Numerical Solution of the Continuous-time Bellman Equation with Poisson Process

Lu Jialei

6729181

September 7, 2022

Contents

Part I

Introduction

As a widely used tool in economics, dynamic programming is bittersweet to economists. Economists love dynamic programming because it transfers overall optimization problems into recursive ones, providing fruitful tools, theorems, and methodologies. Besides, it absorbs the calculus of variations and the optimal control theory and extends the boundary of problems that can be solved. Nevertheless, every coin has two sides. Dynamic programming has its shortcomings. Even though the target functions and restrictions have good characteristics, an analytical solution may not exist. In most cases, the numerical solution, rather than the analytical solution, is the goal of economists. To get more accurate numerical results and solve variental problems, economists and mathematicians develop different approximation methods, which are the focus of this paper.

0.1 Framework: Exact and Approximated Numerical Solutions

Both analytical and numerical methods focus on solving the Bellman Equation. The analytical method deals with the first-order condition of the Bellman Equation, solving it recursively and backward. Analogously, the numerical methods iterate the value or policy functions until reaching the optimality. During the process, approximations are everywhere: discretization, the function approximation, and the derivative approximation. Discretization transfers the continuum space into one with finite states, enabling the computation of the value function and the stochastic process. The function approximation is a well-developed field, including three subfields: the finite difference method, the projection method, and the perturbation method. The derivative approximation follows the function approximation. Different function approximations lead to corresponding derivative approximation.

0.2 Outlines

This paper provides several examples, the analytical solutions, and the numerical solutions to illustrate the dynamic programming under varying situations and highlights the corresponding numerical methods in detail. Part II provides basic framework. Among which, Section 1 demonstrates a deterministic optimal saving model. The discrete-time problem is a stepping stone, while the corresponding continuous-time one is the focus. Next, the paper introduces the numerical methods. After dealing with the deterministic situation, Sections 3 and 4 explore a stochastic example and provide the corresponding numerical results in Section 5. In Part III, XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Part II

Basic Framework and Examples:

1 A Deterministic Optimal Saving Example

1.1 The Setup

A household maximizes its utility by consuming. The utility function follows a CRRA form. The discount rate of the household is ρ.

$$U(t)=\sum_{t}^{\infty}\rho^{-[\tau-t]}u(c(\tau))=\sum_{t}^{\infty}\rho^{-[\tau-t]}\frac{c(\tau)^{1-\sigma}-1}{1-\sigma} \tag{1}$$

The corresponding utility function under the continuous-time situation is as follows:

$$U(t)=\int_{t}^{\infty}e^{-\rho[\tau-t]}u(c(\tau))=\int_{t}^{\infty}e^{-\rho[\tau-t]}\frac{c(\tau)^{1-\sigma}-1}{1-\sigma}d\tau \tag{2}$$

The household's income consists of two parts: the interest income $r \cdot a(\tau)$ and the labor income w at the moment τ. Here $a(\tau)$ is the household's volume of wealth. Meanwhile, the household expends $c(\tau)$ to gain utility and saves the remaining. Hence, the motion of the assets follows:

$$\Delta a = \dot{a}(\tau) = r \cdot a(\tau) + w - c(\tau) \tag{3}$$

1.2 The Exact Solution

To judge the numerical solutions' accuracy, the benchmark is the exact analytical solution. Two methods are widely used in calculating the exact solutions of the optimal problem, the Lagrangian and the Hamiltonian.

1.2.1 The Lagrangian

The Lagrangian method is based on the Lagrangian function with Lagrangian multipliers. In general, the number of multipliers are the same as the number of constraints. However, it is impossible to wrtie down infinite number of multipliers. Fortunately, in this example, the budget constraint in each period has the same effect with the lifetime budget constraint, which takes the following form:

$$\int_t^\infty e^{-r[\tau-t]} c(\tau) d\tau = a(t) + \int_t^\infty e^{-r[\tau-t]} w d\tau$$

The right-hand side represents the lifetime spending and the left-hand side tells the lifetime resources, which contains the original asset and the discounted value of the wages.

