This documents shows how to solve the Merton Portfolio Choice Problem numerically with cubic splines and Chebyshev polynomials.

Contents

1 The Model

The Merton Portfolio Choice Model provides a continuous-time framework for optimal portfolio selection and consumption under uncertainty. The investor seeks to maximize expected utility derived from both flow consumption and terminal wealth over a finite time horizon.

The financial market consists of

- A risk-free asset with constant rate of return r.
- A risky asset whose value S_t evolves according to the following geometric brownian motion

$$dS(t) = \mu S(t)dt + \sigma S(t)dW(t)$$

  where $\mu > 0$ is the expected return, $\sigma > 0$ is the volatility and $W(t)$ is a standard Brownian motion.

The investor's asset wealth $a(t)$ evolves according to the following SDE

$$da(t) = [ra(t) + \pi(t)(\mu - r)a(t) - c(t)]\,dt + \pi(t)\sigma a(t)dW(t)$$

where

- $\pi(t)$ is the amount of wealth allocated to the risky asset at time t,
- $c(t)$ is the consumption rate at time t.

The investor aims to maximize the expected CRRA-utility of flow consumption and terminal wealth. Let time $t \in [0, T]$ with $T < \infty$. Then, the Value Function at each time point is given by

$$V(t, a_t) = \max_{\{c(s), \pi(s)\}_{s \in [t,T]}} \mathbb{E}_t \left[\int_t^T e^{-\rho(s-t)} u(c(s)) ds + e^{-\rho(T-t)} u(a(T)) \right] \qquad \forall t \in [0, T]$$
$$s.t.\ da(s) = [ra(s) + \pi(s)(\mu - r)a(s) - c(s)]\, ds + \pi(s)\sigma a(s) dW(s)$$

where

- $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$ is the utility function with $\gamma > 1$ being the risk aversion parameter.
- $\rho > 0$ is the subjective discount rate.

Notice that the Value Function at $t = T$ is simply given by utility of terminal wealth.

Using dynamic programming, the Hamilton-Jacobi-Bellman (HJB) equation is given by (omitting the argument t and a for V)

$$\rho V = \max_{c_t, \pi_t} \Big\{ u(c_t) + \partial_t V + (ra_t + \pi_t(\mu - r)a_t - c_t)\partial_a V + 0.5(\pi_t \sigma a_t)^2 \partial^2_{a^2} V \Big\},$$

where the optimal policies c_t^* and π_t^* are given by the First Order Conditions

- $c_t^* = [u']^{-1}(\partial_a V)$,
- $\pi_t^* = -\dfrac{\mu - r}{\sigma^2 a_t} \dfrac{\partial_a V}{\partial^2_{a^2} V}$.

Summing up, we must find the Value function $V(t, a)$ for $t \in [0, T]$ and $a > 0$ such that following four conditions are satisfied

$$\rho V = u(c_t^*) + \partial_t V + \big(ra_t + \pi_t^*(\mu - r)a_t - c_t^*\big)\partial_a V + 0.5\big(\pi_t^* \sigma a_t\big)^2 \partial^2_{a^2} V, \tag{1}$$
$$c_t^* = \big[u'\big]^{-1}(\partial_a V), \tag{2}$$
$$\pi_t^* = -\frac{\mu - r}{\sigma^2 a_t} \frac{\partial_a V}{\partial^2_{a^2} V}, \tag{3}$$
$$V(T, a_T) = u(a_T), \tag{4}$$

where (1)-(3) hold $\forall t \in [0, T)$ and $\forall a_t > 0$ while terminal condition (4) holds $\forall a_T > 0$ at $t = T$.

2 General Numerical Scheme

Here I describe how to solve (1)-(4) in an abstract sense.

As we know $V(T, a_T)$, this is evidently a good point to start. This becomes even clearer with the discrete time Bellman Equation, where knowing the future Value Function (the Value Function at time $t + 1$) allows us to find the Value Function at the current time t. Therefore,

- (1)-(4) has to be solved backwards in time starting with the terminal condition $V(T, a_T)$.

Irrespective of the numerical scheme, we must discretise time. Thus, we start with a time grid.

- Define $t_{grid} = \{0, \Delta, 2\Delta, \ldots, T\}$ with $|t_{grid}| = N_t$.
  - There is no requirement to enforce a uniformly spaced time grid, but I have never encountered a justification for an unevenly spaced time grid.

Next, a grid for assets is required. Setting up this grid can be tricky and very nuanced which I will cover later, so for the sake of illustration in this section let's simply fix the following.

- Define $a_{grid} = \{a_{min}, \ldots, a_{max}\}$ with $|a_{grid}| = N_a$.

– This spacing can be uniform, but there are good reasons *not* to pick a uniform grid. More grid points should be placed where the Value Function has the most curvature. Various adaptive and sparse grid procedures, such as Smolyak grids, exist to set grid points optimally based on some notion of curvature.

  Additionally, the dynamic nature of the problem can be utilized: Given policies $\{c(t), \pi(t)\}_{t\in[0,T]}$, the state variable can be simulated using the SDE to find an ergodic set, i.e. a set where assets mostly lie, which is where the most grid points should be (compare Section 4). However, finite difference schemes typically use evenly spaced grids for the state variable.

The main idea now is to solve (1)-(4) by using terminal condition (4) and the time derivative in (1) to link $V(T,\cdot)$ to $V(T-\Delta,\cdot)$ in order to solve for $V(T-\Delta,\cdot)$. Once $V(T-\Delta,\cdot)$ is solved for, an iterative procedure is applied to solve all the way down to $V(T-T,\cdot)$. There are three established ways to do this outlined below.

2.1 General Framework

Re-write the PDE (1) in the following manner

$$-\partial_t V = -\rho V u\big(c_t^*\big) + \big(ra_t + \pi_t^*(\mu - r)a_t - c_t^*\big)\partial_a V + 0.5\big(\pi_t^*\sigma a_t\big)^2\partial_{a^2}^2 V$$

Assume that $V(t+\Delta,\cdot)$ is known (in the first iteration this is the terminal condition). It is common to omit the time argument and replace it with a superscript denoting the iteration step. Thus, denote $V^0(\cdot) = V(T,\cdot)$ or more generally $V^n(\cdot) = V(T-n\Delta,\cdot)$.

When solving PDEs in Physics one usually goes forward in time and the left hand side features $\partial_t V$. Here we go backwards in time and feature $-\partial_t V$ which due to the minus sign is exactly the same[1]. Denote the right hand side by the function $F\big(a, V, \partial_a V, \partial_{a^2}^2 V\big)$. There are three established procedures to solve the PDE which are outlined below. Notice that I do not cover the Crank-Nicolson method which is a mixture of the explicit and implicit method.

