in state space models is possible because the likelihood is a byproduct of the filtering recursions. Given a set of initial parameters, numerical maximization techniques, often quasi-Newton methods, can be applied to find the set of parameters that maximize (locally) the likelihood function, $\mathcal{L}(Y_n \mid \psi)$. In this section we describe how to apply maximum likelihood estimation (MLE) to state space models in Python. First we show how to apply a minimization algorithm in SciPy to maximize the likelihood, using the `loglike` method. Second, we show how the underlying Statsmodels functionality inherited by our subclasses can be used to greatly streamline estimation.

In particular, models extending from the `sm.tsa.statespace.MLEModel` ("`MLEModel`") class can painlessly perform maximum likelihood estimation via a `fit` method. In addition, summary tables, postestimation results, and model diagnostics are available. *Appendix B: Inherited attributes and methods* describes all of the methods and attributes that are available to subclasses of `MLEModel` and to results objects.

4.1 Direct approach

Numerical optimziation routines in Python are available through the Python package SciPy (Jones et al., 2001). Generically, these are in the form of minimizers that accept a function and a set of starting parameters and return the set of parameters that (locally) minimize the function. There are a number of available algorithms, including the popular BFGS (Broyden–Fletcher–Goldfarb–Shannon) method. As is usual when minimization routines are available, in order to maximize the (log) likelihood, we minimize its negative.

The code below demonstrates how to apply maximum likelihood estimation to the `LocalLevel` class defined in the previous section for the Nile dataset. In this case, because we have not bothered to define good starting parameters, we use the Nelder-Mead algorithm that can be more robust than BFGS although it may converge more slowly.

```
# Load the generic minimization function from scipy
from scipy.optimize import minimize

# Create a new function to return the negative of the loglikelihood
nile_model_2 = LocalLevel(nile)
def neg_loglike(params):
    return -nile_model_2.loglike(params)

# Perform numerical optimization
output = minimize(neg_loglike, nile_model_2.start_params, method='Nelder-Mead')

print(output.x)  # [ 15108.31   1463.55]
print(nile_model_2.loglike(output.x))  # -632.537685587
```

The maximizing parameters are very close to those reported by Durbin and Koopman (2012) and achieve a negligibly higher loglikelihood (-632.53769 versus -632.53770).

4.2 Integration with Statsmodels

While likelihood maximization itself can be performed relatively easily, in practice there are often many other desired quantities aside from just the optimal parameters. For example, inference often requires measures of parameter uncertainty (standard errors and confidence intervals). Another issue that arises is that it is most convenient to allow the numerical optimizer to choose parameters across the entire real line. This means that some combinations of parameters chosen by the optimizer may lead to an invalid model specification. It is sometimes possible to use an optimization procedure with constraints or bounds, but it is almost always easier to allow the optimizer to choose in an unconstrained way and then to transform the parameters to fit the model. The implementation of parameter transformations will be discussed at greater length below.

While there is no barrier to users calculating those quantities or implementing transformations, the calculations are standard and there is no reason for each user to implement them separately. Again we turn to the principle of separation of concerns made possible through the object oriented programming approach, this time by making use of the tools available in Statsmodels. In particular, a new method, `fit`, is available to automatically perform maximum likelihood estimation using the starting parameters defined in the `start_params` attribute (see above) and returns a results object.

The following code further refines the local level model by adding a new attribute `param_names` that augments output with descriptive parameter names. There is also a new line in the `update` method that implements parameter transformations: the `params` vector is replaced with the output from the `update` method of the parent class (`MLEModel`). If the parameters are not already transformed, the parent `update` method calls the appropriate transformation functions and returns the transformed parameters. In this class we have not yet defined any transformation functions, so the parent `update` method will simply return the parameters it was given. Later we will improve the class to force the variance parameter to be positive.

```
class FirstMLELocalLevel(sm.tsa.statespace.MLEModel):
    start_params = [1.0, 1.0]
    param_names = ['obs.var', 'level.var']

    def __init__(self, endog):
        super(FirstMLELocalLevel, self).__init__(endog, k_states=1)

        self['design', 0, 0] = 1.0
        self['transition', 0, 0] = 1.0
        self['selection', 0, 0] = 1.0

        self.initialize_approximate_diffuse()
        self.loglikelihood_burn = 1

    def update(self, params, **kwargs):
        # Transform the parameters if they are not yet transformed
        params = super(FirstMLELocalLevel, self).update(params, **kwargs)

        self['obs_cov', 0, 0] = params[0]
        self['state_cov', 0, 0] = params[1]
```

With this new definition, we can instantiate our model and perform maximum likelihood estimation. As one feature of the integration with Statsmodels, the result object has a `summary` method that prints a table of results:

```
nile_mlemodel_1 = FirstMLELocalLevel(nile)

print(nile_mlemodel_1.loglike([15099.0, 1469.1]))  # -632.537695048

# Again we use Nelder-Mead; now specified as method='nm'
nile_mleresults_1 = nile_mlemodel_1.fit(method='nm', maxiter=1000)
print(nile_mleresults_1.summary())
```

```
                           Statespace Model Results
==============================================================================
Dep. Variable:                  volume   No. Observations:                  100
Model:             FirstMLELocalLevel   Log Likelihood                -632.538
Date:                Sat, 28 Jan 2017   AIC                           1269.075
Time:                        15:19:50   BIC                           1274.286
Sample:                    01-01-1871   HQIC                          1271.184
                         - 01-01-1970
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
obs.var     1.511e+04   2586.966      5.840      0.000       1e+04    2.02e+04
level.var   1463.5472    843.717      1.735      0.083    -190.109    3117.203
===================================================================================
Ljung-Box (Q):                       36.00   Jarque-Bera (JB):                 0.05
Prob(Q):                              0.65   Prob(JB):                         0.98
Heteroskedasticity (H):               0.61   Skew:                            -0.03
Prob(H) (two-sided):                  0.16   Kurtosis:                         3.08
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

A second feature is the availability of model diagnostics. Test statistics for tests of the standardized residuals for normality, heteroskedasticity, and serial correlation are reported at the bottom of the summary output. Diagnostic plots can also be produced using the `plot_diagnostics` method, illustrated in Fig. 6.[13] Notice that Statsmodels is aware of the date index of the Nile dataset and uses that information in the summary table and diagnostic plots.

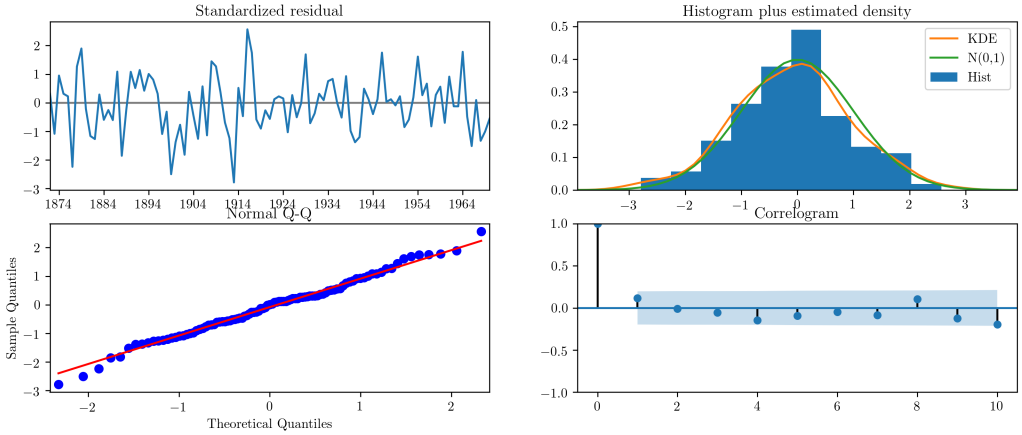

Fig. 6: Diagnostic plots for standardised residuals after maximum likelihood estimation on Nile data.

A third feature is the availability of forecasting (through the `get_forecasts` method) and impulse response functions (through the `impulse_responses` method). Due to the nature of the local level model these are uninteresting here, but will be exhibited in the ARMA(1,1) and real business cycle examples below.

Parameter transformations

As mentioned above, parameter transformations are an important component of maximum likelihood estimation in a wide variety of cases. For example, in the local level model above the two estimated parameters are variances, which cannot theoretically be negative. Although the optimizer avoided the problematic regions in the above example, that will not always be the case. As

[13] See sections 2.12 and 7.5 of Durbin and Koopman (2012) for a description of the standardized residuals and the definitions of the provided diagnostic tests.

another example, ARMA models are typically assumed to be stationary. This requires coefficients that permit inversion of the associated lag polynomial. Parameter transformations can be used to enforce these and other restrictions.

For example, if an unconstrained variance parameter is squared the transformed variance parameter will always be positive. Monahan (1984) and Ansley and Kohn (1986) describe transformations sufficient to induce stationarity in the univariate and multivariate cases, respectively, by taking advantage of the one-to-one correspondence between lag polynomial coefficients and partial autocorrelations.[14]

It is strongly preferred that the transformation function have a well-defined inverse so that starting parameters can be specified in terms of the model space and then "untransformed" to appropriate values in the unconstrained space.

Implementing parameter transformations when using `MLEModel` as the base class is as simple as adding two new methods: `transform_params` and `untransform_params` (if no parameter transformations as required, these methods can simply be omitted from the class definition). The following code redefines the local level model again, this time to include parameter transformations to ensure positive variance parameters.[15]

[14] The transformations to induce stationarity are made available in this package as the functions `sm.tsa.statespace.tools.constrain_stationary_univariate` and `sm.tsa.statespace.tools.constrain_stationary_multivariate`. Their inverses are also available.

[15] Note that in Python, the exponentiation operator is `**`.

```
class MLELocalLevel(sm.tsa.statespace.MLEModel):
    start_params = [1.0, 1.0]
    param_names = ['obs.var', 'level.var']

    def __init__(self, endog):
        super(MLELocalLevel, self).__init__(endog, k_states=1)

        self['design', 0, 0] = 1.0
        self['transition', 0, 0] = 1.0
        self['selection', 0, 0] = 1.0

        self.initialize_approximate_diffuse()
        self.loglikelihood_burn = 1

    def transform_params(self, params):
        return params**2

    def untransform_params(self, params):
        return params**0.5

    def update(self, params, **kwargs):
        # Transform the parameters if they are not yet transformed
        params = super(MLELocalLevel, self).update(params, **kwargs)

        self['obs_cov', 0, 0] = params[0]
        self['state_cov', 0, 0] = params[1]
```

All of the code given above then applies equally to this new model, except that this class is robust to the optimizer selecting negative parameters.

4.3 Example models

In this section, we extend the code from *Representation in Python* to allow for maximum likelihood estimation through Statsmodels integration.

ARMA(1, 1) model

```
from statsmodels.tsa.statespace.tools import (constrain_stationary_univariate,
                                              unconstrain_stationary_univariate)

class ARMA11(sm.tsa.statespace.MLEModel):
    start_params = [0, 0, 1]
    param_names = ['phi', 'theta', 'sigma2']

    def __init__(self, endog):
        super(ARMA11, self).__init__(
            endog, k_states=2, k_posdef=1, initialization='stationary')

        self['design', 0, 0] = 1.
        self['transition', 1, 0] = 1.
        self['selection', 0, 0] = 1.

    def transform_params(self, params):
        phi = constrain_stationary_univariate(params[0:1])
        theta = constrain_stationary_univariate(params[1:2])
        sigma2 = params[2]**2
        return np.r_[phi, theta, sigma2]

    def untransform_params(self, params):
        phi = unconstrain_stationary_univariate(params[0:1])
        theta = unconstrain_stationary_univariate(params[1:2])
        sigma2 = params[2]**0.5
        return np.r_[phi, theta, sigma2]

    def update(self, params, **kwargs):
        # Transform the parameters if they are not yet transformed
        params = super(ARMA11, self).update(params, **kwargs)

        self['design', 0, 1] = params[1]
        self['transition', 0, 0] = params[0]
        self['state_cov', 0, 0] = params[2]
```

The parameters can now be easily estimated via maximum likelihood using the `fit` method. This model also allows us to demonstrate the prediction and forecasting features provided by the Statsmodels integration. In particular, we use the `get_prediction` method to retrieve a prediction object that gives in-sample one-step-ahead predictions and out-of-sample forecasts, as well as confidence intervals. Fig. 7 shows a graph of the output.

