Python and Asset Pricing

Reading Group Presentation

Simina Pușcașu

June 12, 2020

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Table of contents

Python Scientific Libraries

Overview

Main libraries:

- NumPy
- SciPy
- Pandas
- Numba
- Matplotlib (not included in this presentation)

* Text in red contains links to resources.

NumPy

- How to: `import numpy as np`
- Scope: fast array processing
- Data structure: `np.array()`
- Similar functionalities to Matlab:
  See here a comprehensive list of equivalent expressions

SciPy

- How to:

```
from scipy import subpackage 1,..., subpackage n
```

- Scope: algorithms and functions built on NumPy
- Subpackages must be imported separately

SciPy : Useful subpackages

- `constants` : Physical and mathematical constants
- `integrate` : Integration and ODE solvers
- `interpolate` : Interpolation and smoothing splines
- `linalg` : Linear algebra
- `optimize` : Optimization and root-finding routines
- `stats` : Statistical distributions and functions

SciPy : scipy.optimize

- Optimization
  - local: `minimize([...],method ='...')`
  - global: `differential_evolution()`
- Root finding
  - 1-D: `bisect()`, `newton()`, `fixed_point()`
  - n-D: `root()`
- Linear programming
  - `linprog()`
- Other (legacy) functions:
  - `fmin()`,`fsolve()`

Pandas

- How to: `import pandas as pd`
- Scope: data analysis (handling)
- Data structures: `pd.Series()`, `pd.DataFrames()`
- Easily translates to R
- Similar functionalities in Stata
- Related libraries:
  - statsmodels
  - linearmodels
  - scikit-Learn

Pandas : statsmodels

- How to: `import statsmodels.api as sm`
- Scope: statistical models, tests, data exploration
- Useful models are implemented as classes or modules:
  - class `sm.OLS()`
  - module `sm.tsa` with classes such as :
    `.ar_model.AutoReg()`
    `.arima_model.ARMA()`
    `.regime_switching.markov_regression.MarkovRegression()`
  - module `statsmodels.tsa.statespace` with corresponding classes and methods
- Check User guide for full list of functionalities

Pandas : linearmodels

- How to: `from linearmodels.ModuleName import ModelName`
- Scope: complements `statsmodels` with
  - panel data models (including Fama-MacBeth routine)
  - IV estimators
  - factor asset pricing models (including GMM)
  - system regressions (SUR, SURE, 3SLS, GMM)

Pandas : scikit-learn

- How to: `from sklearn import subpackage`
- Scope:
  - Supervised learning
  - Unsupervised learning
  - Model selection and evaluation
  - Inspection
  - Visualization
  - Data transformation and importing

Pandas : scikit-learn supervised learning useful modules

- `linear_model`
  - `linear_model.LinearRegression()`
  - `linear_model.Ridge()`
  - `linear_model.Lasso()`
  - `LogisticRegression()`
- `svm`
- `neighbors`
- `tree`