With λ as the multiplier, the Lagrangian formula takes the following form:

$$\mathcal{L} = \int_t^\infty e^{-\rho[\tau-t]} u(c(\tau)) d\tau - \lambda \left[\int_t^\infty e^{-r[\tau-t]} c(\tau) d\tau - a(t) - \int_t^\infty e^{-r[\tau-t]} w d\tau \right]$$

The first-order derivative with respect to c gives out:

$$u'(c(\tau)) = \lambda e^{-(r-\rho)(\tau-t)}$$

The Keynes-Ramsey rule comes from taking derivative of the above formula's log-form with respect to time:

$$-\frac{u''(c(\tau))}{u'(c(\tau))} \cdot \dot{c}(\tau) = r - \rho \tag{4}$$

Pluging in the CRRA utility function gives out:

$$\frac{\dot{c(\tau)}}{c(\tau)} = \frac{r-\rho}{\sigma}$$

This seperable first-order ODE has an explicit solution:

$$c(\tau) = c_0 e^{\frac{r-\rho}{\sigma}\tau} \tag{5}$$

where, c_0 is the consumption in period 0 and given by some boundary condition.

1.2.2 Hamiltonian

Based on the Pontrygin's maximum theorem, the Hamiltonian defines a present-value Hamiltonian function with $\lambda(\tau)$ as an adjoint equation when discount factor exists.

$$\max_{c(\tau),a(\tau),\lambda(\tau)} \mathcal{H} = u(c(\tau)) + \lambda(\tau) \cdot \dot{a}(\tau)$$

The first-order condition with respect to $c(\tau)$ gives out:

$$\frac{\partial \mathcal{H}}{\partial c(\tau)} = u'(c(\tau)) - \lambda(\tau) = 0$$

And the first-order conditions with respect to $a(\tau)$ and $\lambda(\tau)$ form the Hamiltonian system:

$$\frac{\partial \mathcal{H}}{\partial a(\tau)} = \lambda(\tau) \cdot r = -\dot{\lambda}(\tau) + \rho\lambda\tau$$
$$\frac{\partial \mathcal{H}}{\partial \lambda(\tau)} = \dot{a}(\tau) = r \cdot a(\tau) + w - c(\tau)$$

Solving the above conditions gives out the Keynes-Ramsey rule again.

1.3 The Bellman Equation & Its Solution

Dynamic optimization problems require functionals, the motions of assets or consumptions, as the solutions. However, solving a functional at one time is complex. Complexity leads to many variants of Lagrangian and Hamiltonian to deal with specific settings. Luckily, the principle of optimization affords a novel way: instead of finding a path of consumption, solving an equivalent recursive problem step by step and generating a policy function.

The core of this method is to find and solve the Bellman Equation. The first step is to classify the variables into two categories: the state and the control variables. In this example, the wealth level a is the state variable, and the consumption level c is the control variable. The second step generates the value function. To be simple, I normalize t as 0 and write the value functions as follows:

$$V_t(a) = \sum_0^t \rho^\tau u(c_\tau) = \sum_0^t \rho^\tau \frac{c_\tau^{1-\sigma} - 1}{1-\sigma}$$

Third, the Bellman Equation is the recursive form of the value function:

$$V_t(a) = \max_a \{u(c_\tau) + \rho V_{t+1}(a)\} = \max \left\{ \frac{c_\tau^{1-\sigma} - 1}{1-\sigma} + \rho V_{t+1}(a) \right\}$$

The transfer makes the recursion obvious by decomposing the value function into two parts: the utility in Period t and the discounted cost-to-go.

Parellel to the discrete-time situation, the value function under the continuous-time situation has a similar form.

$$V(a) = \max_c \int_0^\infty e^{\rho\tau} u(c) = \max_c \int_0^\infty e^{\rho\tau} \frac{c^{1-\sigma} - 1}{1-\sigma} d\tau \tag{6}$$

