MFR (Macro Finance Research) Suite

Lars Peter Hansen, Joseph Huang, Paymon Khorrami, Fabrice Tourre*

Version: 0.1.2†

Contents

1 Introduction **4**
1.1 License 4

2 Installation **5**
2.1 Installing MFR Suite 5
2.1.1 Windows 5
2.1.2 macOS/Linux 6
2.1.3 Optional Step (for Exporting Plots as Files with `Plotly`) 7
2.1.4 Potential Errors 8
2.2 Quick Start Guide 8

*We would like to gratefully acknowledge the Macro Financial Modeling project through the generous financial support from the Alfred P. Sloan Foundation and Fidelity Investments and to thank Amy Boonstra, former MFM Executive Director, for her unconditional support. For their feedback, we thank Yu-Ting Chiang (University of Chicago), Jian Li (University of Chicago), Simon Scheidegger (HEC Lausanne), Elisabeth Proehl (University of Amsterdam), and conference participants at the 2nd MMCN, PASC18, University of Zurich, Northwestern University, and participants at the Economic Dynamics Working Group at the University of Chicago. We also would like to thank the Research Computing Center at the University of Chicago for their guidance on high performance computing, in particular Peter Carbonetto and Hossein Pourreza.

†If you have any questions, please contact: jhuang12@uchicago.edu, fabrice@uchicago.edu, paymon@uchicago.edu, hanxuh@uchicago.edu

3 Graphic User Interface **10**
3.1 Start the Interface . . . 10
3.2 Solving a Model . . . 10
3.3 Analyzing a Model . . . 12
3.4 Comparing Models and Advanced Usage . . . 13

4 Generalized Modeling Framework in Hansen, Khorrami, and Tourre **14**
4.1 Model Overview . . . 14
4.1.1 Preferences . . . 14
4.1.2 Technology . . . 15
4.1.3 Markets . . . 15
4.1.4 Single-Agent Optimization Problem . . . 16
4.1.5 Market Equilibrium . . . 16
4.2 Solving the Hansen, Khorrami, Tourre Model: `mfr.modelSoln` . . . 17
4.2.1 Solving a Model . . . 17
4.2.2 Setting Parameters . . . 20
4.2.3 Model Diagnostics I . . . 23
4.2.4 Model Diagnostics II . . . 26
4.2.5 Other Methods . . . 28

5 Stationary Density and Shock Elasticities **31**
5.1 Stationary Density . . . 31
5.1.1 Overview . . . 31
5.1.2 Computing Stationary Distribution: `mfr.sdm.computeDent` . . . 31
5.2 Shock Elasticities . . . 34
5.2.1 Overview . . . 34
5.2.2 Computing Shock Elasticities: `mfr.sem.computeElas` . . . 36

6 Troubleshooting **39**
6.1 Interpreting Errors and Warnings . . . 39
6.1.1 Module `mfr.modelSoln` . . . 39
6.1.2 Modules `mfr.sdm` and `mfr.sem` . . . 41
6.2 Interpreting the `log.txt` File and Potential Bottlenecks . . . 41
6.3 Numerical Algorithm Parameters to Tweak . . . 43
6.4 Models with Capital Misallocation . . . 44

7 Examples **45**
7.1 He and Krushnamurthy (2013) . . . 45
7.2 Bansal and Yaron (2004) . . . 47
7.3 One-Dimensional Hansen, Khorrami, and Tourre (2018) . . . 50
7.4 Two-Dimensional Hansen, Khorrami, and Tourre (2018) . . . 51

8 Appendix **52**
8.1 Parameters for `mfr.modelSoln` . . . 52
8.2 Equilibrium Quantities and Variables in Hansen, Khorrami, and Tourre (2018) 54

9 Change Log **56**
9.1 Version 0.1.1 56
9.2 Version 0.1.2 56

1 Introduction

The MFR (Macro Finance Research) Suite is a set of four user-friendly Python packages:

1. `mfr.modelSoln`: provides model solution to the generalized framework introduced in Hansen, Khorrami, and Tourre (2018) (more in section 4)
2. `mfr.sdm`: provides tools to compute stationary density given stochastic processes (more in section 5.1)
3. `mfr.sem`: provides tools to compute shock elasticities (more in section 5.2)
4. `mfr.jupyterWidgets`: an auxiliary module that provides a Jupyter-based graphical user interface (GUI) that allows the user to visualize model results (more in section 3)

`mfr.sdm` and `mfr.sem` are independent modules and can be used without the other two.

1.1 License

The software is distributed under the MIT License (please refer to file `/license/LICENSE.txt`). We kindly suggest that you cite the Macro Finance Research (MFR) Team and paper Hansen, Khorrami, and Tourre (2018).

2 Installation

2.1 Installing MFR Suite

2.1.0.1 Step 1: The setup and use of the MFR Suite requires a Python 3 distribution with the `pip` package manager. We recommend the Anaconda distribution (www.anaconda.com), since it includes the most relevant packages for scientific computation (even if you don't want to use Anaconda, we recommend that you install `miniconda` to manage your Python environment[1]). If you decide not to use Anaconda, please ensure that you have installed the `numpy`, `scipy`, `matplotlib`, and `cython` packages before running anything else.

If you have not downloaded the MFR Suite from `larspeterhansen.org` already, please do so. Afterwards, please unzip the file and you will end up with folder `mfrSuite`. Enter `mfrSuite` and continue by following the instructions below based on your operating system.

2.1.1 Windows

2.1.1.1 Step 2: If you've never compiled C++ code on your computer before, you probably need to install a C++ compiler. To do so, we recommend that you install Microsoft Visual Studio 2017 (https://visualstudio.microsoft.com/downloads). After you download and open the installer, please select *Visual Studio Community 2017* (figure 1). When choosing which development package/environment/toolkit, select only *Linux development with C++* (figure 2). Note: although the core code of the MFR Suite is written in C++, you don't need any knowledge of C++ to use the software.

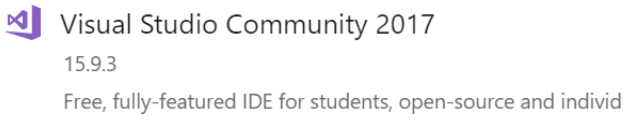


Figure 1: Choosing Visual Studio

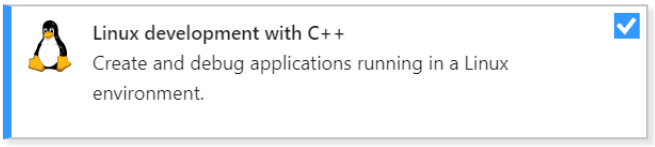


Figure 2: Choosing Linux

2.1.1.2 Step 3: You may need to upgrade your `pip` to the newer version (such as 18.1). To do so, open up the Anaconda command prompt (not the command prompt that you get by typing in `cmd`) and run `pip install --upgrade pip`.

2.1.1.3 Step 4: After upgrading pip, in the Anaconda command prompt, go into the folder of the MFR Suite. Run the installation batch file by running `install.bat` in the Anaconda prompt. Depending on the Python version you're using, you might see an error. Please refer to section 2.1.4 for a solution.

2.1.1.4 Step 5: The installation process is now complete. If the installation was successful, you should see the following message:

[1] It can be downloaded from https://conda.io/miniconda.html

```
mfr.sem was installed successfully.
mfr.sdm was installed successfully.
mfr-modelSoln was installed successfully.
mfr-jupyterWidgets was installed successfully.
modelSolnCore was installed successfully.
```

If not, you would see an error message suggesting which package was not installed successfully. For example, you could see:

```
==========================================
ERROR: mfr.sdm WAS NOT INSTALLED SUCCESSFULLY.
Re-run 'pip install ./src/sdm' in the command line and
look for explanation in documentation on common errors.
==========================================
```

If you do see such an error message, run the command as suggested and see the related error. Afterwards, look for a solution in section 2.1.4.

If the installation was successful, for a quick start guide, please refer to section 2.2. If you wish to test models without doing any coding, please try out the graphical user interface in section 3.

2.1.2 macOS/Linux

2.1.2.1 Step 2: If you've never compiled C++ code on your computer before, you probably need to install a C++ compiler. To check, run `g++` in the Terminal. If you already have a C++ compiler set up, you would see a message similar to `clang: error: no input files`. If not, the Terminal should prompt you to install XCode command line tools (figure 3). If the Terminal does not prompt you to install XCode command line tools, you can run `xcode-select --install` and you would see the same prompt as in figure 3.

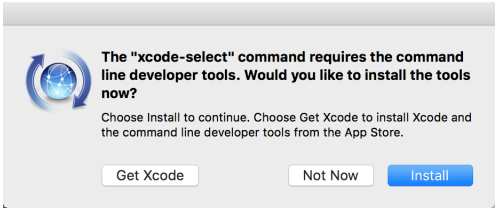


Figure 3: Installing command line tools for macOS

If you use Linux, you can run `sudo apt-get install build-essential` in the terminal. If you have compiled C++ code before, you can skip this step. Note: although the core of the MFR Suite is written in C++, you don't need any knowledge of C++ to use the software.

2.1.2.2 Step 3: Open up Terminal and `cd` into the software folder. Run the `install.sh` script to install the Suite (you can use `./install.sh` in the Terminal). Depending on the Python version you're using, you might see an error. Please refer to section 2.1.4 for a solution.

2.1.2.3 Step 4: The installation process is now complete. If the installation was successful, you should see the following message:

```
mfr.sem was installed successfully.
mfr.sdm was installed successfully.
mfr-modelSoln was installed successfully.
mfr-jupyterWidgets was installed successfully.
modelSolnCore was installed successfully.
```

If not, you would see an error message suggesting which package was not installed successfully. For example, you could see:

```
==========================================
ERROR: mfr.sdm WAS NOT INSTALLED SUCCESSFULLY.
Re-run 'pip install ./src/sdm' in the command line and
look for explanation in documentation on common errors.
==========================================
```

If you do see such an error message, run the command as suggested and see the related error. Afterwards, look for a solution in section 2.1.4.

If the installation was successful, for a quick start guide, please refer to section 2.2. If you wish to test models without doing any coding, please try out the graphical user interface in section 3.

2.1.3 Optional Step (for Exporting Plots as Files with `Plotly`)

The `mfr.modelSoln` module uses either `matplotlib` or `plotly`, another plotting engine, to generate plots. By default (with the exception of the GUI discussed in section 3), the module uses `matplotlib` to display plots or export plots into files. However, if you would like to export `Plotly`-generated plots into files (for example, if you would like to use `.dumpPlots(fancy = True)`, which exports plots into files, as described in section 4.2.5), you need to implement this section[2].

This section contains the steps to install two packages, `plotly-orca` and `psutil`. If you use the Anaconda distribution (or miniconda), you can simply run the following command

[2]This section is only necessary when you want to *export* plots into files with `plotly`. If you want to display plots on the screen with `plotly`, this section is not needed. You only need to run the module in an interactive environment, such as Jupyter, if you want to display `plotly`-based plots on the screen.

in the Terminal if you use macOS or Linux, or the Anaconda prompt if you use Windows:

```
conda install -c plotly plotly-orca psutil
```

If you don't use the Anaconda distribution, please refer to this link and follow the instructions in *npm + pip* or *Standalone Binaries + pip*.

2.1.4 Potential Errors

We use `pip` to install packages but some existing packages might have been installed by `disutils` and could create confusion for `pip`. For example, you might see an error such as:

> Cannot uninstall 'traitlets'. It is a distutils installed project and thus we cannot accurately determine which files belong to it which would lead to only a partial uninstall.

> Cannot uninstall 'llvmlite'. It is a distutils installed project and thus we cannot accurately determine which files belong to it which would lead to only a partial uninstall.

These two errors are essentially the same, except for the involved package. If you see the same or similar error, you need to manually uninstall the package located in `site-packages` of Python's directory. To do so, follow the steps below (suppose the conflicted package is `llvmlite`):

- Find your Python's `site-packages` directory: in the Terminal, run `python -c "import site; print(site.getsitepackages())"` . This command would give you the path of the `site-packages` directory (for example, the directory could be `/anaconda3/lib/python3.5/site-packages`). Enter the directory by doing, for example, `cd /anaconda3/lib/python3.5/site-packages`.
- Delete folder `llvm-0.x.x` (the `x`s in `0.x.x` are variable, depending on the version number; for example, the folder could be called `llvmlite-0.26.0`) by doing, for example, `rm -r llvmlite-0.26.0`.
- Delete egg file `llvmlite-0.26.0-py3.5.egg-info` (note that the numbers in the file name could look different depending on your versions) by doing `rm llvmlite-0.26.0-py3.5.egg-info`.

You may then attempt to re-run the installation file (`install.sh` or `install.bat`).

2.2 Quick Start Guide

We've included a few examples for you to get started (you can find a more detailed explanation on the examples in section 7). To quickly solve a one-dimensional model, please go into the `examples` folder and run `test1DModel.py`. You can try `test2DModel.py` and

`test3DModel.py` for two-dimensional and three-dimensional models respectively.

If you are interested in computing stationary density and shock elasticities, you can use `HeKrushnamurthy.py` and `BansalYaron.py`. These two scripts compute the stationary density and shock elasticities of two models.

3 Graphic User Interface

3.1 Start the Interface

The Suite comes with a graphical user interface (GUI) that allows you to solve models by inputting your own parameters and compute the stationary density and shock elasticities of a solved model, *without any coding at all*. The GUI is built on *Jupyter*, a Python extension (no knowledge of *Jupyter* is needed). If you used the Anaconda distribution as suggested, you already have Jupyter set up and are good to go. If not, you would need to follow the instructions here.

After getting Jupyter set up, open up Terminal (if you use macOS/Linux) or the Anaconda Prompt (if you use Windows), and `cd` into the `examples` folder. Run `jupyter-notebook`. This command will open up Jupyter in your browser. Click `Python Jupyter Interface.ipynb` as indicated in figure 4. (Note: you can copy `Python Jupyter Interface.ipynb` into a location that you prefer; the file can be stored and called anywhere).

Figure 4: Start the GUI

After opening the file, you will see an uninitialized interface (figure 5). Click *here* as shown in the red box. You then see the underlying code. Select the first cell with Python code (as boxed in red in figure 6) by clicking the cell. After that, click button *Run* (as boxed in blue in figure 6). The notebook is then initialized and ready to use.