2.2 Explicit Method

Fix a time period t and take the backward time derivative which leads to

$$\frac{V^{n+1} - V^n}{\Delta} = F^n\big(a, V, \partial_a V, \partial_{a^2}^2 V\big).$$

More elaborately, the to be solved equation is given by

$$\rho V^n = u(c^{n,*}) + \frac{V^n - V^{n+1}}{\Delta} + \big(ra + \pi^{n,*}(\mu - r)a - c^{n,*}\big)\partial_a V^n + 0.5\big(\pi^{n,*}\sigma a\big)^2\partial_{a^2}^2 V^n.$$

It is now immediately possible to solve for the unknown V^{n+1} by simple rearrangements. However, the explicit method is many times unstable and requires very, very small time steps to circumvent this instability.

2.3 Fully Implicit Method

Using the same setting as in Section 2.2, this uses the *Backward Euler Method* while the explicit method used the *Forward Euler Method*. We thus fix a time period $t-\Delta$ and take the forward time derivative (since at $t-\Delta$ we know nothing about $V(t-\Delta)$ and must use the future known Value Function t). In short, this leads to

$$\frac{V^{n+1} - V^n}{\Delta} = F^{n+1}\big(a, V, \partial_a V, \partial_{a^2}^2 V\big)$$

[1] To make this clearer, one can use time inversion and use $\tau(t) = T - t$ which flips the negative sign of the time derivative.

More elaborately, the to be solved equation is given by

$$\rho V^{n+1} = u(c^{n+1,*}) + \frac{V^n - V^{n+1}}{\Delta} + \big(ra + \pi^{n+1,*}(\mu - r)a - c^{n+1,*}\big)\partial_a V^{n+1} + 0.5\big(\pi^{n+1,*}\sigma a\big)^2 \partial^2_{a^2} V^{n+1}.$$

Now, the the known V^n only appears once and the policies are computed from (2)-(3) as a function of the unknown V^{n+1}. Solving this equation for V^{n+1} is in my experience also mostly unstable due to the extreme non-linearities induced by the policy functions.

2.4 Semi-Implicit Method

Due to stability issues of the fully implicit method, a semi-implicit method exists. This features the following equation for the unknown V^{n+1}

$$\rho V^{n+1} = u(c^{n,*}) + \frac{V^{n+1} - V^n}{\Delta} + \big(ra + \pi^{n,*}(\mu - r)a - c^{n,*}\big)\partial_a V^{n+1} + 0.5\big(\pi^{n,*}\sigma a\big)^2 \partial^2_{a^2} V^{n+1}.$$

Hence, the policies are taken from the previous iteration, i.e. the future Value Function, thereby eliminating the troublesome source of non-linearities. This is also reminiscent of a policy-function iteration since if we know the policies we can reverse engineer the Value Function[2]. Therefore, the time step should also be small because the policy is based on $V(t+\Delta, \cdot)$ instead of $V(t, \cdot)$. As the Value Function is continuous, if the time step is small this should not be an issue and $c(t+\Delta, \cdot) \approx c(t, \cdot)$ and $\pi(t+\Delta, \cdot) \approx \pi(t, \cdot)$. Once $\{V^n\}_{n=1}^{N_t}$ have been obtained, the policies can be recomputed using (2)-(3) to verify this.

I found the semi-implict method to be the most reliable, but when solving for a time-dependent solution there is a problem when solving for $n = 1$, i.e. $V(T - \Delta, \cdot)$. If the terminal condition does not allow for an interior solution of the policies, then we must be cautious when using the semi-implicit method at $n = 1$. Since $V(T, a) = u(a)$ this is no issue in this case, but the Merton Model only requires a concave terminal condition[3], so e.g. $V(T, a) = a$ would be possible as linear functions are both concave and convex, but we cannot compute $\pi(T, a)$ using the FOC (3). Thus, we either need to use the explicit or fully-implicit scheme for $n = 1$. However, the explicit scheme can also cause troubles as the policies at the terminal condition can be tricky to define. Additionally, if $\pi(T), c(T)$ are very different from $\pi(T - \Delta), c(T - \Delta)$, which is the case for the Merton Model, this could cause errors in $V(T - \Delta, \cdot)$ which then trickle down every iteration.

3 Numerical Derivatives

Having established a general framework, the question arises how to compute the derivatives.

3.1 Finite Differences

Consider the semi-implicit method from Section 2.4. Let's consider $n = 0$ so that we need to find the function $V^1(a)$ that fulfils

$$\rho V^1(a) = u(c^{0,*}(a)) + \frac{V^1(a) - V^0(a)}{\Delta} + \big(ra + \pi^{0,*}(a)(\mu - r)a - c^{0,*}(a)\big)\partial_a V^1(a) + 0.5\big(\pi^{0,*(a)}\sigma a\big)^2 \partial^2_{a^2} V^1(a).$$

where

$$\begin{aligned} c^{0,*}(a) &= \big[u'\big]^{-1}\big(\partial_a V^0(a)\big), \\ \pi^{0,*}(a) &= -\frac{\mu - r}{\sigma^2 a}\frac{\partial_a V^0(a)}{\partial^2_{a^2} V^0(a)}, \\ V^0(a) &= u(a). \end{aligned}$$

[2] Knowing the policies, one can simply evaluate the objective function which is the Value Function.

[3] The reward function must be *strictly* concave.

Finite Differences work by defining output values of the Value Function at the gridpoints and interpolating between gridpoints to fill the holes. Thus, in finite differences we set $V^1(a_i) = v_i$ such that V^1 is a vector of length N_a and the i-th element is the function output when the i-th gridpoint is the function input. The derivatives $\partial_a V$ and $\partial^2_{a^2} V$ are then approximated by finite differences.

When computing finite differences one can e.g. use forward, backward or central differences. The problem is that at the upper bound $a_i = a_{max}$ forward differences cannot be used as no gridpoint beyond a_{max} exists. In usual PDE solutions boundary conditions must exist, i.e. we must know $V(t, a_{max})$ and $V(t, a_{min})$ *for all* time points so that we can always operate on the interior of the grid, i.e. we only need to solve for $v_2, \dots v_{N_a-1}$ so that any finite difference at any gridpoint is always available. When e.g. pricing American Calls[4] we say that after a very high price of the underlying the option will undoubtedly be exercised and for very low prices of the underlying the value of the option is zero so that we would know $v(t, a_{max})$ and $v(t, a_{min})$.