However, the transfer is not straightforward since the discount rate convergence to zero as time goes infinitesimal. To generate a meaningful equation. Consider a small time period Δ, the value function becomes:

$$\begin{aligned} V(a(\tau)) &= \max_{c(\tau)} \left[\frac{c(\tau)^{1-\sigma} - 1}{1-\sigma} + e^{-\rho\Delta} V(a(\tau + \Delta)) \right] \\ &= \max_{c(\tau)} \left[\frac{c(\tau)^{1-\sigma} - 1}{1-\sigma} + (1 - \rho\Delta) V(a(\tau + \Delta)) \right] \end{aligned}$$

This family of functions (with respect to Δ) has one limit, the Bellman Equation under the continuous-time setting:

$$\rho V(a) = \max_c \left[u(c(\tau)) + \dot{a}\frac{dV}{da} \right] = \max_c \left[\frac{c^{1-\sigma} - 1}{1-\sigma} + \dot{a}\frac{dV}{da} \right] \tag{7}$$

This Bellman Equation provides the policy function, which contains the optimal solution of the original sequential optimal problem. If the following condition meets, the policy function and the original optimal solution are equivalent:

$$\lim_{n \to \infty} \beta^n v(a_n) = 0, \quad \{a_1, a_2, \cdots, a_n, \cdots\} \in \mathit{Feasible\ set\ of\ } a_0$$

The policy function comes from the envelop theorem and the first-order derivative with respect to consumption.

$$\rho V'(a) = rV'(a) + \dot{a}V''(a)$$
$$u'(c) - V'(a) = 0$$

Solving the equations gives out the Keynes-Ramsey rule.

The analytical results above reply on the well-defined problem. In general cases, unable to have an analytical solution leads economists to solving Bellman Equations numerically.

2 Approximated Numerical Methods

2.1 Theoretical Background

Different methods for solving a dynamic programming problem have a common element: searching for the fixed point in the value function space by recursive mapping and generating a policy function as a by-product. This process generates two questions to be answered prior to any executions.

1. Does the algorithm converge to a unique solution?

2. How large is the error?

2.1.1 Existence of the Solution

The mathematic foundation to answer the first question is the Banach Fixed Point Theorem:

Theorem 2.1 (The Banach Fixed Point Theorem) *Let (F, d) be the Banach Space with metric d, and $T : F \to F$ be a contraction mapping on F. i.e., $\forall f, g \in F, d(Tf, Tg) < d(f, g)$. Then there exists a* ***unique*** *fixed point $f^* \in F$, that $Tf^* = f^*$*

This fixed point theorem ensures convergence and a unique solution when the two conditions meet. The first condition requires a complete metric space. Since the utility function is continuous on $(0, \infty)$, the value function is continuous, and the continuous function space is complete. The second condition requires the mapping to contract uniformly. Examing the uniform contraction, in practice, equals a sufficient but easy-to-check theorem, the Blackwell Theorem:

Theorem 2.2 (The Blackwell Theorem) *Let $f, g \in F : \mathbf{R}^l \to \mathbf{R}$ be two bounded functions. $T : F \to F$ is a contraction mapping if the following two conditions meet:*
1) (Monotonicity) if $f(x) \leq g(x), \forall x \in \mathbf{R}^l$ then we have $Tf(x) \leq Tg(x), \forall x \in \mathbf{R}^l$
2) (Discounting) There exists $\delta \in (0, 1)$, such that

$$T(f + b)(x) \leq Tf(x) + \delta b, \forall f \in F, b \geq 0, \forall x \in X$$

Both conditions meet in this optimal saving problem.

2.1.2 Theoretical Error

To illustrate how the discount rate pins down the upper-bound error, I introduce the μ-transfer operator and the Bellman operator. Fix a policy μ_c, the μ-transfer operator, T_μ, maps the value function in Step n, $V^{(n)}$, to $V^{(n+1)}$, which is implicitly defined by:

$$T_\mu : V^{(n)} \to V^{(n+1)} : \rho V^{(n+1)}(a) = \frac{c^{1-\sigma}-1}{1-\sigma} + \dot{a}\frac{dV^{(n+1)}}{da}$$