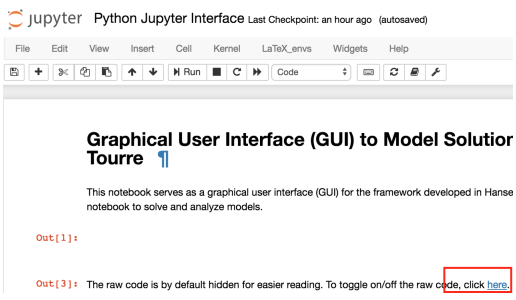


Figure 5: Uninitialized notebook

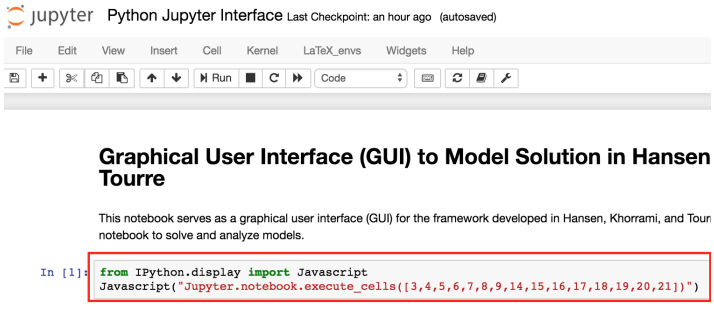


Figure 6: Initializing the notebook

3.2 Solving a Model

3.2.0.1 Solving a model After initializing the notebook, you can start solving models by inputting your parameters (figure 7). By default, the parameters will lead to a one-

dimensional model with log-utility agents. A detailed explanation on the parameters can be found in section 4.2.2.

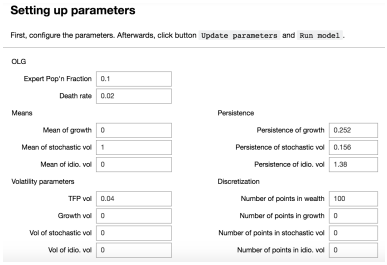


Figure 7: Inputting parameters

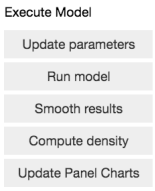


Figure 8: Buttons to solve models

When you finish inputting your parameters, you can solve and analyze the model by using these five buttons (figure 8):

- *Update parameters*: update the parameters of the model using your input. If you don't click this button after modify the parameters, the interface will not register your change.
- *Run model*: solve the model.
- *Smooth results*: after solving a model with capital misallocation, you may or may not need to smooth equilibrium quantities κ and q.
- *Compute density*: compute stationary density of the model *after* it is solved.
- *Display charts*: visualize the model results. Note that this button can be used without computing stationary density.

For example, after solving a 2D model, you can click *Display charts* and one of the equilibrium quantities (functions of two state variables), *Experts' Equity Retention*, would be visualized as in figure 11.

3.2.0.2 Stopping the solution program. If you wish to terminate the model solution after clicking *Run Model*, (for example, if it is taking too much time and you wish to adjust the parameters), you may click the stop button (on top of the GUI) as in figure 9. Alternatively, you may click *Kernel* and then *Interrupt* as in figure 10. These two options are equivalent.[3]

[3]These options do not apply if you use Windows, as Jupyter does not throw the recognizable Windows signal, `CTRL_C_EVENT`. To terminate the program, you have to kill the kernel. However CTRL+C still works if you are using Python directly, regardless of the operating system.

Figure 9: Clicking stop button

Figure 10: Interrupting the kernal

After stopping the program, a message will show up and let you know the location of the `log.txt` file, which contains the history of the program (for a more detailed discussion of this file, please refer to section 6.2).

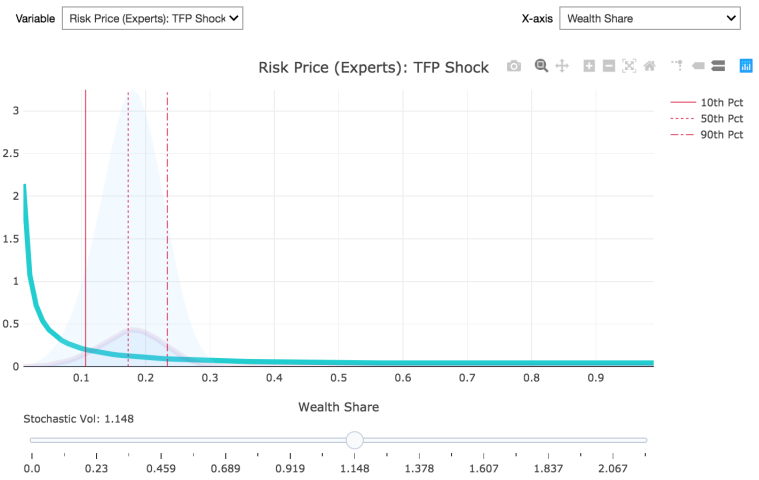


Figure 11: Visualizing an equilibrium quantity from a two-dimensional model. In this chart, the equilibrium quantity is a function of two state variables, one of which, wealth share, is on the x-axis, whereas the other, stochastic volatility, is on the adjustable slider. The 1D plot shows the equilibrium quantity as a function of wealth share, fixed at a specific point of stochastic volatility.

3.3 Analyzing a Model

After solving a model, you may compute its stationary density by clicking button *Compute density*. Afterwards, you may compute the ergodic mean, standard deviation, and correlations of the equilibrium quantities using the functionalities in section *Diagnostics* (fig 12). Similarly, you may compute and visualize shock elasticities by perturbing a selected list of equilibrium quantities (figure 13)

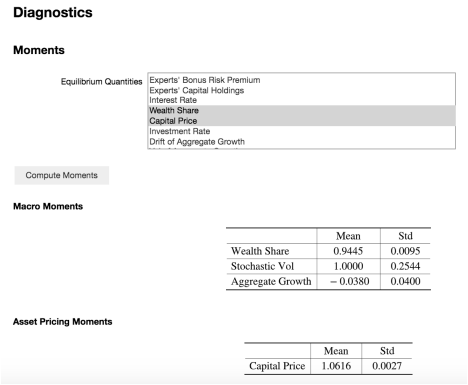


Figure 12: Computing moments

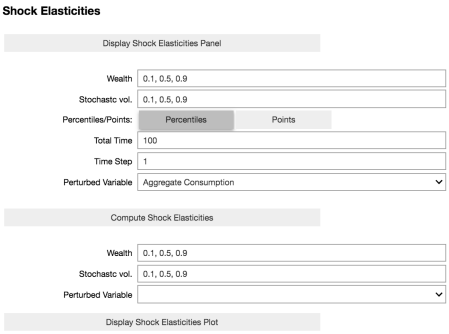


Figure 13: Computing Shock Elasticities

3.4 Comparing Models and Advanced Usage

Note that the notebook `Python Jupyter Interface.ipynb` does not have any location dependence. That is, you can copy this file anywhere. To compare different models, you can simply duplicate this file and run two notebooks simultaneously. We advise against running the same notebook in two different browser windows, as it would create a conflict.

If you are familiar with Python and Jupyter and would like to modify a computed model, you can create a new cell below the notebook (or modify the notebook however you'd like). The underlying object behind the GUI is `defaultModel`. You can call the methods of this object as discussed in section 4.2. For example, you can use `defaultModel.plot('q')` to plot the capital price.

4 Generalized Modeling Framework in Hansen, Khorrami, and Tourre

`mfr.modelSoln` solves the framework developed in Hansen, Khorrami, and Tourre (2018): a dynamic stochastic general equilibrium model featuring two types of agents: experts, which we think of as financial intermediaries and households. We built from the original framework of Brunnermeier and Sannikov (2014), to which we add a constellation of shocks and for which we use generalized recursive preferences. Note that while `mfr.modelSoln` depends on the independent stationary density and shock elasticities toolboxes in section 5, you don't need to go through this section to use the stationary density and shock elasticities toolboxes in section 5.

4.1 Model Overview

Our agent types might differ in their productivities, their attitudes towards risk, their ability to smooth consumption inter-temporally, and their access to financial markets. Both agent types can hold physical capital to produce consumption goods according to a standard linear production technology. Capital holdings expose agents to a constellation of idiosyncratic and aggregate shocks, described in this model overview.

4.1.1 Preferences

- Two types of agents in our economy: "experts" (subscript "e") and "households" (subscript "h")
- Each group features a continuum of measure 1 of agents
- Consider a "representative" expert and a "representative" household
- Each agent type j has recursive ("Kreps-Porteus") preferences over consumption streams.
- Overlapping generation structure:
  - Agents die and are borne at a constant Poisson rate λ_d.
  - Wealth of dying agents is distributed to new-born agents, a fraction ν being given to experts and a fraction $1-\nu$ being given to households.
- Consumption and continuation values of agent j are denoted by $C_{j,t}$ and $U_{j,t}$, respectively, where the continuation value satisfies the continuous time counterpart to the following discrete time recursion:

$$\begin{aligned} U_t &= \left[\left[1-\exp(-\delta\epsilon)\right](C_t)^{1-\rho} + \exp(-\delta\epsilon)\mathbb{R}(U_{t+\epsilon} \mid \mathcal{F}_t)^{1-\rho}\right]^{\frac{1}{1-\rho}} \\ \mathbb{R}(U_{t+\epsilon} \mid \mathcal{F}_t) &= \left(\mathbb{E}\left[(U_{t+\epsilon})^{1-\gamma} \mid \mathcal{F}_t\right]\right)^{\frac{1}{1-\gamma}} \end{aligned} \tag{1}$$

  parameterized by:

- A rate of time preference δ_j
- A relative risk aversion coefficient γ_j
- An elasticity of inter-temporal substitution $1/\rho_j$

4.1.2 Technology

- Capital $K_{j,t}$ of agent type j evolves as:

$$dK_{j,t} = K_{j,t}\left[\left(Z_t + \Phi(I_{j,t}) - \alpha_K\right)dt + \sqrt{V_t}\sigma_K \cdot dW_t + \sqrt{\widetilde{V}_t} \cdot d\widetilde{W}_t\right]$$

  where $I_{j,t}$ is the investment capital ratio.

- Exogenous state variables $\{Z_t, V_t, \widetilde{V}_t\}$ follow

$$\begin{aligned} dZ_t &= \lambda_Z(\overline{Z} - Z_t)dt + \sqrt{V_t}\sigma_Z \cdot dW_t \\ dV_t &= \lambda_V(\overline{V} - V_t)dt + \sqrt{V_t}\sigma_V \cdot dW_t \\ d\widetilde{V}_t &= \lambda_{\widetilde{V}}(\overline{\widetilde{V}} - \widetilde{V}_t)dt + \sqrt{\widetilde{V}_t}\sigma_{\widetilde{V}} \cdot dW_t \end{aligned}$$

  where W_t is a $d \times 1$ vector of aggregate shocks, while $\widetilde{W}_t$ is an idiosyncratic shock, with $\overline{Z}$ normalized to 0 and $\overline{V}$ normalized to 1. Z_t is the expected growth rate, V_t aggregate stochastic variance, and H_t idiosyncratic variance.

- $K_{j,t}$ held between dates t and $t + dt$ produces a net dividend

$$\left[(\mathbf{a}_j - i_{j,t})\right] K_{j,t}dt$$

  where:

$$\Phi(i) = \frac{\log(1 + \phi i)}{\phi}$$

4.1.3 Markets

- Capital is freely traded (subject to a no shorting constraint), at price Q_t that follows

$$dQ_t = Q_t[\mu_{q,t}dt + \sigma_{q,t} \cdot dW_t]$$

- Stochastic discount factors for households and experts

$$dS_{j,t} = -S_{j,t}[R_t dt + \pi_{j,t} \cdot dW_t]$$

  where r_t is the riskfree rate and $\pi_{j,t}$ are the shadow risk price vectors.

- Experts can issue "equity" securities (more precisely "pro-rata strips"), but up to a limit – an unspecified moral-hazard problem forces those experts to retain a minimum amount $\underline{\chi}$ percent exposure to their asset portfolio.

- Experts face skin-in-the-game constraint:

$$\chi_t \geq \underline{\chi}$$

  where χ_t is fraction of aggregate risk exposure held by experts that is "retained."

- Households face dynamically complete markets.

4.1.4 Single-Agent Optimization Problem

- Agent j maximizes its continuation utility $U_{j,t}$ over all possible sequences of consumption $C_{j,t}$, capital holdings $K_{j,t}$, investment rates $I_{j,t}$, aggregate market hedges $\theta_{j,t}$, subject to the following dynamic budget constraint:

$$\begin{aligned}
\frac{dN_{j,t}}{N_{j,t}} &= \left[\mu_{n,j,t} - \frac{C_{j,t}}{N_{j,t}}\right] dt + \sigma_{n,j,t} \cdot dW_t + \widetilde{\sigma}_{n,j,t} d\widetilde{W}_t \\
\mu_{n,j,t} &= r_t + \frac{Q_t K_{j,t}}{N_{i,t}} \left(\mu_{R,j,t} - r_t\right) + \theta_{j,t} \cdot \pi_t \\
\sigma_{n,j,t} &= \frac{Q_t K_{j,t}}{N_{j,t}} \sigma_{R,t} + \theta_{j,t} \\
\widetilde{\sigma}_{n,j,t} &= \frac{Q_t K_{j,t}}{N_{j,t}} \sqrt{\widetilde{V}_t} \\
\theta_{j,t} &\in \Theta_{j,t}
\end{aligned}$$

  where $\mu_{R,j,t}$ is the drift and $\sigma_{R,t}$ vector of Brownian exposures for the investment returns.

- The financial constraint sets $\Theta_{j,t}$ of agent j is equal to:
  - $\Theta_{e,t} = \{(\chi_t - 1)\frac{Q_t K_{j,t}}{N_{j,t}} \sigma_{R,t}, \chi_t \geq \underline{\chi}\}$: "skin-in-the-game" restriction faced by experts
  - $\Theta_{h,t} = \mathbb{R}^d$: unconstrained sets faced by households

4.1.5 Market Equilibrium

- Markov equilibrium – aggregate state vector $X_t = \{W_t, Z_t, V_t, \widetilde{V}_t\}$:
  - Endogenous state: $W_t = \frac{N_{e,t}}{N_{e,t} + N_{h,t}}$ (experts' relative wealth share)
  - Exogenous states: $\{Z_t, V_t, \widetilde{V}_t\}$
- All agent strategies are functions of the state vector X_t:
  - Consumption-wealth ratio $C_{j,t}/N_{j,t}$
  - Investment rate $I_{j,t}$
  - Expert's leverage $\chi_t K_{j,t} Q_t / N_{j,t}$
- Market clearing:
  - r_t equalizes supply and demand of risk-free debt
  - Q_t equalizes supply and demand of physical capital
- The continuation value $U_j(N, X)$ for an agent of type j with wealth N in aggregate state X takes the following form:

$$U_j(N, X) = n \exp\left(\zeta_j(X)\right)$$

4.2 Solving the Hansen, Khorrami, Tourre Model: `mfr.modelSoln`

In this section, we explain and document how to use the set of tools in `mfr.modelSoln` to solve and analyze the generalized framework from the Hansen, Khorrami, and Tourre (2018) paper described in the previous section.

