A Solution Method for Continuous-Time Models[*]

Adrien d'Avernas[†], Damon Petersen[‡], and Quentin Vandeweyer[‡]

July 25, 2023

Abstract

We propose an algorithm capable of solving a general class of continuous-time asset pricing models with heterogeneous agent models, in a fast and standardized way. We rely on a finite difference algorithm and the Stern-Brocot Tree decomposition of Bonnans, Ottenwaelter, and Zidani (2004) for fast and stable convergence in settings with up to two endogenous and stochastic state variables. We provide an open source software package, PyMacroFin, that includes an object-oriented interface for model definition and solution for any model in this general class.

[*]Quentin Vandeweyer thanks the Alfred P. Sloan Foundation, the CME Group Foundation, Fidelity Management & Research, and the MFM initiative for the support when working on this work as part of his PhD dissertation.

[†]Stockholm School of Economics

[‡]University of Chicago Booth School of Business

1 Introduction

The financial crisis of 2008 generated a resurgence of interest in the interaction between macroeconomic and financial variables. In particular, there is a growing demand for models able to capture non-linear dynamics and time-varying risk premia. An important part of this research effort has been undertaken by introducing financial frictions and heterogeneity in classical consumption-based asset pricing models. For instance, seminal articles by Brunnermeier and Sannikov (2014) and He and Krishnamurthy (2013) building on Basak and Cuoco (1998) developed a convenient framework to think about general equilibrium consequences of financial frictions. A second wave of articles looks at more complex dynamics involving more than one state variable to generate movements in the aggregate stochastic discount factor such as Drechsler, Savov, and Schnabl (2017), Di Tella (2017) and Di Tella and Kurlat (2016).

Although this strand of research is showing great potential in incorporating important macro-financial insights into asset pricing models, solving these models is a nontrivial problem. Most heterogeneous agent asset pricing models share a similar mathematical structure. They consist of a Hamilton-Jacobi-Bellman (HJB) equation for each agent coupled with a system of algebraic equations derived from the market clearing conditions, occasionally binding constraints, and financial frictions. Because we are interested in the recursive equilibrium, HJB equations are time-independent, and hence, nonlinear degenerated elliptic PDEs. Solving such a system of PDEs is, a priori, a tedious problem because it might be numerically unstable. Approximation errors tend to amplify to create explosive dynamics. Recent work on algorithms for solving continuous-time models has included a shift toward machine learning techniques to solve high-dimensional and highly nonlinear models. Sauzet (2022) uses neural networks as function approximators to solve continuous-time models in high-dimensional settings. Duarte (2022) poses nonlinear partial differential equations (PDEs) associated with continuous-time model solution as supervised learning problems. Gopalakrishna (2022) solves continuous-time equilibrium models using neural networks with economic information encoded as regularizers. In this paper, we propose an algorithm able to solve a very general class of models in an efficient and standardized way without machine learning. We overcome the stated issues by combining insights from different parts of the numerical methods literature.

First, as is customary in the physics literature, we add a fictitious time dimension (transient) to solve the system over time until convergence to equilibrium to bypass numerical difficulties created by nonlinearity. More precisely, we follow Brunnermeier and Sannikov (2016) and solve the algebraic part of the system statically while solving for the value functions of the different agents dynamically backward in time. The static system is solved in between every time iteration using a simple Newton-Raphson method with the unconstrained solution as an initial guess. Solving for the value function backward in time requires careful attention as the HJB equation inherits some of the inherent instability of the well-known advection equation. Informally, one needs to be particularly cautious in approximating the derivatives to preserve monotonicity of the elliptic operator. With one state variable (or several state variables with uncorrelated laws of motion), we can simply apply a traditional upwind scheme by taking the finite difference approximation according to the sign of the drift of the law of motion of the corresponding state variable. When we have at least two correlated state variables, the problem is more complex as the monotone direction may be inside the state space but not necessarily a primary vector on the discrete grid. In this case, we use the method developed by Bonnans, Ottenwaelter, and Zidani (2004) consisting of using an available degree of freedom in the interpolation problem to rotate the state space with minimized computational time. Last, we treat the nonlinearities arising from the regulated part of the HJB. We follow the suggestion of Candler (1999) to treat the problem as if it were linear and relax the nonlinear part with each iteration. We then solve the system in the time dimension using a fully implicit backward Euler algorithm until convergence. The contribution of this paper to the literature is to demonstrate that, by combining these different insights, we can solve large class of continuous-time macro-finance models. The project is close to Hansen, Tourre, and Khorrami (2019) which also provides a finite difference method to solve for a nested macro-finance model. The algorithm presented in this paper diverges mainly by showing how to deal with correlated Brownian motions by using the algorithm of Bonnans, Ottenwaelter, and Zidani (2004).

2 The General Portfolio Problem

In this section, we recall the structure of Merton's (1973) portfolio problem in continuous time, as it is the basis of the class of models we would like to solve, and to define the Hamilton-Jacobi-Bellman (HJB) equation that is the focus of the finite difference scheme of Section 5. This problem can be written in the following generic form. Agents have a lifetime utility function defined as:

$$U_t = E_t \left[\int_t^{\infty} f(c_t, U_t) du \right],$$

where f_t is a homothetic utility function. We assume that it follows an Epstein-Zin recursive formulation:

$$f(c_t, U_t) = \left(\frac{1-\gamma}{1-1/\zeta} \right) U_t \left(\left[\frac{c_t}{([1-\gamma] U_t)^{1/(1-\gamma)}} \right]^{1-1/\zeta} - \rho \right),$$

where ρ, γ, and ζ are the parameters for time discounting, risk aversion, and intertemporal elasticity of substitution, respectively. Agents maximize U_t under the law of motion of their net worth n_t:

$$\frac{dn_t}{n_t} = \left(r_t + w_t(\mu_t^{r,k} - r_t) - \mathfrak{c}_t \right) dt + w_t \sigma_t^{r,k} dZ_t,$$

where r_t is the risk free-rate, $\mathfrak{c}_t = c_t/n_t$ the consumption to wealth ratio, and w_t the portfolio weight on a risky asset. This risky asset has dividend flows that follows:

$$dr_t^k = \mu_t^{r,k} dt + \sigma_t^{r,k} dZ_t,$$

where $Z_t = \{Z_t \in \mathbb{R}^d; \mathcal{F}_t, t \geq 0\}$ is a standard adapted Brownian motion process. Finally, the HJB of the problem is given by:

$$0 = \max_{w_t, c_t} f(c_t, U_t) + E_t (dU_t).$$

Thanks to the homotheticity of the utility function, we can guess and verify the value function as

$$U(\xi_t, n_t) = \frac{(\xi_t n_t)^{1-\gamma}}{1-\gamma}, \tag{1}$$

where ξ_t is a wealth multiplier variable that tracks changes in the set of investment opportunities driven by changes in the state variables. We postulate its law of motion as

$$\frac{d\xi_t}{\xi_t} = \mu_t^{\xi} dt + \sigma_t^{\xi} dZ_t.$$

Applying Ito's lemma to the HJB equation gives

$$\begin{aligned}E_t\left(dU(\xi_t,n_t)\right) &= \mu_t^\xi \xi U_\xi(\xi_t,n_t) + \mu_t^n n_t U_n(\xi_t,n_t)\\ &\quad + \left(\sigma_t^\xi \xi_t\right)^2 \frac{1}{2} U_{\xi\xi}(\xi_t,n_t) + \left(\sigma_t^n n_t\right)^2 \frac{1}{2} U_{nn}(\xi_t,n_t) + \sigma_t^\xi \xi_t \sigma_t^n n_t U_{\xi n}(\xi_t,n_t),\end{aligned}$$

where the subscript on a function represents the partial derivative with respect to that variable such that

$$F_x(x,y) = \frac{\partial F(x,y)}{\partial x}.$$

Note that, in a recursive equilibrium, state-variables characterize the whole system such that U_t only moves through time as a deterministic function of other variables, and hence, $\dot{U}_t = 0$. Using (20), we can rewrite the HJB equation as

$$\begin{aligned}0 = \max_{\mathfrak{c}_t, w_t} \Bigg\{ &\frac{1}{1-1/\zeta}\left(\left(\frac{\mathfrak{c}_t}{\xi_t}\right)^{1-1/\zeta} - \rho\right) + \mu_t^\xi - \frac{\gamma}{2}\left(\sigma_t^\xi\right)^2 + r_t + w_t\left(\mu_t^{r,k} - r_t\right) - \mathfrak{c}_t \\ &- \frac{\gamma}{2} w_t^2 \left(\sigma_t^{r,k}\right)^2 + (1-\gamma) w_t \sigma_t^{r,k} \sigma_t^\xi \Bigg\}.\end{aligned} \tag{2}$$

The optimality conditions for $\mathfrak{c}_t$ and w_t are given by

$$\mathfrak{c}_t = \xi_t^{1-\zeta}, \tag{3}$$

$$w_t = \frac{\mu_t^{r,k} - r_t + (1-\gamma)\sigma_t^{r,k}\sigma_t^\xi}{\gamma\left(\sigma_t^{r,k}\right)^2}. \tag{4}$$

We can then plug these conditions into the HJB to find a differential equation in ξ_t. From here, models diverge by assuming different types of agents with heterogeneous constraints, number of available assets, technology, financial frictions, and stochastic processes. These differences will eventually determine a set of state-variable(s) affecting the set of investment opportunities in which a recursive equilibrium is determined. Yet, the skeleton of the model remains similar in consisting in a series of algebraic equations, imposing market clearing conditions and constraints, and an HJB equation for each agent.

3 Relaxation of Nonlinearity

We follow the approach of Brunnermeier and Sannikov (2016) in solving the algebraic part of the system as a side problem within each iteration of the differential problem. In a recursive equilibrium, we can write $\xi_t = \xi(\mathbf{X}_t)$ as all variables can be expressed as a function of a set of the state variables vector $\mathbf{X}_t$ following the law of motion:

$$\frac{d\mathbf{X}_t}{\mathbf{X}_t} = \boldsymbol{\mu}_t^X dt + \boldsymbol{\sigma}_t^X dZ_t,$$

where $\boldsymbol{\mu}_t^X$ is the vector of individual drifts and $\boldsymbol{\sigma}_t^X$ a covariance matrix. We can then apply Ito's lemma to $\xi(\boldsymbol{X}_t)$ to find:

$$\begin{aligned} \mu_t^\xi \xi_t &= (\nabla_X \xi_t)^\intercal \boldsymbol{\mu}_t^X + \frac{1}{2} Tr \left[\boldsymbol{\sigma}_t^{X\intercal} (H_X \xi_t) \boldsymbol{\sigma}_t^X \right], \\ \sigma_t^\xi \xi_t &= (\nabla_X \xi_t)^\intercal \boldsymbol{\sigma}_t^X, \end{aligned} \tag{5}$$

where $\nabla_X \xi_t$ is the gradient of ξ_t with respect to $\mathbf{X}_t$ and $H_X \xi_t$ is the Hessian matrix of ξ_t with respect to $\mathbf{X}_t$. By substituting these expressions for μ_t^ξ and σ_t^ξ into (2), one can readily see that the HJB is a second-order nonlinear partial differential equation in $\mathbf{X}_t$.

Unfortunately, because of its nonlinearity, there is no theorem that can be applied to guarantee the stability and convergence of a numerical scheme for this problem. Nonetheless, by treating the equation *as if* it was linear, it is possible to create a scheme that is closer to stability and which works in practice. To make this point,[1] assume that the set of state variables is a scalar $\mathbf{X}_t = \{x_t\}$ and use equations (2), (3), and (4) to isolate μ_t^ξ:

$$\mu_t^\xi = -\frac{1}{1 - 1/\zeta} (\mathfrak{c}_t - \rho) + \frac{\gamma}{2} (\sigma_t^\xi)^2 - r_t + \mathfrak{c}_t - \frac{\gamma}{2} w_t^2 (\sigma_t^{r,k})^2 - (1-\gamma) w_t (\sigma_t^{r,k})^2 \sigma_t^\xi. \tag{6}$$

In our general portfolio problem, ξ_t is raised to the power $1-\zeta$ in the first order condition (3) which makes the HJB equation (2) nonlinear once the optimal controls have been taken into account. Our strategy consists in solving the Ito process in equation (5), rather than directly the HJB, as a linear function by treating μ_t^ξ as a parameter whose value is computed from the previous iteration. We can use this equation to compute a consistent value for μ_t^ξ

[1] The exact same procedure can be used for any number of state variables but making this assumption facilitates exposition at this stage.

to plug in (5).[2] This procedure is commonly referred to as a *relaxation method* to reflect the fact that the nonlinearity is introduced to the problem only in small increments. The next sections will provide a concrete application of this principle.

4 A Finite-Difference Approach

In this section, we provide a short introduction to finite difference schemes to solve systems of PDEs. In particular, we illustrate through the example of the advection equation that the direction of the finite difference approximation is key to the convergence of the scheme. The section is based on Candler (1999) and Tourin (2011).