Among the μ-transfer, the Bellman operator maps the value function to its optimality, maximizes the right-hand side of the last equation, and maps any intial value function to the fixed point recursively:

$$T^* : V^{(n)} \to V^{(n+1)} : \rho V^{(n+1)}(a) = \max_c \left[\frac{c^{1-\sigma}-1}{1-\sigma} + \dot{a}\frac{dV^{(n+1)}}{da}\right]$$

$$(T^*)^n V_0 \to V^*, \quad as\ n \to \infty$$

With the help of defined operators, the error has the following upper-bound:

Theorem 2.3 (The Upper-bound Error) *Let T^* and T_μ be the Bellman operator and the μ-transfer operator. Let the $||\cdot||$ be the sup-norm.*

$$||V|| = \sup_{a\in A} |V(a)|$$

Since all μ-transfers are uniform contractive. The upper-bound errors are:

$$||V^* - V|| \leq \frac{1}{1-\rho}||T^*V - V||$$
$$||V_\mu - V|| \leq \frac{1}{1-\rho}||T_\mu V - V||$$

Here, V_μ is the fixed point of the value functon space under the T_μ mapping.

2.2 The Value and Policy Function Iteration

The fixed point theorem guarantees the existence and uniqueness of the solution; the theoretical upper-bound error measures how well the numerical solution is; the function iteration provides the framework for detecting the solution. There are two main methods: the value function iteration and the policy function iteration.

The value function iteration has three steps. The first step is the initialization step. The algorithm requires values of the model's parameters, the forms of formulas, a grid of the asset (the state variable) space, and the initial guess of the value function as inputs. The second step is the update and iteration step. After traversing the product space of the consumptions and assets and calculating the corresponding values, the algorithm picks the maximum as an updated value function. The third step is the termination check step. The output is optimality as long as the norm between the successive value functions is small enough.

Algorithm 1 Value Function Iteration of The Optimal Saving Problem

```
Input: parameters, utility function, motion function, original value function,
    state variables, actions
Output: final value function, policy function
1: Initialize the parameter $\sigma$,$\rho$, the threshold value $\gamma$
2: Initialize the utility function, the motion equation $\dot{a}$, the original value func-
    tion $V(\cdot)$
3: Initialize the grid of the state variable $a$
4: while the norm between the original and the updated value functions is larger
    than $\gamma$ do
5:     Update the value function by Function 7
6:     Set the function which maximize the value function as the updated policy
    function
7: end while
       return the final value function, the policy function
```

Although the value function iteration is straightforward to apply, it converges slowly, bounded by the discount rate ρ. The closer ρ is to 1, the more iterations the algorithm requires.

Alternatively, the policy function iteration requires fewer iterations to update the value function by solving the policy function directly. If we derive Function 7 concerning the consumption on both sides, we have the first-order condition:

$$u'(c) = c^{-\sigma} = \frac{dV}{da} \tag{8}$$

Given the value function, the solution of Function 8 is the corresponding policy function. With the new policy function on the right-hand side of Function 7, the value function updates.

Algorithm 2 Policy Function Iteration of The Optimal Saving Problem

```
Input: parameters, utility function, motion function, original value function,
    state variables, original policy function
Output: final value function, policy function
1: Initialize the parameter $\sigma$,$\rho$, the threshold value $\gamma$
2: Initialize the utility function, the motion equation $\dot{a}$, the original value function $V(\cdot)$
3: Initialize the grid of the state variable $a$
4: while the norm between the original and the updated policy functions is larger than $\gamma$ do
5:     Update the policy function by solving Function 8
6:     Update the value function given the updated policy function
7: end while
       return the final value function, the policy function
```

The above process includes the information of the first-order condition into the calculation and decreases the number of iterations. However, there is no free lunch. The policy function iteration spends more time on each step to solve Equation 8.

Since the value function iteration spends less time on each step and the policy function iteration needs fewer iterations. A 'hybrid iteration,' which updates the policy and the value function by turns, generates.