4.2.1 Solving a Model

4.2.1.1 Initializing a model. `mfr.modelSoln` contains a Python class called `Model`, which is initialized with a Python dictionary that contains the parameters. To initialize a model, after importing the module, you need to first create a dictionary to store the parameters and initialize a `Model` instance with the dictionary. Section 4.2.2 contains the parameters required to initialize a `Model`. However, a short cut that we recommend is to import the default set of parameters and modify them. For example (the code below is contained in `examples/test2DModel.py` with more detailed comments):

```
## Import modules
import mfr.modelSoln as m
import numpy as np

## Copy default parameters
params = m.paramsDefault.copy()

## Modify copied dictionary
params['nV'] = 50
params['nWealth'] = 100
params['sigma_V_norm'] = 0.132
params['cov21'] = -0.1
```

`params = m.paramsDefault.copy()` copies the default dictionary and the lines of code below it modify the default dictionary. After that, you may initialize the model:

```
## Initialize model
testModel = m.Model(params)
```

If your parameters are not degenerate, the code snippet above should run without giving you any message. The successful outcome would be object `testModel`, initialized with `params` (throughout this section, we will use `testModel` as the example object). However, if you entered a degenerate set of parameters, the code above will generate an error message. For a more detailed discussion on error messages, please refer to section 6.1.

4.2.1.2 Solving a model. After successfully initializing a model, you can try to solve it by using the `.solve()` method. That is,

```
## Solve model
testModel.solve()
```

4.2.1.3 Results. Your call to `.solve()` will start the numerical algorithm to solve the model. When the method is finished, attribute `status` would be altered to one of the following:

1. `status = -3`: The program stopped due to user termination (such as pressing CTRL+C).
2. `status = -1`: The solution method gives an error because the value function contained `NaN` or `inf` during the solution algorithm.
3. `status = 0`: The model is not solved because the maximum number of allowed iterations was reached but the model still did not converge.
4. `status = 1`: The program converged to a model solution.

To check, you can either call the `status` attribute, or use the `.printInfo()` method by calling `testModel.printInfo()`. Also, when `.sovle()` is run, the method generates three files, `log.txt`, `parameters.txt`, and `parameters.json` into a folder specified in the parameters dictionary (i.e. `params['folderName']`).

When `.solve()` is completed, the following files will be exported into `params['folderName']`:

- `log.txt`: the log file; discussed extensively in section 6.2
- `parameters.txt`: a text file that contains the parameters of the model
- `parameters.json`: a json file that contains the parameters of the model (this file contains the same information as in `parameters.txt`, but in a different format)
- `xi_e_final.dat`, `xi_h_final.dat`, `chi_final.dat`, `kappa_final.dat`: value and policy functions of the model
- `dent.txt`: stationary density of the model, if you used `.computeStatDent()` (described in section 4.2.4)

4.2.1.4 Monitoring progress and stopping program. You may use the `log.txt` file to monitor the progress of the program. For more explanation on this file, please refer to section 6.2. If the program is taking too long and you wish to terminate, please press CTRL+C. The program will stop and return `-3` as the `status`.

4.2.1.5 Restarting. After the program stops running (such as after reaching convergence or user termination), `testModel` will store the value and policy functions (`.zetaE()`, `.zetaH()`, `.chi()` and `.kappa()`), regardless of whether convergence is reached. You may restart the program by using the value and policy functions stored as the initial guesses by calling `testModel.restart()`. This function is helpful, for example, when you involuntarily stopped the program.

4.2.1.6 Summary. We summarize the attributes and methods we have covered in this section and document related methods:

- `__init__`: You don't need to explicitly call `__init__`. To initialize a model, simply call `testModel = m.Model(params)`, where `params` is a parameters dictionary.
- `.solve()`: Method to solve a model
- `.status`: (Integer) Status of the model
- `.params`: (Dictionary) Parameters of the model. **Note**: simply modifying this attribute and run `.solve()` would not solve a new model; instead, the program will solve the model before you modified `.params`. To modify the parameters of a model and solve the resulting new model, you need to modify this attribute and call `.updateParameters()` or use `.updateParameters(params = newParams)`, where `newParams` is a dictionary that contains the parameters that you would like to modify and the new values (more below). Afterwards, running `.solve()` would solve the resulting new model.
- `.updateParameters(params = {})`:

  Arguments: (Dictionary) This argument is optional. If you want to change the parameters of an existing object, provide the new parameters in a dictionary and call this function with the new dictionary. For example, suppose you want to change `rho_e` to 2 and `gamma_e` to 5, you can run `testModel.updateParameters({'rho_e' = 2, 'gamma_e' = 5})`. Alternatively, you may modify the `params` attribute of the object and call `.updateParameters()` without any argument. That is, `testModel.params['rho_e'] = 2; testModel.params['gamma_e'] = 5; testModel.updateParameters()`.

  Output: None; this method updates the parameters of a model.
- `.restart()`:

  Arguments: None

  Output: None; this method restarts the numerical algorithm by using the value and policy functions (i.e. `.zetaE()`, `.zetaH()`, `.chi()` and `.kappa()`) stored in the object as the initial guesses.
- `.printInfo()`: Prints status of the `Model`

  Arguments: None

  Output: Depending on the value of attribute `status`, it prints different messages:

  - `-3`: `User terminated program before completion.`
  - `-2`: `There has been no attempt to solve to model. Plesae use method solve().`
  - `-1`: `Program resulted in error. Value functions contain nan or inf.`

  - 0: `Program did not converge after reaching [integer] iterations (the maximum).`
  - 1: `Program converged. Took [integer] iterations and [float] seconds. [float]% of the time was spent on dealing with the linear systems.`

- `.loadParams(fileName)`:

  Arguments: (String) Path of a `.json` file that contains the parameters of a model. Whenever `.solve()` is used, the program exports a `.json` file that contains the parameters. This function provides a convenient way to update a model by loading a `.json` file of a different model. **Caution:** This method does not load `preLoad` in the `.json` file.

  Output: None

4.2.2 Setting Parameters

This section explains the parameters needed to initialize and solve a model. All the parameters are tabulated in the Appendix (Section 8.1). For a more detailed discussion, please refer to paper Hansen, Khorrami, and Tourre (2018).

4.2.2.1 Parameters of the model: In Table 1, we summarize the parameters needed for the model and their symbols, interpretations and default values.

4.2.2.2 A note on `covij`: The parameter `covij` in Table 1 needs more explanation. The number of state variables must be equal to the number of shocks (this will be automatically enforced, as you only need to tell the program the number of state variables). When you have more than one state variable, you must specify the correlation matrix of the shocks, starting at row 2, as the first row will be automatically filled in with 1 in the first coordinate and 0 in the rest. For example, suppose you have two state variables, your correlation matrix would be 2×2, and you need to provide `cov21` as shown in the matrix below, as elements `*` will be automatically computed:

$$\begin{bmatrix} * & * \\ \texttt{cov21} & * \end{bmatrix}$$

Suppose `params['cov21'] = 0.5`, then the program would compute the matrix as

$$\begin{bmatrix} 1.0 & 0 \\ 0.5 & \sqrt{1-0.5^2} \end{bmatrix}$$

If you have three state variables and three shocks, the 3×3 correlation matrix would look like:

$$\begin{bmatrix} * & * & * \\ \texttt{cov21} & * & * \\ \texttt{cov31} & \texttt{cov32} & * \end{bmatrix}$$

Suppose you provide ρ_{21}, ρ_{31}, and ρ_{32} as your inputs for `params[cov21]`, `params[cov31]`, and `params[cov32]`. Then the program will compute the correlation matrix as:

$$\begin{bmatrix} 1 & 0 & 0 \\ \rho_{21} & \sqrt{1-\rho_{21}^2} & 0 \\ \rho_{31} & \rho_{32} & \sqrt{1-\rho_{31}^2-\rho_{32}^2} \end{bmatrix}$$

4.2.2.3 Parameters of the state space grid: You need to discretize the state variables in order to create and solve a model. In Table 2, we tabulate the parameters needed for this purpose. In order to properly initialize the state space, please note the following:

- Whenever you turn on a state variable, both the volatility of that state variable has to be strictly positive and the number of grid points has to be positive. For example, if `params['sigma_V_norm']` is positive, then you must have a nonzero integer for `params['nV']` (and vice versa).

- Furthermore, if $\widetilde{V}$ and V are state variables, in addition to their volatility, the means of $\widetilde{V}$ and V, i.e. `Vtilde_bar` and `V_bar` in Table 1, must be strictly positive, as the means are used to compute their ergodic standard deviations.

- `wMin` and `wMax` are the numerical boundaries of state variable W, which is open set $(0, 1)$. Numerically, these two parameters must be close to, but not, 0 and 1.

- When initializing exogenous state variables stochastic volatility V and idiosyncratic volatility $\widetilde{V}$, the lower bound would be set as $\max(0, \mu - k \times \sigma)$ and upper bound $\mu + k \times \sigma$, where μ is the mean, set by parameters ending in `_bar` in Table 1, and σ is the square root of the volatility parameters (that begin with `sigma_`) in Table 1. k is set by `numSds` in Table 2.

- When initializing exogenous state variable growth Z, the range is $[\mu - k \times \sigma, \mu + k \times \sigma]$, where σ and k are the same as the bullet point above.

4.2.2.4 Parameters of the numerical algorithm: The parameters in Table 3 pertain to the numerical algorithm used to solve the model. We will first explain and then recommend a few guidelines in choosing the parameters. Furthermore, if your model fails to converge, please also took at section 6.3 for more tips.

- `method`: With an explicit scheme (i.e. setting `method` to `1`), the algorithm will take a lot more outer loops to converge, but each iteration is relatively fast. With an implicit scheme (i.e. setting `method` to `2`), the opposite is true: it takes fewer outer loops to converge, but each outer loop takes longer.

- `dt`: Since `dt` is the time step size of the time derivative, regardless of your choice of `method`, a smaller `dt` makes the algorithm more stable as the approximation of the time derivative becomes finer. Therefore, when you're solving a highly nonlinear model, using a small `dt`, such as 0.01, may be necessary. Similarly, regardless of `method`, the numerical algorithm takes more outer loops to converge with smaller `dt`s. However,

there is a subtle difference: with an explicit scheme (i.e. `method` as `1`), the time it takes per outer loop remains the same regardless of the size of `dt`. With an implicit scheme, since we're solving it with conjugate gradient, using a smaller `dt` makes the finite difference matrix more diagonally dominant so the time needed for an outer loop to converge decreases with `dt`. Separately, if you insist on using an explicit scheme, `dt` has to be made small with a finer grid (i.e. high values of `nWealth`, `nZ`, `nV`, and `nVtilde`).

Therefore, in general, we recommend the implicit scheme. With highly nonlinear models that require small `dt`s, the implicit scheme (solved with conjugate gradient) is very effective.

- `dtInner`: This parameter only matters if you were solving a model with capital misallocation (i.e. when $a_h > -\infty$). Since the inner loop required to solve for experts' capital holdings is essentially an explicit scheme to solve an ODE, `dtInner`, the time step size, has to be made small with finer grids. For more information, please refer to section 6.4.

- `CGscale`: When you use an implicit scheme, the numerical algorithm solves the linear system via conjugate gradient, which finds an approximate solution. The tolerance for the approximation is `tol` × `dt` × `CGscale`. Therefore, `CGscale` allows you to control the accuracy of conjugate gradient.

4.2.2.5 Parameters for input/output: By default, whenever `.solve()` is run, the program outputs the two sets of files, (1) `log.txt`, `parameters.txt`, and `parameters.json`; and (2) `xi_e_final.dat`, `xi_h_final.dat`, `chi_final.dat`, and `kappa_final.dat`. (1) contains information of the model solved, whereas (2) contains the solved value and policy functions. Furthermore, when you run `.dumpData()` (more description in section 4.2.5), the program will output the model solution. For inputs, we allow you to set your initial guesses or load solutions to other models as your initial guesses. The parameters in this section are used to control the behavior of these options.

You are required to provide the following parameters in Table 4:

- `folderName`: (String) Folder to which the program will export files. If the provided folder does not exist, the program will create it automatically. If it already exists, the program will either prompt an error message or overwrite the existing folder depending on `overwrite`.

- `overwrite`: (Boolean) If `True`, the program will overwrite the existing folder; if `False`, the program will prompt an error message asking you to change `folderName` or `overwrite`.

- `exportFreq`: (Integer) Frequency at which the program exports data in the course of the numerical algorithm. For example, if `params['exportFreq'] = 100000`, the program will export the value functions and equilibrium quantities into `folderName` every 100,000 outer loops. The exported filenames would be, for example, `zeta_e_100000.dat`. **Caution**: Frequent exports could slow down the numerical algorithm. This feature is provided for numerical diagnostics only.

The following parameters in Table 4 are optional:

- `preLoad`: (String) If you would like to load solutions from another model as the initial guesses, you can set `preLoad` as the folder name where the solutions are saved. For example, if you solved a model with `params['folderName'] = 'defaultModel'` and exported the solutions using `.dumpData()` (more explanation in section 4.2.5). The solutions would be saved with file names such as `zeta_e_final.dat` in folder `'defaultModel'`. Suppose you then want to solve a different model, but use the solutions saved in `'defaultModel'` as your initial guesses, you can simply set `params['preLoad'] = 'defaultModel'`.

- `suffix`: (String) By default, when you call `.dumpData()` (more explanation in section 4.2.5), the solutions of a *solved* model are saved with suffix `_final`, such as `zeta_e_final.dat`. However, you may not always want to load the solutions of a solved model, but the value functions at a specific iteration. For example, you could be exporting the value functions every 10,000 iterations and you would want to re-start solving a model at the 50,000th iteration. If so, you simply need to set `params['suffix'] = '50000'` and the program will load files such as `zeta_e_50000.dat` as the initial guesses. This parameter is only relevant if `preLoad` is a key in the parameters dictionary.

- `zetaEGuess`, `zetaHGuess`, `chiGuess` and `kappaGuess`: (`np.array`s) You can customize your initial guesses for ζ_e, ζ_h, χ, and κ. You must provide your initial guesses in a "vectorized" format. Suppose you want to solve a model `testModel` with two state variables W and V, with 100 discretization points in W and 20 points in V. Then your guesses must be `np.array`s of `(2000, 1)`, where, for example, `params['zetaEGuess'][2]` would correspond to the value of ζ_e at `testModel.W()[2]` and `testModel.V()[2]`. **Note:** If you fill in both `'preLoad'` and customized guesses `'*Guess'`, the customized guesses will be ignored and the program will load the solutions in `'preLoad'` instead.

4.2.3 Model Diagnostics I

After a model is successfully solved (i.e. when `status` is `1`), you may look at the equilibrium quantities such as risk prices, interest rate, etc. To access them, you may call the following methods and attributes (all of these methods and attributes are summarized in Tables 5, 6, 7).

- `stateMat`: a `pandas` dataframe that contains the grid on which the model was solved.

- `W()`, `Z()`, `V()`, and `Vtilde()`: The state variables wealth, growth, stochastic volatility, and idiosyncratic volatility, respectively. If any of these state variables were not included, then its corresponding method would return a vector of that state variable's mean as defined in the parameters dictionary used to initialize the model. If a state variable is included, it is a `np.array` that stores the same data as the corresponding column in `stateMat`.