```
inf_model = ARMA11(inf)
inf_results = inf_model.fit()

inf_forecast = inf_results.get_prediction(start='2005-01-01', end='2020-01-01')
print(inf_forecast.predicted_mean)  # [2005-01-01   2.439005 ...
print(inf_forecast.conf_int())      # [2005-01-01   -2.573556 7.451566 ...
```

If only out-of-sample forecasts had been desired, the `get_forecasts` method could have been used instead, and if only the forecasted values had been desired (and not additional results like confidence intervals), the methods `predict` or `forecast` could have been used.

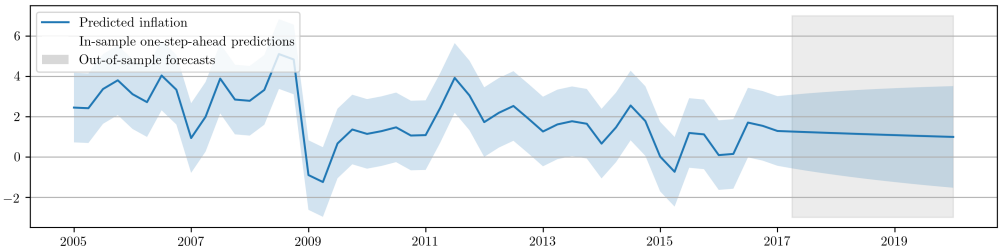

Fig. 7: In-sample one-step-ahead predictions and out-of-sample forecasts for ARMA(1,1) model on US CPI inflation data.

Local level model

See the previous sections for the Python implementation of the local level model.

Real business cycle model

Due to the the complexity of the model, the full code for the model is too long to display inline, but it is provided in the *Appendix C: Real business cycle model code*. It implements the real business cycle model in a class named `SimpleRBC` and allows selecting some of the structural parameters to be estimated while allowing others to be calibrated (set to specific values).

Often in structural models one of the outcomes of interest is the time paths of the observed variables following a hypothetical structural shock; these time paths are called impulse response functions, and they can be generated for any state space model.

In the first application, we will calibrate all of the structural parameters to the values suggested in Ruge-Murcia (2007) and simply estimate the measurement error variances (these do not affect the model dynamics or the impulse responses). Once the model has been estimated, the `impulse_responses` method can be used to generate the time paths.

```
# Calibrate everything except measurement variances
calibrated = {
    'discount_rate': 0.95,
    'disutility_labor': 3.0,
    'capital_share': 0.36,
    'depreciation_rate': 0.025,
    'technology_shock_persistence': 0.85,
    'technology_shock_var': 0.04**2
}
calibrated_mod = SimpleRBC(rbc_data, calibrated=calibrated)
calibrated_res = calibrated_mod.fit()

calibrated_irfs = calibrated_res.impulse_responses(40, orthogonalized=True) * 100
```

The calculated impulse responses are displayed in Fig. 8. By calibrating fewer parameters we can expand estimation to include some of the structural parameters. For example, we may consider also estimating the two parameters describing the technology shock. Implementing this only requires eliminating the last two elements from the `calibrated` dictionary. The impulse responses corresponding to this second exercise are displayed in Fig. 9.[16]

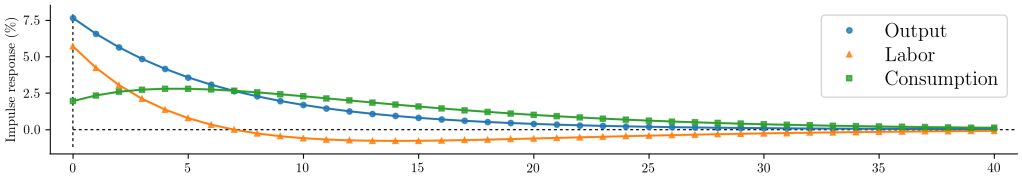

Fig. 8: Impulse response functions corresponding to a fully calibrated RBC model.

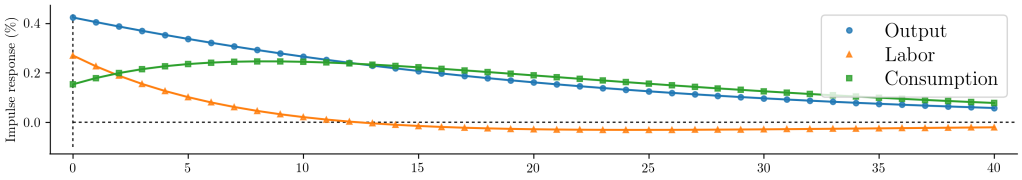

Fig. 9: Impulse response functions corresponding to a partially estimated RBC model.

Recall that the RBC model has three observables, output, labor, and consumption, and two unobserved states, capital and the technology process. The Kalman filter provides optimal estimates of these unobserved series at time t based on on all data up to time t, and the state smoother provides

[16] We note again that this example is merely by way of illustration; it does not represent best-practices for careful RBC estimation.

optimal estimates based on the full dataset. These can be retrieved from the results object. Fig. 10 displays the smoothed state values and confidence intervals for the partially estimated case.

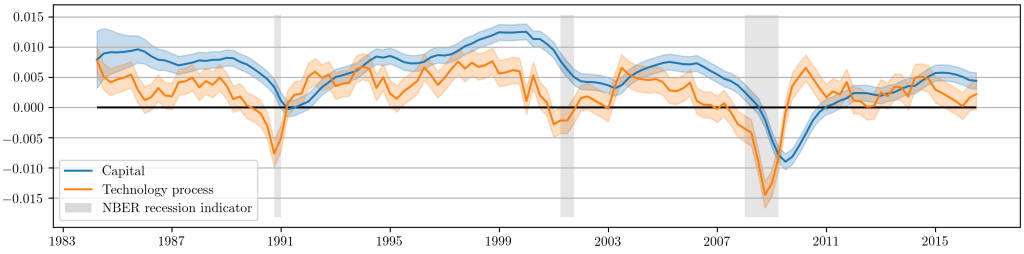

Fig. 10: Smoothed estimates of capital and the technology process from the partially estimated RBC model.

5 Posterior Simulation

State space models are also amenable to parameter estimation by Bayesian methods. We consider posterior simulation by Markov chain Monte Carlo (MCMC) methods, and in particular using the Metropolis-Hastings and Gibbs sampling algorithms. This section describes how to use the above models in Bayesian estimation, but fortunately no further modifications need be made; classes defined as in the maximum likelihood section (i.e. classes that extend from `sm.tsa.statespace.MLEModel`) can be used for either maximum likelihood estimation or Bayesian estimation. Thus the example code here is only tasked with *applying* the previously defined state space models.

A full discussion of Bayesian techniques is beyond the scope of this paper, but interested readers can consult Koop (2003) for a general introduction to Bayesian econometrics, West and Harrison (1999) for a comprehensive Bayesian approach to state space models, and Kim and Nelson (1999) for a excellent practical text on parameter estimation in state space models. The following introduction to Bayesian methods is drawn from these references.

The Bayesian approach to parameter estimation begins by considering parameters as random vari-

ables. Bayes' theorem is applied to derive a distribution for the parameters conditional on the observed data. This "posterior" distribution is proportional to the likelihood function multiplied by a "prior" distribution for the parameters. The prior summarizes all information the researcher has on the parameter values prior to observing the data. Denoting the prior as $\pi(\psi)$, the likelihood function as $\mathcal{L}(Y_n \mid \psi)$, and the posterior as $\pi(\psi \mid Y_n)$, we have

$$\pi(\psi \mid Y_n) \propto \mathcal{L}(Y_n \mid \psi)\pi(\psi)$$

The posterior distribution is the quantity of interest; the difficulty of working with it depends on the prior specified by the researcher and the likelihood function entailed by the selected model. In specific cases (for example the special case of "conjugate priors") the analytic form of the posterior distribution can be found and used for analysis directly. More often the posterior is not available analytically so other methods must be used to explore its properties.

Posterior simulation is a method available when a procedure exists to *sample* from the posterior distribution even though the analytic form of the distribution may not be known. Posterior simulation considers drawing samples $\psi_s, s = 1 \ldots S$. Under fairly weak conditions a law of large numbers can be applied so that, given the S samples, sample averages can be used to approximate population quantities

$$\frac{1}{S}\sum_{s=1}^{S} g(\psi_s) \rightarrow \int g(\psi)\pi(\psi \mid Y_n)d\psi = E_{\pi(\cdot|Y_n)}\left[g(\psi)\right]$$

For example, the posterior mean is often of interest and corresponds to $g(\psi) = \psi$. Histograms can be used to examine the shapes of the marginal distributions of individual parameters.

It may seem that sampling from an unknown distribution is impossible, but MCMC methods allow the *eventual* sampling from an unknown distribution by applying an algorithm designed to ensure that the unknown distribution is an invariant distribution of a Markov chain. The Markov chain is initialized with an arbitrary value, and then a transition density, denoted $f(\psi_s \mid \psi_{s-1})$, is applied

to draw subsequent values conditional only on the previous value. The appropriate selection of the transition densities can usually ensure that there exists some value $\hat{s}$ such that every subsequently drawn sample $\psi_s,\ s > \hat{s}$ is marginally distributed according to the unknown distribution of interest.[17] The two methods discussed below differ in the specification of the transition density.

5.1 Markov chain Monte Carlo algorithms

Metropolis-Hastings algorithm[18]

The Metropolis-Hastings algorithm is a very general strategy for constructing a Markov chain with the desired invariant distribution. The transition density is specified in the following way:

1. Given the current value of the chain, ψ_{s-1}, a proposal value, ψ^*, is selected according to a proposal $q(\psi; \psi_{s-1})$ which is a fixed density function for a given value ψ_{s-1}.

2. With probability $\alpha(\psi_{s-1}, \psi^*)$ (defined below) the proposed value is accepted so that the next value of the chain is set to $\psi_s = \psi^*$; if it is not accepted, the chain remains in place $\psi_s = \psi_{s-1}$.

$$\alpha(\psi_{s-1}, \psi^*) = \min \left\{ \frac{\pi(\psi^* \mid Y_n) q(\psi^*; \psi_{s-1})}{\pi(\psi_{s-1} \mid Y_n) q(\psi_{s-1}; \psi^*)}, 1 \right\}$$

Practically speaking, the important component of this algorithm is that only the ratio of posterior quantities is required. Recalling from above that the posterior is proportional to the likelihood and the prior we can rewrite the probability of acceptance as

$$\alpha(\psi_{s-1}, \psi^*) = \min \left\{ \frac{\mathcal{L}(Y_n \mid \psi^*) \pi(\psi^*) q(\psi^*; \psi_{s-1})}{\mathcal{L}(Y_n \mid \psi_{s-1}) \pi(\psi_{s-1}) q(\psi_{s-1}; \psi^*)}, 1 \right\}$$

Given a particular specification for the prior and proposal distributions, *this ratio can be computed*,

[17] Of course the value $\hat{s}$ is unknown and can in some cases be quite large, although statistical tests do exist that can explore this issue.

[18] This discussion is somewhat loose; see Tierney (1994) and Chib and Greenberg (1995) for careful treatments.

where the likelihood function is evaluated as a byproduct of the Kalman filter iterations. In the special case that the proposal distribution satistifes $q(\psi_{s-1}; \psi^*) = q(\psi^*; \psi_{s-1})$ (as will be the case in the examples below), we can again rewrite the probabilty of acceptance as

$$\alpha(\psi_{s-1}, \psi^*) = \min \left\{ \frac{\mathcal{L}(Y_n \mid \psi^*)\pi(\psi^*)}{\mathcal{L}(Y_n \mid \psi_{s-1})\pi(\psi_{s-1})}, 1 \right\}$$

One convenient choice of proposal distribution that allows this is the so-called random walk proposal with Gaussian increment, defined such that

$$\psi^* = \psi_{s-1} + \epsilon_s, \qquad \epsilon_s \sim N(0, \Sigma_\epsilon)$$

Notice that to use this proposal distribution, we must set the variance Σ_ϵ. This is often calibrated to achieve some target acceptance rate (ratio of accepted to rejected draws); see the references above for more details.

Gibbs sampling algorithm

Suppose that we can block the parameter vector into K subvectors, so that $\psi = \{\psi^{(1)}, \psi^{(2)}, \ldots, \psi^{(K)}\}$, and further suppose that all *conditional* posterior distributions of the form $\pi(\psi^{(k)} \mid \psi^{(-k)}, Y_n),\ k = 1, \ldots, K$ can be sampled from. Then the transition density moving from ψ_{s-1} to ψ_s can be defined as follows:

1. Given the current value of the chain ψ_{s-1}, sample $\psi_s^{(1)}$ according to the density $\pi(\psi^{(1)} \mid \psi_{s-1}^{(-1)}, Y_n)$.
2. Sample $\psi_s^{(2)}$ according to the density $\pi(\psi^{(1)} \mid \psi_{s-1}^{(-1,2)}, \psi_s^{(1)}, Y_n)$
3. [repeat for $k = 3, \ldots, K$]
4. Then $\psi_s = \{\psi_s^{(1)}, \psi_s^{(2)}, \ldots, \psi_s^{(K)}\}$

In the case of state space models, we can augment the parameter vector to include the unobserved states. Notice then that the conditional posterior distribution for the states is exactly the distribution

from which the simulation smoother produces simulated states; i.e. $\tilde{\alpha}$ is drawn according to $\pi(\alpha \mid \psi, Y_n)$.

The conditional distributions for the parameter vector must be identified on a case-by-case basis. However, notice that the conditional posterior distribution conditions on the unobserved states, so that in many cases the conditional distributions follow from well known econometric problems. For example, if the observation covariance matrix is diagonal, the rows of the observation equation can be viewed as equation-by-equation OLS.

Metropolis-within-Gibbs sampling algorithm

In the case that the parameter vector can be blocked as above but some of the conditional posterior distributions cannot be directly sampled from, a hybrid MCMC approach can be taken. The Gibbs sampling algorithm is used as defined above, except that for any block k such that the conditional posterior cannot be sampled from, the Metropolis-Hastings algorithm is applied for that block (i.e. a proposal is generated and accepted with the probability defined above).

5.2 Implementing Metropolis-Hastings: the local level model

In this section we describe implementing the Metropolis-Hastings algorithm to estimate unknown parameters of a state space model. First, it is illuminating to consider a direct approach where all code is explicit. Second, we consider using the another Python library (PyMC) to streamline the estimation process.

The local level, as written above, has two variance parameters σ_ε^2 and σ_η^2. In practice we will sample the standard deviations σ_ε and σ_η. Recalling the Metropolis-Hastings algorithm, in order to proceed we will need to evaluate the likelihood and the prior and specify a proposal distribution. The likelihood will be evaluated using the Kalman filter via the `loglike` method introduced earlier. The parameters are chosen to have independent inverse-gamma priors, with the shape and scale

Table 5: Priors for the local level model applied to Nile data.

Parameter	Prior distribution	Shape	Scale	Prior mean	Prior variance
σ_ε	Inverse-gamma	3	300	150	22,500
σ_η	Inverse-gamma	3	120	60	3,600

parameters set as in Table 5.[19] We will use the random walk proposal, which simply requires drawing a value from a multivariate normal distribution each iteration. We set the variance of the random walk innovation to be the identity matrix times ten. The prior densities can be evaluated and variates drawn from the multivariate normal using the Python package SciPy.

For each iteration, the acceptance probability can be calculated from the above elements, and the decision to accept or reject can be made by comparing the acceptance probability to a random variate from a standard uniform distribution.

Direct approach

Given the existence of the local level class (`MLELocalLevel`) for calculating the loglikelihood, the code for performing an MCMC exercise is relatively simple. First, we initialize the priors and the proposal distribution