2.3 The Function Approximation Methods

Function iterations are proper tools for solving the discrete-time dynamic problems but are powerless before the continuous-time ones. The ways to transfer the continuous-time problems to corresponding discrete-time ones are three approximation methods. The finite difference method acts on the derivative term di-

Algorithm 3 Hybid Iteration of The Optimal Saving Problem

```
Input: parameters, utility function, motion function, original value function,
    state variables, actions
Output: final value function, policy function
 1: Initialize the parameter $\sigma$,$\rho$, the threshold value $\gamma$
 2: Initialize the utility function, the motion equation $\dot{a}$, the original value func-
    tion $V(\cdot)$
 3: Initialize the grid of the state variable $a$
 4: Innitialize the inner iteration number $M$.
 5: while the norm between the original and the updated value functions is larger
    than $\gamma$ do
 6:     Based on the latest value function, calculate the corresponding policy
    function by sovling Function 8
 7:     Perform the 'Value Function Iteration M' times
 8: end while
        return the final value function, the policy function
```

rectly, constructing the derivative by the upwind method. The projection method uses interpolation to approximate the value function and calculates its derivative recursively. The perturbation method focuses on the equation of optimization. Combining the Taylor Series or the Padé formula with the implicit formula theorem, this method searches for the optimal value function through symbolic and numerical computation.

2.3.1 The Finite Difference Method

The finite difference method revolves around the derivative of the value functon $\frac{dV}{da}$. As its name suggests, it uses the tangent value of the neighboring points on the state space's grid to approximate the derivative: the forward difference, $V'_{i,F} = \frac{V_{i+1}-V_i}{\Delta a}$, the backward difference, $V'_{i,B} = \frac{V_i - V_{i-1}}{\Delta a}$, and the steady-state derivative, $V'_{ss} = \{u'(c)|\dot{a} = 0\}$. Then it applies the upwind scheme for combining the three approximations:

$$\frac{dV_i}{da} = \frac{V_{i+1} - V_i}{\Delta a}\mathbf{1}_{\{\dot{a}>0\}} + \frac{V_i - V_{i-1}}{\Delta a}\mathbf{1}_{\{\dot{a}<0\}} + V'_{ss}\mathbf{1}_{\{\dot{a}=0\}} \tag{9}$$

Here, $\mathbf{1}_{\{\cdot\}}$ is the indicator function. Function 9 is easy to understand. Since the value function is an increasing function of asset a. If $\dot{a}$ is larger than 0, the forward difference is a good candidate for the derivative. On the contrary, if $\dot{a}$ is smaller than 0, the backward difference is suitable to approximate the derivative. When $\dot{a}$ is 0, the steady-state derivative enters the function.

Given the upwind derivative, the Bellman Equation is as follows:

$$\begin{aligned}\rho V_i &= \frac{c_i^{1-\sigma} - 1}{1-\sigma} + \frac{V_{i+1} - V_i}{\Delta a}\dot{a}^+_{i,F} + \frac{V_i - V_{i-1}}{\Delta a}\dot{a}^-_{i,B} \\ &= \frac{c_i^{1-\sigma} - 1}{1-\sigma} + [\begin{matrix}\frac{-\dot{a}^-_{i,B}}{\Delta a} & \frac{\dot{a}^-_{i,B} - \dot{a}^+_{i,F}}{\Delta a} & \frac{\dot{a}^+_{i,F}}{\Delta a}\end{matrix}]\begin{bmatrix} V_{i-1} \\ V_i \\ V_{i+1}\end{bmatrix}\end{aligned}$$

Here c_i is the solution of the first-order condition (Function 8) with the upwind derivative as the approximation. $\dot{a}^+_i \equiv max(\dot{a}_i, 0)$, $\dot{a}^-_i \equiv min(\dot{a}_i, 0)$, while $\dot{a}_{i,F}$ and $\dot{a}_{i,B}$ come from the asset motion function given c_i.

Based on the approximated Bellman Equation, the update formula is as follows:

$$\frac{V_i^{n+1} - V_i^n}{\Delta} = \frac{(c_i^n)^{1-\sigma} - 1}{1-\sigma} + \frac{V_{i+1}^{n+1} - V_i^{n+1}}{\Delta a}\dot{a}^{n}{}^{+}_{i,F} + \frac{V_i^{n+1} - V_{i-1}^{n+1}}{\Delta a}\dot{a}^{n}{}^{-}_{i,B} - \rho V_i^{n+1} \tag{10}$$

Here n represents the n^{th} iteration, and Δ is the step. The intuition is simple. When $V^n = V^\star$, the right-hand side equals to 0 by the Bellman Equation; there is no update. Else, the value function evolves to a new one by some proportion of the difference.