4.1 Introduction to the Finite-Difference Scheme

Designing a finite difference scheme starts from defining a series of points (a grid) in the dimension(s) of the state variable(s). For simplicity, we assume a time t and state variable x on a grid equispaced in both time and state with a distance of respectively Δt and Δx between two points. Grid nodes are then referred by numbering them along the two dimensions: $\{t_1, t_2, \ldots, t_T\}$ and $\{x_1, x_2, \ldots, x_W\}$. A function $V(t, x)$ evaluated at a point (n, i) on the grid is then:

$$V_i^n = V(t_n, x_i) = V(n\Delta t, i\Delta x)$$

We recall the definition of a partial derivative with respect to the state variable x as:

$$\frac{\partial V(t,x)}{\partial x} = \lim_{\Delta x \to 0} \frac{V(t, x + \Delta x) - V(t,x)}{\Delta x}.$$

A finite difference approximation consists in the evaluation of the previous expression for a finite distance Δx. As our grid features various points, one could potentially use different nodes to compute the approximation. In theory, a finite difference approximation can be done through any linear combinations of the nodes in the grid. The most commonly used

[2]In a later example, we will derive $\boldsymbol{\mu}_t^X$ and $\boldsymbol{\sigma}_t^X$ using the definition of the state(s) variable(s).

local approximations involving only two neighboring points are:

$$\begin{aligned}
\frac{\partial V(t_n, x_i)}{\partial x} &= \frac{V_{i+1}^n - V_{i-1}^n}{2\Delta x} + O\left(\Delta x^2\right) && \text{Central Approximation} \\
\frac{\partial V(t_n, x_i)}{\partial x} &= \frac{V_{i+1}^n - V_i^n}{\Delta x} + O\left(\Delta x\right) && \text{Forward Approximation} \\
\frac{\partial V(t_n, x_i)}{\partial x} &= \frac{V_i^n - V_{i-1}^n}{\Delta x} + O\left(\Delta x\right) && \text{Backward Approximation.}
\end{aligned}$$

The order of the approximation error can be computed by taking a Taylor expansion. Approximations that are the most centered and feature the most points will have a higher order of error. This is reflected in the central approximation having an error of order 2 while the forward and backward ones have only errors of order 1. At this stage, one could be tempted to conclude that the central approximation dominates the other two as it is more accurate when using only two nodes. Yet, as will be clear in the next section, it is not the case as we also care about convergence properties of the numerical scheme.

4.2 Instability in the Advection Equation

In this subsection, we introduce the advection (or wave) equation which features the same stiffness characteristics as the HJB equation we are concerned with. This example is often used in introductory fluid dynamics classes. Let's consider the advection equation:

$$V_t + aV_x = 0. \tag{7}$$

This equation has a well-known exact solution as $V(t, x) = V(0, x - at)$, given an initial condition $V(0, x)$. We solve this problem by applying the three FD approximations from the last section with respect to the state variable and forward in time.

$$\begin{aligned}
\frac{V_i^{n+1} - V_i^n}{\Delta t} &= -a\frac{V_{i+1}^n - V_{i-1}^n}{2\Delta x} && \text{Central} \\
\frac{V_i^{n+1} - V_i^n}{\Delta t} &= -a\frac{V_{i+1}^n - V_i^n}{\Delta x} && \text{Forward} \\
\frac{V_i^{n+1} - V_i^n}{\Delta t} &= -a\frac{V_i^n - V_{i-1}^n}{\Delta x} && \text{Backward}
\end{aligned}$$

We can express these three equations as an explicit function of V_i^{n+1} as:

$$V_i^{n+1} = V_i^n - a\Delta t \frac{V_{i+1}^n - V_{i-1}^n}{2\Delta x} \qquad \text{Central} \tag{8}$$
$$V_i^{n+1} = V_i^n - a\Delta t \frac{V_{i+1}^n - V_i^n}{\Delta x} \qquad \text{Forward} \tag{9}$$
$$V_i^{n+1} = V_i^n - a\Delta t \frac{V_i^n - V_{i-1}^n}{\Delta x} \qquad \text{Backward.} \tag{10}$$

We can then compute the value for V across the grid iteratively through time starting from the given initial condition $V(t = 0) = V_0$. Figure (1) provides the results (using algorithm 1 below) of this procedure for the three given approximations and parameters: $\Delta t = 0.2; a = 0.5; dx = 0.17$ on a grid from 0 to 10 and starting from an initial state where $V_0 = 2$ for $x \in [0, 5]$ and $V_0 = 1$ for $x \in [5, 10]$.

Algorithm 1: Explicit Euler

1. Define a finite grid over the state variable x, set $V(t = 0) = V_0$ for any nodes on the grid.
2. Iterate through time by increment Δt.
3. Iterate through each point in the state space from i=1 to i=I and use one either (8), (9), and (10) to solve for V_i^n given V_i^{n-1}, V_{i+1}^{n-1} and V_{i-1}^{n-1}.
4. Go back to Step 2 until t=T.

The last panel of Figure 1 displays the analytical solution of the advection equation at different time steps. As the coefficient a is negative, the initial condition is expanded from the left to the right in the analytical solution. This process occurs through time, until reaching a steady state position where $V = 2$ for the whole state space. The first and second panels show that both the central and the forward difference approximation do not provide satisfactory results as the scheme exhibits large oscillations reflecting growing approximation errors. These errors are increasing in the number of time iterations, which will therefore never converge to its steady-state value. One can see that the backward difference approximation is more satisfying in matching the analytical solution and in converging to the analytical steady-state solution.

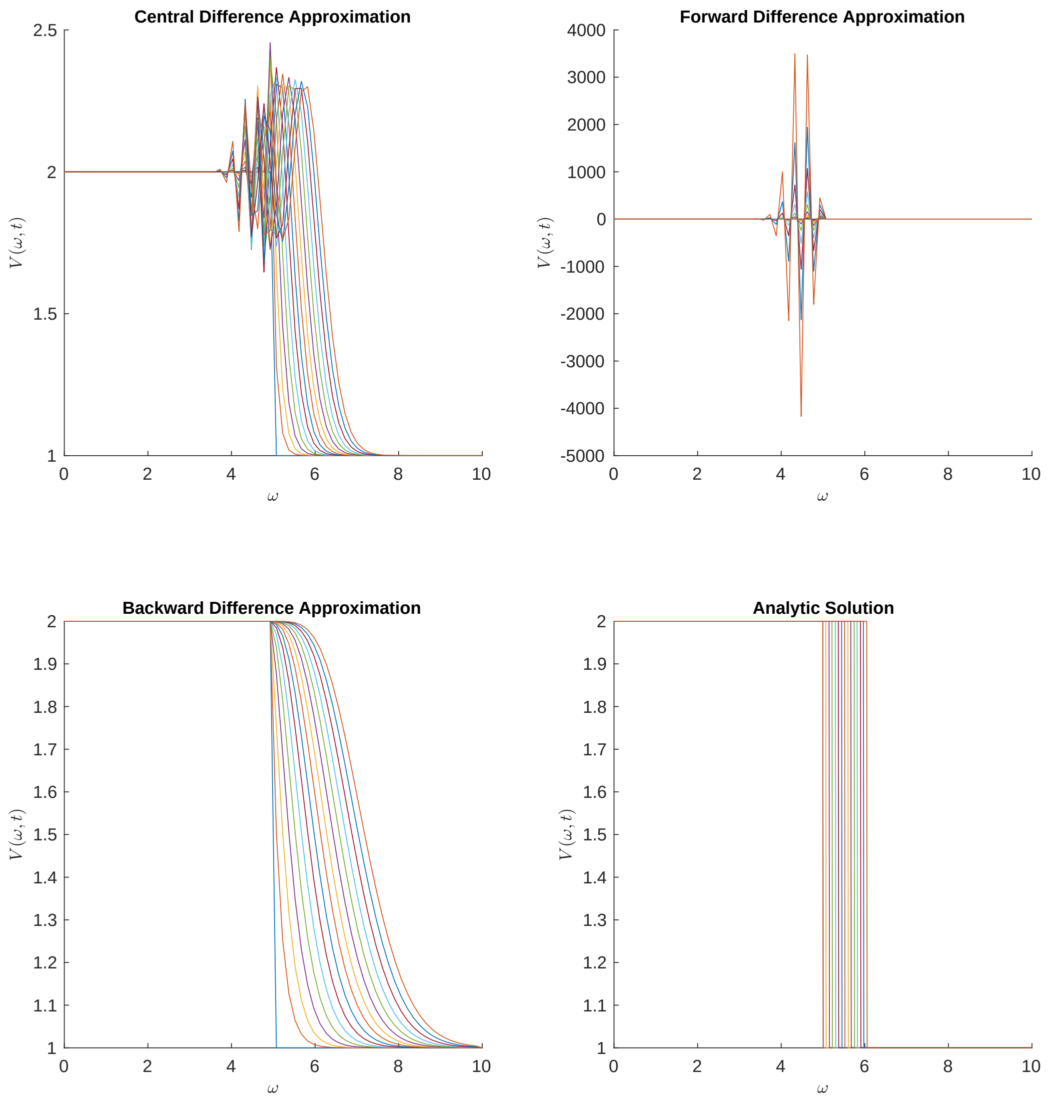

Figure 1: Solving the advection equation (7) with different approximations. The first three panels display the result of solving for the movement of the wave equation across time using three different approximations to the space derivative. The last panel shows the exact analytical solution of the problem moving from the left to the right. Each line of a different color is the solution (exact or approximated) at a given point in time.

This result is well-known in the numerical literature (see Candler, 1999). When a is positive, the backward approximation has the property of being taken to the left of the wave being propagated to the right. Such a scheme is called *upwinding* or *upstreaming* to reflect that, by taking the derivative left from the wave, we only take into consideration information coming from upstream of the flow. In the case of advection equation with $a > 0$ this is natural as the wave is transported from left to right and the information on its right is, hence, irrelevant for the wave's evolution. Crucially, the wave equation is what is called in physics *a pure conservation*, meaning that the energy (the solution) is purely transported and does not diffuse into the domain. Providing a numerical approximation of a pure conservation equation is difficult because any approximation has a diffusive nature. One has, therefore, to be careful about how this artificial diffusion (also called artificial viscosity) is introduced to ensure that it is not amplified through time. The key concept in this regard is that the approximation *preserves the monotonicity* of the solution through each time iteration and does not add a new local maximum. One can see that this condition is indeed broken in the central and forward approximations as taking information from downstream breaks the conservation of the solution. This spurious diffusion going in the wrong direction is particularly problematic as it amplifies at each iteration and prevents the solution from converging to its steady-state. On the other hand, the backward approximation, even if it introduces more diffusion in comparison to the central approximation, is doing so in the right direction and, therefore, preserves the monotonicity of the scheme provided that the Courant-Friedrichs-Lewy condition is satisfied $\left|\frac{a\Delta t}{\Delta\omega}\right| \leq 1$.[3]

4.3 An Implicit Scheme

In this subsection we introduce implicit (backward) schemes that are more stable for the PDE in question than explicit (forward) schemes. Note that in the last subsection, we approximated our advection equation (7) forward in time but we could also have done

[3]We refer to Candler (1999) for a the rigorous Von Neuman analysis of the dynamics.

backward as:

$$\begin{aligned}
\frac{V_i^n - V_i^{n-1}}{\Delta t} &= -a\frac{V_{i+1}^n - V_{i-1}^n}{2\Delta x}, && \text{Central,}\\
\frac{V_i^n - V_i^{n-1}}{\Delta t} &= -a\frac{V_{i+1}^n - V_i^n}{\Delta x}, && \text{Forward}\\
\frac{V_i^n - V_i^{n-1}}{\Delta t} &= -a\frac{V_i^n - V_{i-1}^n}{\Delta x}, && \text{Backward.}
\end{aligned}$$

In this case, we cannot use algorithm 1 as described in the previous section because V_i^n is now an *implicit* function which requires determining jointly the value of its neighboring points. One therefore needs to solve the following system:

$$\mathbf{V}^n = \mathbf{A}^{-1}\mathbf{V}^{n-1} \tag{11}$$

where $\mathbf{V}^n$ is a vector of V_i^n and A is a IxI matrix given by:

$$\mathbf{A^{CE}} = \begin{bmatrix} 1 & \frac{a\Delta t}{2\Delta x} & \cdot & \cdot \\ \frac{-a\Delta t}{2\Delta x} & 1 & \frac{a\Delta t}{2\Delta x} & \cdot \\ \cdot & \ddots & \ddots & \ddots \\ \cdot & \cdot & \frac{-a\Delta t}{2\Delta x} & 1 \end{bmatrix}$$

$$\mathbf{A^{FW}} = \begin{bmatrix} \left[1 - \frac{a\Delta t}{\Delta x}\right] & \frac{a\Delta t}{\Delta x} & \cdot & \cdot \\ \cdot & \left[1 - \frac{a\Delta t}{\Delta x}\right] & \frac{a\Delta t}{\Delta x} & \cdot \\ \cdot & \ddots & \ddots & \ddots \\ \cdot & \cdot & \cdot & \left[1 - \frac{a\Delta t}{\Delta x}\right] \end{bmatrix}$$

$$\mathbf{A^{BW}} = \begin{bmatrix} \left[1 + \frac{a\Delta t}{\Delta x}\right] & \cdot & \cdot & \cdot \\ \frac{-a\Delta t}{\Delta x} & \left[1 + \frac{a\Delta t}{\Delta x}\right] & \cdot & \cdot \\ \cdot & \ddots & \ddots & \ddots \\ \cdot & \cdot & \frac{-a\Delta t}{\Delta x} & \left[1 + \frac{a\Delta t}{\Delta x}\right] \end{bmatrix}$$

where $\mathbf{A^{CE}}$, $\mathbf{A^{FW}}$, and $\mathbf{A^{BW}}$ are the matrices corresponding to central, forward, and backward approximations, respectively.