```
from scipy.stats import multivariate_normal, invgamma, uniform

# Create the model for likelihood evaluation
model = MLELocalLevel(nile)

# Specify priors
prior_obs = invgamma(3, scale=300)
prior_level = invgamma(3, scale=120)

# Specify the random walk proposal
rw_proposal = multivariate_normal(cov=np.eye(2)*10)
```

Next, we perform 10,000 Metropolis-Hastings iterations as follows. The resultant histograms and

[19] To be clear, since there are multiple ways to parameterize the inverse-gamma distribution, with $x \sim \text{IG}(\alpha, \beta)$ the density we consider is

$$p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} e^{-\frac{\beta}{x}}$$

traces in terms of the variances, as well as a plot of the acceptance ratio over the iterations, are given in Fig. 11.[20]

```
# Create storage arrays for the traces
n_iterations = 10000
trace = np.zeros((n_iterations + 1, 2))
trace_accepts = np.zeros(n_iterations)
trace[0] = [120, 30]  # Initial values

# Iterations
for s in range(1, n_iterations + 1):
    proposed = trace[s-1] + rw_proposal.rvs()

    acceptance_probability = np.exp(
        model.loglike(proposed**2) - model.loglike(trace[s-1]**2) +
        prior_obs.logpdf(proposed[0]) + prior_level.logpdf(proposed[1]) -
        prior_obs.logpdf(trace[s-1, 0]) - prior_level.logpdf(trace[s-1, 1]))

    if acceptance_probability > uniform.rvs():
        trace[s] = proposed
        trace_accepts[s-1] = 1
    else:
        trace[s] = trace[s-1]
```

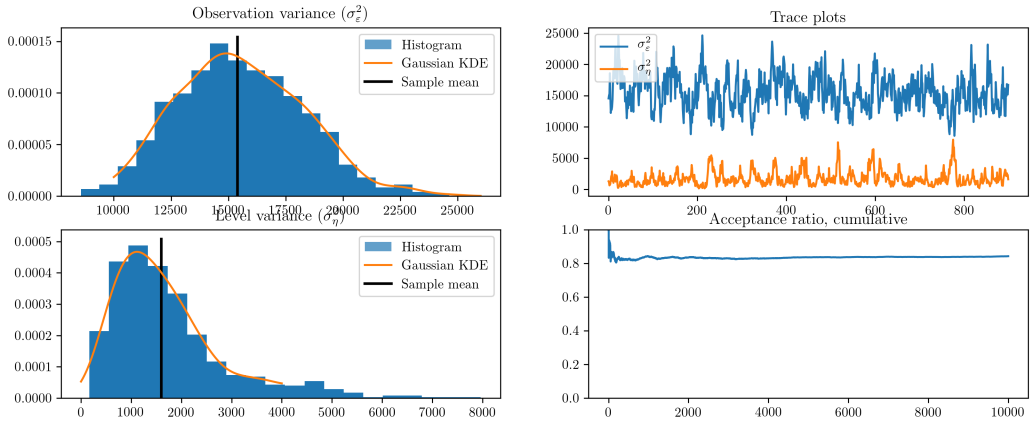

Fig. 11: Output from Metropolis-Hastings posterior simulation on Nile data.

Integration with PyMC

Parameters can also be simply estimated by taking advantage of the PyMC library (Patil et al., 2010). A full discussion of the features and use of this library is beyond the scope of this paper and

[20] The output figures are ultimately based on 900 simulated values for each parameter. Of the 10,000 simulations performed, the first 1,000 were eliminated as the burn-in period and the remaining 9,000 were thinned by only taking each 10th sample, to reduce the effects of autocorrelated draws.

instead we only introduce the features we need for estimation of this model. A similar approach would handle most state space models, and the PyMC documentation can be consulted for more advanced usage, including sophisticated sampling techniques such as slice sampling and No-U-Turn sampling.

As above, we need to create objects representing the selected priors and an object representing the likelihood function. The former are referred to by PyMC as "stochastic" elements, and the latter as a "data" element (which is a stochastic element that has already been "observed" and so is not sampled from). The priors and likelihood function using the `MLELocalLevel` class defined above can be implemented with PyMC in the following way

```
import pymc as mc

# Priors as "stochastic" elements
prior_obs = mc.InverseGamma('obs', 3, 300)
prior_level = mc.InverseGamma('level', 3, 120)

# Create the model for likelihood evaluation
model = MLELocalLevel(nile)

# Create the "data" component (stochastic and observed)
@mc.stochastic(dtype=sm.tsa.statespace.MLEModel, observed=True)
def loglikelihood(value=model, obs_std=prior_obs, level_std=prior_level):
    return value.loglike([obs_std**2, level_std**2])
```

We do not need to explicitly specify the proposal; PyMC uses an adaptive proposal by default. Instead, we simply need to create a "model", which unifies the priors and likelihood, and a "sampler". The sampler is an object used to perform the simulations and return the trace objects. The resultant histograms and traces in terms of the variances from 10,000 iterations are given in Fig. 12.[21]

```
# Create the PyMC model
pymc_model = mc.Model((prior_obs, prior_level, loglikelihood))