Algorithm 4 Finite Difference Method of The Optimal Saving Problem

```
Input: parameters, utility function, motion function, original value function,
    state variables, original policy function
Output: final value function, policy function
1: Initialize the parameter $\sigma$,$\rho$, the threshold value $\gamma$
2: Initialize the utility function, the motion equation $\dot{a}$, the original value function $V(\cdot)$
3: Initialize the grid of the state variable $a$
4: while the norm between the original and the updated value functions is larger than $\gamma$ do
5:     Calculate the forward difference, the backward difference, and the steady state derivative given the original value function
6:     Use the motion equation to calculate the forward and backward $\dot{a}$
7:     Calculate the upwind derivative by Function 9 and the consumption by Function 8
8:     Update the value funtion by Function 10
9: end while
       return the final value function, the policy function
```

2.3.2 The Projection Method

Rather than approximating the derivative directly, the projection method focuses on approximating the value function and policy function by polynomials. According to the Stone-Weierstrass Theorem, since the polynomial set is dense in the continuous function space defined on a closed interval. Polynomials can represent any continuous functions with any precision. Thus, to find a polynomial close enough to the original function is the core of this method:

$$\min_{\theta_1,\cdots,\theta_n} R\{F(x) - \sum_{i=1}^{n} \theta_i \Psi_i(x)\}$$

Here, $F(x)$ is the function we want to approximate. $\Psi_i(x)$s are basis functions. We choose the coefficient vector $\{\theta_1, \cdots, \theta_n\}$ to minimize the resudual. $R\{\cdot\}$.

In practice, three elements are needed: the nodes, $\{x_j\}$, the basis functions, and the criteria.

A natural candidate for $\{x_j\}$ is evenly spaced nodes: $x_i = a + \frac{j-1}{N-1}(b-a)$. However, such a choice has two shortcomings. First, evenly spaced nodes may not produce an accurate approximant. Second, polynomial approximants with evenly spaced nodes may rapidly deteriorate, rather than improve, as the degree of approximation n rises.

Thus, an alternative candidate is the Chebyshev nodes,

$$x_j = \frac{a+b}{2} + \frac{b-a}{2} cos(\frac{N-j+\frac{1}{2}}{N}\pi)$$

which is the solution minimizing the approximation error term $\Pi_{j=1}^{N}(x_j - x)$.

There are two categories of basis functions. The finite element method uses basis functions that are nonzero over only a subinterval of the domain of approximation. While the spectral method uses basis functions that are nonzero over the entire domain of the target function. Both methods require proper polynomials.

Similar to the node-choice case, a natural polynomial candidate is the monomial. However, the monomial has severe collinearity and rounding error problems. Thus, other orthogonal polynomials or splines (e.g., the Chebyshev polynomial) have a broad application:

$$T_i(x) = cos(arccos(x)i)$$

In practice, its recursive form is more popular:

$$T_0 = 1, \quad T_1 = x, \quad T_i(x) = 2xT_{i-1}(x) - T_{i-2}(x) \tag{11}$$

Using Chebyshev polynomials as the basis function has several advantages. First, the error is bounded above: $error \leq \sum_{i=n+1}^{\infty} \theta_i$. Second, unlike the monomial interpolant, Chebyshev interpolant converges to the target function rapidly as the degree of interpolation increases. Third, the derivative of the Chebyshev basis functions also has a recursive form, simplifying the computation.

$$T_0' = 0, \quad T_1' = 1, \quad T_i'(x) = 2T_{i-1}(x) + 2xT_{i-1}'(x) - T_{i-2}'(x) \tag{12}$$

The last element to choose is the criterion, which measures the distance between the approximation and its target. Though the preference on the Chebyshev nodes and the Chebyshev polynomial is everywhere, a thousand economists have a thousand choices of the criteria. Four most used are as follows:

1. The Least Square criterion: $\min_{\theta_1,\cdots,\theta_n} \langle F(x) - \sum_{i=1}^{n} \theta_i \Psi_i(x), F(x) - \sum_{i=1}^{n} \theta_i \Psi_i(x) \rangle$
2. The Galerkin criterion: $\langle F(x) - \sum_{i=1}^{n} \theta_i \Psi_i(x), \Psi_i(x) \rangle = 0$
3. The Method of Moments: $\langle F(x) - \sum_{i=1}^{n} \theta_i \Psi_i(x), x^{i-1} \rangle = 0$
4. The Collocation: $F(x_j) - \sum_{i=1}^{n} \theta_i \Psi_i(x_j) = 0$

Here, $\langle \cdot, \cdot \rangle$ is the inner product. Though different criteria have different formulas. They share the similar intuitions: to limit the error by making it as orthogonal to the n-degree polynomial space as possible.

The proper nodes, the suitable basis functions, and the appropriate criteira enable the approximation of the value and the policy function as follows:

$$V(a) = \sum_{i=1}^{n} \theta_i^V \Psi_i(a) \tag{13}$$

$$c(a) = \sum_{i=1}^{n} \theta_i^c \Psi_i(a) \tag{14}$$

$\{\theta_i^c\}$ and $\{\theta_i^V\}$ evolve recurssively to minimize the residuals based on the first-order condition (Function 8) and the Bellman Equation (Function 7). When both sequences converge, the optimality generates.

Algorithm 5 Projection Method of The Optimal Saving Problem

```
Input: parameters, utility function, motion function, state variables, basis functions, original $\{\theta_i^V\}$ and $\{\theta_i^c\}$
Output: final value function, policy function
1: Initialize the parameter $\sigma$,$\rho$, the threshold value $\gamma$
2: Initialize the utility function, the motion equation $\dot{a}$, the original $\{\theta_i^V\}$, and residual function $R(\cdot)$
3: Initialize the grid of the state variable $a$
4: while the norms between the original and the updated $\{\theta_i^V\}$ and $\{\theta_i^c\}$ are larger than $\gamma$ do
5:     Choose the proper $\{\theta_i^c\}$ to minimize $R(u'(c) - V'(a))$ (from Function 8)
6:     Update the policy function by $c(a) = \sum_{i=1}^{n} \theta_i^c \Psi_i(a)$
7:     Choose the proper $\{\theta_i^v\}$ to minimize $R(\rho V(a) - u(c) - \dot{a}V'(a))$ (from Function 7)
8:     Update the value funtion by $V(a) = \sum_{i=1}^{n} \theta_i^V \Psi_i(a)$
9: end while
       return the final value function, the policy function
```

2.3.3 The Perturbation Method

Solving formulas rather than approximating the functions distinguishes the perturbation method from other methods. Based on the implicit function theorem, the perturbation method aims to pin down the undetermined coefficients of the value function or the policy function in form of their Taylor expansions. The expansion of the consumption is as follows:

$$C(a) = C(a^\star) + C'(a^\star)(a - a^\star) + \frac{C''(a^\star)}{2!}(a - a^\star)^2 + \frac{C^{(3)}(a^\star)}{3!}(a - a^\star)^3 + \cdots \tag{15}$$

Here $a^\star$ is the steady-state value of capital. The target is to calculate the values of $a^\star, C(a^\star), C'(a^\star), C''(a^\star), C^{(3)}(a^\star), \cdots$.

$C(a^\star)$ and $a^\star$ are steady-state consumptions and assets, which are solutions of the first-order derivative of the Bellman Equation 7 with respect to consumption $c(a)$ and capital a with $\dot{a}$ be zero. Meanwhile, an equation including higher-order derivatives generates:

$$0 = u_{cc}c'(f + w - c) + (f_a - \rho)u_c \tag{16}$$

Here, X_a means the partial derivative of a on X. In general, the n^{th} derivative of the Bellman Equation with respect to capital a only contains three catergories variables: the known steady-state capital $a^\star$ and consumption $c(a^\star)$, the solved lower-than-nth-order derivative of consumption $C^{(m)}(a), m < n$, and the n^{th}-order derivative of consumption $C^{(n)}(a)$ as the only unknown variable.