Algorithm 1 is therefore amended in replacing Step 3 with solving (11). This requires the inversion of the bi-diagonal or tri-diagonal matrix $\mathbf{A}$. This step is more computationally involved than solving explicitly for every node as in algorithm 1 but can still be done very efficiently by exploiting the sparsity of the matrix using, for instance, a standard Thomas algorithm.[4] In the case of the linear advection problem, these implicit schemes can be shown to be unconditionally stable and hence, do not need to respect the Courant–Friedrichs–Lewy condition. Implicit schemes are in general more diffusive in nature as values of the solution at each node impact each other. For this reason, it is not frequently used in the numerical fluid dynamics literature to approximate the advection (a pure conservation) equation as they introduce unnecessary approximation error. Our case is different as we are interested in finding a recursive equilibrium that is time independent. We are therefore interested in adding as much diffusion as possible in order to take larger time steps to minimize computational time.

5 A Monotonic Scheme for the Portfolio Problem

In this section, we provide an implicit upwinding finite-difference scheme that can be applied to the HJB of the general portfolio problem. We tackle in turn the one-dimensional and two-dimensional cases.

5.1 Finite-Difference Scheme in One Dimension

In the unidimensional case, we are interested in solving an elliptic ordinary differential equation as (5) that does not depend on time. Though, because of the inherent instability of the non-linear HJB, it is easier to add a false *transcient* (time) dimension and solve it until convergence to a steady state. In doing so, we build our numerical scheme to be as diffusive as possible to be able to take large time steps and minimize the computational time needed for convergence. Note that, in this regard, we are interested in the accuracy of the approximation at a particular step in time only with respect to its impact on the

[4]The Thomas algorithm (named after Llewellyn Thomas), is an efficient Gaussian elimination technique that can be used to invert tridiagonal matrices. See, for example, Niyogi (2006)

convergence property of the scheme. Moreover, we solve the system backward in time rather than forward, as this will allow us not to define exogenous boundary conditions when the system admits a globally absorbing steady-state strictly inside the state space. We will come back to this point in actual examples. At the moment, let us consider the following linear[5] parabolic (time-dependent) partial differential equation:

$$r(x)F(x,t) = u(x) + \mu(x)\frac{\partial F(x,t)}{\partial x} + \frac{\sigma(x)^2}{2}\frac{\partial^2 F(x,t)}{\partial x \partial x} + \frac{\partial F(x,t)}{\partial t}. \tag{12}$$

We recall our definition of the grid from the previous section along the time t and state variable x on a grid equispaced in both time and state with respectively Δt and Δx distance between two points. Grid nodes are then referred to by numbering them ordinally along the two dimensions: $t \in \{t_1, t_2, \ldots, t_n, \ldots, t_N\}$ and $\{x_1, x_2, \ldots, x_i, \ldots, x_I\}$. A function $F(t,x)$ evaluated at a point (n,i) on the grid is then noted as:

$$F_i^n = F(t_n, x_i) = F(n\Delta t, i\Delta x).$$

A finite difference approximation consists in the evaluation of the previous expression for a finite distance of Δx. As our grid features various points, one could potentially use different nodes to compute the approximation. In theory, a finite-difference approximation can be done through any linear combinations of the nodes in the grid. The most commonly used local approximations involving only two neighboring points are the forward, backward, and central approximations of the previous section.

As we have illustrated previously, a wave equation with a positive directional parameter (moving to the right) requires a backward approximation while a negative directional parameter (moving to the left) requires a forward approximation. By allowing the directional parameter to vary according to its position between negative and positive values, we change the direction approximation dynamically. This is what the following *upwinding* approximation does:

$$\frac{\partial F(t,x)}{\partial x} \approx \mu_i^+ \frac{F_{i+1}^n - F_i^n}{\Delta x} + \mu_i^- \frac{F_i^n - F_{i-1}^n}{\Delta x},$$

[5]This equation would correspond to equation 5. As explained at the end of Section 2, we solve our non-linear equation as if it was linear and introduce the non-linearity slowly through time iterations.

where

$$\mu_i^+ = \begin{cases} \mu_i & \text{if } \mu_i > 0 \\ 0 & \text{else,} \end{cases} \qquad \mu_i^- = \begin{cases} \mu_i & \text{if } \mu_i < 0 \\ 0 & \text{else.} \end{cases}$$

This approximation preserves *monotonicity* of the solution through each time iteration; that is, it does not add a new local maximum.

We use an *implicit* upwind finite-difference scheme. An *implicit* method, while more complex to program and requiring more computational effort in each solution step, is more stable and allows for large time-step sizes. Explicit methods calculate the state of a system at a later time from the state of the system at the current time, while implicit methods find a solution by solving an equation involving both the current state of the system and the later one. Mathematically, if $\mathbf{F}^n$ is the current value function vector on the discrete equispaced grid I and $\mathbf{F}^{n+1}$ is the state at the later time, then, for an explicit method, we solve

$$\mathbf{F}^{n+1} = T(\mathbf{F}^n)$$

while, for an implicit method, we solve

$$T(\mathbf{F}^{n+1}, \mathbf{F}^n) = 0$$

to find $\mathbf{F}^{n+1}$. The upwind finite-difference scheme approximation of equation (12) for time $t \in T$ on the discrete equispaced grid $i \in I$ is given by

$$r_i F_i^t = u_i + \mu_i^+ \frac{F_{i+1}^t - F_i^t}{\Delta x} + \mu_i^- \frac{F_i^t - F_{i-1}^t}{\Delta x} + \frac{\sigma_i^2}{2} \left(\frac{F_{i+1}^t - 2F_i^t + F_{i-1}^t}{\Delta x^2} \right) + \frac{F_i^{t+1} - F_i^t}{\Delta t}.$$

We are looking for an implicit system of equations given our parameters and guess from the previous time iteration but solving backward in time, setting $\mathbf{F}^t \equiv \mathbf{F}^{n+1}$ and $\mathbf{F}^{t+1} \equiv \mathbf{F}^n$.

We can therefore write our numerical scheme in the fixed-point form as:

$$\underbrace{\left[r_i + \frac{1}{\Delta t} + \frac{\mu_i^+ - \mu_i^-}{\Delta x} + \frac{\sigma_i^2}{\Delta x^2}\right]}_{M_i} F_i^{n+1} = \underbrace{\left[\frac{\mu_i^+}{\Delta x} + \frac{\sigma_i^2}{2\Delta x^2}\right]}_{-U_i} F_{i+1}^{n+1} - \underbrace{\left[\frac{\mu_i^-}{\Delta x} - \frac{\sigma_i^2}{2\Delta x^2}\right]}_{D_i} F_{i-1}^{n+1} + u_i + \frac{F_i^n}{\Delta t}.$$

Because $U_i > 0$, $M_i > 0$, and $D_i < 0$ for all i, the scheme is unconditionally monotone in F_{i-1}^{n+1}, F_i^n, and F_{i+1}^{n+1}. Note that the centered second derivative term in front of the volatility, since always positive, is not an issue for the monotonicity of the scheme. Theoretically, for a linear problem, we could, therefore, take an arbitrarily large time step in solving the equation. In practice, the non-linearity of the scheme restricts the size of the time step we can take. There is no theorem available to determine this limit, and it can only be found through by running simulations.

Going backward in time, we solve for $\mathbf{F}^{n+1}$ as a function of $\mathbf{F}^n$. In order to do so, we can now write our parabolic partial differential equation in matrix form as:

$$\mathbf{F}^{n+1} = \mathbf{A}^{-1}\left[\mathbf{u} + \frac{\mathbf{F}^n}{\Delta t}\right]$$

where the matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \begin{bmatrix} M_1 & U_1 & \cdot & \cdot & \cdot \\ D_2 & M_2 & U_2 & \cdot & \cdot \\ \cdot & \ddots & \ddots & \ddots & \cdot \\ \cdot & \cdot & D_{I-1} & M_{I-1} & U_{I-1} \\ \cdot & \cdot & \cdot & D_I & M_I \end{bmatrix}$$

and

$$\mathbf{F}^n = \begin{bmatrix} F_1^n \\ \vdots \\ F_I^n \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_D \end{bmatrix}.$$

Note that by writing the equation in this form we are *not* assuming any boundary condition on the edge of the grid for the value function $F(t,x)$ in terms of the state variable x. We assume that we do not need to do so because the value function will drift right at the left boundary and left at the right boundary. This is equivalent to assuming that there exists an interior absorbing stochastic steady-state. In most macro-finance applications, the two edges of the state grid are degenerating points where the volatility σ_i goes to 0. Therefore, we can solve numerically for equation (12) with the following algorithm:

Algorithm 1 *Implicit Euler*

```
1. Define a finite grid over the state variable $x$ and set an initial guess for $\mathbf{F}^0$.
2. Invert the sparse matrix $\mathbf{A}$ using Thomas algorithm to solve for $\mathbf{F}^{n+1}$ in (??).
3. Iterate on 2 until convergence.
```

5.2 Finite-Difference Scheme in Two Dimensions

Several models in macro-finance feature two state variables (i.e., Silva, 2016; Di Tella and Kurlat, 2016; Drechsler, Savov, and Schnabl, 2017). In this case, the state-space becomes a plane and the grid is defined on two coordinates. We write the generalization of (12) in multiple dimensions as:

$$r(\mathbf{X})F(\mathbf{X},t) = u(x) + \sum_{i=1}^{m} \mu_i(\mathbf{X})\frac{\partial F(\mathbf{X},t)}{\partial x_i} + \sum_{i=1}^{m}\sum_{j=1}^{m} \frac{\sigma_i(\mathbf{X})\sigma_j(\mathbf{X})}{2}\frac{\partial^2 F(\mathbf{X},t)}{\partial x_i \partial x_j} + \frac{\partial F(\mathbf{X},t)}{\partial t}. \tag{13}$$

In this section, we are interested in the two-dimensional case and we, therefore, set $m = 2$. We define $F^n_{i,j}$ as the value of $F(\mathbf{X},t_n)$ on the i-th point of the two-dimensional grid in the first dimension of size d_1 and j-th point in the second dimension of size d_2. Finding a monotone scheme in the multidimensional case is a significantly more involved problem than the single state variable one, leading to important instability and convergence issues if not tackled properly. The first reason is that we now need to approximate the cross-derivative of $F(\mathbf{X},t)$ while ensuring monotonicity. For instance, the following approximation

$$\frac{\partial^2 F(\mathbf{X},t_n)}{\partial x_i \partial x_j} \approx \frac{F^n_{i+1,j+1} + F^n_{i-1,j-1} - F^n_{i+1,j-1} - F^n_{i-1,j+1}}{4\Delta x} \tag{14}$$

is not monotone because both $F^n_{i+1,j+1}$ and $F^n_{i-1,j-1}$ have the wrong sign. The second reason is that even if we have identified the upwinding direction, there is no guarantee that there is an actual node in this particular direction and one must take an interpolation in order to estimate this particular point. In this case, this interpolation should be made in a way that preserves monotonicity.