# Create a PyMC sample and perform sampling
sampler = mc.MCMC(pymc_model)
sampler.sample(iter=10000, burn=1000, thin=10)
```

[21] The acceptance ratio is not provided by PyMC when the adaptive proposal is used.

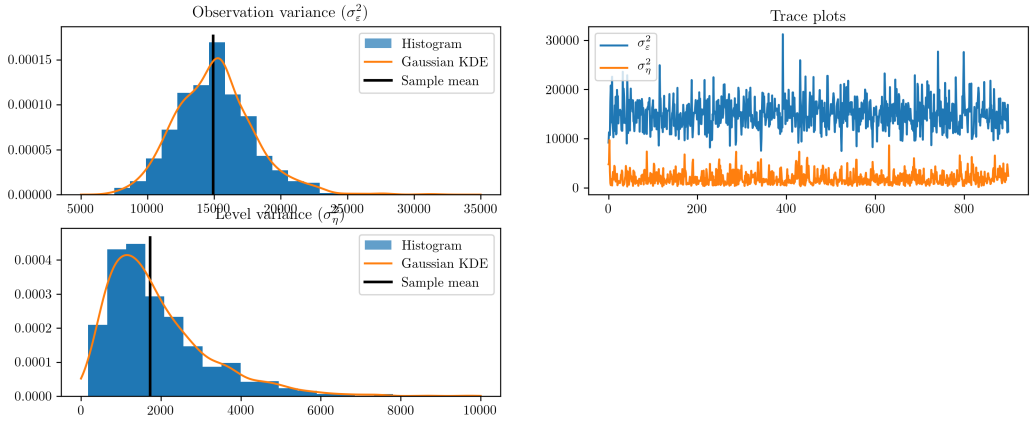

Fig. 12: Output from Metropolis-Hastings posterior simulation on Nile data, using the PyMC library.

5.3 Implementing Gibbs sampling: the ARMA(1,1) model

In this section we describe implementing the Gibbs sampling algorithm to estimation unknown parameters of a state space model. Only the direct approach is presented here (as of now, PyMC only has preliminary support for Gibbs sampling). The Metropolis-within-Gibbs approach is used to demonstrate both how to apply Gibbs sampling and how to apply a hybrid approach.

Recalling the Gibbs sampling algorithm, in order to proceed we need to block the parameters and the unobserved states such the the conditional distributions can be found. We will choose four blocks, so that the unobserved states are in the first block, the autoregressive coefficient is in the second block, the variance is in the third block, and the moving average coefficient is in the last block. In notation, this means that $\psi = \{\psi^{(1)}, \psi^{(2)}, \psi^{(3)}, \psi^{(4)}\} = \{\alpha, \phi, \sigma^2, \theta\}$. We will apply Gibbs steps for the first, second, and third blocks and a Metropolis step for the fourth block.

We select priors for the parameters so that the conditional posterior distributions that we require can be constructed. For the autogressive coefficients we select a multivariate normal distribution - conditional on the variance - with an identity covariance matrix and restricted to the space such that the corresponding lag polynomial is invertible. To be precise, the prior is $\phi \mid \sigma^2 \sim N(0, I)$

such that $\phi(L)$ is invertible.

For the variance, we select an inverse-gamma distribution - conditional on the autoregressive coefficients - with the shape and scale parameters both set to three. To be precise, the prior is $\sigma^2 \mid \phi \sim IG(3,3)$. These choices will be convenient due to their status as conjugate priors for the linear regression model; they will lead to known conditional posterior distributions.

Finally, the prior for the moving-average coefficient is specified to be uniform over the interval $(-1,1)$, so that $\theta \sim \text{unif}(-1,1)$. Notice that the prior density for all values in the range is equal, and so the acceptance probability is either zero, in the case that the proposed value is outside the range, or else simplifies to the ratio of the likelihoods because the prior values cancel out. We will use a random walk proposal with standard Gaussian increment.

Now, conditional on the model parameters, a draw of $\psi^{(1)}$ can be taken by applying the simulation smoother as shown in previous sections. Next notice that, given the values of the states, the first row of the transition equation in (2.7) is simply a linear regression:

$$\alpha_{1,t+1} = \phi\alpha_{1,t} + \varepsilon_{t+1}$$

Stacking these equations across all t into matrix form yields $Z = X\phi + \varepsilon$. A standard result applying conjugate priors to the linear regression model (see for example Kim and Nelson, 1999) is that the conditional posterior distribution for the coefficients is Gaussian and the conditional posterior distribution for the variance is inverse-gamma. To be precise, given our choice of prior hyperparameters here we have

$$\begin{aligned}
\phi \mid \sigma^2, \alpha, Y_n &\sim N\Big((\sigma^2 I + X'X)^{-1}X'Z, (I + \sigma^{-2}X'X)^{-1}\Big) \\
\sigma^2 \mid \phi, \alpha, Y_n &\sim IG\Big(3+n, 3+(Z-X\phi)'(Z-X\phi)\Big)
\end{aligned}$$

Making draws from these conditional posteriors can be implemented in the following way

```
from scipy.stats import multivariate_normal, invgamma

def draw_posterior_phi(model, states, sigma2):
    Z = states[0:1, 1:]
    X = states[0:1, :-1]

    tmp = np.linalg.inv(sigma2 * np.eye(1) + np.dot(X, X.T))
    post_mean = np.dot(tmp, np.dot(X, Z.T))
    post_var = tmp * sigma2

    return multivariate_normal(post_mean, post_var).rvs()

def draw_posterior_sigma2(model, states, phi):
    resid = states[0, 1:] - phi * states[0, :-1]
    post_shape = 3 + model.nobs
    post_scale = 3 + np.sum(resid**2)

    return invgamma(post_shape, scale=post_scale).rvs()
```

Implementing the hybrid method then consists of the following steps for each iteration, given the previous value ψ_{s-1}.

1. Apply the simulation smoother to retrieve a draw of the unobserved states, yielding $\tilde{\alpha} = \psi_s^{(1)}$.
2. Draw a value for $\phi = \psi_1^{(2)}$ from its conditional posterior distribution, conditioning on the states drawn in step 1 and the parameters from the previous iteration.
3. Draw a value for $\sigma^2 = \psi_s^{(3)}$ from its conditional posterior distribution, conditioning on the state states drawn in step 1 and the autoregression coefficients drawn in step 2.
4. Propose a new value for $\theta = \psi_s^{(4)}$ using the random walk proposal, and calculate the acceptance probability using the `loglike` function.

The implementation code is below, and the resultant histograms and traces from 10,000 iterations are given in Fig. 13.

```
from scipy.stats import norm, uniform
from statsmodels.tsa.statespace.tools import is_invertible

# Create the model for likelihood evaluation and the simulation smoother
model = ARMA11(inf)
sim_smoother = model.simulation_smoother()

# Create the random walk and comparison random variables
rw_proposal = norm(scale=0.3)

# Create storage arrays for the traces
n_iterations = 10000
trace = np.zeros((n_iterations + 1, 3))
trace_accepts = np.zeros(n_iterations)
trace[0] = [0, 0, 1.]  # Initial values

# Iterations
for s in range(1, n_iterations + 1):
    # 1. Gibbs step: draw the states using the simulation smoother
    model.update(trace[s-1], transformed=True)
    sim_smoother.simulate()
    states = sim_smoother.simulated_state[:, :-1]

    # 2. Gibbs step: draw the autoregressive parameters, and apply
    # rejection sampling to ensure an invertible lag polynomial
    phi = draw_posterior_phi(model, states, trace[s-1, 2])
    while not is_invertible([1, -phi]):
        phi = draw_posterior_phi(model, states, trace[s-1, 2])
    trace[s, 0] = phi

    # 3. Gibbs step: draw the variance parameter
    sigma2 = draw_posterior_sigma2(model, states, phi)
    trace[s, 2] = sigma2

    # 4. Metropolis-step for the moving-average parameter
    theta = trace[s-1, 1]
    proposal = theta + rw_proposal.rvs()
    if proposal > -1 and proposal < 1:
        acceptance_probability = np.exp(
            model.loglike([phi, proposal, sigma2]) -
            model.loglike([phi, theta, sigma2]))