- `muW()`, `muZ()`, `muV()`, and `muVtilde()`: The calculated drifts of the state variables. Note that only state variables with a positive number of grid points allocated and a positive volatility norm parameter will have drifts available.
- `sigmaW()`, `sigmaZ()`, `sigmaV()`, and `sigmaVtilde()`: The calculated volatilities of the state variables.
- `piE{# shock}()`: The expert risk price for the shock number placed in the brackets. For example, suppose you include an idiosyncratic volatility shock along with the standard TFP shock and no other shocks. Then to get the expert risk price associated with the TFP shock it would be `piE1()` and the expert risk price associated with the idiosyncratic volatility shock is `piE2()`.
- `piH{# shock}()`: The same as `piE{# shock}()`, except with household risk prices.
- `muQ()` and `sigmaQ()`: The calculated drift and volatility of capital price.
- `xiE()` and `xiH()`: Value functions for the experts and households, respectively.
- `chatE()` and `chatH()`: Experts' and households' consumption to wealth ratios
- `kappa()`: Experts' capital retention
- `chi()`: Experts' equity holdings
- `r()`: Interest rate
- `q()`: Capital price
- `deltaE()` and `deltaH()`: Experts' and households' bonus risk premium
- `excessReturnKExperts()` and `excessReturnKHouseholds()`: Experts' and households' excess return on capital
- `I()`: Investment rate
- `CoverI()`: Aggregate consumption-to-investment ratio
- `IoverK()`: Aggregate investment-to-capital ratio
- `CoverK()`: Aggregate consumption-to-capital ratio
- `IoverY()`: Aggregate investment-to-output ratio
- `CoverY()`: Aggregate consumption-to-output ratio
- `muK()` and `sigmaK()`: Drift and volatility of capital.
- `muY()` and `sigmaY()`: Drift and volatility of aggregate growth.
- `piETilde()` and `piHTilde()`: Experts' and households' idiosyncratic risk prices.

- `leverage()`: Experts' leverage.
- `muC()`, `muCh()`, and `muCe()`: Drifts of aggregate consumption, household consumption, and expert consumption respectively.
- `sigmaC()`, `sigmaCh()`, and `sigmaCe()`: Volatility of aggregate consumption, household consumption, and expert consumption respectively.
- `dent`: stationary density of the model. By default, it is `None` and will be changed to a `np.array` after successfully calling `.computeStatDent()` (more in section 4.2.4)
- `FKmat`: the Feynman-Kac matrix, the transpose of which is associated with the stationary density of the model. By default, it is `None` and will be changed to a `np.array` after successfully calling `.computeStatDent()` (more in section 4.2.4). It is the output `FKmat` as described in section 5.1.2.
- `marginals`: a dictionary that maps from strings to `np.array`s. Each key-value pair contains the marginal distribution of a state variable (indicated by the key). For example, `testModel.marginals['V']` returns a `np.array` that contains the marginal distribution of state variable stochastic volatility.
- `apMoments`, `macroMoments`: dictionaries that store the asset pricing moments and macroeconomic moments. By default, they are empty and will be populated after calling `.computeMoments(varNames = [])` (more in section 4.2.4). Each key in the dictionaries references to a list of two elements, with the first element being the ergodic mean and second the ergodic standard deviation.
- `corrs`: dictionary that contains the ergodic correlations of the equilibrium quantities. By default, it is empty and will be populated after calling `.computeCorrs(varNames = [])` (more in section 4.2.4).
- `inverseCDFs`: a dictionary of inverse CDFs of the state variables. Each key-value pair maps a string, such as `'W'`, into a function handle (a `scipy` interpolant). For example, if you were to find the median of W, you can use `testModel.inverseCDFs['W'](0.5)`.
- `params`: a dictionary that stores the model's parameters
- `gridSizeList`: a list of grid sizes in the order of `[nWealth, nZ, nV, nVtilde]` (the list would only include the state variables that are active. For example, when a model has W and V as state variables, the attribute would be `[nWealth, nV]`.

All the equilibrium quantities are stored in a "vectorized" format. Suppose you solved a model, `testModel`, with two state variables W and V, with 100 discretization points in W and 20 points in V (that is, in this parametrization, `params['sigma_V_norm']` is strictly greater than zero, and `params['nWealth'] = 100` and `params['nV'] = 20`). Then `testModel.muW()`, for example, will be a 2000×1 `np.array`, with each point in the array corresponding to the point in state space. For example, `testModel.muW()[2]` is the value of μ_w at `testModel.W()[2]` and `testModel.V()[2]`. Sometimes you may want to `reshape`

the "vectorized" array into an n-dimensional array. If so, you can do `testModel.muW().reshape(testModel.gridSizeList, order = 'F')`.

To diagnose the performance of the numerical algorithm, you may use the following attributes:

- `timeOuterloop`: Time taken to solve each iteration of the algorithm.
- `timeLynSys`: Time taken to solve the linear systems within each outer loop.
- `errorsE` and `errorsH`: The errors in the value functions of experts and households.
- `CGEIters` and `CGHIters`: The number of conjugate gradient iterations for each outer loop for experts and households respectively; only useful if you use the implicit scheme.

4.2.4 Model Diagnostics II

The `Model` class contains a set of methods to analyze a solved model.

4.2.4.1 Stationary density: `.computeStatDent()` The module uses `mfr.sdm` to compute stationary density (for a more detailed discussion, refer to section 5.1.2). No need to import `mfr.sdm` separately, as it is already imported with `mfr.modelSoln`. After successfully solving a model (i.e. when `status` is `1`), you can call `.computeStatDent()` to find its stationary density. If you've specified a folder name for `params['preLoad']`, this method, by default, will search for exported stationary density in the folder. To force the function to compute stationary density from scratch, regardless of whether a solution already exists, set `forceCompute` to `True` (more below).

- Arguments:
  - Same as those in `mfr.sdm.computeDent`; documented in section 5.1.2)
  - forceCompute: (Boolean) By setting this argument to `True`, the function will compute the model's stationary density, regardless of whether stationary density has already been exported. If `False`, this function will first search for exported stationary density files, and will only move onto computing stationary density if it fails to find any existing files. This argument defaults to `False`.
- Output: None; it adds the discretized stationary density, a `np.array`, to attribute `dent`. It also exports the discretized stationary density as `dent.txt` in the folder specified in `params['folderName']`.

4.2.4.2 Moments: `.computeMoments(varNames = [])` This method finds the ergodic mean and standard deviation of the equilibrium quantities in section 4.2.3.

- Arguments:

  - varNames: (List) A list of strings, where each string is the method used to call the equilibrium quantity without the parentheses. For example, if you were to compute the ergodic mean and standard deviation of wealth share and the interest rate, you would call `testModel.computeMoments(['W', 'r'])` (Recall that the corresponding methods to call wealth share and interest rate are `.W()` and `.r()`). By default, it appends the state variables to argument `varNames` and computes moments for aggregate growth (In other words, if you put in an empty list as the argument, the method will compute the moments for the state variables and aggregate growth). The list of variables for which you can compute moments is in Table 7.

- Output: None; it populates attributes `.apMoments` and `.macroMoments`, dictionaries where each key refers to a list of two elements, with the first element being the ergodic mean and second the ergodic standard deviation.

4.2.4.3 Correlations: `.computeCorrs(varNames = [])` This method finds the ergodic correlations among the equilibrium quantities in section 4.2.3.

- Arguments:
  - varNames: (List) A list of strings, where each string is the method used to call the equilibrium quantity without the parentheses. Same as the argument in `.computeMoments(varNames = [])`, you would use `testModel.computeCorrs(['W', 'r'])` to find the correlation between wealth share and the interest rate. By default, it appends the state variables to argument `varNames` (In other words, if you put in an empty list as the argument, the method will compute the correlations among the state variables). The list of variables for which you can compute correlations is in Table 7.

- Output: None; it populates attribute `.corrs`, dictionary where each key contains the correlation between a set of two equilibrium quantities.

4.2.4.4 Shock elasticities: `.computeShockElas(pcts = {'W':[.1,.5,.9]}, points = np.matrix([]), T = 100, dt = 1, perturb = 'C', bc = {})` The module uses `mfr.sem` to compute shock elasticities of the model (for a more detailed discussion, refer to section 5.1.2). No need to import `mfr.sem` separately, as it is already imported with `mfr.modelSoln`.

- Arguments:
  - `pcts`: (Dictionary) A dictionary of percentiles from each variable from which to start elasticity calculations (x_0). For example, if I wanted to use the 30th and 70th percentiles of both wealth and stochastic volatility, then I would have `pcts = {'W':[.3,.7], 'V': [0.3, 0.7]}` (a total of four starting points). The number of keys in the dictionary should equal to the number of state variables in the model. This argument requires that the stationary density be calculated already.

- `points`: (Numpy array or matrix) This is an alternative to the `pcts` argument. Rather than specify percentiles, you have the option to pass in the actual points of each of the states. For example, if I wanted to start from wealth share equal to 0.2 and 0.4 and aggregate stochastic volatility equal to 1.0 and 1.5, I would use `points = np.matrix('0.2, 0.4; 0.5, 1.5')`. The dimension of this matrix should be $k \times n$, where k is the number of starting points and n the number of state variables. If you've used `pcts`, your input for `points` would be ignored.
- `T`: (Integer) The number of time periods for which to calculate the elasticities. Defaults to 100.
- `dt`: (Float) The size of the time step to use in calculating the shock elasticities. For example, if your model parameters are quarterly and you want to see 50 years of shock elasticities, you would want to set `dt` to 0.25 and `T` to 200. `dt` defaults to 1.
- `perturb`: (String) The name of the variable that you want to shock as part of the elasticity computation. Possible parameters are `'W'` (the wealth share), `'Q'` (the capital price), `'C'` (aggregate consumption), `'Ce'` (expert consumption), and `'Ch'` (household consumption). Additionally, `'Z'` (growth), `'V'` (stochastic volatility volatility), and `'Vtilde'` (idiosyncratic volatility) can be included if they were given positive grid points and positive volatility in the parameters. Default is `'C'`.
- `bc`: (Dictionary) Boundary conditions for the PDE. Refer to the documentation for `bc` in Section 5.2.2 for more information. By default the dictionary is empty. When you use an empty dictionary for `bc`, the function uses zero first derivatives as the boundary conditions.

- Output: None; it populates the following attributes
  - `expoElasMap`: (Dictionary) a dictionary of `pandas` dataframes that contain the shock *exposure* elasticities. Each key refers to a perturbed variable for which you've computed the shock elasticities. For example, if you've computed shock elasticities for both households' consumption and experts' consumption, `expoElasMap` would contain two keys, `Ce` and `Ch`. `testModel.expoElasMap['Ce']` would return a `pandas` dataframe, with the shock exposure elasticities at each period.
  - `priceElasExpertsMap`: (Dictionary) similar to `expoElasMap`, but the dataframes contain the shock *price* elasticities using *experts'* SDF.
  - `priceElasHouseholdsMap`: (Dictionary) similar to `priceElasExpertsMap`, but the dataframes contain the shock *price* elasticities using *households'* SDF.

4.2.5 Other Methods

- `.smoothResults(degree = 9)`: When a model with capital misallocation is solved (i.e. $a_h > 0$), the results may need to be smoothed. This method fits a k degree polynomial for κ and q, while k is determined by argument `degree`.

  Arguments:

- `degree`: (Integer) The degree of polynomials such that the program will use to approximate κ and q.

Output: None; the program will use the fitted functions for κ and q to recompute their derivatives and the equilibrium quantities that depend on κ, q, and their derivatives.

- `.findIndices(pcts, useQuintiles = True)`: Given a set of points or of percentiles, this function returns the indices of where the state variables equal those points or percentiles. For example, suppose you solve a model with 300 points in wealth share and 20 points in idiosyncratic volatility. Then if you want to find experts' risk price for the first shock at the median level of idiosyncratic risk, you would call `testModel.piE1()[testModel.findIndices({'Vtilde': 0.5})]` to get a `(300,)` numpy vector of expert risk prices as a function of wealth share.

  Arguments:

  - `pcts`: (Dictionary) Depending on the value used for the following argument (`useQuintiles`), a dictionary of either percentiles or points for which to find the indices. If percentiles are used, then the function will return the index for the points at that percentile. If points are used and the point used is represented exactly in the state stape somewhere, the function will return the indices where that state variable exactly equals the point. If the point is not exactly represented, then the function will provide a warning and return the indices where the state is closest to equaling that point. At most one point may be used for each state.
  - `useQuintiles`: (Boolean) A boolean representing whether percentiles are used. Set `False` to use points instead of percentiles. Defaults to `True`.

  Output: a `np.array` of indices

- `.plot(varName, pts = [], col = 0, useQuintiles = True, title = '', height = 500, width = 500, show = True, fancy = False)`: Plots a variable as a one-dimensional function, holding all other state variables fixed at a user-specified point. Arguments:

  - `varName`: (String) The equilibrium quantity to plot. Options include all the variables specified in section 4.2.3. For example, if you were to plot μ_w, you would have `varName = 'muW'`.
  - `pts`: (List) A list of dictionaries where each dictionary contains represents a line in the chart and contains the percentile or value at which state variables should be fixed. This argument should be left as an empty list when the model is one dimensional. With $n > 1$ state variables this argument should be a list dictionaries. Each dictionary represents one line on the chart and should contain $n - 1$ keys, with each key representing a state variable, and the remaining state variable not included in the dictionaries would be the x-axis. For example, suppose you solved a model with W and V, you can do the following `testModel.plot('piE1', pts = [{'V': .5}, {'V': .9}])` to plot the risk price of experts with respect to the TFP shock as a function of W by fixing V at the 50th and 90th percentiles. The resulting chart would contain two lines, both as a function of W.

- `col`: (Integer) The column of the variable that you would like to plot. For example, suppose you solved a 3d model and `sigmaW` would have three columns, each column corresponding to the loading of the shock. Suppose you would like to plot the second loading of the shock, you then need to set `col = 1` (the count starts at `0`).
- `useQuintiles`: (Boolean) Similar to the same-named argument for the `findIndices` function. If `True`, argument `pts` would be interpreted as percentiles. If `False`, argument `pts` would be interpreted as values. Defaults to `True`.
- `title`: (String) A title to apply to the graph once plotted.
- `height`: (Integer) Specifies the image height. Defaults to 500.
- `width`: (Integer) Specified the image width. Defaults to 500.
- `show`: (Boolean) If `True`, the function will display a chart. If not, the function will `return` the plot as an object.
- `fancy`: (Boolean) If `True`, the function will use package `plotly` to render charts. If `False`, the function will use `matplotlib`.

Output: depends on the `show` argument. By default, `show` is `True` so it displays the chart using `matplotlib`. If `show` is `False`, it will return an object of the figure. Note: To use `plotly`, you must be in a Jupyter notebook environment, and `show` and `fancy` must be `True`.

- `.dumpPlots(pts = [], fancy = False)`: This function exports a series of plots of selected equilibrium quantities. This function simply calls `.plot()` repeatedly to generate a series of plots.

  Argument:

  - `pts`: (List) The argument is the same as `pts` in `plot` (documented right above).
  - `fancy`: (Boolean) If `True`, the function will use package `plotly` to render charts. If `False`, the function will use `matplotlib`.

  Output: it exports a series of `.png` files. By default, it generates plots using `matplotlib`. To generate plots using `plotly`, you must have implemented the optional installation step in section 2.1.3.