To do so, we follow Bonnans, Ottenwaelter, and Zidani (2004) with a fast algorithm based on a walk on the Stern-Brocot tree. We accordingly write the upwind scheme that preserves the monotonicity with the following finite-difference approximation for time $t \in T$ and vector of state variables $\mathbf{x_k}$. We define $\mathbf{k}$ as the coordinate vector of the position of $\mathbf{x_k}$ on the discrete multidimensional grid $\mathbf{k} \in \mathbb{N}_0^m$. That is, if $\mathbf{k} = [2,5]^\intercal$, it means that $x_{1,\mathbf{k}}$ is the 2nd point in the first dimension and $x_{2,\mathbf{k}}$ is the 5th point in the second dimension. We rewrite the partial differential equation (13) as:

$$r_{\mathbf{k}} F^t_{\mathbf{k}} = u_{\mathbf{k}} + \sum_{i=1}^{m} \mu^+_{i,\mathbf{k}} \frac{F^t_{\mathbf{k}+\mathbf{e}_i} - F^t_{\mathbf{k}}}{\Delta x_{i,\mathbf{k}+\mathbf{e}_i}} + \sum_{i=1}^{m} \mu^-_{i,\mathbf{k}} \frac{F^t_{\mathbf{k}} - F^t_{\mathbf{k}-\mathbf{e}_i}}{\Delta x_{i,\mathbf{k}-\mathbf{e}_i}} \tag{15}$$

$$+ \sum_{\boldsymbol{\xi}_{\mathbf{k}} \in \Xi_{\mathbf{k}}} \eta_{\boldsymbol{\xi}_{\mathbf{k}},\mathbf{k}} \left(F^t_{\mathbf{k}+\boldsymbol{\xi}_{\mathbf{k}}} + F^t_{\mathbf{k}-\boldsymbol{\xi}_{\mathbf{k}}} - 2F^t_{\mathbf{k}} \right) + \frac{F^{t+1}_{\mathbf{k}} - F^t_{\mathbf{k}}}{\Delta t}, \tag{16}$$

where $\mathbf{e}_i$ is the directional vector such that the i-th component is equal to 1 and 0 otherwise. The vectors $\boldsymbol{\xi}_{\mathbf{k}} \in \Xi_{\mathbf{k}}$ for the grid point $\mathbf{k}$ are found using the following *stencil decomposition* consisting in a collection of *nonnegative* coefficients $\eta_{\boldsymbol{\xi},\mathbf{k}}$ such that:

$$\sum_{\boldsymbol{\xi}_{\mathbf{k}} \in \Xi_{\mathbf{k}}} \eta_{\boldsymbol{\xi},\mathbf{k}} \xi_{i,\mathbf{k}} \xi_{j,\mathbf{k}} = \frac{\sigma_{i,\mathbf{k}} \sigma_{j,\mathbf{k}}}{2 h_i h_j}$$

where h_i is the distance between grid points in the i-th dimension and the elements of the vectors $\boldsymbol{\xi}_{\mathbf{k}}$ are integers. Using a stencil decomposition, that imposes that the coefficient $\eta_{\boldsymbol{\xi},\mathbf{k}}$ are nonnegative, guarantees that the implicit scheme is monotonic and converges to the unique solution. The stencil decomposition is reminiscent of the one using eigenvalues, with the important difference that the set of vectors is now constrained to belong to the stencil. We characterize the size of the stencil with P as the highest norm of the elements of the vectors $\boldsymbol{\xi}_{\mathbf{k}}$. We note that the following algorithm is limited to stencil decompositions in two dimensions, which makes this method infeasible for a higher dimensional problem.

Consider the covariance matrix

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix}$$

where $\sigma_{12} = \sigma_{21}$. When a covariance matrix is diagonal dominant, we have the well-known decomposition

$$\begin{aligned} \mathbf{\Sigma} = {} & (\sigma_{11} - |\sigma_{12}|) \begin{pmatrix} 1 \\ 0 \end{pmatrix} (1 \;\; 0) + (\sigma_{22} - |\sigma_{12}|) \begin{pmatrix} 0 \\ 1 \end{pmatrix} (0 \;\; 1) \\ & + \max(\sigma_{12}, 0) \begin{pmatrix} 1 \\ 1 \end{pmatrix} (1 \;\; 1) - \min(\sigma_{12}, 0) \begin{pmatrix} -1 \\ 1 \end{pmatrix} (-1 \;\; 1). \end{aligned} \tag{17}$$

If the matrix $\mathbf{\Sigma}$ is not diagonally dominant, the decomposition requires an algorithm to find a stencil decomposition. It suffices to discuss the case when the matrix is such that $\sigma_{22} < \sigma_{12} < \sigma_{11}$ as it is easy to reduce to this case by permutation of variables and change of sign of one of the element of the stencils. The stencil decomposition algorithm[6] is as follows:

Algorithm 2 *Stencil Decomposition (Bonnans, Ottenwaelter, and Zidani, 2004)*

1. *Initiate with $q_0 = 0$, $p_0 = 1$, $q'_0 = 1$, and $p'_0 = 1$.*
2. *If $\mathbf{\Sigma}$ is diagonal dominant, use equation (17) and stop.*
3. *Begin iteration n by computing the following*

$$\boldsymbol{\xi} = \begin{pmatrix} p_n \\ q_n \end{pmatrix} \qquad \boldsymbol{\xi}' = \begin{pmatrix} p'_n \\ q'_n \end{pmatrix} \qquad \mathbf{X} = \boldsymbol{\xi}\boldsymbol{\xi}^\intercal \qquad \mathbf{X}' = \boldsymbol{\xi}'\boldsymbol{\xi}'^\intercal$$

$$\mathbf{V} = \begin{pmatrix} x_{11} \\ \sqrt{2}x_{12} \\ x_{22} \end{pmatrix} \qquad \mathbf{V}' = \begin{pmatrix} x'_{11} \\ \sqrt{2}x'_{12} \\ x'_{22} \end{pmatrix} \qquad \overline{\mathbf{V}} = (\mathbf{V} \;\; \mathbf{V}') \qquad \mathbf{S} = \begin{pmatrix} \sigma_{11} \\ \sqrt{2}\sigma_{12} \\ \sigma_{22} \end{pmatrix}$$

[6] We refer the reader to Bonnans, Ottenwaelter, and Zidani (2004) for an in-depth exposition.

4. *Take the cross product of* $\mathbf{V}$ *and* $\mathbf{V}'$

$$\mathbf{N} = \mathbf{V} \times \mathbf{V}'$$

and project $\mathbf{S}$ *on the plane with normal vector* $\mathbf{N}$

$$\mathbf{K} = \mathbf{S} - \tau \mathbf{N}$$

where

$$\tau = ||\mathbf{N}||^{-2} \mathbf{N}^{\mathsf{T}} \mathbf{S}$$

and $|| \cdot ||$ *is the Euclidean norm.*

5. *If* $p + p' \geq P$ *or* $||\mathbf{S} - \mathbf{K}|| \leq \varepsilon$, *then stop and the decomposition is such that*

$$\eta_1 \xi_i \xi_j + \eta_2 \xi'_i \xi'_j \approx \sigma_{ij},$$

where

$$\boldsymbol{\eta} = \overline{\mathbf{V}} \backslash \mathbf{K}.$$

The function $\backslash$ *is the solution in the least squares sense to the underdetermined system of equations* $\overline{\mathbf{V}} \boldsymbol{\eta} = \mathbf{P}$.

6. *If* $p + p' < P$ *and* $||\mathbf{S} - \mathbf{P}|| > \varepsilon$, *then* $q''_n = q_n + q'_n$, $p''_n = p_n + p'_n$ *and compute*

$$\boldsymbol{\xi}'' = \begin{pmatrix} p''_n \\ q''_n \end{pmatrix}, \qquad \mathbf{X}'' = \boldsymbol{\xi}'' \boldsymbol{\xi}''^{\mathsf{T}}, \qquad \mathbf{V}'' = \begin{pmatrix} x''_{11} \\ \sqrt{2} x''_{12} \\ x''_{22} \end{pmatrix}, \qquad \overline{\overline{\mathbf{V}}} = \left(\mathbf{V} \ \mathbf{V}' \ \mathbf{V}'' \right),$$

$$\mathbf{N} = \mathbf{V} \times \mathbf{V}'', \qquad \tau = ||\mathbf{N}||^{-2} \mathbf{N}^{\mathsf{T}} \mathbf{S}, \qquad \mathbf{K} = \mathbf{S} - \tau \mathbf{N}, \qquad \boldsymbol{\eta} = \overline{\overline{\mathbf{V}}}^{-1} \mathbf{K}.$$

- *If each element of the vector* $\boldsymbol{\eta}$ *is positive, then stop and the decomposition is* *such that*

$$\eta_1 \xi_i \xi_j + \eta_2 \xi_i' \xi_j' + \eta_3 \xi_i'' \xi_j'' \approx \sigma_{ij}.$$

- *If each element of the vector* $\boldsymbol{\eta}$ *is not positive and*

$$s\mathbf{N}^{\intercal}\mathbf{P} \leq 0,$$

where

$$\mathbf{H} = \begin{pmatrix} 0.5 \\ 0 \\ 0.5 \end{pmatrix} \qquad s = sign\left(\mathbf{N}^{\intercal}\mathbf{H}\right),$$

then $q_{n+1} = q_n$*,* $p_{n+1} = p_n$*,* $q'_{n+1} = q''_n$*,* $p'_{n+1} = p''_n$*, and go to (3) for next iteration* $n = n + 1$*.*

- *If each element of the vector* $\boldsymbol{\eta}$ *is not positive and*

$$s\mathbf{N}^{\intercal}\mathbf{P} > 0,$$

then $q_{n+1} = q''_n$*,* $p_{n+1} = p''_n$*,* $q'_{n+1} = q'_n$*,* $p'_{n+1} = p'_n$*, and go to (3) for next iteration* $n = n + 1$*.*

The intuition of Algorithm 2 is as follows. A two dimensional variance covariance matrix can be represented in two dimensions (since $\sigma_{12} = \sigma_{21}$, $\boldsymbol{\Sigma}$ has three coordinates). If the 3D representation of a variance-covariance matrix $\boldsymbol{\Sigma}$ is close enough to the projection of $\boldsymbol{\Sigma}$ on the plane generated by the vectors $\mathbf{X}$ and $\mathbf{X}'$, then we can generate $\boldsymbol{\Sigma}$ by a linear combination of $\mathbf{X}$ and $\mathbf{X}'$. If the 3D representation of a variance-covariance matrix $\boldsymbol{\Sigma}$ is inside the convex cone generated by the vectors $\mathbf{X}$, $\mathbf{X}'$, and $\mathbf{X}''$, then we can generate A by a conical combination of $\mathbf{X}$, $\mathbf{X}'$, and $\mathbf{X}''$. If none of the above is true, we need to update $\mathbf{X}$ and $\mathbf{X}'$ such that one of the two above is eventually true. If $\boldsymbol{\Sigma}$ is outside the half plane generated by $\mathbf{X}$ and $\mathbf{X}''$, update such that $q' = q''$ and $p' = p''$. Otherwise, $\boldsymbol{\Sigma}$ has to be outside of the half plane generated by $\mathbf{X}$ and $\mathbf{X}''$ and update such that $q = q''$ and $p = p''$.

As in the single dimensional case, we are looking for a solution that solves backward in time, that is an implicit system of equations in $F^t_{\mathbf{k}}$ given our parameters and guess from the previous time iteration. We can rearrange equation (15) to get:

$$\sum_{i=1}^{m} D_{i,\mathbf{k}} F^{n+1}_{\mathbf{k}-\mathbf{e}_i} + M_{\mathbf{k}} F^{n+1}_{\mathbf{k}} + S_{\mathbf{k}} F^{n+1}_{\mathbf{k}} + \sum_{i=1}^{m} U_{i,\mathbf{k}} F^{n+1}_{\mathbf{k}+\mathbf{e}_i} - \sum_{\boldsymbol{\xi}_{\mathbf{k}} \in \Xi_{\mathbf{k}}} \eta_{\boldsymbol{\xi}_{\mathbf{k}},\mathbf{k}} \left(F^{n+1}_{\mathbf{k}+\boldsymbol{\xi}_{\mathbf{k}}} + F^{n+1}_{\mathbf{k}-\boldsymbol{\xi}_{\mathbf{k}}} \right) \quad (18)$$
$$= u_{\mathbf{k}} + \frac{F^n_{\mathbf{k}}}{\Delta t},$$

where

$$\begin{aligned} D_{i,\mathbf{k}} &= \frac{\mu^-_{i,\mathbf{k}}}{\Delta x_{\mathbf{k}-\mathbf{e}_i}}, \\ M_{\mathbf{k}} &= r_{\mathbf{k}} + \frac{1}{\Delta t} + \sum_{i=1}^{m} \frac{\mu^+_{i,\mathbf{k}}}{\Delta x_{\mathbf{k}+\mathbf{e}_i}} - \sum_{i=1}^{m} \frac{\mu^-_{i,\mathbf{k}}}{\Delta x_{\mathbf{k}-\mathbf{e}_i}}, \\ S_{\mathbf{k}} &= 2 \sum_{\boldsymbol{\xi}_{\mathbf{k}} \in \Xi_{\mathbf{k}}} \eta_{\boldsymbol{\xi},\mathbf{k}}, \\ U_{i,\mathbf{k}} &= -\frac{\mu^+_{i,\mathbf{k}}}{\Delta x_{\mathbf{k}+\mathbf{e}_i}}. \end{aligned}$$