However, with usual stochastic optimal control problems we do not know any boundary value of the Value Function. To rectify the issue, the state variable is reflected at the bounds a_{min} and a_{max} (reflective barriers). Thus, at $a_i = a_{max}$ we must impose restrictions on the control policies such that the agent cannot increase his assets (if this happens to be the interior solution) and at $a_i = a_{min}$ we must restrict the control such that the agent cannot decrease his assets (if this happens to be the interior solution). Moreover, at $a_i = a_{max}$ only backward and at $a_i = a_{min}$ only forward differences can be used. Generally, whenever the drift of the state variable is positive the forward difference and else-wise the backward difference ought to be used which is called the finite upwind difference scheme.

Implementing finite differences, especially with upwinding. is much more tedious than the next two approaches which is why I chose not to do this.

3.2 Splines

The advantage of using splines is that they are easy to implement and they have known analytical derivatives. Thus, I initialise any $V^{n+1}(a)$ as a cubic spline with the *not-a-knot* boundary condition. This is called a **projection method** as the unknown function V^{n+1} is approximated with known basis functions (in this case splines).

For any spline to be uniquely defined, conditions on the boundary are required as there is no point above a_{max} and no point below a_{min} for the spline to interpolate and infer curvature. The not-a-knot boundary condition sets the first and second cubic spline at a curve end to be identical which will of course make it harder to fit a_{min} and its adjacent gridpoint and a_{max} and its adjacent gridpoint. Unfortunately, Python's Scipy library does not support quadratic splines which simply restrict the first and last cubic spline to be quadratic. It is important not to use splines where the derivatives at the boundary are zero as this must not be the case for the Value Function and can cause numerical convergence to fail.

Cubic splines are analytically and continuously twice differentiable, so that the first and second derivative of V^{n+1}_{spline} are immediately computable. Moreover, the cubic spline also computes $\partial_a V^{n+1}_{spline}(a_{max})$, $\partial^2_{a^2} V^{n+1}_{spline}(a_{max})$ and likewise for a_{min}. However, this does not entail that we can neglect the nature of the optimal control problem and still either require reflective barriers or time dependent grids. This I discuss in the Section 4.

[4] Of course, the American Call Price is equal to the European Call Price.

3.3 Chebyshev Polynomials

When using Chebyshev polynomials we set

$$V^n_{cheby}(a) = \sum_{k=0}^{N_a-1} \theta^n_k T_k(\omega(a)),$$
$$\omega(a) = \frac{a - 0.5(a_{max} + a_{min})}{0.5(a_{max} - a_{min})},$$

where T_k is the k-th Chebyshev polynomial. Hence, $V^n_{cheby}(a)$ is parametrised by $\{\theta_k\}_{k=0}^{N_a-1} \in \mathbb{R}^{N_a-1}$ and consist of known basis functions $\{T_k\}_{k=0}^{N_a-1}$.

The inputs of the Chebyshev polynomials must be scaled into $[-1, 1]$ and are constructed from the Chebyshev nodes. Chebyshev polynomials are continuously differentiable so that obtaining the derivatives of $V^n_{cheby}(a)$ is no problem. The only thing to keep in mind is the chain rule, i.e.

$$V'_{cheby}(a) = \sum_{k=0}^{N_a-1} \theta_k T'_k(\omega)\omega'(a) \qquad \text{and} \qquad V''_{cheby}(a) = \sum_{k=0}^{N_a-1} \theta_k T''_k(\omega)[\omega'(a)]^2$$

where $\omega'(a) = \frac{2}{a_{max}-a_{min}}$ is a constant. Both $\sum_{k=0}^{N_a-1} \theta_k T'_k(\omega)$ and $\sum_{k=0}^{N_a-1} \theta_k T''_k(\omega)$ are computed from the software package so that these outputs solely require to be multiplied with the constant $\omega'(a)$ and respectively $[\omega'(a)]^2$ to obtain the first and second derivative of $V^n_{cheby}(a)$.

A nice aspect of cubic splines is that this rescaling of inputs is not necessary which makes both the code neater and prevents a source of bugs or errors.

4 Grids

Stochastic optimal control problems are dynamic in nature. This means that the state variable's ergodic set changes over time. To illustrate this, let's assume Merton investors start with wealth levels $a_0 \in [0.5, 2]$. Let's assume none of them consume and only invest in the riskless asset. Then, $a_t \in [0.5e^{rt}, 2e^{rt}]$. Thus, as time progresses, the set a_t lies in will be different than the one a_0 lied in. Naturally, the Merton model is stochastic so that if investors invest into the risky asset there is a probability distribution where a_t will lie in. The ergodic set is the subset $[a_{min,t}, a_{max,t}] \subseteq [0, \infty]$ where most probability mass of $a(t)$ lies in. It is upon interpretation what *most* probability mass is. This ergodic set can be simulated with Monte Carlo by computing the evolution of the probability distribution of $a(t)$ which is given by the Kolmogorov Forward Equation which requires either the knowledge of the optimal policies or approximations of them.

4.1 Infinite Time Horizon

In an infinite time horizon the solution object is a stationary Value Function, i.e. a Value Function that does not depend on the time point. To see this, consider the following two

$$V(t_1, a_0) = \max_{\{c(s),\pi(s)\}_{s\in[t_1,\infty]}} \mathbb{E}_{t_1}\left[\int_{t_1}^{\infty} e^{-\rho(s-t_1)}u(c(s))ds\right]$$
$$V(t_2, a_0) = \max_{\{c(s),\pi(s)\}_{s\in[t_2,\infty]}} \mathbb{E}_{t_2}\left[\int_{t_2}^{\infty} e^{-\rho(s-t_2)}u(c(s))ds\right]$$

which for the same a_0 are the exact same problem. Thus, the Value Function does not depend on time (stationary) so that $\partial_t V = 0$. Note: the Merton model as outlined in Section 1 could potentially not make sense in an infinite time horizon as $a(t)$ converges to infinity, allowing infinite consumption and $U(\infty) = \infty$. Therefore, a terminal condition $\lim_{t\to\infty} e^{-\rho t}V(t, a_t) = 0$ is required. Since a_t follows a geometric Brownian Motion, $U(c_t)$ could asymptotically outgrow $e^{-\rho t}$ depending on the calibration which would make the model ill-posed.

Since a stationary solution is solved for, it is very helpful numerically if the grid a_{grid} is stationary as well. To achieve this, it must be ensured that the agent never wants to leave a_{grid} as explained in Section 3.1. In practice, the algorithm works as with a finite time horizon. An arbitrary terminal condition (Value Function)[5] is taken and iterated backwards so long until subsequent updates of the Value Function no longer change, i.e. $\partial_t V \approx 0$. In this case, even in the semi-implicit method large time steps are in order as we want to iterate the Value Function far back in time.