Numba

- Scope: Python code → fast-running optimized code
- Works for functions and classes
- Key aspect: infers input type to speed things up

```
from numba import jit
import random

@jit(nopython=True) # decorator
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Object-oriented programming

OOP

- In Python everything is an object.
- Examples of objects:
  - functions
  - modules (scripts)
  - variables such as integers, strings...
- An object is characterized by
  - type (string, integer, function, class etc.)
  - identity (its address in memory)
  - content (data)
  - methods (functions/ callable attributes)

OOP

- A **class** acts as a blueprint for creating your own objects
- An **object** is therefore an instance of a class
- Objects will have their own content and can use the methods provided in the class definition

Application

Bansal and Yaron (2004)

$$x_{t+1} = \rho x_t + \varphi_x \sigma_t \varepsilon^x_{t+1} \tag{1}$$

$$g^c_{t+1} = \mu + x_t + \sigma_t \varepsilon^c_{t+1} \tag{2}$$

$$g^d_{t+1} = \mu_d + \phi x_t + \varphi_d \sigma_t \varepsilon^d_{t+1} + \varphi_{dc} \sigma_t \varepsilon^c_{t+1} \tag{3}$$

$$\sigma^2_{t+1} = \sigma^2 + \nu_1 (\sigma^2_t - \sigma^2) + \varphi_s \varepsilon^\sigma_{t+1} \tag{4}$$

Bansal and Yaron (2004)

$$x_{t+1} = \rho x_t + \varphi_x \sigma_t \varepsilon^x_{t+1} \tag{1}$$

$$g^c_{t+1} = \mu + x_t + \sigma_t \varepsilon^c_{t+1} \tag{2}$$

$$g^d_{t+1} = \mu_d + \phi x_t + \varphi_d \sigma_t \varepsilon^d_{t+1} + \varphi_{dc} \sigma_t \varepsilon^c_{t+1} \tag{3}$$

$$\sigma^2_{t+1} = \sigma^2 + \nu_1(\sigma^2_t - \sigma^2) + \varphi_s \varepsilon^\sigma_{t+1} \tag{4}$$

$$r^c_{t+1} = \kappa_0 + \kappa_1 wc_{t+1} - wc_t + g^c_{t+1} \tag{5}$$

$$wc_t = A_0 + A_1 x_t + A_2 \sigma^2_t \tag{6}$$

$$r^d_{t+1} = \kappa^m_0 + \kappa^m_1 pd_{t+1} - pd_t + g^d_{t+1} \tag{7}$$

$$pd_t = A^m_0 + A^m_1 x_t + A^m_2 \sigma^2_t \tag{8}$$

Goal

- Build LLR model class
- Instances of the class: BY2004 with and without stochastic volatility for various parametrizations
- Class methods:
  - Simulate the stochastic variables
  - Produce log-linear approximation solution
  - Produce projection method solution (implemented: the collocation method as in Pohl, Schmedders and Wilms (2018))
  - Compute asset pricing moments
  - *Methods for visualizing model output

Goal

- ~~Build LLR model class~~
- ~~Instances of the class: BY2004 with and without stochastic volatility for various parametrizations~~
- Class methods:
  - ~~Simulate the stochastic variables~~
  - ~~Produce log-linear approximation solution~~
  - ~~Produce projection method solution (implemented: the collocation method as in Pohl, Schmedders and Wilms (2018))~~
  - Compute asset pricing moments
  - *Methods for visualizing model output

Numerical approximation: the collocation method

- The time invariant Euler equation for consumption is:

$$0 = \int \left[1 - exp\Big(\theta\Big(\log\delta + \Big(1 - \frac{1}{\psi}\Big)\Delta c(s'|s) + z(s') - \log(\exp(z(s)) - 1)\Big)\Big)\right] df_{\varepsilon}$$

  - $s = (x, \sigma^2)$
  - $s' = (x, \sigma^2; \varepsilon)$
  - z as wc ratio

Numerical approximation: the collocation method

- Approximate the wc ratio using Chebyshev polynomial series, $\Lambda_k = T_k\left(\frac{2x-a-b}{b-a}\right)$, $\Lambda_h = T_h\left(\frac{2\sigma^2-c-d}{d-c}\right)$, with $x \in [a, b]$, $\sigma^2 \in [c, d]$:

$$\hat{z}(x, \sigma^2; \alpha) = \sum_{k=0}^{n} \sum_{h=0}^{n} \alpha_{kh} \Lambda_k(x) \Lambda_h(\sigma^2)$$

- $T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

Numerical approximation: the collocation method

$$\hat{z}(x,\sigma^2;\alpha) = \sum_{k=0}^{n}\sum_{h=0}^{n} \alpha_{kh}\Lambda_k(x)\Lambda_h(\sigma^2)$$

- The coefficients in α are obtained by imposing that the Euler equation holds at each $s_{ij} = (x_i, \sigma_j^2)$ pair with $i, j \in \{1, ..., n\}$ (approximation nodes) with $s'_{ij}(x_i, \sigma_j^2; \varepsilon)$ as next period state:

$$0 = \int \left[1 - exp\left(\theta\left(\log\delta + \left(1 - \frac{1}{\psi}\right)\Delta c(s'_{ij}|s_{ij}) + \hat{z}(s'_{ij},\alpha) - \log(\exp(\hat{z}(s_{ij},\alpha)) - 1)\right)\right)\right] df_{\varepsilon}$$

Numerical approximation : the collocation method cont'd

$$0 = \int \Big[1 - exp\Big(\theta\Big(\log\delta + \Big(1 - \frac{1}{\psi}\Big)\Delta c(s'_{ij}|s_{ij}) + \hat{z}(s'_{ij}, \alpha) - \log(\exp(\hat{z}(s_{ij}, \alpha)) - 1)\Big)\Big)\Big] df_{\varepsilon}$$

- Finally, the integral above is computed using Gauss-Hermite quadrature of degree $\mathcal{L}$ over each of the 4 normal shocks:

$$0 = \sum_{\ell=1}^{(\mathcal{L})^4} \Big[1 - exp\Big(...\Big)\Big] w_{\ell}(\varepsilon)$$

- A similar procedure is performed for the pd ratio. Details in Judd (1998) or in the appendix of Pohl, Schmedders and Wilms (2018)

Implementation

- My Python code
- The original Matlab code for PSW(2018) (downloads code on click)
- 1:1 translation of PSW(2018) Matlab code to Python