Later we will see that we need to keep $M_{\mathbf{k}}$ and $S_{\mathbf{k}}$ separate to handle points too close from the boundary. Going backward in time, we solve for $\mathbf{F}^{n+1} \equiv \mathbf{F}^t$ as a function of $\mathbf{F}^n \equiv \mathbf{F}^{t+1}$. In order to do so, we can now write (18) in matrix form as:

$$\mathbf{F}^{n+1} = \mathbf{A}^{-1} \left[\mathbf{u} + \frac{\mathbf{F}^n}{\Delta t} \right] \quad (19)$$

where

$$\mathbf{F}^n = \begin{bmatrix} F^n_{\mathbf{k}_1} \\ \vdots \\ F^n_{\mathbf{k}_D} \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} u_{\mathbf{k}_1} \\ \vdots \\ u_{\mathbf{k}_I} \end{bmatrix},$$

and

$$\mathbf{K} = \begin{bmatrix} \mathbf{k}_1 & \mathbf{k}_2 & \cdots & \mathbf{k}_D \end{bmatrix} = \begin{bmatrix} 1 & 2 & \cdots & d_1 & 1 & 2 & \cdots & d_1 & \cdots & d_1 \\ 1 & 1 & \cdots & 1 & 2 & 2 & \cdots & 2 & \cdots & d_2 \\ 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots & d_3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots & d_n \end{bmatrix}.$$

We denote $D = \prod_{i=1}^{n} d_i$ where d_i is the size of the n-dimensional grid in the i-th dimension, and $d_0 = 1$. The matrix $\mathbf{A} = \mathbf{A}^D + \mathbf{A}^M + \mathbf{A}^S + \mathbf{A}^U + \mathbf{A}^\eta$ is such that

$$\mathbf{A}^M (j, j) = M_{\mathbf{k}_j},$$

$$\mathbf{A}^S (j, j) = S_{\mathbf{k}_j},$$

$$\mathbf{A}^D \left(j, j - \textstyle\prod_{l=0}^{i-1} d_l\right) = D_{i,\mathbf{k}_j}, \qquad \mathbf{A}^U \left(j, j + \textstyle\prod_{l=0}^{i-1} d_l\right) = U_{i,\mathbf{k}_j},$$

$$\mathbf{A}^\eta \left(j, j - \textstyle\sum_{i=1}^{n} \xi_{i,\mathbf{k}} \prod_{l=0}^{i-1} d_l\right) = -\eta_{\boldsymbol{\xi},\mathbf{k}_j}, \qquad \mathbf{A}^\eta \left(j, j + \textstyle\sum_{i=1}^{n} \xi_{i,\mathbf{k}} \prod_{l=0}^{i-1} d_l\right) = -\eta_{\boldsymbol{\xi},\mathbf{k}_j}.$$

Now that we have our monotonic approximation, we can apply algorithm 3 exactly as we did with one dimension. Here as well, we are not assuming any boundary condition on the edge of the grid for the value function $\mathbf{F}^n$ in terms the vector of state variables $\mathbf{X}$. Implicitly, we assume that we do not need to do so because the value function will drift right at the left boundary and left at the right boundary. This is equivalent to assuming that there exists an interior absorbing stochastic steady state or that $\mu_{i,\mathbf{k}} > 0$ on the left boundary and $\mu_{i,\mathbf{k}} < 0$ on the right boundary for all dimensions i.

6 Example Application

In this section we present an example application of the numerical methods presented in this paper to a general equilibrium model with two state variables. We present a general extension of Brunnermeier and Sannikov (2014) where two agents have Epstein and Zin (1989) utility functions and aggregate volatility is time-varying. The framework can easily be modified to any other general equilibrium framework with two state variables.

6.1 Model Definition

Preferences There are two agent types: : households $h \in H$ and intermediaries $i \in I$. Both agents have stochastic differential utility, as developed by Duffie and Epstein (1992). The utility of agent j over his consumption process c_t^j is defined as

$$U_t^j = \mathbb{E}_t \left(\int_t^\infty f\left(c_s^j, U_s^j\right) ds \right).$$

The function $f_j(c,u)$ is a normalized aggregator of consumption and continuation value in each period defined as

$$f(c,U) = \frac{1-\gamma}{1-1/\zeta} U \left[\left(\frac{c}{((1-\gamma)U)^{1/(1-\gamma)}} \right)^{1-1/\zeta} - \rho \right]$$

where ρ is the rate of time preference, γ is the coefficient of relative risk aversion, and ζ determines the elasticity of intertemporal substitution. Each agent chooses its optimal consumption c_t^j, investment risk σ_t^r, and portfolio weight w_t^j on capital holdings in order to maximize discounted infinite life time expected utilities U_t^j. At any time, the following budget constraint has to be satisfied:

$$\frac{dn_t^j}{n_t^j} = \left((1-w_t^j)r_t + w_t^j \mu_t^{r,j} - \mathfrak{c}_t^j\right)dt + w_t^j \sigma_t^{q,\sigma} dZ_t^\sigma + w_t^j \left(\sigma_t + \sigma_t^{q,k}\right) dZ^k,$$

where n_t^j is the wealth of agent j, $\mathfrak{c}_t^j = c_t^j / n_t^j$ his consumption rate, and the portfolio weight w_t^j are choice variables. Z_t^σ and Z_t^k are two standard Brownian motions that hit aggregate volatility and capital growth, respectively.

Technology The production technology in the economy is given by:

$$y_t^j = \left(a^j - \iota_t^j\right) k_t^j$$

and

$$\frac{dk_t}{k_t} = \mu_t^k dt + \sigma_t dZ_t^k,$$

where μ_t^k is given by

$$\mu_t^k = \psi_t \Phi(\iota_t^i) + (1-\psi_t)\Phi(\iota_t^h),$$

where $\Phi(\cdot)$ is a concave investment function and ψ_t is the share of capital in the hands of intermediaries:

$$\psi_t \equiv \frac{w_t^i n_t^i}{w_t^i n_t^i + w_t^h n_t^h}.$$

In this example, we work with the following functional form for the investment function:

$$\Phi(\iota_t^j) = \log(1+\kappa_p \iota_t^j)/\kappa_p - \delta^j.$$

The price of a unit of capital is q_t. The volatility of capital returns follows a diffusion:

$$\frac{d\sigma_t}{\sigma_t} = \kappa\big(\overline{\sigma} - \sigma_t\big)dt + \varsigma dZ_t^\sigma.$$

The stochastic law of motion of q_t follows

$$\frac{dq_t}{q_t} = \mu_t^q dt + \sigma_t^{q,\sigma} dZ_t^\sigma + \sigma_t^{q,k} dZ_t^k.$$

The variables μ_t^q, $\sigma_t^{q,k}$, and $\sigma_t^{q,\sigma}$ are to be determined endogenously. We can use Ito's lemma to write the process of the value of capital:

$$\frac{d(q_t k_t^j)}{q_t k_t^j} = \big(\Phi_t + \mu_t^q + \sigma_t \sigma_t^{q,k}\big)dt + \sigma_t^{q,\sigma} dZ_t^\sigma + \big(\sigma_t + \sigma_t^{q,k}\big)dZ^k.$$

Hence, the return on physical asset is given by

$$dr_t^j = \underbrace{\left(\frac{a^j - \iota_t}{q_t} + \Phi_t + \mu_t^q + \sigma_t \sigma_t^{q,k}\right)}_{\mu_t^{r,j}} dt + \sigma_t^{q,\sigma} dZ_t^\sigma + \big(\sigma_t + \sigma_t^{q,k}\big)dZ^k.$$

6.2 Model Solution

Solving the HJB We will guess and verify that the homotheticity of preferences allows us to write the value function for agents of type j as:

$$U\left(n_t^j, \xi_t^j\right) = \frac{\left(n_t^j\right)^{1-\gamma} \xi_t^j}{1-\gamma}, \tag{20}$$

where variable ξ_t^j follows

$$\frac{d\xi_t^j}{\xi_t^j} = \mu_t^{\xi,j} dt + \sigma_t^{\xi,\sigma,j} dZ_t^\sigma + \sigma_t^{\xi,k,j} dZ_t^k.$$

We can write the HJB equation corresponding to the problem of agent j as

$$\begin{aligned} 0 = &\max_{\mathfrak{c}_t^j, \iota_t^j, w_t^j} f\left(\mathfrak{c}_t^j n_t^j, U_t^j\right) \\ &+\left((1-w_t^j)r_t + w_t^j \mu_t^{r,j} - \mathfrak{c}_t^j\right) n_t^j U_n(n_t^j, \xi_t^j) + \mu_t^{\xi,j} \xi_t^j U_\xi(n_t^j, \xi_t^j) \\ &+\frac{1}{2}\left[\left(w_t^j \sigma_t^{q,\sigma} n_t^j\right)^2 + \left(w_t^j(\sigma_t + \sigma_t^{q,k}) n_t^j\right)^2\right] U_{nn}(n_t^j, \xi_t^j) \\ &+\frac{1}{2}\left[\left(\sigma^{\xi,\sigma,j}\xi_t^j\right)^2 + \left(\sigma^{\xi,k,j}\xi_t^j\right)^2\right] U_{\xi\xi}(n_t^j, \xi_t^j) \\ &+\left[w_t^j \sigma_t^{q,\sigma} n_t^j \sigma_t^{\xi,\sigma,j} \xi_t + w_t^j(\sigma_t + \sigma_t^{q,k}) n_t^j \sigma_t^{\xi,k,j} \xi_t^j\right] U_{n\xi}(n_t^j, \xi_t^j). \end{aligned} \tag{21}$$

Substituting the guess from equation (20), the HJB becomes

$$\begin{aligned} 0 = &\max_{\mathfrak{c}_t^j, \iota_t^j, w_t^j} \frac{1}{1-1/\zeta}\left[\frac{\left(\mathfrak{c}_t^j\right)^{1-1/\zeta}}{\left(\xi_t^j\right)^{\frac{1-1/\zeta}{1-\gamma}}} - \rho\right] + (1-w_t^j)r_t + w_t^j \mu_t^{r,j} - \mathfrak{c}_t^j + \frac{\mu^{\xi,j}}{1-\gamma} \\ &-\frac{\gamma}{2}\left(w_t^j \sigma_t^{q,\sigma}\right)^2 - \frac{\gamma}{2}\left(w_t^j \sigma_t + w_t^j \sigma_t^{q,k}\right)^2 + w_t^j \sigma_t^{q,\sigma} \sigma_t^{\xi,\sigma,j} + w_t^j(\sigma_t + \sigma_t^{q,k})\sigma_t^{\xi,k,j}. \end{aligned} \tag{22}$$

Optimality Conditions The first order conditions with respect to $\mathfrak{c}_t^j, \iota_t^j$, and w_t^j are given by

$$\left(\mathfrak{c}_t^j\right)^{-1/\zeta} = \left(\xi_t^j\right)^{\frac{1-1/\zeta}{1-\gamma}},$$

$$1/q_t = \Phi_\iota(\iota_t),$$

$$\mu_t^{r,j} - r_t - \gamma w_t^j \left(\sigma_t^{q,\sigma}\right)^2 - \gamma w_t^j \left(\sigma_t + \sigma_t^{q,k}\right)^2 + \sigma_t^{q,\sigma}\sigma_t^{\xi,\sigma,j} + \left(\sigma_t + \sigma_t^{q,k}\right)\sigma_t^{\xi,k,j} = 0.$$

Plugging in the optimality conditions in the HJB gives:

$$0 = \frac{1}{1-1/\zeta}\left(\mathfrak{c}_t^j - \rho\right) + r_t - \mathfrak{c}_t^j + \frac{\gamma}{2}\left(w_t^j \sigma_t^{q,\sigma}\right)^2 + \frac{\gamma}{2}\left(w_t^j \sigma_t + w_t^j \sigma_t^{q,k}\right)^2 + \frac{\mu^{\xi,j}}{1-\gamma}. \tag{23}$$

Market Clearing Conditions We can use the market clearing condition for consumption to find q_t:

$$\left(\mathfrak{c}_t^i \eta_t + \mathfrak{c}_t^h (1-\eta_t)\right) q_t = \psi_t(a^i - \iota_t) + (1-\psi_t)(a^h - \iota_t).$$