To obtain a useful grid of the state variable, for infinite time we cannot calibrate the grid according to the terminal condition as there is no terminal condition. Therefore, simply set some reasonable constant policy functions, simulate multiple paths of the state variable by Monte Carlo for a large time period and compute the Value Function by approximating the integral of flow utility. The integral is then simply the average given by the different paths. This way, an artificial terminal condition exists and a grid of the state variable can be set since the ergodic set, i.e. the set where the most probability mass of state variables lie, can be computed. After iterating backwards reasonably many times this approach can be re-done to refine the ergodic set. Notice that it is might be good to reflect the state variable in an interval if after the simulation the domain of the state variable is too large, which could happen since the policies are only rough guesses[6].

4.2 Finite Time Horizon

In this case, time-dependent grids should be used. As we solve system (1)-(4) backwards in time, the grid a_{grid} applies to terminal time $t = T$. However, if we go sufficiently backward in time, there is no reason to believe that the ergodic set of a_0 is similar to the one we imposed on a_T with a_{grid}. Put differently, due to the positive drift coefficient assets grow over time (from $t = 0$ to $t = T$) so that when going from $t = T$ to $t = 0$ they should decrease. Additionally, due to the diffusion coefficient, the dispersion gets larger from $t = 0$ to $t = T$, so that when going backwards in time this diffusion ought to get smaller.

One way to implement time-dependent grids would be to start with some a_{grid} at $t = T$, go a few periods backwards using the same grid and then use the Kolmogorov Forward Equation of $a(t)$ to simulate the density of the state variable forward to $t = T$. With this density we can determine the ergodic set. One problem now is that if $a^{n+1}_{grid} \neq a^n_{grid}$, then the Value Function V^n would have to potentially be extrapolated if the minimum and maximum value of the two grids don't align. With splines this is immediately possible, but this is further justification to use small time steps as extrapolation can be very imprecise and should only be done for small changes. Proceeding iteratively, a time-dependent grid for all time periods can be constructed.

5 Implementation Semi-implicit Method with Cubic Splines

Firstly, I choose the following calibration

- $\rho = 0.05$, $\gamma = 2$, $r = 0.02$, $\mu = 0.06$ and $\sigma = 0.2$, $T = 1$.

I choose the semi-implicit method as this was the most stable. Due to this, I take small time steps. The time grid is given by

$$t_{grid} = \{0, \Delta, 2\Delta, \dots, 1\} \qquad \Delta = 0.02,$$

so that there are $N_t = 50$ time steps.

[5] In this case, it is mostly the easiest to simply take a forever constant, i.e. not changing with the state variable, reasonable policy and solve or approximate the objective function of the agent. This constitutes the terminal Value Function

[6] Reflection can be achieved by changing the policies at e.g. a_{max} and a_{min} such that the drift of a is weakly positive at a_{min} and weakly negative at a_{max}.

I *do not* use time-dependent grids for the state variable as these are more involved and the time period is short enough to permit time-independent grids[7]. Consequently, for the fix asset grid I choose

- $a_{min} = 0.05$, $a_{max} = 15$, $N_a = 150$.

since the terminal Value Function exhibits a nice shape in this domain.

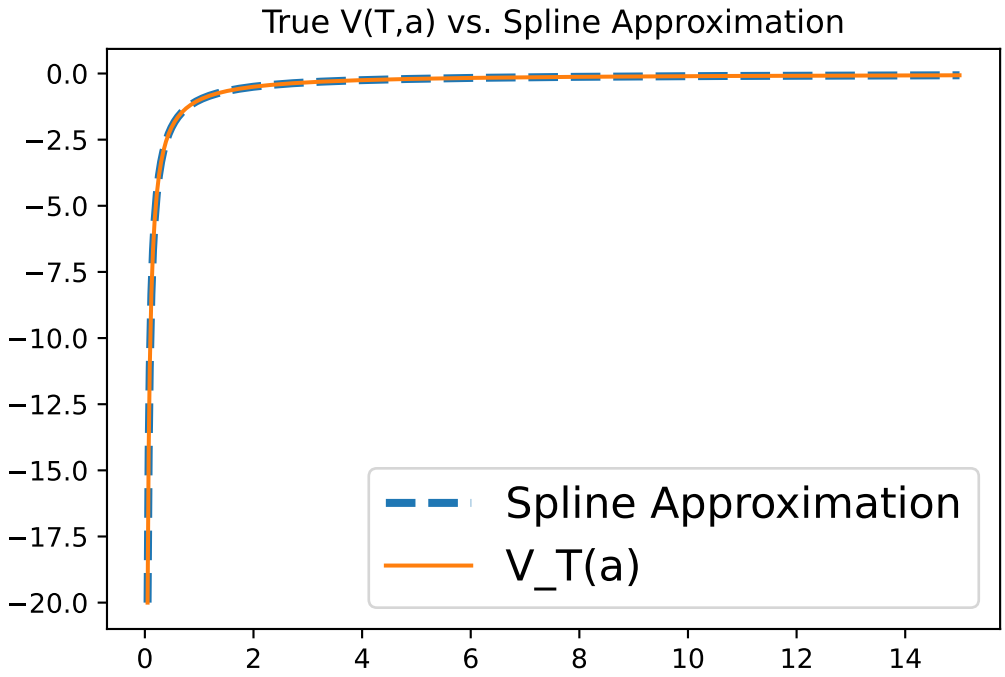
For the spacing of gridpoints I use Chebyshev nodes, i.e. the Chebyshev nodes are linearly transformed to fill the interval $[a_{min}, a_{max}]$. Notice that by using the Chebyshev nodes de-facto the asset grid is in (a_{min}, a_{max}).
I have also implemented an exponentially spaced grid which gives good approximations of the Value Function close to a_{min}, but worse approximations close to a_{max}. A linearly spaced grid also works very well apart from the edges.

I then approximate the terminal Value Function with a spline

$$V^0_{spline} = spline\Big(a_{grid}, u(a_{grid})\Big).$$

This is not necessary as we know the terminal Value Function, but it streamlines the code. Figure 1 shows the approximation which is virtually indistinguishable from the actual function. The numerical

Figure 1



algorithm works as follows

1. Initialise $V^1_{s|y} = spline\Big(a_{grid}, \boldsymbol{y}\Big)$ with $\boldsymbol{y} = V_T(a_{grid})$.
    - The vector $\boldsymbol{y}$ is of size N_a.
    - The subscript $s|\boldsymbol{y}$ denotes a spline fitted to the values given in the vector $\boldsymbol{y}$.
    - As I keep the asset grid fix, there is no need to introduce it into the notation so I leave out the collocation points.

[7] In other words, since I set a_{grid} as the interesting points to consider the terminal Value Function, not going too far back in time implies that these same points are considered to be the interesting points of $V(0, \cdot)$.