        if acceptance_probability > uniform.rvs():
            theta = proposal
            trace_accepts[s-1] = 1
    trace[s, 1] = theta
```

5.4 Implementing Gibbs sampling: real business cycle model

Finally, we can apply the same techniques as above to perform Metropolis-within-Gibbs estimation of the real business cycle model parameters. It is often difficult to estimate all of the parameters of the RBC model, or other structural models, by maximum likelihood. Indeed, above we only estimated two of the six structural parameters. By choosing appropriately tight priors it is often feasible to estimate more parameters; in this example we estimate four of the six structural pa-

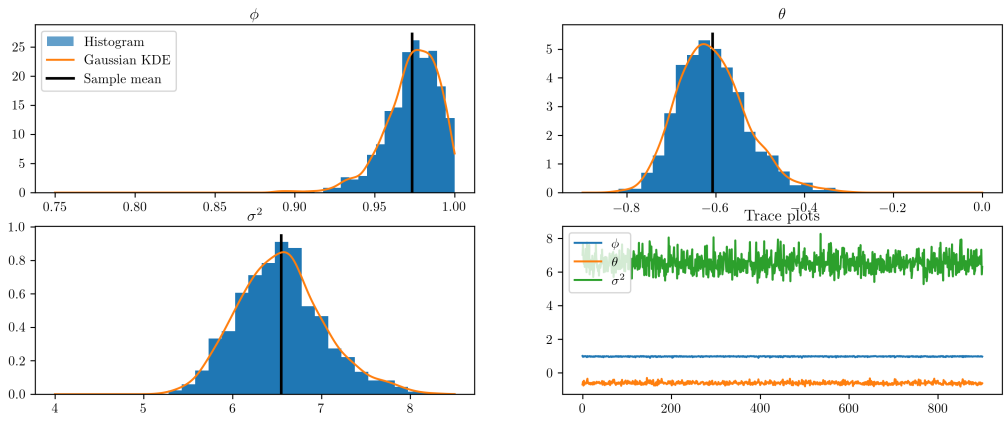

Fig. 13: Output from Metropolis-within-Gibbs posterior simulation on US CPI inflation data.

rameters: the discount rate, capital share, and the two technology shock parameters. Of the two remaining parameters, the disutility of labor only serves to pin down steady-state values and so the model presented above is independent of its value (since it considers data in in deviation-from-steady-state values), and the depreciation rate is best calibrated when the observation datasets do not speak to to depreciation (see, for example, the discussion in Smets and Wouters (2007)).

For the Metropolis-within-Gibbs simulation, we consider 8 blocks. The first three blocks are sampled using Gibbs steps, and are very similar to the ARMA(1,1) example; the first block samples the unobserved states, and the second and third blocks sample the two technology shock parameters. Noticing that the second row of the transition equation is simply an autoregression, conditional on the states, we can use the same approach as before. Thus the priors on these parameters are the Gaussian and inverse-gamma conjugate priors and the unobserved states are sampled using the simulation smoother.

The remaining blocks apply Metropolis steps to sample the remaining five parameters: the discount rate, capital share, and the three measurement variances. The priors on these parameters are as in Smets and Wouters (2007). All priors are listed in Table 6, along with statistics describing the posterior draws.

[6] If the discount rate is denoted β, then the Gamma prior actually applies to the transformation $100(\beta^{-1} - 1)$.

Table 6: Priors and posteriors for the real business cycle model.

	Prior distribution			Posterior distribution			
	Distribution	Mean	Std. Dev.	Mode	Mean	5 percent	95 percent
Discount rate [6]	Gamma	0.25	0.1	0.997	0.997	0.994	0.998
Capital share	Normal	0.3	0.01	0.325	0.325	0.308	0.341
Technology shock persistence	Normal	0	1	0.672	0.637	-0.271	0.940
Technology shock variance	Inverse-gamma	0.01	1.414	8.65e-5	8.98e-5	7.67e-5	1.05e-4
Output error standard deviation	Inverse-gamma	0.1	2	2.02e-5	2.29e-5	1.46e-5	3.34e-05
Labor error standard deviation	Inverse-gamma	0.1	2	3.06e-5	3.21e-5	2.25e-5	4.34e-05
Consumption error standard deviation	Inverse-gamma	0.1	2	2.46e-5	2.57e-5	1.94e-5	3.28e-05

Again, the code is slightly too long to display inline, so it can be found in *Appendix C: Real business cycle model code*. We perform 100,000 draws and burn the first 10,000. Of the remaining 90,000 draws, each tenth draw is saved, so that the results below are ultimately based on 9,000 draws. Histograms of the four estimated structural parameters are presented in Fig. 14.

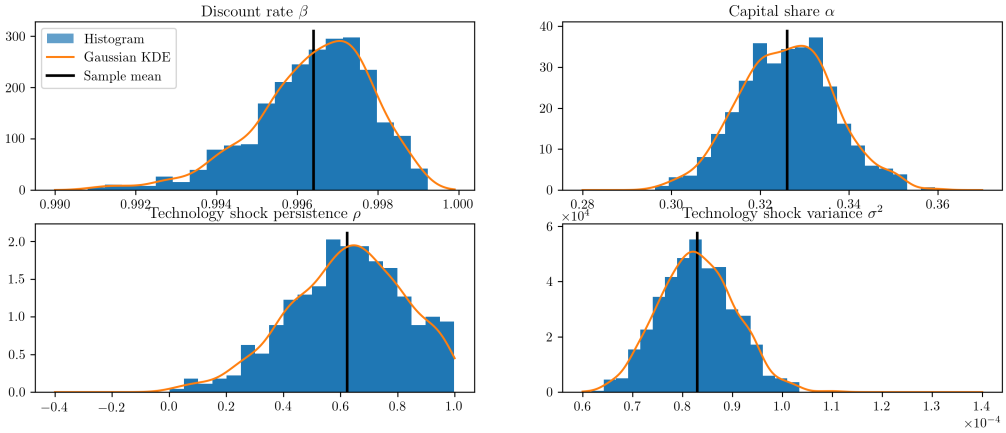

Fig. 14: Output from Metropolis-within-Gibbs posterior simulation of the real business cycle.

As before, we may be interested in the implied impulse response functions and the smoothed state values; here we calculate these by applying the Kalman filter and smoother to the model based on the median parameter values. Fig. 15 displays the impulse responses and Fig. 16 displays the smoothed states and confidence intervals.

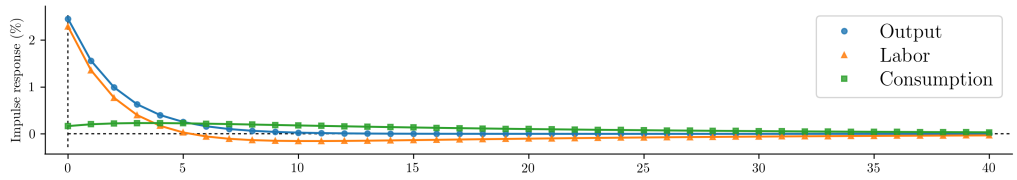

Fig. 15: Impulse response functions corresponding to Metropolis-within-Gibbs estimation of the real business cycle.

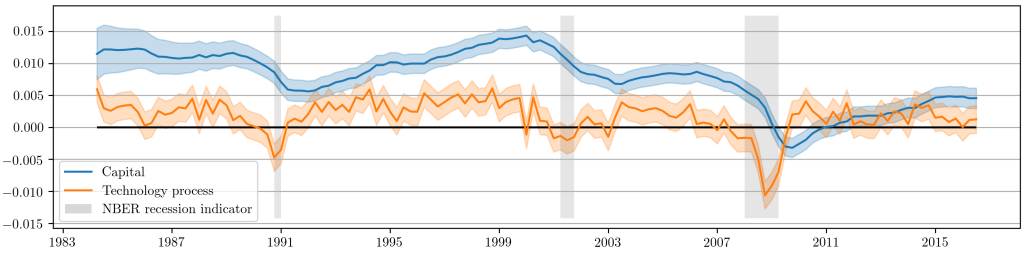

Fig. 16: Smoothed estimates of capital and the technology process from Metropolis-within-Gibbs estimation of the real business cycle.

6 Out-of-the-box models

This paper has focused on demonstrating the creation of classes to specify and estimate arbitrary state space models. However, it is worth noting that classes implementing state space models for four of the most popular models in time series analysis are built in. These classes have been created exactly as described above (e.g. they are all subclasses of `sm.tsa.statespace.MLEModel`), and can be used directly or even extended with their own subclasses. The source code is available, so that they also serve as advanced examples of what can be accomplished in this framework.

Maximum likelihood estimation is available immediately simply by calling the `fit` method. Features include the calculation of reasonable starting values, the use of appropriate parameter transformations, and enhanced results classes. Bayesian estimation via posterior simulation can be performed as described in this paper by taking advantage of the `loglike` method and the simulation smoother. Of course the selection of priors, parameter blocking, etc. must be manually implemented, as above.

In this section, we briefly describe each time series model and provide examples.

6.1 SARIMAX

The seasonal autoregressive integrated moving-average with exogenous regressors (SARIMAX) model is a generalization of the familiar ARIMA model to allow for seasonal effects and explanatory variables. It is typically denoted SARIMAX $(p, d, q) \times (P, D, Q, s)$ and can be written as

$$y_t = \beta_t x_t + u_t$$
$$\phi_p(L)\tilde{\phi}_P(L^s)\Delta^d\Delta_s^D u_t = A(t) + \theta_q(L)\tilde{\theta}_Q(L^s)\zeta_t$$

where y_t is the observed time series and x_t are explanatory regressors. $\phi_p(L), \tilde{\phi}_P(L^s), \theta_q(L)$, and $\tilde{\theta}_Q(L^s)$ are lag polynomials and Δ^d is the differencing operator Δ, applied d times. This model is sometimes described as regression with SARIMA errors.

It is straightforward to apply this model to data by creating an instance of the class `sm.tsa.SARIMAX`. For example, if we wanted to estimate an ARMA(1,1) model for US CPI inflation data using this class, the following code could be used

```
model_1 = sm.tsa.SARIMAX(inf, order=(1, 0, 1))
results_1 = model_1.fit()
print(model_1.loglike(results_1.params))  # -432.375194381
```

We can also extend this example to take into account annual seasonality. Below we estimate an SARIMA(1,0,1)x(1,0,1,12) model. This model achieves a lower value for the Akaike information criterion (AIC), which indicates a potentially better fit.[22]

[22] The Akaike information criterion, as well as several other information criteria, is available for all models that extend the `sm.tsa.statespace.MLEModel` class. See the tables in *Appendix B: Inherited attributes and methods* for all available attributes and methods.

```
model_2 = sm.tsa.SARIMAX(inf, order=(1, 0, 1), seasonal_order=(1, 0, 1, 12))
results_2 = model_2.fit()

# Compare the two models on the basis of the Akaike information criterion
print(results_1.aic)  # 870.750388763
print(results_2.aic)  # 844.623363003
```

6.2 Unobserved components

Unobserved components models, also known as structural time series models, decompose a univariate time series into trend, seasonal, cyclical, and irregular components. They can be written as:

$$y_t = \mu_t + \gamma_t + c_t + \varepsilon_t$$

where y_t refers to the observation vector at time t, μ_t refers to the trend component, γ_t refers to the seasonal component, c_t refers to the cycle, and ε_t is the irregular. The modeling details of these components can be found in the package documentation. These models are also described in depth in Chapter 3 of Durbin and Koopman (2012). The class corresponding to these models is `sm.tsa.UnobservedComponents`.

As an example, consider extending the model previously applied to the Nile river data to include a stochastic cycle, as suggested in Mendelssohn (2011). This is straightforward with the built-in model; the below example fits the model and plots the unobserved components, in this case a level and a cycle, in Fig. 17.

```
model = sm.tsa.UnobservedComponents(nile, 'llevel', cycle=True, stochastic_cycle=True)
results = model.fit()
fig = results.plot_components(observed=False)
```

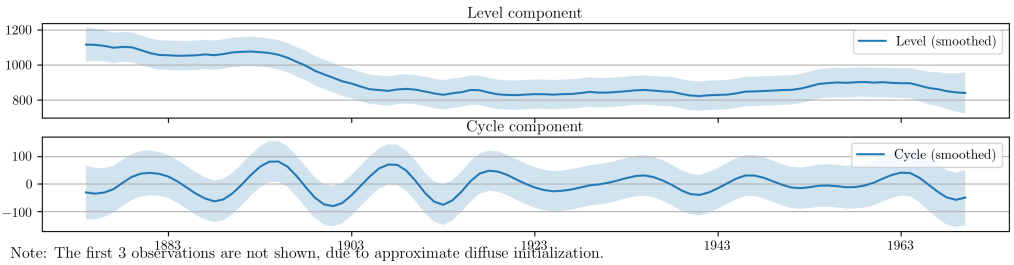

Fig. 17: Estimates of the unobserved level and cyclical components of Nile river volume.

6.3 VAR

Vector autoregressions are important tools for reduced form time series analysis of multiple variables. Their form looks similar to an AR(p) model except that the variables are vectors and the coefficients are matrices.

$$y_t = \Phi_1 y_{t-1} + \cdots + \Phi_p y_{t-p} + \varepsilon_t$$

These models can be estimated using the `sm.tsa.VARMAX` class, which also allows estimation of vector moving average models and optionally models with exogenous regressors.[23] The following code estimates a vector autoregression as a state space model (the starting parameters are the OLS estimates) and generates orthogonalized impulse response functions for shocks to each of the endogenous variables; these responses are plotted in Fig. 18.[24]

```
model = sm.tsa.VARMAX(rbc_data, order=(1, 0))
results = model.fit()