Similarly, we use the market clearing condition for capital to find r_t:

$$w_t^i \eta_t + w_t^h (1-\eta_t) = 1.$$

Numerical Procedure We want to solve the model recursively in a minimal number of state variables summarizing time variations in the equilibrium. We start by providing the definition of such an equilibrium in the state variables $\{\eta_t, \sigma_t\}$, where η_t is defined as the share of wealth in the hands of the intermediaries:

$$\eta_t = \frac{n_t^i}{n_t^h + n_t^i} = \frac{n_t^i}{q_t k_t}.$$

We can therefore use Ito's lemma to write the law of motion of η_t as:

$$\begin{aligned}\frac{d\eta_t}{\eta_t} =& \Big(r_t + w_t^i(\mu_t^{r,j} - r_t) - \mathfrak{c}_t^i - \Phi_t - \mu_t^q - \sigma_t\sigma_t^{q,k} \\ & - \omega_t^i\,(\sigma_t^{q,\sigma})^2 + (\sigma_t^{q,\sigma})^2 + \left(\sigma_t + \sigma_t^{q,k}\right)^2 - w_t^i\left(\sigma_t + \sigma_t^{q,k}\right)^2\Big)dt \\ & + \left(w_t^i - 1\right)\sigma_t^{q,\sigma} dZ_t^\sigma + \left(w_t^i - 1\right)\left(\sigma_t + \sigma_t^{q,k}\right)dZ^k.\end{aligned} \tag{24}$$

Definition 1 *A Markov Equilibrium in $\{\eta, \sigma\}$ is a set of functions $q(\eta,\sigma)$, $\psi(\eta,\sigma)$, $r(\eta,\sigma)$, $w^i(\eta,\sigma)$, $w^h(\eta,\sigma)$, $\iota(\eta,\sigma)$, $\mathfrak{c}^i(\eta,\sigma)$, $\mathfrak{c}^h(\eta,g)$, $\xi^i(\eta,\sigma)$ and $\xi^h(\eta,\sigma)$ and diffusions $\mu^\eta(\eta,\sigma)$, $\sigma^\eta(\eta,\sigma)$, $\mu^q(\eta,\sigma)$, $\sigma^q(\eta,\sigma)$ such that:*

1. *ξ^i and ξ^h solve their respective HJB equations (23).*
2. *Taking prices q, r and the law of motion of η and q as given, policy variables w^i, w^h, ι, $\mathfrak{c}^i$, $\mathfrak{c}^h$ solve their respective optimization problems.*
3. *Law of motions for the state variables η and σ are given by (**??**) and (24).*

Algorithm 3 *Implicit Euler for Two-Dimensional General Equilibrium Model*

1. *Define a finite grid over the state variables η, σ and set guess for ξ_n^i and ξ_n^h at the initial iteration $n = 0$.*
2. *Given ξ_n^i and ξ_n^h solve for all equilibrium variables $q(\eta,\sigma)$, $\psi(\eta,\sigma)$, $r(\eta,\sigma)$, $w^i(\eta,\sigma)$, $w^h(\eta,\sigma)$, $\iota(\eta,\sigma)$, $\mathfrak{c}^i(\eta,\sigma)$, $\mathfrak{c}^h(\eta,g)$, $\xi^i(\eta,\sigma)$ and $\xi^h(\eta,\sigma)$ and diffusions $\sigma^{q,\sigma}(\eta,\sigma)$, $\sigma^{q,k}(\eta,\sigma)$, $\sigma^{\xi,\sigma,j}(\eta,\sigma)$, $\sigma^{\xi,k,j}(\eta,\sigma)$, and $\mu^q(\eta,\sigma)$ using first order conditions and market clearing conditions. One can solve this nonlinear system of equation using a Newton-Raphson method.*[7]
3. *Solve for the next iteration of ξ_{n+1}^i and ξ_{n+1}^h using the method described in Section 5.2.*
4. *Iterate on 2-3 until convergence.*

[7]To provide a good first guess for the Newton-Raphson algorithm, we solve the nonlinear system of equation setting the derivatives of $q(\sigma,\eta)$ to 0.

We can now solve the model according to algorithm 3. The procedure works in two steps. At each iteration, we first solve for all equilibrium variables recursively in the state variables and then iterate on the value function multiplier. The key for this second step is to use the finite difference approximation that preserves the monotonicity of the HJB equation as described in section 5. Since we get $\mu^{\xi,j}$ from (23), we can apply the method of finite difference to

$$\begin{aligned}\xi^j(\eta_t,\sigma_t)\mu_t^{\xi,j} &= \xi_\sigma^j(\eta_t,\sigma_t)\mu_t^\sigma\sigma_t + \xi_\eta^j(\eta_t,\sigma_t)\mu_t^\eta\eta_t + \frac{1}{2}\xi_{\sigma\sigma}^j(\eta_t,\sigma_t)\big(\varsigma\sigma_t\big)^2 \\ &+ \frac{1}{2}\xi_{\eta\eta}^j(\eta_t,\sigma_t)\left[\big((\omega_t^i-1)\sigma_t^{q,\sigma}\eta_t\big)^2 + \big((\omega_t^i-1)\big(\sigma_t+\sigma_t^{q,k}\big)\eta_t\big)^2\right] \\ &+ \xi_{\sigma\eta}^j(\eta_t,\sigma_t)\varsigma\sigma_t\big(\omega_t^i-1\big)\sigma_t^{q,\sigma}\eta_t + \xi_t^j(\eta_t,\sigma_t).\end{aligned}$$

By applying Ito's lemma, we can find $\sigma_t^{q,\sigma}$, $\sigma_t^{q,k}$, $\sigma_t^{\xi,\sigma,j}$, $\sigma_t^{\xi,k,j}$, and μ_t^q from:

$$\begin{aligned}q(\sigma_t,\eta_t)\sigma_t^{q,\sigma} &= q_\sigma(\sigma_t,\eta_t)\varsigma\sigma_t + q_\eta(\sigma_t,\eta_t)\big(w_t^i-1\big)\sigma_t^{q,\sigma}\eta_t, \\ q(\sigma_t,\eta_t)\sigma_t^{q,k} &= q_\eta(\sigma_t,\eta_t)\big(w_t^i-1\big)\big(\sigma_t+\sigma_t^{q,k}\big)\eta_t, \\ \xi^j(\sigma_t,\eta_t)\sigma_t^{\xi,\sigma,j} &= \xi_\sigma^j(\sigma_t,\eta_t)\varsigma\sigma_t + \xi_\eta^j(\sigma_t,\eta_t)\big(w_t^i-1\big)\sigma_t^{q,\sigma}\eta_t, \\ \xi^j(\sigma_t,\eta_t)\sigma_t^{\xi,k,j} &= \xi_\eta^j(\sigma_t,\eta_t)\big(w_t^i-1\big)\big(\sigma_t+\sigma_t^{q,k}\big)\eta_t, \\ q(\sigma_t,\eta_t)\mu_t^q &= q_\sigma(\sigma_t,\eta_t)\mu_t^\sigma\sigma_t + q_\eta(\sigma_t,\eta_t)\mu_t^\eta\eta_t + \frac{1}{2}q_{\sigma\sigma}(\sigma_t,\eta_t)\big(\varsigma\sigma_t\big)^2 \\ &+ \frac{1}{2}q_{\eta\eta}(\sigma_t,\eta_t)\left[\big((\omega_t^i-1)\sigma_t^{q,\sigma}\eta_t\big)^2 + \big((\omega_t^i-1)\big(\sigma_t+\sigma_t^{q,k}\big)\eta_t\big)^2\right] \\ &+ q_{\sigma\eta}(\sigma_t,\eta_t)\varsigma\sigma_t\big(\omega_t^i-1\big)\sigma_t^{q,\sigma}\eta_t.\end{aligned}$$

7 Numerical Implementation: PyMacroFin

PyMacroFin is an open source package[8] developed in Python by the authors that implements the numerical solution methodology presented in this paper. PyMacroFin allows users to input a model to be solved using an intuitive object-oriented interface in Python. In this section we introduce the package functionality by illustrating how the package can be used to solve the example application presented in the previous section as well as a

[8] The package is available for download via the PyPi package manager: https://pypi.org/project/PyMacroFin/. Download instructions are also included in the package documentation: https://adriendavernas.com/pymacrofin/index.html.

one-dimensional example from Brunnermeier and Sannikov (2014).

7.1 Model Definition

The core of an implementation of a model in PyMacroFin is a model object which will hold model equations, parameters, settings, as well as results. A model object should be defined and given a name as follows:

```
from PyMacroFin.model import macro_model
m = macro_model(name='BruSan')
```

Model parameters (constants) are specified and given values using the following command:

```
m.params.add_parameter('parameter_name', parameter_value).
```

Model variables are defined differently depending on the type of variable. State variables are defined with the method `m.set_state()`, value variables are defined with the method `m.set_value()`, endogenous variables are defined by the method `m.set_endog`, and intermediate variables are defined with `m.equation()` commands. In our model, the state variables are η_t and σ_t which we define as `e` and `z`, respectively, as follows:

```
m.set_state(['e','z'])
```

This sets the state variables of the model. The value function or wealth multipliers ξ_t^i and ξ_t^h will be defined as `vi` and `vh`, respectively, as follows:

```
m.set_value(['vi','vh'])
```

Endogenous variables are variables whose values are solved for using market clearing and first order conditions at each iteration backward through the transient, or false time, dimension. In our example, these include q_t, ψ_t, μ_t^η, $\sigma_t^{q,k}$, and $\sigma_t^{q,\sigma}$. This is defined as follows:

```
m.set_endog(['q','psi','mue','sigqk','sigqs'])
```

Although not demonstrated here for expository purposes, these variable definition commands also include optional arguments to provide initial numeric guess values, in the form of either constants or functions of state variable values to provide variation across the finite difference grid, for the values of the variables as well as optional arguments to provide latex rendering of variable names for visualization purposes. For details on command syntax the reader is referred to the package documentation.[9] Intermediate variables can be assigned by defining equations in terms of endogenous, state, or value variables or other intermediate variables. An example is $w_t^i = \psi_t/\eta_t$ which we define via the following command:

```
m.equation("wi = psi/e")
```

To inform the package how to solve the equilibrium at each node of the finite difference grid at each transient iteration, a system of equations is required of the same dimension as the vector of endogenous variables. In our example, we use the definition of μ_t^η, the market clearing condition for consumption, a combination of the first order conditions for w_t^i and w_t^h, and definitions of $\sigma_t^{q,k}$ and $\sigma_t^{q,\sigma}$ from Ito's Lemma. All other necessary equilibrium conditions not in these equations are included in intermediate variable definitions leading to these equations. We define these equations in the form $0 = f(\mathbf{X})$ where $\mathbf{X}$ is the vector of endogenous variables. The equations are as follows:

$$\begin{aligned}
0 = &\frac{\kappa_L}{\eta_t}(\overline{\eta} - \eta_t) + (1-\eta_t)(\mu_t^{n,i} - \mu_t^q - \mu_t^k - \sigma_t\sigma_t^{q,k} + (\sigma_t^{q,k} + \sigma_t)^2 \\
&+ (\sigma_t^{q,\sigma})^2 - w^i(\sigma_t^{q,\sigma})^2 - w_t^i(\sigma_t^{q,k} + \sigma_t)^2) - \mu_t^\eta
\end{aligned} \tag{25}$$

$$0 =(\mathfrak{c}_t^i\eta_t + \mathfrak{c}_t^h(1-\eta_t))q_t - (a^i - \iota_t^i)\eta_t w_t^i - (a^h - \iota_t^h)(1-\eta_t)w_t^h \tag{26}$$

$$\begin{aligned}
0 =&\mu_t^{r,i} - \mu_t^{r,h} + \gamma^h w_t^h((\sigma_t^{q,\sigma})^2 + (\sigma_t + \sigma_t^{q,k})^2) - \gamma^i w_t^i((\sigma_t^{q,\sigma})^2 + (\sigma_t + \sigma_t^{q,k})^2) \\
&+ \sigma_t^{q,\sigma}\sigma_t^{\xi,i,\sigma} + (\sigma_t + \sigma_t^{q,k})\sigma_t^{\xi,i,k} - \sigma_t^{q,\sigma}\sigma_t^{\xi,h,\sigma} - (\sigma_t + \sigma_t^{q,k})\sigma_t^{\xi,h,k}
\end{aligned} \tag{27}$$

$$0 =(\varsigma q_\sigma \sigma_t + \sigma_t^{\eta,\sigma} q_\eta \eta) - \sigma_t^{q,\sigma} q_t \tag{28}$$

$$0 =\sigma_t^{\eta,k} q_\eta \eta - \sigma_t^{q,k} q_t \tag{29}$$

Endogenous equations, using Equation 29 as an example, are input as follows:

```
m.endog_equation("sigek*qe*e-sigqk*q")
```

where $\sigma_t^{\eta,k}$ is defined as `sigek`, q_η is defined as `qe`, and q_t is defined as `q`.