2. Solve the following root-finding problem where $\boldsymbol{y}$ is the running variable

$$
\begin{aligned}
0 = u(c^{0,*}(a)) + &\frac{V_s^0(a) - V_{s|\boldsymbol{y}}^1(a)}{\Delta} \\
&+ \big(ra + \pi^{0,*}(a)(\mu - r)a - c^{0,*}(a)\big)\partial_a V_{s|\boldsymbol{y}}^1(a) \\
&+ 0.5\big(\pi^{0,*(a)}\sigma a\big)^2 \partial_{a^2}^2 V_{s|\boldsymbol{y}}^1(a) - \rho V_{s|\boldsymbol{y}}^1(a) \qquad \forall a \in a_{grid}.
\end{aligned} \tag{5}
$$

The loss on the right hand side is a vector of length N_a as the loss on the right hand side is evaluated at each asset gridpoint in a_{grid}. Notice that the policies are computed from (2)-(3) as a function of V^0 such that they remain constant irrespective of how $\boldsymbol{y}$ is changed. Changing $\boldsymbol{y}$ at the collocation points a_{grid} changes the spline and thus the function V^1 which is how we find the new Value Function that sets the loss to zero at the collocation points.

- I use Scipy's hybrid broyden algorithm, but any root-finding algorithm can be used.
  - Depending on a_{min} and a_{max} and the curvature of the Value Function close to these values, the numerical accuracy must be forced to be much higher. The reason is that the Value Function is very steep when a is close to a_{min} so that even the slightest numerical error in the gradient and the Hessian can blow up. The same applies to a close to a_{max} where the Value Function is extremely flat so that even the slightest slope can cause the policies to blow up. Setting more gridpoints here cannot completely remedy the problem if the numerical accuracy is not increased.
  - For simplicity, I simply keep the same ordinary tolerance level imposed by Scipy's library.
- It is possible to solve the PDE as a minimisation problem. Let $r_{s|\boldsymbol{y}}(a)$ denote the right hand side of (5) and find $\boldsymbol{y}$ that solves

$$
\min_{\boldsymbol{y}} \int_{a_{min}}^{a_{max}} r_{s|\boldsymbol{y}}^2(a) da
$$

where the integral (MSE) must be discretised at points which can be any number of points, including the collocation points, on a_{grid}.

  - This can e.g. be solved with a BFGS-algorithm.
  - In my experience collocation with root-finding is more stable compared to minimisation, but this needn't necessarily be the general case so that minimisation is worth exploring.

3. Once $V_{s|\boldsymbol{y}}^1(a)$ has been obtained, repeat step 2 iteratively until all $\{V_s^n\}_{n=0}^{N_t}$ have been obtained.

In general, the derivative of $r_{s|\boldsymbol{y}}$ with respect to $\boldsymbol{y}$ can be computed with automatic differentiation if the spline is coded with e.g. `jax`. Unfortunately, while I was writing this document, Scipy did not support automatic differentiation with autograd or jax.

5.1 Results

Here I show the results of the numerical approximation.

I start with the comparison of the Value Function at $t = 0$ and all its derivatives in Figure 2. As can be seen, the numerical scheme can fit the Value Function extremely well aside from values close to a_{min}. However, a_{min} is chosen to be very small where the Value Function is very steep so that the cubic splines find it hart to fit the Value Function there.

Therefore, a truncation of a_{min} at 0.5 is depicted in Figure 3. As can be seen, the fit is extremely good. Next, the evolution of the numerically computed Value Function is displayed in Figure 4. As can be seen, the Value Function gets steeper and steeper the further we go back in time. Eventually, the cubic spline will not be able to capture this steepness anymore and the algorithm breaks. Lastly,