- `.dumpData()`: This function exports the data for each of the variables calculated by the `solve` function as binary-encoded `.dat` files in the output folder specified by the `'folderName'` parameter. It should only be called after a model has been succesfully solved.

  Argument: None

  Output: exports a set of `.dat` files.

- `.printParams()`: Prints the model parameters.

  Arguments: None

  Output: Prints model parameters

5 Stationary Density and Shock Elasticities

The MFR Suite has two other modular, independent toolboxes for computing stationary density and shock elasticities. Note that you don't have to go through section 4 in order to use these two independent toolboxes.

> Note: These two modules use Python package `numba` with Just-in-Time (JIT) compilation. Therefore, in each given session, the first application of these two modules would be slightly slow (since `numba` has to compile). After the first application, the modules will run faster.

5.1 Stationary Density

5.1.1 Overview

`mfr.sdm` computes the stationary density of a stochastic process for an arbitrary number of dimensions. Consider the following $n \times 1$ vector X_t, satisfying the Markov diffusion:

$$dX_t = \mu(X_t)dt + \sigma(X_t)dW_t \tag{2}$$

Note $\Sigma(X) := \sigma(X_t)\sigma(X_t)'$, then the stationary density $h(X)$ satisfies the Kolmogorov forward equation (a.k.a., Fokker-Planck equation:

$$0 = -\sum_{i=1}^{d} \partial_{x_i} [\mu_i h] + \frac{1}{2} \sum_{1 \le I,j \le d} \partial_{x_i x_j} [\Sigma_{ij} h] \tag{3}$$

Rather than directly solving the forward equation, our numerical algorithm finds the stationary density for the process X_t by discretizing the Kolmogorov backward operator associated with the stochastic process X_t, using a finite difference scheme that is monotone (in the sense of Barles and Souganidis, 1991). Note that this step was performed when solving for the equilibrium of the model. The resulting matrix P can be interpreted as an intensity matrix (with row sums adding up to zero, diagonal entries that are all negative and off diagonal entries that are all positive), which represents an approximating discretized process. Computing the ergodic density of the state vector X_t then consists in finding the ergodic distribution of the approximating process. From the theory of continuous-time Markov chains, this is the positive left eigen-vector associated with the zero eigen-value of P, i.e., the vector h that solves $h'P = 0$. In practice, we transpose P and solve $P'h = 0$ for h.

5.1.2 Computing Stationary Distribution: `mfr.sdm.computeDent`

This function solves Fokker-Planck formulation of the PDE governing the stationary distribution of the state variables. The function takes in the following inputs, defined in a similar way as for the `computeElas` function described in 5.2.2:

- `stateMat`: (Tuple or `numpy` matrix) 1. A tuple containing a numpy-array formatted domain for each of the states. For example, suppose the model has two state variables and each state variable has 100 grid points. The `stateMat` argument would be a tuple containing two numpy arrays of shape `(100,)`. 2. A `numpy` matrix that contains the numerical grid on which you would like to compute the stationary density. Using the same example of two state variables with 100 grid points in each, attribute `.shape` of the matrix would be `(10000,2)`.

- `model`:
  (Dictionary) A dictionary that contains the drifts and diffusions of the model. These are its keys:

  - `muX`: (Function or `numpy` matrix) 1. A function[4] that computes the drifts of the state variables (i.e. μ_X as in (2)) for each of the grid points in `stateMat`. Suppose there are n state variables. Given a $k \times n$ `numpy` matrix `points`, where each row represents a point, `model['muX'](points)` must return a $k \times n$ `numpy` matrix, with each row corresponding to the drift vector of the state variables. 2. Alternatively, it could be a `numpy` matrix of $K \times n$, where K is the total number of grid points, such that the matrix contains the numerical values of the drifts of all the grid points.
  - `sigmaX`: (List of functions or list of `numpy` matrices) 1. A list of functions, where each function computes the diffusion of a state variable in the order of the list (e.g. the *first* element of the list computes the diffusion of the *first* state variable in `stateMat`). For example, suppose there are n state variables and m shocks. `model['sigmaX']` is a list of two elements. Further suppose that `points` is a $k \times n$ matrix, where each row represents a point. Then `model['sigmaX'][0](points)` returns a $k \times m$ matrix that contains the diffusion of the first state variable at these points. 2. A list of `numpy` matrices. Similar to a list of functions, instead of containing functions, each element is a matrix which contains the numerical values of the diffusions of the state variables. Using the same example, `model['sigmaX'][0]` would be a $K \times m$ matrix (K is the total number of grid points), where each row is the diffusion vector of the first state variable at each of the grid points.

- `bc`:
  (Dictionary) A dictionary that specifies the boundary conditions in this form:

$$0 = a_0 + a_1 \cdot \frac{\partial}{\partial X} h(X) + a_2 \cdot \frac{\partial^2}{\partial X^2} h(X) \tag{4}$$

  where a_0 is a scalar, a_1 and a_2 are $1 \times n$ vectors, and X is a $1 \times n$ vector of state variables (n is the number of state variables). The user should configure `bc` in this way:

[4] If you only have numerical values of the drift(s) of your state variable(s) (instead of closed-form expressions, e.g. the state variable(s) are endogenous, you can create an interpolant using scipy; the same applies to the items below.

- `a0`: a scalar that corresponds to a_0 in (4)
- `first`: a $1 \times n$ numpy matrix that corresponds to a_1 in (4)
- `second`: a $1 \times n$ numpy matrix that corresponds to a_2 in (4)
- `natural`: a boolean (`True` or `False`) that determines whether to use natural boundaries[5]; if `True`, the fields above would be ignored.

Note: By default, we set `bc['natural'] = True` and recommend that the user stay with this default, as there is typically no reason to fill in other boundary conditions.

- `usePardiso`: (Boolean) A boolean option to use the Intel package called Pardiso[6] to solve the linear system. Defaults to `False`.
- `iparms`: (Dictionary) If `usePardiso` is set to `True`, Pardiso's default parameters[7] will be used. The `iparms` parameter allows the user to input a dictionary with changes for those default parameters. Note that while the Pardiso site indexes the parameters at one, this function argument indexes them at zero. For example, a user wishing to set what is listed on the website as iparm(1) to -1 would set `iparms = {0:-1}`.

When shocks are correlated, the program might need to use the explicit scheme to solve the PDE, as the sufficient condition of monotonicity (Barles and Souganidis, 1991) is not satisfied[8]. The following (optional) arguments are for the explicit scheme:

- `explicit`: (Boolean) `False` by default. It is used to turn on the explicit scheme. If `False`, the rest of the arguments in this list are irrelevant;
- `dt`: (Float) the time step size used in the explicit scheme;
- `tol`: (Float) tolerance for the explicit scheme;
- `maxIters`: (Integer) the maximum number of iterations;
- `verb`: (Boolean) `False` by default. if `True`, program will print out information for each iteration (`verb` is short for *verbatim*).

Lastly, there is one more optional argument:

[5]Natural boundaries simply take the limit of PDE (12), where second derivatives at the boundary are approximated by one grid point inside the boundary.

[6]Pardiso is a sparse matrix solver that factorizes matrices with parallel processors. A detailed explanation can be found here: https://software.intel.com/en-us/mkl-developer-reference-fortran-intel-mkl-pardiso-parallel-direct-sparse-solver-interface

[7]The default parameters can be found here https://software.intel.com/en-us/mkl-developer-reference-fortran-pardiso-iparm-parameter

[8]Note: We suggest that the user attempt the default method (i.e. `explicit = False`) first, as the default method is computationally cheap and does not necessarily fail since monotoncity is not a necessary condition.

- `betterCP`: (Boolean) `True` by default. If `False`, the program will use the following finite difference approximation scheme for cross partials:

$$f_{xy}(x,y) = \frac{f(x+h,y+k) - f(x+h,y-k) - f(x-h,y+k) + f(x-h,y-k)}{4hk}$$

if `True`, the program will use the alternative finite difference approximation for cross partials. With the alternative approximation, one can have a better chance of preserving monotonicity:

$$\begin{aligned} f_{xy}(x,y) =& \frac{1}{2hk}\Big(f(x+h,y+k) - f(x+h,y) - f(x,y+k) \\ & + 2f(x,y) - f(x-h,y) - f(x,y-k) + f(x-h,y-k) \Big) \end{aligned}$$

The function outputs the following:

- `dent`: a `numpy` array of the unconditional probability mass of each of the state variables. The array is in a *vectorized* format, with each point corresponding to the value of the discretized density function at a grid point in `stateGrid` (described below). This array can be reshaped; for example, suppose there are two state variables having `n1` and `n2` points respectively. Then if `density = dent.reshape([n1, n2], order = 'F')`, the i,j entry of density is the unconditional probability of the ith point of state 1 and the jth point of state 2.

- `FKmat`: a `scipy` sparse matrix object that contains the Faynman-Kac matrix, the transpose of which is used to calculate the stationary density.

- `stateGrid`: an $K \times n$ matrix, where k is the total number of grid points. For example, in the case where there are two state variables with 70 and 100 points respectively, then `stateGrid` is 7000×2.

5.2 Shock Elasticities

5.2.1 Overview

`mfr.sem` computes shock elasticities defined as follows. For a full discussion on shock elasticities, please refer to Borovička and Hansen (2016), Borovička, Hansen, and Scheinkman (2014), and Hansen (2012).

Assume we are given stochastic processes

$$\begin{aligned} dX_t &= \mu_X(X_t)dt + \sigma_X(X_t)dW_t \\ d\log C_t &= \mu_C(X_t)dt + \sigma_C(X_t)dW_t \\ d\log S_t &= \mu_S(X_t)dt + \sigma_S(X_t)dW_t \end{aligned} \tag{5}$$

where X is n-dimensional, the Brownian motion dW_t has k shocks (hence k dimensions), and

$$\begin{aligned}\mu_X &: \mathbb{R}^n \mapsto \mathbb{R}^n \\ \sigma_X &: \mathbb{R}^n \mapsto \mathbb{R}^{n\times k} \\ \mu_C, \mu_S &: \mathbb{R}^n \mapsto \mathbb{R} \\ \sigma_C, \sigma_S &: \mathbb{R}^n \mapsto \mathbb{R}^{1\times k}\end{aligned}$$

In the equations above, C_t is the cash flow or consumption process and S_t is the stochastic discount factor. We are interested in computing the **shock exposure elasticities** and **shock price elasticities**.

For a given cash flow or consumption process, its shock **exposure elasticity** can be computed as

$$\varepsilon_C^1(x,t) = \nu(x)\cdot\left[\sigma_X'(x)\left(\frac{\partial}{\partial x}\log E\Big[\frac{C_t}{C_0}|X_0=x\Big]\right)+\sigma_C(x)\right] \tag{6}$$

$$\varepsilon_C^2(x,t) = \frac{E\Big[\frac{C_t}{C_0}\nu(X_t)\cdot\sigma_C(X_t)|X_0=x\Big]}{E\Big[\frac{C_t}{C_0}|X_0=x\Big]} \tag{7}$$

where $\nu:\mathbb{R}\mapsto\mathbb{R}^k$ encodes the direction of the shocks.

To compute the price elasticities for a given cash flow process C_t and stochastic discount factor S_t, we first compute the shock cost elasticities by substituting process C in (6) and (7) with a new process SC (and define them as ε_{SC}^1 and ε_{SC}^2). To get the **price elasticities**, we compute:

$$\varepsilon_C^1(x,t) - \varepsilon_{SC}^1(x,t) \tag{8}$$

and

$$\varepsilon_C^2(x,t) - \varepsilon_{SC}^2(x,t) \tag{9}$$

All expressions in (6) and (7) can be computed explicitly, with the exception of conditional expectations

$$E\Big[\frac{C_t}{C_0}|X_0=x\Big] \tag{10}$$

$$E\Big[\frac{C_t}{C_0}\nu(X_t)\cdot\sigma_C(X_t)|X_0=x\Big] \tag{11}$$

We compute the conditional expectations by using the Feynman-Kac formula.

Define

$$\phi_t(x) \equiv E\Big[\frac{C_t}{C_0}\phi_0(X_t)|X_0 = x\Big]$$

Then $\phi_t(x)$ satisfies:

$$\frac{\partial\phi}{\partial t} = \Big(\mu_C + \frac{1}{2}|\sigma_C|^2\Big)\phi + \Big(\mu_X + \sigma_X\sigma_C\Big)\frac{\partial\phi}{\partial x} + \frac{1}{2}\text{trace}[\sigma_X\partial_{XX'}\phi\sigma_X'] \tag{12}$$

Let $\phi_0(x) = 1$ and $\phi_0(x) = \nu(x)\cdot\sigma_C(x)$ for (10) and (11) respectively.

5.2.2 Computing Shock Elasticities: `mfr.sem.computeElas`

This is the function that computes the shock exposure and price elasticities (both the first and second types). Note that this function is independent from the Hansen, Khorrami, and Tourre (2018) model. Although it shares similar inputs as `mfr.sdm.computeStatDent()` (as in section 5.1.2), there are some differences. It takes in the following arguments:

- `stateMat`: (Tuple) A tuple containing a numpy-array formatted domain for each of the states. For example, suppose the model has two state variables and each state variable has 100 grid points. The `stateMat` argument would be a tuple containing two numpy arrays of shape `(100,)`.

- `model`:
  (Dictionary) A dictionary that contains the information described in (5). Specifically, it has the following keys:

  - `muX`: (Function) A function[9] that computes the drifts of the state variables (i.e. μ_X as in (5)) for each of the grid points in `stateMat`. Suppose there are n state variables. Given a $k \times n$ `numpy` matrix `points`, where each row represents a point, `model['muX'](points)` must return a $k \times n$ `numpy` matrix, with each row corresponding to the drift vector of the state variables.

  - `sigmaX`: (List of functions) A list of functions, where each function computes the diffusion of a state variable (i.e. σ_X as in (5)) in order of the list (i.e. the *first* element of the list computes the diffusion of the *first* state variable in `stateMat`). For example, suppose there are n state variables and m shocks. `model['sigmaX']` is a list of two elements. Further suppose that `points` is a $k \times n$ matrix, where each row represents a point. Then `model['sigmaX'][0](points)` returns a $k \times m$ matrix that contains the diffusion of the first state variable at these points.

[9] If you only have numerical values of the drift(s) of your state variable(s) (instead of closed-form expressions, e.g. the state variable(s) are endogenous, you can create an interpolant using scipy; the same applies to the items below.