[9] https://adriendavernas.com/pymacrofin/index.html

The variables associated with the HJB equations must also be specified to inform the package how to iterate backward through the transient dimension to find the steady-state equilibrium solution to the system. The partial differential equation takes the following form as outlined in Section 5:

$$
\begin{aligned}
r(\mathbf{X})F(\mathbf{X},t) = u(\mathbf{X}) &+ \sum_{i=1}^{m} \mu_i(\mathbf{X}) \frac{\partial F(\mathbf{X},t)}{\partial x_i} \\
&+ \sum_{i=1}^{m}\sum_{j=1}^{m} \frac{\sigma_i(\mathbf{X})\sigma_j(\mathbf{X})}{2} \frac{\partial^2 F(\mathbf{X},t)}{\partial x_i \partial x_j} + \frac{\partial F(\mathbf{X},t)}{\partial t}
\end{aligned}
\tag{30}
$$

where x_i are state variables, $F(\mathbf{X},t)$ are value variables, and $\mathbf{X}$ is a vector of all variables. The `m.hjb_equation()` method is used to inform the model object of the values to be used in this partial differential equation. The method links variables in Equation 30 to variables defined in a model. In our example, the drifts $\mu_i(\mathbf{X})$ in Equation 30 are linked to $\mu_t^\eta \eta_t$ and $\mu_t^\sigma \sigma_t$ by the following commands:

```
m.hjb_equation('mu','e','mue*e')
m.hjb_equation('mu','z','muz*z')
```

where the first two arguments specify that we are setting the value for the drift, $\mu_i(\mathbf{X})$, for state variable $i = \eta_t$, and the last argument specifies the value to be used in terms of variables defined as endogenous, value, state, or intermediate variables (`mue` representing μ_t^η and `muz` representing μ_t^σ). The details of linking syntax for other variables in Equation 30 is left to the package documentation.

Although unnecessary in our example, PyMacroFin also includes flexible methods for defining boundary conditions for endogenous variables, either as functions or as constants. PyMacroFin also includes flexible constraint management, such that a different system of endogenous equations may be solved in the case of a binding constraint. For details on these additional features the reader is directed to the package documentation.

7.2 Model Solution

Once a model is defined in PyMacroFin, running the model is very simple:

```
m.run()
```

Updates will be printed to Python standard output while iterations progress, if requested, and solutions will be saved to a user specified location. A live dashboard with 2-dimensional or 3-dimensional plots of requested variable values can also be displayed while the model solves. For syntax for these requests the reader is directed to the package documentation. The size of the finite difference grid, error tolerances, minimum and maximum iterations, and time step used can be specified by setting model options. For example, the time step can be set as follows before running the model:

```
m.options.dt = 0.1
```

Full documentation of all available options is detailed in the package documentation. The package also includes a method which can be used to obtain the stationary distribution of a system using the Kolmogorov Forward Equation from arrays of values of state variables with associated drifts and volatilities. The solution of the model from the previous section as well as the code to produce the solution are provided in the Appendix. As an example of a one-dimensional solution, the solution to the model of Brunnermeier and Sannikov (2014) and code used to produce the solution are also included in the Appendix. On a Macbook Pro with a M2 chip, the method converges to a solution in less than 2 minutes.

8 Conclusion

In this article, we provide a fast method to solve globally for a broad class of continuous time macro-economic models with Brownians shocks and up to two endogenous (and correlated) state variables. Due to its speed, generality, and robustness, this method opens the door to further research in macroeconomics and asset pricing. For instance, it could be used to solve models with banks with interest for monetary policy or more complex asset pricing model involving heterogeneous agents, financial frictions, and production. The speed of the method also makes it possible to run estimations on the global model without linearization and capture complex amplification dynamics. We also present an open source software package which implements the solution methodology and allows for flexible and intuitive model definitions.

References

S. Basak and D. Cuoco. An equilibrium model with restricted stock market participation. *Review of Financial Studies*, 11(2):309–341, 1998.

F. Bonnans, E. Ottenwaelter, and H. Zidani. A fast algorithm for the two dimensional HJB equation of stochastic control. *ESAIM: Mathematical Modelling and Numerical Analysis*, 38(4):723–735, 2004.

M. Brunnermeier and Y. Sannikov. A macroeconomic model with a financial sector. *The American Economic Review*, 104(2):379–421, 2014.

M. Brunnermeier and Y. Sannikov. Macro, money, and finance: A continuous-time approach. *Handbook of Macroeconomics*, 2:1497–1545, 2016.

G. Candler. Finite-difference methods for continuous-time dynamic programming. In R. Marimon and A. Scott, editors, *Computational Methods for the Study of Dynamic Economies*, chapter 8, pages 172–194. Oxford University Press, Oxford, 1999.

S. Di Tella. Uncertainty shocks and balance sheet recessions. *The Journal of Political Economy*, 125(26):2038–2081, 2017.

S. Di Tella and P. Kurlat. Why are bank balance sheets exposed to monetary policy? Working Paper 24076, NBER, 2016.

I. Drechsler, A. Savov, and P. Schnabl. Uncertainty shocks and balance sheet recessions. *The Journal of Finance*, 73(1):317–373, 2017.

V. Duarte. Machine learning for continuous-time finance. Working paper, University of Illinois Urbana-Champaign, 2022.

D. Duffie and L. Epstein. Stochastic differential utility. *Econometrica*, 60(2):353–394, 1992.

L. Epstein and S. Zin. Substitution, risk aversion, and the temporal behavior of consumption and asset returns: A theoretical framework. *Econometrica*, 57(4):937–969, 1989.

G. Gopalakrishna. Aliens and continuous time economics. Working paper, Swiss Finance Institute at École Polytechnique Fédérale de Lausanne, October 2022.

L. P. Hansen, F. Tourre, and P. Khorrami. Comparative valuation dynamics in models with financing restrictions. Technical report, University of Chicago and Northwestern University, 2019.

Z. He and A. Krishnamurthy. Intermediary asset pricing. *The American Economic Review*, 103(2):732–770, 2013.

R. Merton. An intertemporal capital asset pricing model. *Econometrica*, 41(5):867–887, 1973.

P. Niyogi. *Introduction to Computational Fluid Dynamics*. Pearson Education India, 2006.

M. Sauzet. Projection methods via neural networks for continuous-time models. Working Paper 3981838, SSRN, 2022.

D. Silva. The risk channel of unconventional monetary policy. Working paper, Krannert School of Management at Purdue University, 2016.

A. Tourin. An introduction to finite difference methods for pdes in finance. Technical report, Fields Institute, 2011.

Appendix

A Code and Results

A.1 Model with Two State Variables

In this section we provide the code and results for the model presented in Section 6.

A.1.1 Code

```
from PyMacroFin.model import macro_model
import numpy as np
import pandas as pd
import time
import PyMacroFin.utilities as util

def define_model():
        m = macro_model(name='BruSan')

        m.set_endog(['q','psi','mue','sigqk','sigqs'],init=[1,0.95,0,0,0],
                latex=[r'$q$',r'$\psi$',r'$\mu^{\eta}$',
                       r'$\sigma^{q,k}$',r'$\sigma^{q,\sigma}$'])
        m.prices = ['q']
        m.set_state(['e','z'])
        m.set_value(['vi','vh'],init=[0.04,0.04],latex=[r'$\xi^i$',r'$\xi^h$'])

        m.params.add_parameter('gammai',2)
        m.params.add_parameter('gammah',3)
        m.params.add_parameter('ai',.1)
        m.params.add_parameter('ah',.1)
        m.params.add_parameter('rhoi',.04)
        m.params.add_parameter('rhoh',.04)
        m.params.add_parameter('sigz',.01)
        m.params.add_parameter('sigbar',.5)
        m.params.add_parameter('deltai',.04)
        m.params.add_parameter('deltah',.04)
        m.params.add_parameter('kappa_p',2)
        m.params.add_parameter('kappa_z',5)
```

```
m.params.add_parameter('zetai',1.15)
m.params.add_parameter('zetah',1.15)
m.params.add_parameter('kappa_l',.9)
m.params.add_parameter('ebar',0.5)

m.equation("sigma = z")
m.equation("wi = psi/e")
m.equation("wh = (1-psi)/(1-e)")
m.equation("ci = vi**((1-zetai)/(1-gammai))")
m.equation("ch = vh**((1-zetah)/(1-gammah))")
m.equation("iotai = (q-1)/kappa_p")
m.equation("iotah = (q-1)/kappa_p")
m.equation("phii = log(1+kappa_p*iotai)/kappa_p-deltai")
m.equation("phih = log(1+kappa_p*iotah)/kappa_p-deltah")
m.equation("muz = kappa_z*(sigbar-sigma)")
m.equation("muk = psi*phii+(1-psi)*phih")
m.equation("signis = wi*sigqs")
m.equation("signhs = wh*sigqs")
m.equation("signik = wi*(sigqk+sigma)")
m.equation("signhk = wh*(sigqk+sigma)")
m.equation("siges = e*(1-e)*(signis -sigqs)")
m.equation("sigek = e*(1-e)*(signik - (sigqk+sigma))")
m.equation("sigxik = d(vi,e)/vi*sigek*e")
m.equation("sigxhk = d(vh,e)/vh*sigek*e")
m.equation("sigxis = d(vi,e)/vi*siges*e + d(vi,z)/vi*sigz*z")
m.equation("sigxhs = d(vh,e)/vh*siges*e + d(vh,z)/vh*sigz*z")
m.equation("muq = d(q,e)/q*mue*e + d(q,z)/q*muz*z + \
                        1/2*d(q,e,e)/q*((siges*e)**2 + (sigek*e)**2) + \
                        1/2*d(q,z,z)/q*(sigz*z)**2 + d(q,e,z)/q*siges*e*sigz*z")
m.equation("muri = (ai-iotai)/q + phii + muq + sigma*sigqk")
m.equation("murh = (ah-iotah)/q + phih + muq + sigma*sigqk")
m.equation("r = muri - gammai*wi*((sigqs**2)+(sigma+sigqk)**2) + \
                        sigqs*sigxis + (sigqk+sigma)*sigxik")
m.equation("muni = r + wi*(muri-r)-ci")
m.equation("munh = r + wh*(murh-r)-ch")

m.endog_equation("kappa_l/e*(ebar-e)+(1-e)*(muni - muk - muq\
                                  - sigma*sigqk + (sigqk+sigma)**2 + sigqs**2 \
                                  - wi*sigqs**2 - wi*(sigqk+sigma)**2) - mue")
m.endog_equation("(ci*e+ch*(1-e))*q - psi*(ai-iotai) - (1-psi)*(ah-iotah)")
m.endog_equation("muri - murh + gammah*wh*((sigqs**2)+(sigqk+sigma)**2) - \
                                          gammai*wi*((sigqs)**2+(sigqk+sigma)**2) + sigqs*sigxis + \
                                          (sigqk+sigma)*sigxik - sigqs*sigxhs - (sigqk+sigma)*sigxhk")
        m.endog_equation("(sigz*z*d(q,z) + siges*e*d(q,e))-sigqs*q")
        m.endog_equation("sigek*e*d(q,e) - sigqk*q")

        m.hjb_equation('mu','e','mue*e')
        m.hjb_equation('mu','z','muz*z')
        m.hjb_equation('sig','e',"(siges*e)**2 + (sigek*e)**2")
        m.hjb_equation('sig','z',"(sigz*z)**2")
        m.hjb_equation('sig','cross',"siges*e*sigz*z")
        m.hjb_equation('u','vi',0)
        m.hjb_equation('u','vh',0)
        m.hjb_equation('r','vi',"-1*(1-gammai)*(1/(1-1/zetai)*(ci-(rhoi+kappa_l))\
                                        +r-ci+gammai/2*(wi*(sigqs)**2 +wi*(sigqk+sigma)**2))")
        m.hjb_equation('r','vh',"-1*(1-gammah)*(1/(1-1/zetah)*(ch-(rhoh+kappa_l))\
                                        +r-ch+gammah/2*(wh*(sigqs)**2 +wh*(sigqk+sigma)**2))")

        m.options.loop = False
        m.options.outer_plot = True
        m.options.n0 = 50
        m.options.n1 = 50
        m.options.start0 = 0.05
        m.options.start1 = 0.05
        m.options.end0 = 0.95
        m.options.end1 = 0.95
        m.options.inner_solver = 'newton-raphson'
        m.options.parallel = True

        return m

if __name__=='__main__':
        tic = time.time()
        m = define_model()
        util.deploy_dash(m)
        m.run()
        toc = time.time()
        print('elapsed time: {}'.format(toc-tic))
```