Figure 2

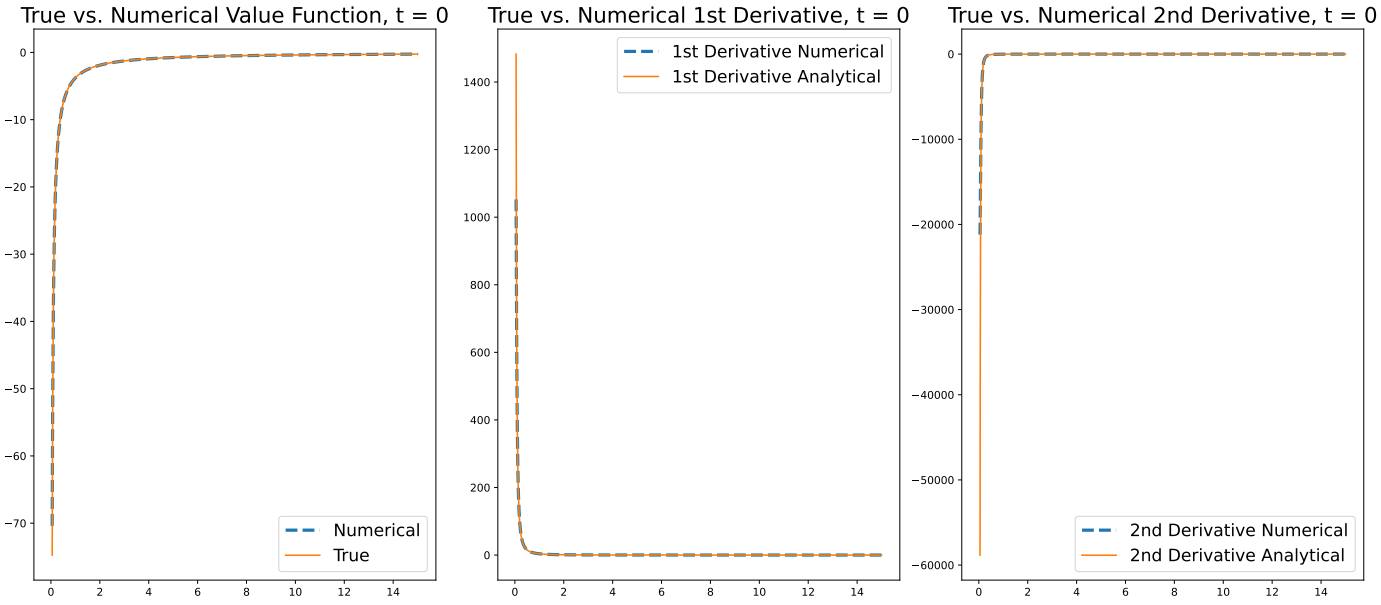


Figure 3

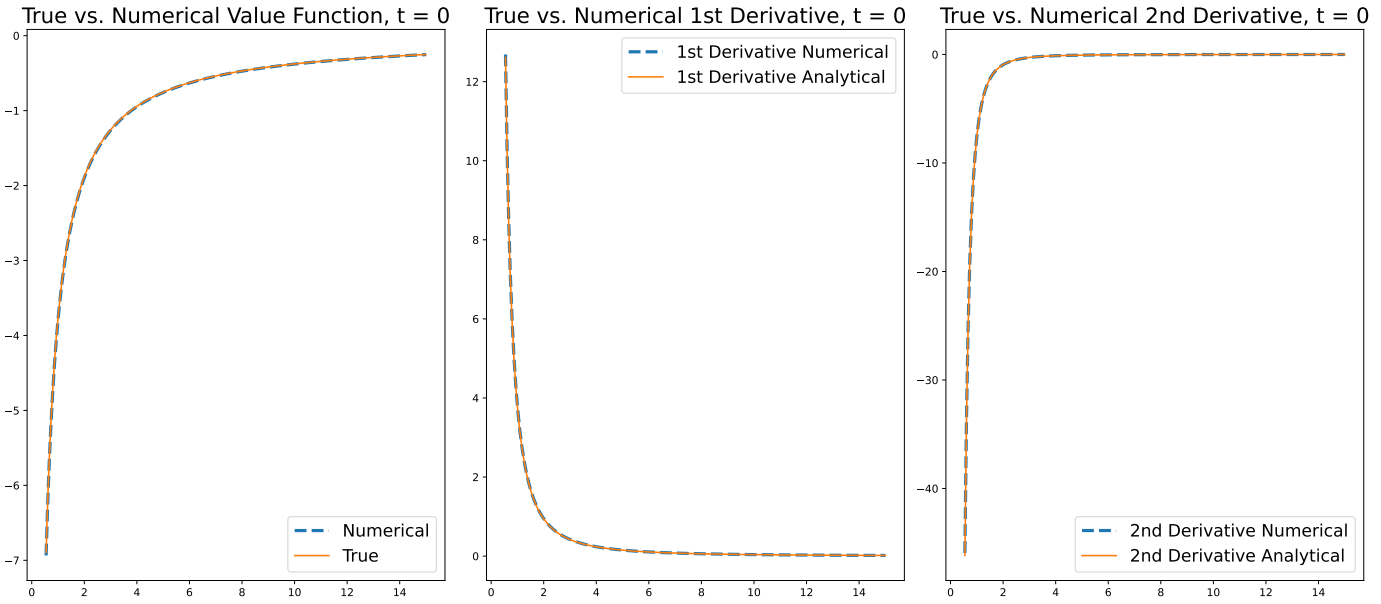
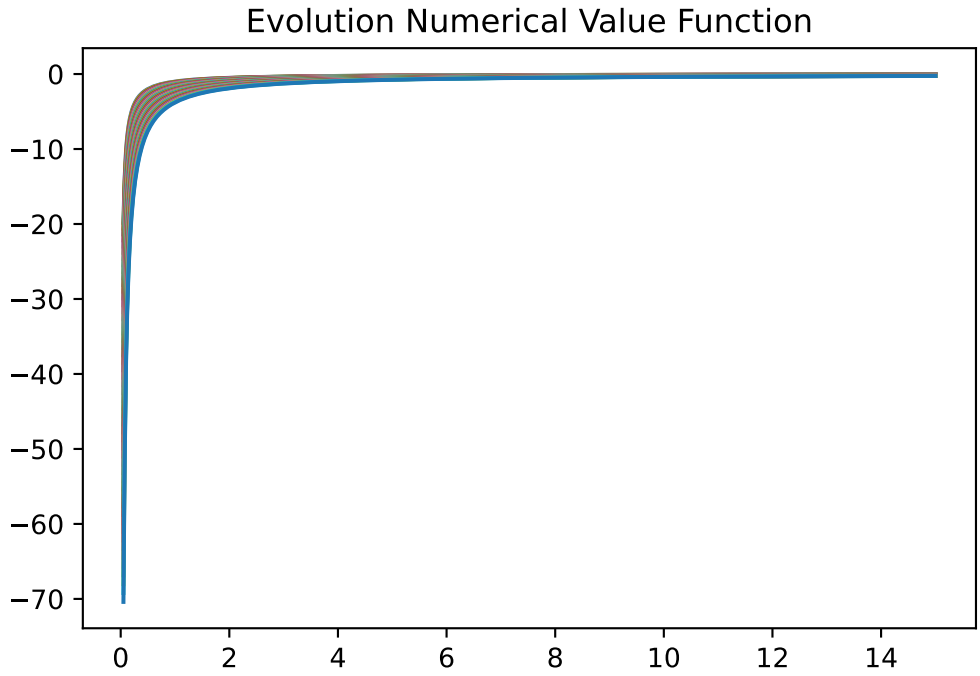


Figure 4



the policy functions must be depicted which is displayed in Figure 5. As can be seen, the fit of the policy functions is not as good as the Value Function. This is particularly true for $\pi(t)$ which is harder to fit as it depends both on the gradient and the Hessian. The dependence on the Hessian is even highly non-linear so that even the slightest numerical error can cause vast discrepancies to the analytical solution.

Since $a_{min} = 0.05$ is very small and the Value Function is very explosive at these values, a truncated plot is displayed in Figure 6. Notice the scaling for the plot of π. The relative error is at most 0.3% which is an extremely good fit.

6 Implementation (Semi)-Implicit Method with Chebyshev

I use the same calibration and time grid as in Section 5. For the asset grid I compute the first N_a Chebyshev nodes $\{\omega_i\}_{i=1}^{N_a} \in [-1, 1]$ so that the asset grid is given by
$a_i = 0.5(a_{max} + a_{min}) + 0.5(a_{max} - a_{min})\omega_i$. I further use the following calibration

- $a_{min} = 0.8$, $a_{max} = 10$, $N_a = 30$

for the simple reason that the Chebyshev polynomials perform worse than the cubic splines which forced me to reduce the domain of the state variable in order to prevent the algorithm from failing. Although $N_a = 30$ can be increased, the algorithm gets much slower much more quickly than for cubic splines and even fails eventually. I explain below why.

Chebyshev polynomials perform worse than cubic splines for three main reasons:

1. Unlike with splines, which are more flexible and allow for locally bounded changes[8], changing one θ_k has global effects. Thus, changing one θ_k changes the entire function $V^n_{cheby}(a)$ and thus every entry of the PDE loss. In other words, the Jacobian of the PDE loss with respect to any θ^n_k is not sparse. Thus, when increasing the number of gridpoints N_a can result in issues

[8]See the function `compute_jacobian` in the Code which outputs a sparse matrix for the cubic splines.