- `muC`: (Function) Similar to `muX`, a function that computes μ_C, the drift of the perturbed process, as in (5). Suppose there are n state variables. `model['muC'](points)` must return a $k \times 1$ `numpy` matrix, where `points` is a $k \times n$ matrix. **Note:** the number of columns of `points` is n, but the dimensions of the output are $k \times 1$.
- `sigmaC`: (Function) A function that computes σ_C, the diffusion of the perturbed process, as in (5). Suppose `points` is a $k \times n$ matrix. Then `model['sigmaC'](points)` must return a $k \times m$ matrix, where n is the number of state variables and m is the number of shocks (Note that m is not necessarily equal to n).
- `muS`: (Function) Similar to `muX` and `muC`, a function that computes μ_S, the drift of the stochastic discount factor, as in (5). Using the same example, `model['muS'](points)` must return a $k \times 1$ `numpy` matrix, where `points` is a $k \times n$ matrix.
- `sigmaS`: (Function) A function that computes σ_S, the diffusion of the perturbed process, as in (5). Suppose `points` is a $k \times n$ matrix. Then `model['sigmaS;'](points)` must return a $k \times m$ matrix, where n is the number of state variables and m is the number of shocks.
- `dt`: (Float) The length of time step;
- `T`: (Integer) Total number of time periods. Suppose `dt = 1/12` implies one month and `model['T'] = 120`. The total time that the elasticities are solved for is 10 years.

- `bc`:
  (Dictionary) A dictionary that specifies the boundary conditions in this form:

$$0 = a_0 + a_1 \cdot \phi(X) + a_2 \cdot \frac{\partial}{\partial X}\phi(X) + a_3 \cdot \frac{\partial^2}{\partial X^2}\phi(X) \tag{13}$$

  where a_0 is a scalar, a_1, a_2, and a_3 are $1 \times n$ vectors, and X is a $1 \times n$ vector of state variables. The user should configure `bc` in this way:

  - `a0`: a scalar that corresponds to a_0 in (13)
  - `level`: a $1 \times n$ numpy matrix that corresponds to a_1 in (13)
  - `first`: a $1 \times n$ numpy matrix that corresponds to a_2 in (13)
  - `second`: a $1 \times n$ numpy matrix that corresponds to a_3 in (13)
  - `natural`: a boolean (`True` or `False`) that determines whether to use natural boundaries[10]; if `True`, the fields above would be ignored.

  Note that right now, we don't allow the user to specify their own cross partial derivatives at the boundary.

  For example, to implement the boundary conditions $0 = \frac{\partial}{\partial x}\phi(x)$ (i.e. first derivatives are set to zero), one should set up `bc` by doing:

[10]Natural boundaries simply take the limit of PDE (12), where second derivatives at the boundary are approximated by one grid point inside the boundary.

```
bc = {}
bc['a0']          = 0
bc['first'] = np.matrix([1, 1], 'd')
bc['second'] = np.matrix([0, 0], 'd')
bc['third'] = np.matrix([0, 0], 'd')
bc['level'] = np.matrix([0, 0], 'd')
bc['natural'] = False
```

- `x0`: (`numpy` matrix) A numpy matrix that contains all the starting points (i.e. x in (6)). For l starting points, `x0` should be an $l \times n$ matrix (n being the number of states).
- `usePardiso`: (Boolean) A boolean option to use the Intel package called Pardiso[11] to solve the linear system. Defaults to `False`.
- `iparms`: (Dictionary) If `usePardiso` is set to `True`, Pardiso's default parameters[12] will be used. The `iparms` parameter allows the user to input a dictionary with changes for those default parameters. Note that while the Pardiso site indexes the parameters at one, this function argument indexes them at zero. For example, a user wishing to set what is listed on the website as iparm(1) to -1 would set `iparms = {0:-1}`.

Finally, we have one optional argument:

- `betterCP`: (Boolean) `True` by default. This argument is the same as the same-named argument in `mfr.sdm.computeDent` in section 5.1.2.

The function returns exposure and price elasticities. Suppose the user inputs l starting points (i.e. `x0` has l rows) and T time periods. Then the function will return the following:

- `expoElas`: an object with two attributes: the `firstType` and `secondType`. Each attribute is an $l \times k \times T$ matrix. The first index corresponds to the starting point, the second index corresponds to the shock number, and the third corresponds to time. As the name suggests, `firstType` is the first type shock exposure elasticities for the ith starting point (the ith row in input `x0`) up to T time periods, whereas `secondType` is the second type shock exposure elasticities.
- `priceElas`: similar to `expoElas`, but contains the price elasticities.

[11] Pardiso is a sparse matrix solver that factorizes matrices with parallel processors. A detailed explanation can be found here: https://software.intel.com/en-us/mkl-developer-reference-fortran-intel-mkl-pardiso-parallel-direct-sparse-solver-interface

[12] The default parameters can be found here https://software.intel.com/en-us/mkl-developer-reference-fortran-pardiso-iparm-parameter

6 Troubleshooting

6.1 Interpreting Errors and Warnings

The MFR Suite has a set of built-in error messages and warnings that aim to inform you of wrong settings and degenerate parameters. We provide a more detailed explanation in the two sections below, the first of which deals with module `mfr.modelSoln`, described in section 4, while the second `mfr.sdm` and `mfr.sem`, described in section 5.

6.1.1 Module `mfr.modelSoln`

6.1.1.1 Error: `Trivial volatility is assigned to [state variable] but [number] grid points are assigned to [state variable]. Set the number of grid points to zero.`

Explanation: You probably set one of the parameters `nZ`, `nV`, and `nVtilde` to be nonzero but left the corresponding volatility (`sigma_Z_norm`, `sigma_V_norm`, or `sigma_Vtilde_norm`) to be zero. In order to have a state variable, the volatility of that state variable must be strictly positive.

6.1.1.2 Error: `Nontrivial volatility is assigned to [state variable] but [number] grid points are assigned to [state variable]. Increase the number of grid points in [state variable].`

Explanation: You probably set one of the parameters `sigma_Z_norm`, `sigma_V_norm`, or `sigma_Vtilde_norm` to be nonzero but left the corresponding number of grid points (`nZ`, `nV`, and `nVtilde`) to be zero. In order to have a state variable, you need to specify a strictly positive number of grid points for that state variable.

6.1.1.3 Error: `Please insert a positive number for numSds`

Explanation: When you include an exogenous variable, the program will initialize k numbers of standard deviations below and above the mean. `numSds` specifies the number k so it has to be positive.

6.1.1.4 Error: `Vol of idiosyncratic vol is positive implies mean of idiosyncratic vol has to be positive.`

Explanation: When you include idiosyncratic volatility as an exogenous state variable, the mean of idiosyncratic volatility has to be positive, as idiosyncratic volatility follows a Cox–Ingersoll–Ross process whose mean is used to compute the ergodic standard deviation.

6.1.1.5 Error: `Inputs for the covariance matrix are incorrect. Check the inputs for shock [number].`

Explanation: The program requires you to input the lower triangular of the local correlation matrix starting at row 2. For example, if you have three shocks, you should, in theory, have a 3×3 matrix. In the program, you need to specify:

$$\begin{bmatrix} * & * & * \\ \texttt{cov21} & * & * \\ \texttt{cov31} & \texttt{cov32} & * \end{bmatrix}$$

where matrix elements `cov**`, such as `cov12`, are required inputs from you, and matrix elements `*` are not required.

6.1.1.6 Warning: `Time step is [float]; beware that a small time step could take excessive time.`

Explanation: Given our false transient method in solving the underlying PDEs, the time step parameters (`dt` and `dtInner`) are the step size of the finite difference approximation for the time derivative. A smaller time step implies a more accurate approximation and more stability, but longer time to converge.

6.1.1.7 Warning: `One of the following is prompting this warning: (1) chiUnderline is very close to zero, (2) chiUnderline is less than nu_newborn, or (3) a_e and a_h are very close to each other. You may get a degenerate wealth distribution or may not compute stationary density at all since the model is closed to a complete markets model (cases 1 and 3) or the distribution becomes degenerate in case 2 (check the HKT paper).`

Explanation: If experts and households have symmetric preferences, when experts have access to perfect risk sharing (case 1) or when households and experts are equally productive (case 3), the model is close to a complete markets model and resembles a single agent model, thereby making the wealth distribution irrelevant and numerically degenerate. When `params['chiUnderline']` is less than `params[nu_newborn ]` (case 2), the wealth distribution is degenerate: check Hansen, Khorrami, and Tourre (2018) for details.

6.1.1.8 Warning: `Model solution failed to converge with explicit scheme. We suggest that you reduce the time step size of the inner loop (dtInner) [or outer loop (dt)] to [float].`

Explanation: When you decide to use the explicit scheme to solve a model or when $a_h > 0$ (i.e. when an explicit scheme is used to solve the ODE related to capital holdings), the outcome of the model solution is very sensitive to the time step size. With a finer grid, we recommend, through this warning, a time step size when the model solution fails to converge. Note that this time step size is a recommendation based on the Courant–Friedrichs–Lewy condition, but not a rule.

6.1.2 Modules `mfr.sdm` and `mfr.sem`

6.1.2.1 Warning: `Stationary density computed could be degenerate. We suggest that you use the explicit scheme by setting explicit = True.`

Explanation: This warning props up when the default method of `mfr.sdm.computeDent()` uses returns a degenerate distribution (this could happen, for example, when the state variables have nonzero local correlations). If so, the function prompts the user to deploy the explicit scheme instead, which could entail longer run time.

6.1.2.2 Warning: `Stationary density computed could be degenerate. We suggest that you reduce the time step size dt to [float]`

Explanation: When using the explicit scheme to compute the stationary distribution, i.e. `mfr.sdm.computeDent(explicit = True)`, the convergence of the scheme is very sensitive to time step size `dt`. This warning gives a suggestion (not a rule) on the choice of `dt`.

6.2 Interpreting the `log.txt` File and Potential Bottlenecks

Whenever `.solve()` is run, the program will generate a set of three files: `parameters.txt`, `parameters.json`, and `log.txt`. The first two contain information of the parameters in `.txt` and `.json` formats. The last file, `log.txt`, contains information of the numerical algorithm and records its performance. Whenever a model fails to converge, the `log.txt` file is the first place to check. The `log.txt` file contains information of each iteration. Below, we paste the log of an iteration as an example where the user has chosen the implicit scheme (i.e. `params['method'] = 2`):

```
 ========================= Iteration:  26 ============================
Error for xi_e:  -0.116764
Error for xi_h:  -0.127147
Tolerance not met; keep iterating.
Inner loop is completed; time elapsed:  0.002284; took 1 iterations.
Finished updating equilibrium quantities.  Next step:  update matrices.
Finished updating matrix.
Finished updating the known vector.
Solving systems using the implicit scheme via CG...
Running conjugate gradient solver using 1 threads.
CG error tolerance:  1e-07
Conjugate gradient took 16 (experts) and 18 (households) iterations.
Solved linear systems.
Iteration 26 is completed; time elapsed:  0.007551; time spent on solving
linear systems (or computing matrix vector product in explicit scheme):
0.002026
```

Next, we break the log file into sections and highlight the potential bottlenecks.

6.2.0.1 Part 1: The firs two lines (pasted below) record the time derivative (or error) of the two value functions. The program declares convergence when both of them are smaller than `params['tol']`.

```
Error for xi_e:  -0.116764
Error for xi_h:  -0.127147
```

In a successful run, we expect the time derivative to drop to zero *continuously*, although not necessarily monotonically. Therefore, if you see oscillations in the time derivatives, it probably means that it is not converging and that you need a smaller `dt`.

6.2.0.2 Part 2: Next, the file records information about the inner loop:

```
Inner loop is completed; time elapsed:  0.002284; took 1 iterations.
```

The inner loop is essentially the solution method to solve for experts' capital holdings κ and experts' equity retention χ, which involves an explicit scheme to solve an ODE and an iterative method to find roots. If it is taking too long or it reaches the maximum number of iterations (i.e. `params['maxItersInner']`), it could mean that the solution method is not converging. Since the solution method is an explicit scheme, the choice of `dtInner` is especially sensitive. The first thing to try is to lower `dtInner`.

6.2.0.3 Part 3: The program then moves onto updating equilibrium quantities and constructing the linear systems. If you have chosen the implicit scheme, the program will use conjugate gradient to find an approximate solution. We expect this part to go smoothly, but would like to highlight the last two lines: the second to the last tells you the number of threads used to compute conjugate gradient, and the last tells you the error tolerance for conjugate gradient.

```
Finished updating equilibrium quantities.  Next step:  update matrices.
Finished updating matrix.
Finished updating the known vector.
Solving systems using the implicit scheme via CG...
Running conjugate gradient solver using 1 threads.
CG error tolerance:  1e-07
```

6.2.0.4 Part 4: The program uses conjugate gradient to solve the linear systems and prints out the relevant information. In the last line, it tells you the total amount of time spent on this iteration (i.e. `time elapsed:  0.007551`) and then the amount of time spent on solving the linear systems (i.e `time spent on solving linear systems (or computing matrix vector product in explicit scheme):  0.002026`).

```
Conjugate gradient took 16 (experts) and 18 (households) iterations.
Solved linear systems.
Iteration 26 is completed; time elapsed:  0.007551; time spent on solving
linear systems (or computing matrix vector product in explicit scheme):
0.002026
```

Note: The program starts conjugate gradient with the solution to the previous iteration as its guess (hence a "smart guess"), so we expect the amount of time spent on the linear systems and the number of iterations taken by CG (i.e. the numbers in `Conjugate gradient took 16 (experts) and 18 (households) iterations`) to go down as the program progresses.

6.3 Numerical Algorithm Parameters to Tweak

Because of the nonlinearity and dimensionality of models in the HKT framework, successful numerical solutions may require some user intervention and troubleshooting. In this section, we recommend a few guidelines on tweaking the numerical algorithm parameters in Tables 2 and 3.

6.3.0.1 Time steps: `dt` and `dtInner`. This is the first place to look when your model is not converging. The model solution consists of two loops: the inner loop and the outer loop. The inner loop involves an explicit scheme to solve for an ODE, the solution of which is the experts' capital holdings (κ). Therefore, if you set $a_h > 0$, the inner loop would be nontrivial and could take a significant amount of time and be a major bottleneck. If the number of iterations of the inner loop maxed out, it could mean that the inner loop could not find a solution for κ. You can stabilize the solution by choosing a smaller `dtInner`. When doing so, recall that given the nature of the explicit scheme, the size of `dtInner` is inversely related to the fineness of the grid. That is, for high values of `nWealth`, `nZ`, `nV`, and `nVtilde`, you need a small `dtInner`. When $a_h < 0$, it is unlikely that the inner loop is causing an issue.

Regardless of a_h, the outer loop could fail to converge when the parameter set makes the model highly nonlinear. If you see oscillations in the time derivatives of the value functions (to see this, you can either check the `log.txt` file or plot attributes `.errorsE()` and `.errorsH()`), it could mean that `dt`, which is the time step size, is too big and that you need a smaller `dt`.

Side note: if your model requires a small `dt`, we recommend that you use the implicit scheme (i.e. `params['method'] = 2`), as conjugate gradient benefits from having a diagonally dominant matrix, which the finite difference method with a false transient and a small `dt` would produce. Second, a small `dt` requires more outer loops to converge, but our model solution starts conjugate gradient with solutions to previous time steps as the initial guess ("smart guesses") - the amount of time needed to solve the linear systems goes down as the model solution progresses because the initial guess for conjugate gradient becomes "smarter" over time.