# Generate impulse response functions; the `impluse` argument is used to
# specify which shock is pulsed.
output_irfs = results.impulse_responses(15, impulse=0, orthogonalized=True) * 100
labor_irfs = results.impulse_responses(15, impulse=1, orthogonalized=True) * 100
consumption_irfs = results.impulse_responses(15, impulse=2, orthogonalized=True) * 100
```

[23] Estimation of VARMA(p,q) models is practically possible, although it is not recommended because no measures are in place to ensure identification (for example, the use of Kronecker indices is not yet available).

[24] Note that the orthogonalization is by Cholesky decomposition, which implicitly enforces a a causal ordering to the variables. The order is as defined in the provided dataset. Here `rbc_data` orders the variables as output, labor, consumption.

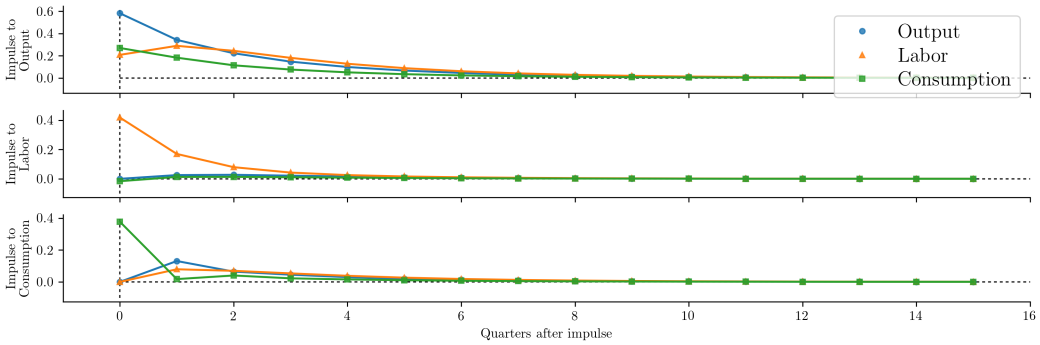

Fig. 18: Impulse response functions derived from a vector autoregression.

6.4 Dynamic factors

Dynamic factor models are another set of important reduced form multivariate models. They can be used to extract a common component from multifarious data. The general form of the model available here is the so-called static form of the dynamic factor model and can be written

$$y_t = \Lambda f_t + Bx_t + u_t$$
$$f_t = A_1 f_{t-1} + \cdots + A_p f_{t-p} + \eta_t$$
$$u_t = C_1 u_{t-1} + \cdots + C_1 f_{t-q} + \varepsilon_t$$

where y_t is the endogenous data, f_t are the unobserved factors which follow a vector autoregression, and x_t are optional exogenous regressors. η_t and ε_t are white noise error terms, and u_t allows the possibility of autoregressive (or vector autoregressive) errors. In order to identify the factors, $Var(\eta_t) \equiv I$.

The following code extracts a single factor that follows an AR(2) process. The error term is not assumed to be autoregressive, so in this case $u_t = \varepsilon_t$. By default the model assumes the elements of ε_t are not cross-sectionally correlated (this assumption can be relaxed if desired). Fig. 19 plots the responses of the endogenous variables to an impulse in the unobserved factor.

```
model = sm.tsa.DynamicFactor(rbc_data, k_factors=1, factor_order=2)
results = model.fit()
print(results.coefficients_of_determination)  # [ 0.957   0.545   0.603 ]

# Because the estimated factor turned out to be inversely related to the
# three variables, we want to consider the negative of the impulse
dfm_irfs = -results.impulse_responses(15, impulse=0, orthogonalized=True) * 100
```

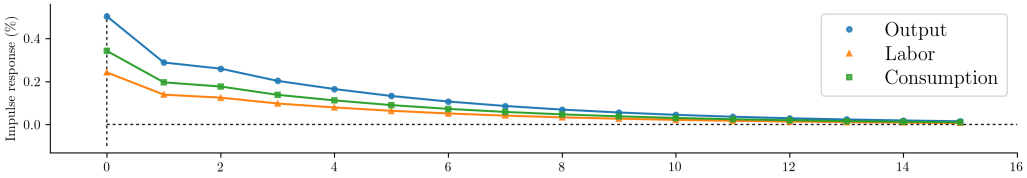

Fig. 19: Impulse response functions derived from a dynamic factor model.

It is often difficult to directly interpret either the filtered estimates of the unobserved factors or the estimated coefficients of the Λ matrix (called the matrix of factor loadings) due to identification issues related to the factors. For example, notice that $\Lambda f_t = (-\Lambda)(-f_t)$ so that reversing the signs of the factors and loadings results in an identical model. It is often informative instead to examine the extent to which each unobserved factor explains each endogenous variable (see for example Jungbacker and Koopman (2014)). This can be explored using the R^2 value from the regression of each endogenous variable on each estimated factor and a constant. These values are available in the results attribute `coefficients_of_determination`. For the model estimated above, it is clear that the estimated factor largely tracks output.

7 Conclusion

This paper describes the use of the Statsmodels Python library for the specification and estimation of state space models. It begins by presenting the notation and equations describing state space models and the filtering, smoothing, and simulation smoothing operations required for estimation. Next, it maps these concepts to programming code using the the technique of object oriented programming and describes a simple method for the specification of state space models. Brief theoretical introductions to maximum likelihood estimation and Bayesian posterior simulation are

given and mapped to programming code; the object oriented representation of state space models makes parameter estimation simple and straightforward.

Three examples, an ARMA(1,1) model, the local level model, and a simple real business cycle model are developed throughout, first theoretically and then as models specified in programming code. Classical and Bayesian estimation of the parameters of each model is performed. Finally, four flexible generic time series models provided in Statsmodels are described. Using these built-in classes, two of the example models, the ARMA(1,1) model and the local level model, are re-estimated and then extended to more complex, better fitting models.

Appendix A: Installation

To use all of the features described in this paper, at least version 0.9.0 of Statsmodels must be used. Many of the features are also available in version 0.8.0. Some of the features not available in 0.8.0 include simulation smoothing and the univariate filtering and smoothing method.

The most straightforward way to install the correct version of Statsmodels is using pip. The following steps should be followed.

1. Install git. Instructions are available many places, for example at https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
2. Install the development version of Statsmodels using the following command:

```
pip install git+git://github.com/statsmodels/statsmodels.git
```

At this point, the package should installed. If you have the Nose package installed, you can test for a successful installation by running the following command (this may take a few minutes):

```
python -c "import statsmodels.tsa.statespace as ssm; ssm.test();"
```

There should be no failures (although a number of Warnings are to be expected).

Dependencies

The Statsmodels library requires the "standard Python stack" of scientific libraries:

- NumPy
- SciPy >= 0.17.1
- Pandas >= 0.18.1
- Cython >= 0.22.0

- Git (this is required to install the development version of Statsmodels)

There are also a few optional dependencies:

- Matplotlib; this is required for plotting functionality
- Nose; this is required for running the test suite
- IPython / Jupyter; this is required for running the examples or building the documentation

Table 7: Methods available to subclasses of `sm.tsa.statespace.MLEModel`

Method	Description
`filter`	Kalman filtering
`fit`	Fits the model by maximum likelihood via Kalman filter.
`loglike`	Joint loglikelihood evaluation
`loglikeobs`	Loglikelihood evaluation
`set_filter_method`	Set the filtering method
`set_inversion_method`	Set the inversion method
`set_stability_method`	Set the numerical stability method
`set_conserve_memory`	Set the memory conservation method
`set_smoother_output`	Set the smoother output
`simulation_smoother`	Retrieve a simulation smoother for the statespace model.
`initialize_known`	Initialize the Kalman filter with known values
`initialize_approximate_diffuse`	Specify approximate diffuse Kalman filter initialization
`initialize_stationary`	Initialize the statespace model as stationary
`simulate`	Simulate a new time series following the state space model
`impulse_responses`	Impulse response function

Table 8: Attributes available to subclasses of `sm.tsa.statespace.MLEModel`

Attribute	Description
`endog`	The observed (endogenous) dataset
`exog`	The dataset of explanatory variables (if applicable)
`start_params`	Parameter vector used to initialize parameter estimation iterations
`param_names`	Human-readable names of parameters
`initialization`	The selected method for Kalman filter initialization
`initial_variance`	The initial variance to use in approximate diffuse initialization
`loglikelihood_burn`	The number of observations during which the likelihood is not evaluated
`tolerance`	The tolerance at which the Kalman filter determines convergence to steady-state

Appendix B: Inherited attributes and methods

`sm.tsa.statespace.MLEModel`

The methods available to all classes inheriting from the base classes `sm.tsa.statespace.MLEModel` are listed in Table 7 and the attributes are listed in Table 8.

The `fit`, `filter`, and `smooth` methods return a `sm.tsa.statespace.MLEResults` object; its methods and attributes are given below.

The `simulation_smoother` method returns a `SimulationSmoothResults` object; its meth-

Table 9: Slice keys available to subclasses of `sm.tsa.statespace.MLEModel`

Attribute	Description
`'obs_intercept'`	Observation intercept; d_t
`'design'`	Design matrix; Z_t
`'obs_cov'`	Observation disturbance covariance matrix; H_t
`'state_intercept'`	State intercept; c_t
`'transition'`	Transition matrix; T_t
`'selection'`	Selection matrix; R_t
`'state_cov'`	State disturbance covariance matrix; Q_t

Table 10: Methods available to results objects from `fit`, `filter`, and `smooth`

Method	Description
`test_normality`	Jarque-Bera for normality of standardized residuals.
`test_heteroskedasticity`	Test for heteroskedasticity (break in the variance) of standardized residuals
`test_serial_correlation`	Ljung-box test for no serial correlation of standardized residuals
`get_prediction`	In-sample prediction and out-of-sample forecasting; returns all prediction results
`get_forecast`	Out-of-sample forecasts; returns all forecasting results
`predict`	In-sample prediction and out-of-sample forecasting; only returns predicted values
`forecast`	Out-of-sample forecasts; only returns forecasted values
`simulate`	Simulate a new time series following the state space model
`impulse_responses`	Impulse response function
`plot_diagnostics`	Diagnostic plots for standardized residuals of one endogenous variable
`summary`	Summarize the results

ods and attributes are also given below.

sm.tsa.statespace.MLEResults

The methods available to these results objects are listed in Table 10 and the attributes are listed in Table 11.

SimulationSmoothResults

The only method of a `SimulationSmoothResults` object is given in Table 12. After this method is called, the attributes in Table 13 are populated. Each time the method is called, these attributes change to the newly simulated values.

Table 11: Attributes available to results objects from `fit`, `filter`, and `smooth`

Attribute	Description
`aic`	Akaike Information Criterion
`bic`	Bayes Information Criterion
`bse`	Standard errors of fitted parameters
`conf_int`	Returns the confidence interval of the fitted parameters
`cov_params_default`	Covariance matrix of fitted parameters
`filtered_state`	Filtered state mean; $a_{t\|t}$
`filtered_state_cov`	Filtered state covariance matrix; $P_{t\|t}$
`fittedvalues`	Fitted values of the model; alias for forecasts.
`forecasts`	Forecasts; $\hat{y}_t = Z_t a_t$
`forecasts_error`	Forecast errors; v_t
`forecasts_error_cov`	Forecast error covariance matrix; F_t
`hqic`	Hannan-Quinn Information Criterion
`kalman_gain`	Kalman gain; K_t
`llf_obs`	The values of the loglikelihood function at the fitted parameters; $\log L(y_t)$
`llf`	The value of the joint loglikelihood function at the fitted parameters; $\log L(Y_n)$
`loglikelihood_burn`	The number of observations during which the likelihood is not evaluated
`nobs`	The number of observations in the dataset
`params`	The fitted parameters
`predicted_state`	Predicted state mean; a_t
`predicted_state_cov`	Predicted state covariance matrix; P_t
`pvalues`	The p-values associated with the z-statistics of the coefficients
`resid`	Residuals of the model; alias for forecasts_errors
`smoothed_measurement_disturbance`	Smoothed observation disturbance mean; $\hat{\varepsilon}_t$
`smoothed_measurement_disturbance_cov`	Smoothed observation disturbance covariance matrix; $Var(\varepsilon_t \mid Y_n)$
`smoothed_state`	Smoothed state mean; $\hat{\alpha}_t$
`smoothed_state_cov`	Smoothed state covariance matrix; V_t
`smoothed_state_disturbance`	Smoothed state disturbance mean; $\hat{\eta}_t$
`smoothed_state_disturbance_cov`	Smoothed state disturbance covariance matrix; $Var(\eta_t \mid Y_n)$
`zvalues`	The z-values of the standard errors of fitted parameters

Table 12: Methods available to results objects from `simulation_smoother`

Method	Description
`simulate`	Perform simulation smoothing

Table 13: Attributes available to results objects from `simulation_smoother`

Attribute	Description
`simulated_state`	Simulated state vector; $\tilde{\alpha}_t$
`simulated_measurement_disturbance`	Simulated measurment disturbance; $\tilde{\varepsilon}_t$
`simulated_state_disturbance`	Simulated state disturbance; $\tilde{\eta}_t$

Appendix C: Real business cycle model code

This appendix presents Python code implementing the full real business cycle model, including solution of the linear rational expectations model, as described in *Representation in Python*. It also presents code for the parameter estimation by classical (see *Maximum Likelihood Estimation*) and Bayesian (see *Posterior Simulation*) methods.

The following code implements the real business cycle model in Python as a state space model.