Figure 5

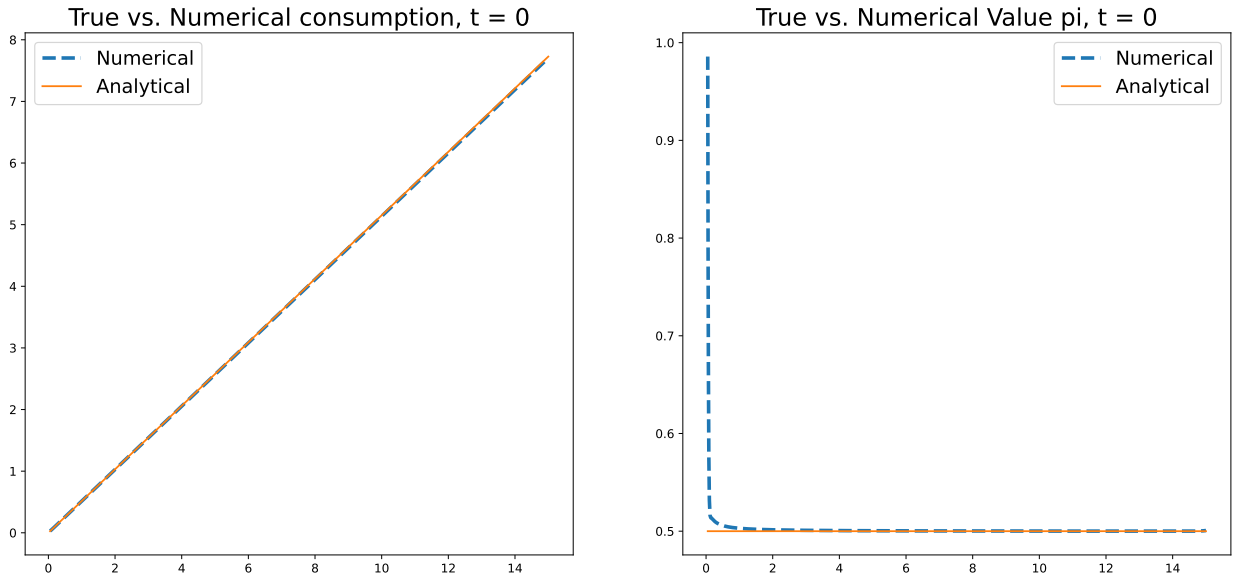
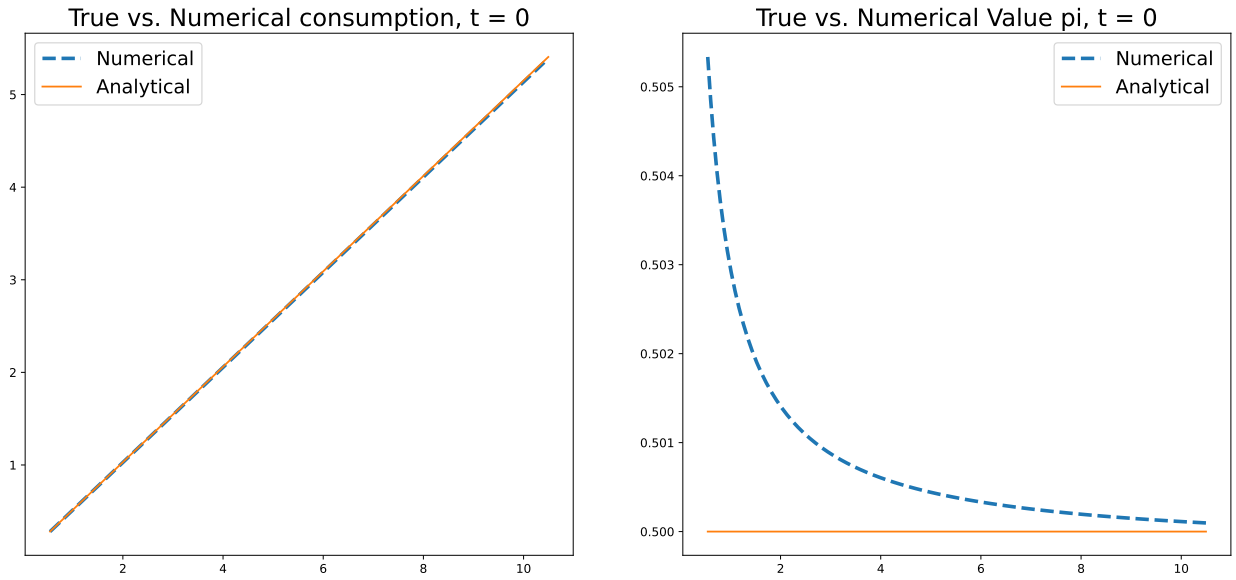


Figure 6



because under collocation more Chebyshev polynomials are used which decreased stability as the Jacobian is not sparse.

2. Chebyshev polynomials oscillate and it is harder to force them to obey monotonicity[9]. Monotonicity of the Value Function is absolutely crucial as consumption depends on the gradient of the Value Function and this gradient must be positive, else-wise the model breaks down.

3. Chebyshev polynomials are simply less flexible than cubic splines and thus provide fewer degree of freedoms. **If** the Value Function has a nice curvature (not extremely steep and not extremely flat anywhere), then a sparse parametrisation of Chebyshev polynomials can probably achieve results as good as a highly parametrised cubic spline. For a one-dimensional state variable this is probably never necessary, but it could provide computational speed gains in higher dimensions. However, if the Value Function is so steep as here, the Chebyshev polynomials cannot match the steepness from minimising the PDE loss.