6.3.0.2 Using solutions from similar models. It is reasonable to assume that the value functions are continuous in the parameters. Therefore, if you encounter a difficult model, you could try to solve a model with similar parameters that are less nonlinear. Also, you can start with a simple model, use the solution as the guess for a slightly more difficult model, and repeat until you reach your model of choice. For example, suppose that your model with `params['gamma_e'] = 10` does not converge to a solution, but a model, all else equal, with `params['gamma_e'] = 1` converges. You can use the solutions of the model with `params['gamma_e'] = 1` as the initial guess to solve a model with `params['gamma_e'] = 2`, and repeat until you reach `params['gamma_e'] = 10`.

6.3.0.3 Grid Sizes. Given our solution method is a finite difference approach, a finer grid (high values of `nWealth`, `nZ`, `nV`, and `nVtilde`) implies more accurate approximations and more stability. In particular, endogenous state variable W and exogenous state variable $\widetilde{V}$ might require finer approximations (i.e. `nWealth` and `nVtilde` need to be higher) when a model is highly nonlinear or has badly behaved endogenous constraints (such as $a_h > 0$ and $\kappa < 1$).

6.4 Models with Capital Misallocation

For models with capital misallocation (i.e. when `params['a_h']` is set to be strictly positive), you should take note of the following: (1) `dtInner` is very sensitive to the total number of grid points and should be made small with finer grids. Furthermore, the total number of grid points increases exponentially with the number of state variables. (2) The time it takes to converge is inversely related to the value of `dtInner`. In other words, with two or more state variables, it could take a very long time to solve a model. To see the progress of the model, you may set `params['verbatim'] = 1` so that the program prints out the error of the inner loop, which should continuously (but not necessarily monotonically) go towards `params['innerTol']`.

7 Examples

7.1 He and Krushnamurthy (2013)

This example borrows from He and Krishnamurthy (2013). In the model, the economy has intermediaries (similar to experts in Hansen, Khorrami, and Tourre (2018)) and households, and x is the wealth share of intermediaries (similar to W in Hansen, Khorrami, and Tourre (2018)). All agents have log utility and parameters correspond closely to those used in He and Krishnamurthy (2013).

The model specifications are as follows.

The state variable is characterized by

$$\begin{aligned}\mu_X(x) &= x\Big[-\frac{\ell}{1+\ell}\rho + (\alpha(x)-1)^2\sigma_Y^2\Big]\\ \sigma_X(x) &= x(\alpha(x)-1)\sigma_Y\end{aligned}$$

And the stochastic discount factor by:

$$\begin{aligned}\mu_S(x) &= \frac{\rho}{1+\ell} + \mu_Y - \alpha(x)\sigma_Y^2 + \frac{1}{2}\alpha(x)^2\sigma_Y^2\\ \sigma_S(x) &= \alpha(x)\sigma_Y\end{aligned}$$

We are interested in two cash flows C^1 (aggregate endowment) and C^2 (expert consumption):

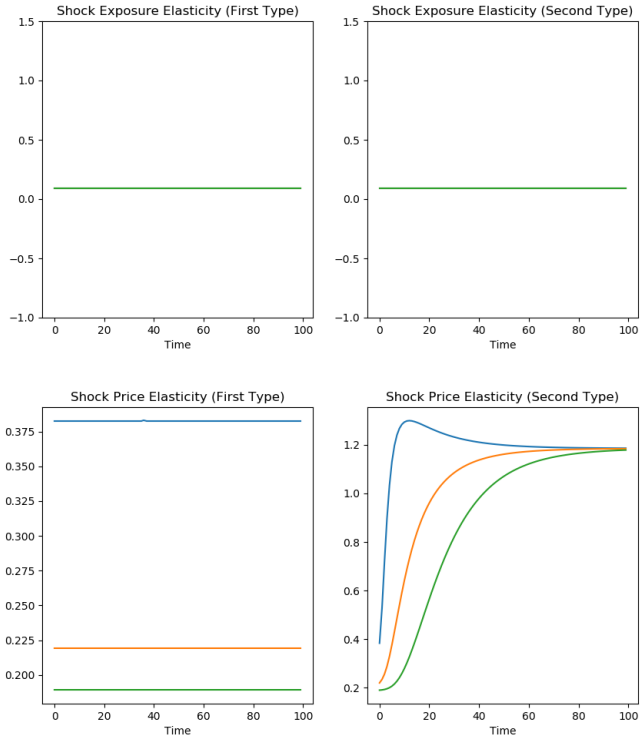
$$\begin{aligned}\mu_{C^1}(x) &= \mu_Y - \frac{1}{2}\sigma_Y^2\\ \sigma_{C^1}(x) &= \sigma_Y\\ \mu_{C^2}(x) &= \mu_S(x) - \rho\\ \sigma_{C^2}(x) &= \sigma_S(x).\end{aligned}$$

The parameters in the equations above are either computed or set as:

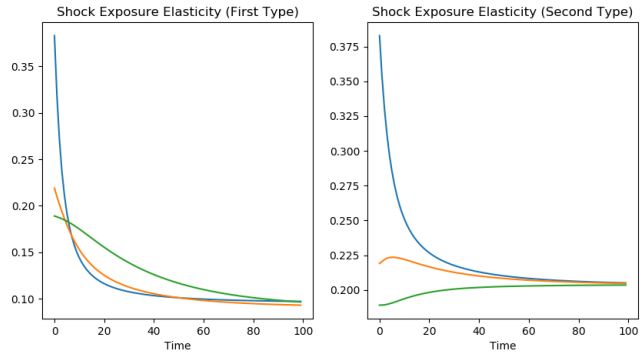
$$\begin{aligned}\mu_Y = 0.02 \quad \sigma_Y = 0.09 \quad m = 4\\ \lambda = 0.6 \quad \rho = 0.04 \quad \ell = 1.84\\ \alpha(x) = \begin{cases}[1-\lambda(1-x)]^{-1}, & \text{if } x > x^*\\ (1+m)^{-1}x^{-1}, & \text{if } x \le x^*.\end{cases}\\ x^* = \frac{1-\lambda}{1-\lambda+m} \in (0,1).\end{aligned}$$

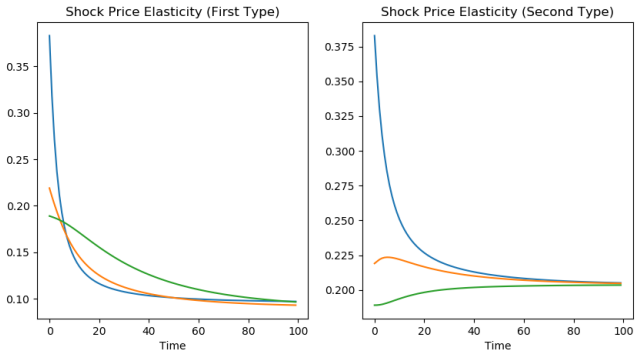
The file `HeKrushnamurthy.py` computes the shock elasticities for both cash flows and the stationary density through `mfr.sem` and `mfr.sdm`. It will render charts below.

The elasticities of the first cash flow C^1:



The elasticities of the second cash flow C^2:





7.2 Bansal and Yaron (2004)

This example `BansalYaron.py` is adapted from Bansal and Yaron (2004). Different from the example in 7.1, the model has two state variables ($n = 2$) and three shocks ($k = 3$). The example computes elasticities for two utility functions (i.e. two SDFs).

The model has two state variables that follow stochastic processes:

$$dX_t^{[1]} = -0.021 X_t^{[1]} dt + \sqrt{X_t^{[2]}} \begin{bmatrix} 0.00031 & -0.00015 & 0 \end{bmatrix} dW_t$$
$$dX_t^{[2]} = -0.013 (X_t^{[2]} - 1) dt + \sqrt{X_t^{[2]}} \begin{bmatrix} 0 & 0 & -0.038 \end{bmatrix} dW_t$$

Where aggregate consumption follows:

$$d \log C_t = 0.0015 dt + X_t^{[1]} dt + \sqrt{X_t^{[2]}} \begin{bmatrix} 0.0034 & 0.007 & 0 \end{bmatrix} dW_t$$

Please note that the parameter settings and specific continuous-time specification comes from the handbook chapter of Hansen, Heaton, Lee, and Roussanov (2007) to well approximate the dynamics used by Bansal and Yaron (2004). As in Hansen (2012), we transform two of the shocks to the macroeconomy in order that one is permanent and the other is transitory. Furthermore, the intertemporal elasticity parameter is unity and the risk aversion parameter is 8.

Suppose that the state dynamics are specified by

$$\mu(x) = \begin{bmatrix} \bar{\mu}_{11} & \bar{\mu}_{12} \\ 0 & \bar{\mu}_{22} \end{bmatrix} \begin{bmatrix} x^{[1]} + \iota_1 \\ x^{[2]} + \iota_2 \end{bmatrix} \text{ and } \sigma(x) = \sqrt{x^{[2]}} \begin{bmatrix} \bar{\sigma}_1 \\ \bar{\sigma}_2 \end{bmatrix}$$

and the consumption dynamics take the following form:

$$\beta(x) = \bar{\beta}_{c,0} + \bar{\beta}_{c,1} (x^{[1]} - \iota_1) + \bar{\beta}_{c,2} (x^{[2]} - \iota_2) \text{ and } \alpha(x) = \sqrt{x^{[2]}} \bar{\alpha}_c$$

We can recast the model by setting:

$$\begin{bmatrix}\bar{\mu}_{11} & \bar{\mu}_{12}\\ 0 & \bar{\mu}_{22}\end{bmatrix} = \begin{bmatrix}-0.021 & 0\\ 0 & -0.013\end{bmatrix}, \begin{bmatrix}\iota_1\\ \iota_2\end{bmatrix} = \begin{bmatrix}0\\ -1\end{bmatrix} \text{ and } \begin{bmatrix}\bar{\sigma}_1\\ \bar{\sigma}_2\end{bmatrix} = \begin{bmatrix}0.00031 & -0.00015 & 0\\ 0 & 0 & -0.038\end{bmatrix}$$
$$\bar{\beta}_{c,0} = 0.0015;\ \bar{\beta}_{c,1} = 1;\ \bar{\beta}_{c,2} = 0$$
$$\bar{\alpha}_c = \begin{bmatrix}0.0034 & 0.007 & 0\end{bmatrix}$$

We compute the shock elasticities for both the recursive utility and power utility SDFs. The SDF of the recursive utility is given by:

$$S_t = \exp(-\delta t)\left(\frac{C_t}{C_0}\right)^{-1}\tilde{M}_t$$

where $d\log\widetilde{M}_t = -\frac{|\tilde{\alpha}|^2}{2}X_t^{[2]}dt + \sqrt{X_t^{[2]}}\tilde{\alpha}\cdot dW_t$.

When we take logs:

$$\begin{aligned}
d\log(S_t) &= -\delta dt - d\log(C_t) + \sqrt{X_t^{[2]}}\tilde{\alpha}\cdot dW_t - \frac{|\tilde{\alpha}|^2}{2}X_t^{[2]}dt\\
d\log(S_t) &= -\delta dt - \left[\bar{\beta}_{c,0}dt + \bar{\beta}_{c,1}X_t^{[1]}dt + \bar{\beta}_{c,2}(X_t^{[2]}-1)dt + \sqrt{X_t^{[2]}}\bar{\alpha}_c\cdot dW_t\right]\\
&\quad + \sqrt{X_t^{[2]}}\tilde{\alpha}\cdot dW_t - \frac{|\tilde{\alpha}|^2}{2}X_t^{[2]}dt\\
&= \left[-\delta - \bar{\beta}_{c,0} - \bar{\beta}_{c,1}X_t^{[1]} - \bar{\beta}_{c,2}(X_t^{[2]}-1) - \frac{|\tilde{\alpha}|^2}{2}X_t^{[2]}\right]dt + \sqrt{X_t^{[2]}}(\tilde{\alpha}-\bar{\alpha}_c)dW_t
\end{aligned}$$

Where $\tilde{\alpha} = (1-\gamma)[(\bar{\sigma}_1)'\bar{v}_1 + (\bar{\sigma}_2)'\bar{v}_2 + \bar{\alpha}_c]$ and $\bar{v}_1$ and $\bar{v}_2$ are computed by:

$$\begin{aligned}
\bar{v}_1 &= \frac{-\bar{\beta}_{(c,1)}}{\bar{\mu}_{11}-\delta}\\
\bar{v}_2 &= \frac{-B \pm \sqrt{B^2-4AD}}{2A}
\end{aligned}$$

Where

$$\begin{aligned}
A &= \frac{1-\gamma}{2}(\bar{\sigma}_1^2 + \bar{\sigma}_2^2)\\
B &= -\delta + \bar{\mu}_{22} + (1-\gamma)(\bar{\alpha}_c)'(\bar{\sigma}_2)' + 2v_1\frac{1-\gamma}{2}\bar{\sigma}_1\bar{\sigma}_2\\
D &= \bar{\mu}_{12}\bar{v}_1 + \bar{\beta}_{c,2} + (1-\gamma)(\bar{\alpha}_c)'(\bar{\sigma}_1)'\bar{v}_1 + \frac{1-\gamma}{2}(\bar{\sigma}_1^2\bar{v}_1^2 - \bar{\alpha}_c^2)
\end{aligned}$$

We also compute the shock elasticities for power utility, whose SDF is given by

$$S_t = \exp(-\delta t)\left(\frac{C_t}{C_0}\right)^{-\gamma}$$

Taking logs yields:

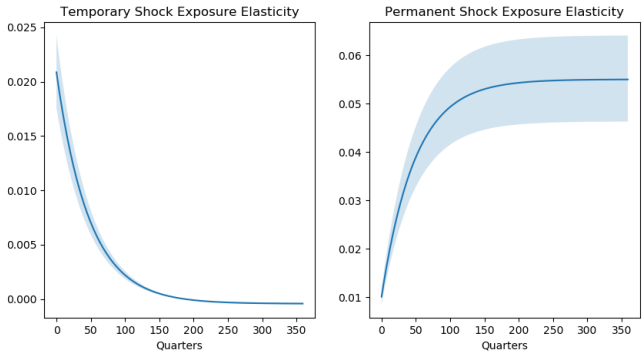
$$\begin{aligned} d\log(S_t) &= -\delta dt - \gamma d\log(C_t) \\ &= -\delta dt - \gamma\left[\mu_C(x)dt + \sqrt{X_t^{[2]}}\bar{\alpha}_c \cdot dW_t\right] \\ &= \left[-\delta - \gamma\mu_C(x)\right]dt + \gamma\sqrt{X_t^{[2]}}(-\bar{\alpha}_c)dW_t \end{aligned}$$

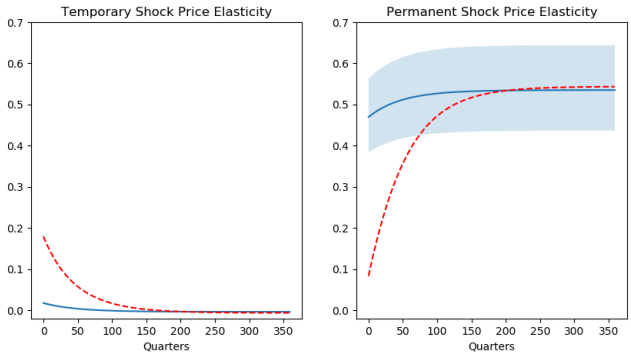
With these equations, we are ready to compute shock elasticities using the toolbox. We will compute the shock elasticities for starting points at the mean of $X^{[1]}$ and the 10th, 50th, and 90th percentiles of $X^{[2]}$ (3 starting points in total). In particular, some of the arguments for function `computeElas` are as follows -