```
from collections import OrderedDict
class SimpleRBC(sm.tsa.statespace.MLEModel):

    parameters = OrderedDict([
        ('discount_rate', 0.95),
        ('disutility_labor', 3.),
        ('depreciation_rate', 0.025),
        ('capital_share', 0.36),
        ('technology_shock_persistence', 0.85),
        ('technology_shock_var', 0.04**2)
    ])

    def __init__(self, endog, calibrated=None):
        super(SimpleRBC, self).__init__(
            endog, k_states=2, k_posdef=1, initialization='stationary')
        self.k_predetermined = 1

        # Save the calibrated vs. estimated parameters
        parameters = self.parameters.keys()
        calibrated = calibrated or {}
        self.calibrated = OrderedDict([
            (param, calibrated[param]) for param in parameters
            if param in calibrated
        ])
        self.idx_calibrated = np.array([
            param in self.calibrated for param in parameters])
        self.idx_estimated = ~self.idx_calibrated

        self.k_params = len(self.parameters)
        self.k_calibrated = len(self.calibrated)
        self.k_estimated = self.k_params - self.k_calibrated

        self.idx_cap_share = parameters.index('capital_share')
        self.idx_tech_pers = parameters.index('technology_shock_persistence')
        self.idx_tech_var = parameters.index('technology_shock_var')

        # Setup fixed elements of system matrices
        self['selection', 1, 0] = 1

    @property
    def start_params(self):
        structural_params = np.array(self.parameters.values())[self.idx_estimated]
        measurement_variances = [0.1] * 3
        return np.r_[structural_params, measurement_variances]

    @property
    def param_names(self):
```

```
        structural_params = np.array(self.parameters.keys())[self.idx_estimated]
        measurement_variances = ['%s.var' % name for name in self.endog_names]
        return structural_params.tolist() + measurement_variances

    def log_linearize(self, params):
        # Extract the parameters
        (discount_rate, disutility_labor, depreciation_rate, capital_share,
         technology_shock_persistence, technology_shock_var) = params

        # Temporary values
        tmp = (1. / discount_rate - (1. - depreciation_rate))
        theta = (capital_share / tmp)**(1. / (1. - capital_share))
        gamma = 1. - depreciation_rate * theta**(1. - capital_share)
        zeta = capital_share * discount_rate * theta**(capital_share - 1)

        # Coefficient matrices from linearization
        A = np.eye(2)

        B11 = 1 + depreciation_rate * (gamma / (1 - gamma))
        B12 = (-depreciation_rate *
               (1 - capital_share + gamma * capital_share) /
               (capital_share * (1 - gamma)))
        B21 = 0
        B22 = capital_share / (zeta + capital_share*(1 - zeta))
        B = np.array([[B11, B12], [B21, B22]])

        C1 = depreciation_rate / (capital_share * (1 - gamma))
        C2 = (zeta * technology_shock_persistence /
              (zeta + capital_share*(1 - zeta)))
        C = np.array([[C1], [C2]])

        return A, B, C

    def solve(self, params):
        capital_share = params[self.idx_cap_share]
        technology_shock_persistence = params[self.idx_tech_pers]

        # Get the coefficient matrices from linearization
        A, B, C = self.log_linearize(params)

        # Jordan decomposition of B
        eigvals, right_eigvecs = np.linalg.eig(np.transpose(B))
        left_eigvecs = np.transpose(right_eigvecs)

        # Re-order, ascending
        idx = np.argsort(eigvals)
        eigvals = np.diag(eigvals[idx])
        left_eigvecs = left_eigvecs[idx, :]

        # Blanchard-Khan conditions
        k_nonpredetermined = self.k_states - self.k_predetermined
        k_stable = len(np.where(eigvals.diagonal() < 1)[0])
        k_unstable = self.k_states - k_stable
        if not k_stable == self.k_predetermined:
            raise RuntimeError('Blanchard-Kahn condition not met.'
                               ' Unique solution does not exist.')

        # Create partition indices
        k = self.k_predetermined
        p1 = np.s_[:k]
        p2 = np.s_[k:]

        p11 = np.s_[:k, :k]
        p12 = np.s_[:k, k:]
        p21 = np.s_[k:, :k]
        p22 = np.s_[k:, k:]
```

```
        # Decouple the system
        decoupled_C = np.dot(left_eigvecs, C)

        # Solve the explosive component (controls) in terms of the
        # non-explosive component (states) and shocks
        tmp = np.linalg.inv(left_eigvecs[p22])

        # This is \phi_{ck}, above
        policy_state = - np.dot(tmp, left_eigvecs[p21]).squeeze()
        # This is \phi_{cz}, above
        policy_shock = -(
            np.dot(tmp, 1. / eigvals[p22]).dot(
                np.linalg.inv(
                    np.eye(k_nonpredetermined) -
                    technology_shock_persistence / eigvals[p22]
                )
            ).dot(decoupled_C[p2])
        ).squeeze()

        # Solve for the non-explosive transition
        # This is T_{kk}, above
        transition_state = np.squeeze(B[p11] + np.dot(B[p12], policy_state))
        # This is T_{kz}, above
        transition_shock = np.squeeze(np.dot(B[p12], policy_shock) + C[p1])

        # Create the full design matrix
        tmp = (1 - capital_share) / capital_share
        tmp1 = 1. / capital_share
        design = np.array([[1 - tmp * policy_state, tmp1 - tmp * policy_shock],
                           [1 - tmp1 * policy_state, tmp1 * (1-policy_shock)],
                           [policy_state,            policy_shock]])

        # Create the transition matrix
        transition = (
            np.array([[transition_state, transition_shock],
                      [0,               technology_shock_persistence]]))

        return design, transition

    def transform_discount_rate(self, param, untransform=False):
        # Discount rate must be between 0 and 1
        epsilon = 1e-4  # bound it slightly away from exactly 0 or 1
        if not untransform:
            return np.abs(1 / (1 + np.exp(param)) - epsilon)
        else:
            return np.log((1 - param + epsilon) / (param + epsilon))

    def transform_disutility_labor(self, param, untransform=False):
        # Disutility of labor must be positive
        return param**2 if not untransform else param**0.5

    def transform_depreciation_rate(self, param, untransform=False):
        # Depreciation rate must be positive
        return param**2 if not untransform else param**0.5

    def transform_capital_share(self, param, untransform=False):
        # Capital share must be between 0 and 1
        epsilon = 1e-4  # bound it slightly away from exactly 0 or 1
        if not untransform:
            return np.abs(1 / (1 + np.exp(param)) - epsilon)
        else:
            return np.log((1 - param + epsilon) / (param + epsilon))

    def transform_technology_shock_persistence(self, param, untransform=False):
        # Persistence parameter must be between -1 and 1
        if not untransform:
            return param / (1 + np.abs(param))
```

```
        else:
            return param / (1 - param)

    def transform_technology_shock_var(self, unconstrained, untransform=False):
        # Variances must be positive
        return unconstrained**2 if not untransform else unconstrained**0.5

    def transform_params(self, unconstrained):
        constrained = np.zeros(unconstrained.shape, unconstrained.dtype)

        i = 0
        for param in self.parameters.keys():
            if param not in self.calibrated:
                method = getattr(self, 'transform_%s' % param)
                constrained[i] = method(unconstrained[i])
                i += 1

        # Measurement error variances must be positive
        constrained[self.k_estimated:] = unconstrained[self.k_estimated:]**2

        return constrained

    def untransform_params(self, constrained):
        unconstrained = np.zeros(constrained.shape, constrained.dtype)

        i = 0
        for param in self.parameters.keys():
            if param not in self.calibrated:
                method = getattr(self, 'transform_%s' % param)
                unconstrained[i] = method(constrained[i], untransform=True)
                i += 1

        # Measurement error variances must be positive
        unconstrained[self.k_estimated:] = constrained[self.k_estimated:]**0.5

        return unconstrained

    def update(self, params, **kwargs):
        params = super(SimpleRBC, self).update(params, **kwargs)

        # Reconstruct the full parameter vector from the
        # estimated and calibrated parameters
        structural_params = np.zeros(self.k_params, dtype=params.dtype)
        structural_params[self.idx_calibrated] = self.calibrated.values()
        structural_params[self.idx_estimated] = params[:self.k_estimated]
        measurement_variances = params[self.k_estimated:]

        # Solve the model
        design, transition = self.solve(structural_params)

        # Update the statespace representation
        self['design'] = design
        self['obs_cov', 0, 0] = measurement_variances[0]
        self['obs_cov', 1, 1] = measurement_variances[1]
        self['obs_cov', 2, 2] = measurement_variances[2]
        self['transition'] = transition
        self['state_cov', 0, 0] = structural_params[self.idx_tech_var]
```

The following code estimates the three measurement variances as well as the two technology shock parameters via maximum likelihood estimation