A.1.2 Results

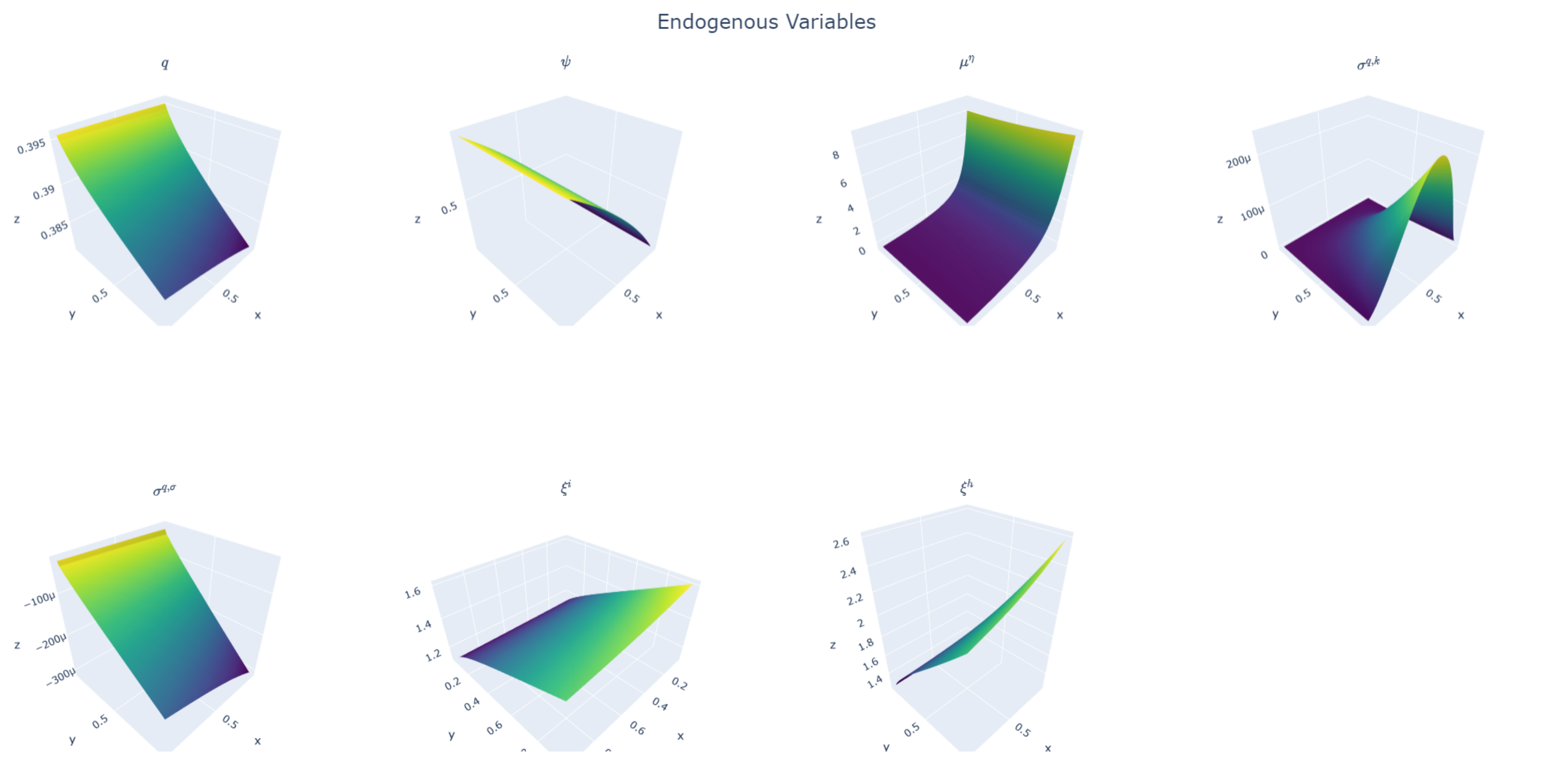

Figure 2: Endogenous variable solutions for the model presented in Section 6.

A.2 Model with One State Variable

In this section we present the code and results replicating the model of Brunnermeier and Sannikov (2014).

A.2.1 Code

```
from PyMacroFin.model import macro_model
import numpy as np
import pandas as pd
import time
import PyMacroFin.utilities as util
from PyMacroFin.system import system

# initial guess function for endogenous variables
def init_fcn(e,c):
        if e<.3:
                q = 1.05+.06/.3*e
                psi = 1/.3*e
                sigq = -.1*(e-.3)**2+.008
        else:
                psi = 1
                sigq = 0
                q = 1.1 - .03/.7*e
        return [q,psi]

# boundary condition function for eta == 0
def eta_minimum(d):
        psi = 0
        q = (2*d['ah']*d['kappa']+(d['kappa']*d['r'])**2.+1)**0.5 - d['kappa']*d['r']
        return [q,psi]


def define_model(npoints):
        m = macro_model(name='BruSan14_log_utility')

        m.set_endog(['q','psi'],init=[1.05,0.5])
        m.prices = ['q']
        m.set_state(['e'])
```

```
m.params.add_parameter('sig',.1)
m.params.add_parameter('deltae',.05)
m.params.add_parameter('deltah',.05)
m.params.add_parameter('rho',.06)
m.params.add_parameter('r',.05)
m.params.add_parameter('ae',.11)
m.params.add_parameter('ah',.07)
m.params.add_parameter('kappa',2)

m.equation('iota = (q**2-1)/(2*kappa)')
m.equation('phi = 1/kappa*((1+2*kappa*iota)**0.5-1)')
m.equation('sigq = (((ae-ah)/q+deltah-deltae)/(psi/e-(1-psi)/(1-e)))**0.5 - sig',plot=True,latex=r'$\si
m.equation('sige = (psi-e)/e*(sig+sigq)')
m.equation('mue = sige**2 + (ae-iota)/q + (1-psi)*(deltah-deltae)-rho')
m.equation('er = psi/e*(sig+sigq)**2',plot=True,latex=r'$E[dr_t^k-dr_t]/dt$')
m.equation('sigee = sige*e',plot=True,latex=r'$\sigma^{\eta} \eta$')
m.equation('muee = mue*e',plot=True,latex=r'$\mu^{\eta} \eta$')

m.endog_equation('q*(r*(1-e)+rho*e) - psi*ae - (1-psi)*ah + iota')
m.endog_equation('(psi-e)*d(q,e) - q*(1-sig/(sig+sigq))')

m.hjb_equation('mu','e','mue')
m.hjb_equation('sig','e','sige')

m.constraint('psi','<=',1,label='upper_psi')
m.constraint('psi','>=',0,label='lower_psi')

m.boundary_condition({'e':'min'},eta_minimum)

s = system(['upper_psi'],m)
s.equation('sigq = sig/(1-(psi-e)*d(q,e)/q) - sig')
s.endog_equation('1 - psi')
s.endog_equation('q*(r*(1-e)+rho*e) - ae + iota')

m.systems.append(s)

m.options.ignore_HJB_loop = True
m.options.import_guess = False
m.options.guess_function = init_fcn
m.options.inner_plot = False
m.options.outer_plot = False
```

```
        m.options.final_plot = True
        m.options.n0 = npoints
        m.options.start0 = 0.0
        m.options.end0 = 0.95
        m.options.inner_solver = 'least_squares'
        m.options.derivative_plotting = [('q','e')]
        m.options.min_iter_outer_static = 5
        m.options.min_iter_inner_static = 0
        m.options.max_iter_outer_static = 50
        m.options.return_solution = True
        m.options.save_solution = False
        m.options.price_derivative_method = 'backward'

        return m

if __name__=='__main__':
        npoints = 100
        tic = time.time()
        m = define_model(npoints)
        df = m.run()
        toc = time.time()
        print('elapsed time: {}'.format(toc-tic))
```

A.2.2 Results

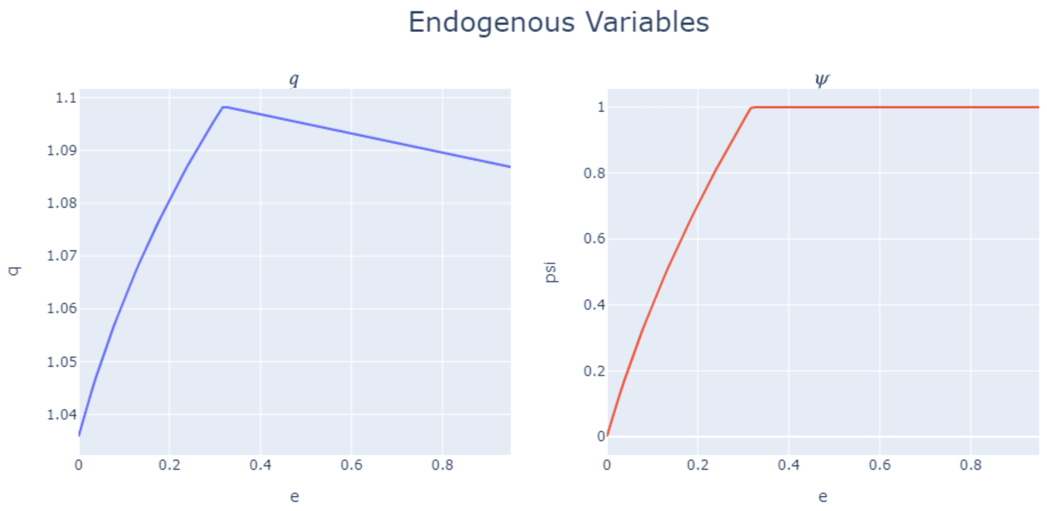

Figure 3: Endogenous variable solutions for the model from Brunnermeier and Sannikov (2014).

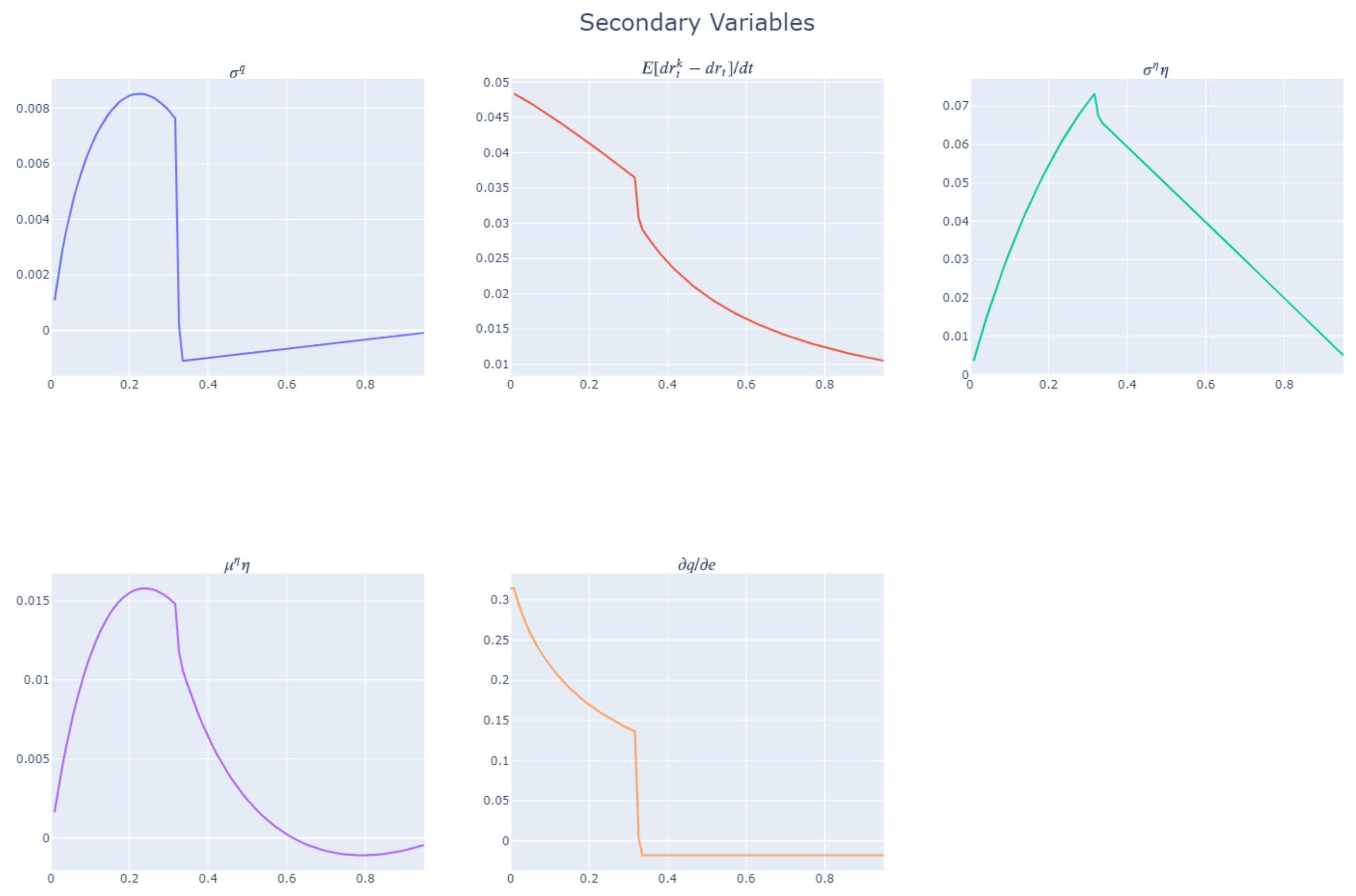

Figure 4: Secondary variable solutions for the model from Brunnermeier and Sannikov (2014).