Let $\tilde{S}$ be the total number of grid points:

- `model`
  - `muC`: a function that returns an $\tilde{S} \times 1$ numpy matrix
  - `sigmaC`: a function that returns an $\tilde{S} \times 3$ numpy matrix (remember, there are **3** shocks)
  - `muS`: a function that returns an $\tilde{S} \times 1$ numpy matrix
  - `sigmaS`: a function that returns an $\tilde{S} \times 3$ numpy matrix
  - `muX`: a function that returns an $\tilde{S} \times 2$ numpy matrix (there are **2** state variables)
  - `sigmaX` a list that contains 2 functions, each of which returns an $\tilde{S} \times 3$ numpy matrix
- `x0`: a 3×2 matrix (there are **3** starting points and **2** dimensions)

You should expect the elasticities to look like:





7.3 One-Dimensional Hansen, Khorrami, and Tourre (2018)

The file `test1DModel.py` demonstrates how to use the `modelSoln` class to solve the one-dimensional version of the model in Hansen, Khorrami, and Tourre (2018). In other words, the only shock in this model is the TFP shock. Here we are only interested in using the default parameters, so we create a model object and borrow default parameters from the class.

```
1 import mfr.modelSoln as m
2 params = m.paramsDefault.copy()
3 testModel = m.Model(params)
```

Now that the model object has been defined, we call the `solve` command to get a solution. In addition to solving the model, the program in `test1DModel.py` does the following:

- Computes the stationary density of the state variables with `computeStatDent()`
- Computes the mean and standard deviation of wealth and interest rates, using `computeMoments()`
- Computes the correlation of wealth share and interest rates using `computeCorrs()`
- Computes the shock elasticities starting at the 30th and 70th percentiles of wealth share using `computeShockElas()`. Using the same method, it also computes the shock elasticities starting at wealth share equal to 0.5.

Upon running the example file, you will see the following printed:

- The solution info (`printInfo()`)
- The parameters (`printParams()`)
- The mean and standard deviation of wealth share and interest rate (the `moments` attribute after `computeMoments(['W', 'r'])`)
- The correlation between the wealth share and interest rates (the `corrs` attribute after `computeCorrs(['W', 'r'])`).

7.4 Two-Dimensional Hansen, Khorrami, and Tourre (2018)

In `test2DModel.py` we examine the Hansen, Khorrami, and Tourre (2018) model with both the standard TFP shock as well as a shock to aggregate TFP volatility. To effect this change, we need to alter some of the default parameters.

```
import mfr.modelSoln as m

params = m.paramsDefault.copy()

params['nV']           = 50
params['nWealth']      = 100
params['sigma_V_norm'] = 0.132
params['cov21']        = -0.1

testModel = m.Model(params)
```

Note that since we are adding a second dimension with 50 grid points, we reduce the number of grid points allocated to wealth share from the default 300 to 100 for computational simplicity. Recall that we only need to input the `covij` parameter for $1 < i \leq k$ and $1 \leq j < i$ where k is the number of shocks, equal to 2 in this case.

From here, the example code is identical to that of `test1DModel.py`, except that the elasticity starting points are different since they must be specified for both of the state variables wealth share and aggregate volatility. We use the 30th and 70th percentiles of wealth share and aggregate volatility first, then use wealth share equal to 0.2 and 1.0 with aggregate volaility equal to 0.5 and 1.5. Refer to the documentation on `test1DModel.py` for more information.

8 Appendix

8.1 Parameters for `mfr.modelSoln`

Table 1: Model parameters and their default values (last column)

Model Parameters			
`nu_newborn`	ν	Fraction of newborns designated experts	0.1
`lambda_d`	λ_d	Birth/death rate	0.02
`lambda_Z`	λ_Z	Mean-reversion exogenous TFP growth	0.252
`lambda_V`	λ_V	Mean-reversion aggregate TFP volatility	0.156
`lambda_Vtilde`	$\lambda_{\widetilde{V}}$	Mean-reversion idiosyncratic TFP volatility	1.38
`Vtilde_bar`	$\widetilde{V}$	Mean idiosyncratic TFP volatility	0.0
`Z_bar`	$\bar{Z}$	Mean exogenous TFP growth	0.0
`V_bar`	$\bar{V}$	Mean "normalized" aggregate TFP volatility	1.0
`delta_e`	δ_e	Expert rate of time preference	0.05
`delta_h`	δ_h	Household rate of time preference	0.05
`a_e`	a_e	Expert TFP	0.14
`a_h`	a_h	Household TFP[1]	-1
`rho_e`	ρ_e	Expert inverse EIS	1.0
`rho_h`	ρ_h	Household inverse EIS	1.0
`phi`	ϕ	Adjustment cost parameter	3.0
`gamma_e`	γ_e	Expert RRA	1.0
`gamma_h`	γ_h	Household RRA	1.0
`sigma_K_norm`	$\|\|\sigma_K\|\|$	Mean TFP volatility	0.04
`sigma_Z_norm`	$\|\|\sigma_Z\|\|$	Exogenous TFP growth volatility	0.00
`sigma_V_norm`	$\|\|\sigma_V\|\|$	Volatility of "normalized" aggregate TFP volaility	0.0
`sigma_Vtilde_norm`	$\|\|\sigma_{\widetilde{V}}\|\|$	Volatility of "normalized" idiosyncratic TFP volaility	0.0
`chiUnderline`	$\underline{\chi}$	Minimum expert skin-in-the-game	0.5
`alpha_K`	α_K	Depreciation rate	0.05
`equityIss`		1: Expert's constant equity retention constraint 2: Minimum expert's equity retention constraint[2]	2
`covij`		Coordinates of the correlation matrix of the shocks[3]	0

[1] Any negative number for `a_h` means $-\infty$.

[2] If `1`, experts' equity retention would be set as `chiUnderline` everywhere; if `2`, the minimum of experts' equity retention is bounded from below by `chiUnderline`, but it is not necessarily the case that equity retention is `chiUnderline` everywhere.

[3] We give an extensive explanation on this set of parameters in 4.2.2.2.

Table 2: Grid parameters and their default values (last column)

Grid Parameters		
`numSds`	Number of standard deviations (for exogenous states)	5
`nWealth`	Number of grid points for wealth	300
`nZ`	Number of points for growth	0
`nV`	Number of points for stochastic volatility	0
`nVtilde`	Number of points for idiosyncratic volatility	0
`wMin`	Minimum for wealth grid	0.01
`wMax`	Maximum for wealth grid	0.99

Table 3: Numerical algorithm parameters and their default values (last column)

Numerical Algorithm Parameters		
`method`	1: Use the explicit scheme 2: Use the implicit scheme	2
`dt`	Outer loop time step	0.1
`dtInner`	Inner loop time step	0.1
`tol`	Algorithm error tolerance	0.00001
`innerTol`	Algorithm inner loop error tolerance	0.00001
`verbatim`	1: Print out iteration information of the inner loop -1: Do not print out iteration information of inner loop	-1
`maxIters`	Maximum iterations allowed for the algorithm	4,000
`maxItersInner`	Maximum iterations allowed for the algorithm inner loop	2,000,000
`CGscale`	CG tolerance scaler	1.0

Table 4: I/O parameters and their default values (last column)

I/O Parameters		
`folderName`	Name of output folder to save all output	'model0'
`overwrite`	'Yes' to overwrite output folder, 'No' otherwise	'Yes'
`exportFreq`	Data export frequency	10000
`preLoad`	Folder from which the program loads guesses (optional)	empty
`suffix`	Suffix of the `.dat` files (optional)	`'final'`
`zetaEGuess`	guess for ζ_e (optional)	`np.array`
`zetaHGuess`	guess for ζ_h (optional)	`np.array`
`chiGuess`	guess for χ (optional)	`np.array`
`kappaGuess`	guess for κ (optional)	`np.array`

8.2 Equilibrium Quantities and Variables in Hansen, Khorrami, and Tourre (2018)

Table 5: Attributes related to the modeling freamwork in Hansen, Khorrami, and Tourre (2018)

`stateMat`	a `pandas` dataframe that contains the grid on which the model was solved.
`dent`	Stationary density of the model
`FKmat`	Feynman-Kac matrix, the transpose of which is used for the stationary density
`marginals`	Dictionary of marginal densities of the state variables
`apMoments`, `macroMoments`	Ergodic moments (mean and standard deviation) of equilibrium quantities
`corrs`	Ergodic correlations of equilibrium quantities
`inverseCDFs`	Dictionary of inverse marginal CDFs of the state variables
`params`	Parameters of the model
`gridSizeList`	List of grid sizes of the state variables

Table 6: Attributes useful in diagnosing the performance of the numerical algorithm

`timeOuterloop`	Time taken to solve each iteration of the algorithm (outer loop)
`timeLynSys`	Time taken to solve the linear systems within each outer loop
`errorsE`, `errorsH`	Error (i.e. time derivative) of experts and households per iteration
`CGEIters`, `CGHIters`	Number of steps taken for conjugate gradient in each outer loop for experts' and households' value functions

Table 7: Methods to access the equilibrium quantities in Hansen, Khorrami, and Tourre (2018)

`W()`	Wealth share of the experts	W
`Z()`	Aggregate growth rate	Z
`V()`	Stochastic volatility	V
`Vtilde()`	Idiosyncratic volatility	$\widetilde{V}$
`muW()`, `sigmaW()`	Drift and volatility of wealth share	μ_W, σ_W
`muZ()`, `sigmaZ()`	Drift and volatility of aggregate growth	μ_Z, σ_Z
`muV()`, `sigmaV()`	Drift and volatility of stochastic volatility	μ_V, σ_V
`muVTilde()`, `sigmaVTilde()`	Drift and volatility of idiosyncratic volatility	$\mu_{\widetilde{V}}$, $\sigma_{\widetilde{V}}$
`piE{# shock}()` `piH{# shock}()`	Risk prices of experts/households for shock j, indicated by {# shock}	π_e^j, π_h^j
`deltaE()` `deltaH()`	Experts'/Households' Bonus Risk Premium	Δ_e, Δ_h
`excessReturnKExperts()` `excessReturnKHouseholds()`	Experts'/Households' Excess return on capital	$\mu_{R_e} - r$ $\mu_{R_h} - r$
`chatE()` `chatH()`	Experts'/Households' consumption to wealth ratio	$\hat{c}_e$, $\hat{c}_h$
`xiE()` `xiH()`	Experts'/Households' value functions	ξ_e, ξ_h
`muQ()`, `sigmaQ()`	Drift and volatility of capital price	μ_q, σ_q
`kappa()`	Experts' capital retention	κ
`chi()`	Experts' equity holdings	χ
`r()`	Interest rate	r
`q()`	Capital price	q
`I()`	Investment rate	I
`CoverI()`	Consumption-to-investment ratio	C/I
`IoverK()`	Investment-to-capital ratio	I/K
`CoverK()`	Consumption-to-capital ratio	C/K
`IoverY()`	Investment-to-output ratio	I/Y
`CoverY()`	Consumption-to-output ratio	C/Y
`Ieverage()`	Experts' leverage	$\kappa\chi/W$
`muK()`, `sigmaK()`	Drift and volatility of capital	μ_k, σ_k
`muY()`, `sigmaY()`	Drift and volatility of aggregate growth	μ_Y, σ_Y
`piETilde()`, `piHTilde()`	Experts' and households' idiosyncratic risk prices	$\tilde{\pi}_e$, $\tilde{\pi}_h$
`muC()` `muCe()`, `muCh()`	Drift of aggregate consumption, experts' consumption, and households' consumption	μ_C μ_{C_e} μ_{C_h}
`sigmaC()` `sigmaCe()`, `sigmaCh()`	Volatility of aggregate consumption, experts' consumption, and households' consumption	σ_C σ_{C_e} σ_{C_h}
`muSe()`, `sigmaSe()`	Drift and volatility of experts' SDF	μ_{S_e}, σ_{S_e}
`muSh()`, `sigmaSh()`	Drift and volatility of households' SDF	μ_{S_h}, σ_{S_h}

9 Change Log

9.1 Version 0.1.1

- Added the option to compute shock elasticities in the Jupyter GUI by perturbing wealth share (W)
- Added plots of return on capital diffusion (σ_R) in the Jupyter GUI
- Added plots of risk price differences ($\pi_e - \pi_h$) in the Jupyter GUI
- Changed `init_notebook_mode(True)` to `init_notebook_mode(False)` so that the GUI can display charts when offline
- Removed `displayModeBar = True` in functions that involve Plotly so that Plotly plots won't display the modebar by default
- Fixed bug where when preloading κ and χ as solutions
- In `.computeStatDent()` of `mfr.modelSoln`, added the feature to export stationary density and preload stationary density. Added the argument `forceCompute` so that the user can opt to compute stationary density even if the density is already computed

9.2 Version 0.1.2

- Fixed the calculation of experts' capital retention (κ) and experts' equity holding (χ) when 1) households' productivity (a_h) equals experts' productivity (a_e), and 2) there is no idiosyncratic risk state.
- Added a new method `.calDriftDiffusion()` to `mfr.modelSoln`. Advanced users can pass into any $f(X)$ (a function of state X, defined on the state grids. For example, f(X) could be model.muCe()). The method will return the drift and diffusion coefficients of $f(X)$, which are also defined on the state grids.
- Fixed y-axis scaling issue where when showing exposure elasticity plots in the Jupyter GUI.

References

Bansal, Ravi, and Amir Yaron, 2004, Risks for the long run: A potential resolution of asset pricing puzzles, *Journal of Finance*.

Borovička, Jaroslav, and Lars Peter Hansen, 2016, Term structure of uncertainty in the macroeconomy, *Handbook of Macroeconomics* 2, 1641–1696.

Borovička, Jaroslav, Lars Peter Hansen, and José A. Scheinkman, 2014, Shock elasticities and impulse responses, *Mathematics and Financial Economics* 8.

Brunnermeier, Markus K., and Yuliy Sannikov, 2014, A macroeconomic model with a financial sector, *American Economic Review* 104, 379–421.

Hansen, Lars Peter, 2012, Dynamic Valuation Decomposition within Stochastic Economies, *Econometrica* 80.

——— , John Heaton, Junghoon Lee, and Nikolai Roussanov, 2007, Intertemporal substitution and risk aversion, *Handbook of Econometrics, Volume 6A*.

Hansen, Lars Peter, Paymon Khorrami, and Fabrice Tourre, 2018, Comparative valuation dynamics in models with financing restrictions, Working Paper.

He, Zhiguo, and Arvind Krishnamurthy, 2013, Intermediary Asset Pricing, *The American Economic Review* 103(2).