```
# Now, estimate the discount rate and the shock parameters
partially_calibrated = {
    'discount_rate': 0.95,
    'disutility_labor': 3.0,
    'capital_share': 0.36,
    'depreciation_rate': 0.025,
}
mod = SimpleRBC(rbc_data, calibrated=partially_calibrated)
res = mod.fit(maxiter=1000)
res = mod.fit(res.params, method='nm', maxiter=1000, disp=False)
print(res.summary())

estimated_irfs = res.impulse_responses(40, orthogonalized=True) * 100
```

Finally, the following code estimates all parameters except the disutility of labor and the depreciation rate via the Metropolis-within-Gibbs algorithm

```
from scipy.stats import truncnorm, norm, invgamma

def draw_posterior_rho(model, states, sigma2, truncate=False):
    Z = states[1:2, 1:]
    X = states[1:2, :-1]

    tmp = 1 / (sigma2 + np.sum(X**2))
    post_mean = tmp * np.squeeze(np.dot(X, Z.T))
    post_var = tmp * sigma2

    if truncate:
        lower = (-1 - post_mean) / post_var**0.5
        upper = (1 - post_mean) / post_var**0.5
        rvs = truncnorm(lower, upper, loc=post_mean, scale=post_var**0.5).rvs()
    else:
        rvs = norm(post_mean, post_var**0.5).rvs()
    return rvs

def draw_posterior_sigma2(model, states, rho):
    resid = states[1, 1:] - rho * states[1, :-1]
    post_shape = 2.00005 + model.nobs
    post_scale = 0.0100005 + np.sum(resid**2)

    return invgamma(post_shape, scale=post_scale).rvs()
```

```
np.random.seed(SEED)

from statsmodels.tsa.statespace.tools import is_invertible
from scipy.stats import multivariate_normal, gamma, invgamma, beta, uniform

# Create the model for likelihood evaluation
calibrated = {
    'disutility_labor': 3.0,
    'depreciation_rate': 0.025,
}
model = SimpleRBC(rbc_data, calibrated=calibrated)
sim_smoother = model.simulation_smoother()

# Specify priors
prior_discount = gamma(6.25, scale=0.04)
prior_cap_share = norm(0.3, scale=0.01)
prior_meas_err = invgamma(2.0025, scale=0.10025)
```

```
# Proposals
rw_discount = norm(scale=0.3)
rw_cap_share = norm(scale=0.01)
rw_meas_err = norm(scale=0.003)

# Create storage arrays for the traces
n_iterations = 10000
trace = np.zeros((n_iterations + 1, 7))
trace_accepts = np.zeros((n_iterations, 5))
trace[0] = model.start_params
trace[0, 0] = 100 * ((1 / trace[0, 0]) - 1)

loglike = None

# Iterations
for s in range(1, n_iterations + 1):
    if s % 10000 == 0:
        print s
    # Get the parameters from the trace
    discount_rate = 1 / (1 + (trace[s-1, 0] / 100))
    capital_share = trace[s-1, 1]
    rho = trace[s-1, 2]
    sigma2 = trace[s-1, 3]
    meas_vars = trace[s-1, 4:]**2

    # 1. Gibbs step: draw the states using the simulation smoother
    model.update(np.r_[discount_rate, capital_share, rho, sigma2, meas_vars])
    sim_smoother.simulate()
    states = sim_smoother.simulated_state[:, :-1]

    # 2. Gibbs step: draw the autoregressive parameter, and apply
    # rejection sampling to ensure an invertible lag polynomial
    # In rare cases due to the combinations of other parameters,
    # the mean of the normal posterior will be greater than one
    # and it becomes difficult to draw from a normal distribution
    # even with rejection sampling. In those cases we draw from a
    # truncated normal.
    rho = draw_posterior_rho(model, states, sigma2)
    i = 0
    while rho < -1 or rho > 1:
        if i < 1e2:
            rho = draw_posterior_rho(model, states, sigma2)
        else:
            rho = draw_posterior_rho(model, states, sigma2, truncate=True)
        i += 1
    trace[s, 2] = rho

    # 3. Gibbs step: draw the variance parameter
    sigma2 = draw_posterior_sigma2(model, states, rho)
    trace[s, 3] = sigma2

    # Calculate the loglikelihood
    loglike = model.loglike(np.r_[discount_rate, capital_share, rho, sigma2, meas_vars])

    # 4. Metropolis-step for the discount rate
    discount_param = trace[s-1, 0]
    proposal_param = discount_param + rw_discount.rvs()
    proposal_rate = 1 / (1 + (proposal_param / 100))
    if proposal_rate < 1:
        proposal_loglike = model.loglike(np.r_[proposal_rate, capital_share, rho, sigma2, meas_
↪vars])
        acceptance_probability = np.exp(
            proposal_loglike - loglike +
            prior_discount.logpdf(proposal_param) -
            prior_discount.logpdf(discount_param))

        if acceptance_probability > uniform.rvs():
```

```
            discount_param = proposal_param
            discount_rate = proposal_rate
            loglike = proposal_loglike
            trace_accepts[s-1, 0] = 1

    trace[s, 0] = discount_param

    # 5. Metropolis-step for the capital-share
    proposal = capital_share + rw_cap_share.rvs()
    if proposal > 0 and proposal < 1:
        proposal_loglike = model.loglike(np.r_[discount_rate, proposal, rho, sigma2, meas_vars])
        acceptance_probability = np.exp(
            proposal_loglike - loglike +
            prior_cap_share.logpdf(proposal) -
            prior_cap_share.logpdf(capital_share))

        if acceptance_probability > uniform.rvs():
            capital_share = proposal
            trace_accepts[s-1, 1] = 1
            loglike = proposal_loglike
    trace[s, 1] = capital_share

    # 6. Metropolis-step for the measurement errors
    for i in range(3):
        meas_std = meas_vars[i]**0.5
        proposal = meas_std + rw_meas_err.rvs()
        proposal_vars = meas_vars.copy()
        proposal_vars[i] = proposal**2
        if proposal > 0:
            proposal_loglike = model.loglike(np.r_[discount_rate, capital_share, rho, sigma2,␣
↪proposal_vars])
            acceptance_probability = np.exp(
                proposal_loglike - loglike +
                prior_meas_err.logpdf(proposal) -
                prior_meas_err.logpdf(meas_std))

            if acceptance_probability > uniform.rvs():
                meas_std = proposal
                trace_accepts[s-1, 2+i] = 1
                loglike = proposal_loglike
                meas_vars[i] = proposal_vars[i]
        trace[s, 4+i] = meas_std
```

References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1999). *LAPACK Users' Guide* (Third ed.). Philadelphia, PA: Society for Industrial and Applied Mathematics.

Ansley, C. F. and R. Kohn (1986, June). A note on reparameterizing a vector autoregressive moving average model to enforce stationarity. *Journal of Statistical Computation and Simulation*, 99–106.

Behnel, S., R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith (2011, March). Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 31–39.

Blanchard, O. J. and C. M. Kahn (1980, July). The Solution of Linear Difference Models under Rational Expectations. *Econometrica*, 1305–1311.

Carter, C. K. and R. Kohn (1994, September). On Gibbs sampling for state space models. *Biometrika*, 541–553.

Chib, S. and E. Greenberg (1995, November). Understanding the Metropolis-Hastings Algorithm. *The American Statistician*, 327–335.

Commandeur, J. J. F., S. J. Koopman, and M. Ooms (2011). Statistical Software for State Space Methods. *Journal of Statistical Software*, 1–18.

DeJong, D. N. and C. Dave (2011, October). *Structural Macroeconometrics: (Second Edition)*. Princeton University Press.

Durbin, J. and S. J. Koopman (2002, August). A simple and efficient simulation smoother for state space time series analysis. *Biometrika*, 603–616.

Durbin, J. and S. J. Koopman (2012, May). *Time Series Analysis by State Space Methods: Second Edition*. Oxford University Press.

Grewal, M. and A. Andrews (2014, December). *Kalman Filtering: Theory and Practice with MATLAB* (4 edition ed.). Hoboken, New Jersey: Wiley-IEEE Press.

Hamilton, J. D. (1994, January). *Time Series Analysis*. Princeton University Press.

Jones, E., T. Oliphant, and P. Peterson (2001). SciPy: Open source scientific tools for Python.

Jungbacker, B. and S. J. Koopman (2014, June). Likelihood-based dynamic factor analysis for measurement and forecasting. *The Econometrics Journal*, n/a–n/a.

Kalman, R. E. (1960, March). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 35–45.

Kim, C.-J. and C. R. Nelson (1999). State-Space Models with Regime Switching: Classical and Gibbs-Sampling Approaches with Applications. MIT Press Books, The MIT Press.

Klein, P. (2000, September). Using the generalized Schur form to solve a multivariate linear rational expectations model. *Journal of Economic Dynamics and Control*, 1405–1423.

Koop, G. (2003, July). *Bayesian Econometrics* (1 edition ed.). Chichester ; Hoboken, N.J: Wiley-Interscience.

Koopman, S. and J. Durbin (2003, January). Filtering and smoothing of state vector for diffuse state–space models. *Journal of Time Series Analysis*, 85–98.

Koopman, S. J. (1993, March). Disturbance Smoother for State Space Models. *Biometrika*, 117–126.

Koopman, S. J. and J. Durbin (2000, May). Fast Filtering and Smoothing for Multivariate State Space Models. *Journal of Time Series Analysis*, 281–296.

McCullough, B. D. and H. D. Vinod (1999, June). The Numerical Reliability of Econometric Software. *Journal of Economic Literature*, 633–665.

Mendelssohn, R. (2011). The STAMP Software for State Space Models. *Journal of Statistical Software*, 1–18.

Monahan, J. F. (1984, August). A note on enforcing stationarity in autoregressive-moving average models. *Biometrika*, 403–404.

Morf, M. and T. Kailath (1975, August). Square-root algorithms for least-squares estimation. *IEEE Transactions on Automatic Control*, 487–497.

Patil, A., D. Huard, and C. J. Fonnesbeck (2010). PyMC: Bayesian Stochastic Modelling in Python. *Journal of Statistical Software*, 1–81.

Ruge-Murcia, F. J. (2007, August). Methods to estimate dynamic stochastic general equilibrium models. *Journal of Economic Dynamics and Control*, 2599–2636.

Seabold, S. and J. Perktold (2010). Statsmodels: Econometric and Statistical Modeling with Python. In *Proceedings of the 9th Python in Science Conference*, pp. 57–61.

Sims, C. A. (2002, October). Solving Linear Rational Expectations Models. *Computational Economics*, 1–20.

Smets, F. and R. Wouters (2007, June). Shocks and Frictions in US Business Cycles: A Bayesian DSGE Approach. *The American Economic Review*, 586–606.

Strickland, C., R. Burdett, K. Mengersen, and R. Denham (2014). PySSM: A Python Module for Bayesian Inference of Linear Gaussian State Space Models. *Journal of Statistical Software*, ??–??

Tierney, L. (1994, December). Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics*, 1701–1728.

Wegner, P. (1990, August). Concepts and Paradigms of Object-oriented Programming. *SIGPLAN OOPS Mess.*, 7–87.

West, M. and J. Harrison (1999, March). *Bayesian Forecasting and Dynamic Models* (2nd edition ed.). New York: Springer.