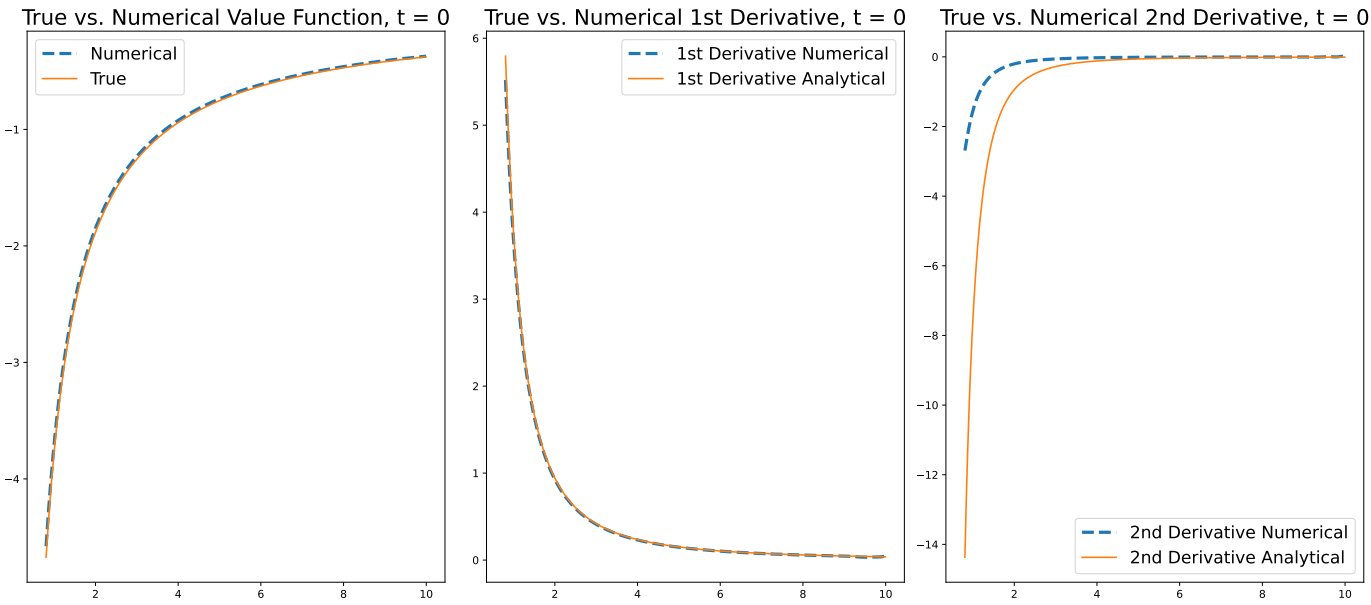
On top of that, as with splines, I have found that minimising the PDE loss causes the algorithm to fail as $V^n_{cheby}(a)$ then fails to adhere to monotonicity quickly.

Despite their worse performance, Chebyshev polynomials can solve the fully implicit method easily as opposed to the cubic splines where I cannot even get one iteration to work in the fully implicit method[10]. However, the results of the Chebyshev polynomials under the fully implicit method are almost the same as for the semi implicit method and are implemented in the code.

6.1 Results

Here I show the approximation of the Value Function under the semi-implicit method. Figure 7 shows the fit of the Value Function. As can be seen, at the lower bound the Chebyshev polynomial cannot fit the second derivative well because it is too steep. Do keep in mind that a_{min} was increased substantially compared to the cubic spline so that this is not a good result. Figure 8 displays the

Figure 7

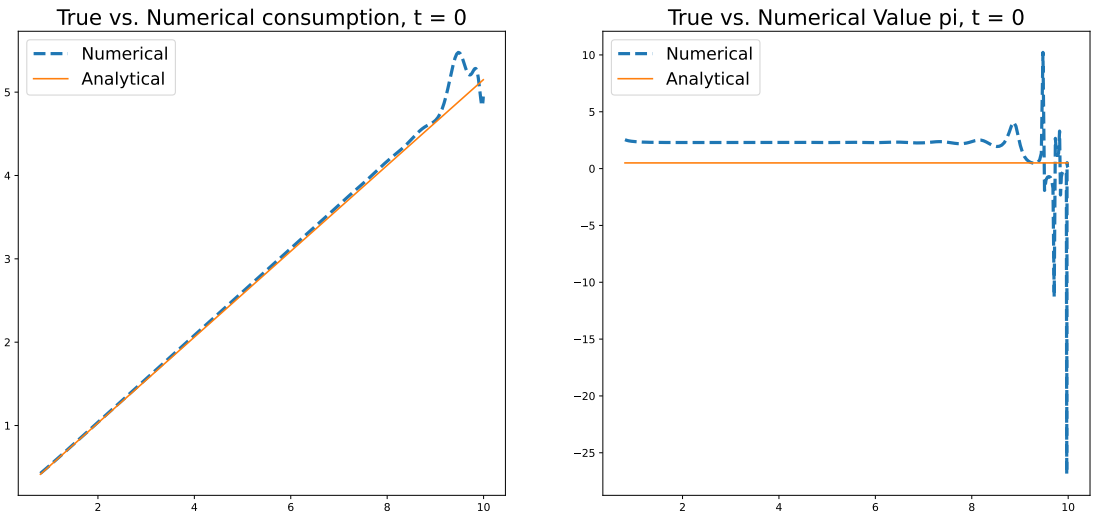


consequences since the policies are not well approximated. Despite the clear errors at the upper bound, even in the interior the grid the policy π is not even on average where it should be. Its average is 2.5 which is far higher than the required 0.5.

[9] In fact, even when fitting the terminal condition, then depending on the grid and the number of nodes, the Chebyshev Polynomial $V^0_{cheby}(a)$ is not necessarily monotonic.

[10] Probably when coding the Newton-Method by hand would this work or at least the source of the problem would become more evident.

Figure 8



7 Outlook

For higher dimensional problems, i.e. a Value Function with 2 inputs, splines and Chebyshev polynomials can still be used. The Chebyshev polynomial would then take the form

$$V(a,b) = \sum_{k=0}^{N_a-1} \sum_{j=0}^{N_b-1} \theta_{k,j} T_k\big(\omega(a)\big) T_j\big(\omega(b)\big).$$

Unfortunately, in Python such a Chebyshev polynomial would mostly have to be built by hand.

For inputs of dimension 4 and more it is probably good to use a Neural Network such that $V^n(a,b,c,d) = NN^n(a,b,c,d)$ and the semi-implicit method can be used as was done before since the derivatives of the Neural Network can be computed with automatic differentiation. Having a separate neural network per time period will provide more flexibility and help the individual Neural Net to fit the Value Function it has to fit. This will be memory intensive, but usually memory is not a bottleneck and at the very least it is easy to improve memory capacity.

Generally, the question arises how to know whether a numerical approximation is good without the knowledge of the analytical solution. This is impossible to tell ex-ante, but a few points can be made.

1. The numerical error of the root-finding or a minimisation problem should be small.
2. If possible, i.e. if the dimension of the state space is low enough, plot your policies or look at contour plots for a 3 dimensional input space. Policy functions in Econ & Finance models are usually monotonic and smooth which should also be the case of the numerical solution. If not, something likely went wrong and the basis function cannot fit the PDE.
3. Change the baseline parameters a bit. If the numerical solution looks vastly different or fails to converge, the baseline numerical solution is not stable and it could very well be a poor approximation.

As general advice for (stochastic) optimal control problems, it is imperative to be on a domain of the Value Function where it has a nice curvature and is not extremely steep or completely flat. On top of that, if your optimal control problem has no control policy for the diffusion, the controls will only dependent on the gradient which means that the problem is easier to solve as fitting higher order of the derivatives becomes exponentially harder. Hence, if you don't need to control policy in the diffusion, then leave it out.