For an example, the twice dirivatives of the Bellman Equation is as follows:

$$\begin{aligned}&3u_{cc}c'f_{aa} + u_c f_{aaa} + (u_{cc} + u_{ccc})f_a(c')^2 - (u_{cc} + u_{ccc})(c')^3 + \\ &(f + w - c)[u_{cc}(c'' \cdot c' + c''') + u_{ccc} \cdot 2c'c'' + u_{cccc}(c')^3] + (f_a - \rho)u_{cc}[(c')^2 + c''] + \\ &(f_a + u_{cc}f_a - c' - 2u_{cc}c')c'' = 0\end{aligned}$$

The steady state conditons, $f + w - c = 0$ and $f_a - \rho = 0$, erase the whole second row. While the values of $C(a^\star)$ and $C'(a^\star)$ are solutions in precious steps. So the only unknown variable is $C''(a^\star)$, which is the result of this equation and the third coefficient of the Taylor expansion, i.e., $C''(a^\star)$.

Two disadvantages of this method are apparent. First, the symbolic computation requires the higher performance of computers and the well-defined function's form. Second, the approximation accuracy of the Taylor expansion or the Padé expansion is high only in the radius of convergence.

Algorithm 6 Perturbation Method of The Optimal Saving Problem

```
Input: parameters, utility function, production function, state variables,
Output: final value function, policy function
 1: Initialize the parameter $\sigma$,$\rho$
 2: Initialize the utility function, the production function (in the symbolic form)
 3: Initialize Taylor expansion of consumption, which has N+1 terms (in the
    symbolic form)
 4: Initialize Function 16 with consumption as its Taylor expansion form.
 5: Calculate the steady state value $c(a^\star)$ and $a^\star$
 6: for i do from 1 to N:
 7:     Derivative function i times
 8:     Plug in the steady state conditions
 9:     Solve the equation
10: end for
        return the final value function, the policy function
```

3 Apply to the Deterministic Optimal Saving Problem

3.1 Results Discussion

In this subsection, I show numerical results based on the finite difference method and illustrate the results' rationality.

Figure 1 presents the value functions, the motion functions of assets, and the policy funcitons corresponding to the different values of discount rate ρ. The global concavity and the monotone increasing of the value functions meet the setting of the utility function and the discount rate. Meanwhile, higher value of the discount rate decreases the aggregated utility and the value function. Higher discount rate forces households weight more on consumption today, less on consumption tomorrow, and less on saving. This intertemporal equilibrium causes a lower asset changes and higher policy function.

Figure 1: Optimal Saving Problem $\rho = 0.1 \sim 0.9$

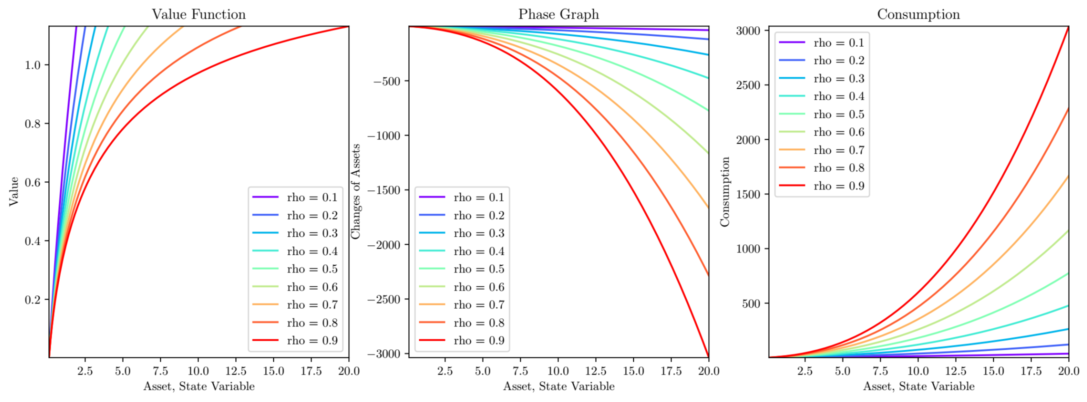


Figure 2 shows the corresponding functions when the model has different interest rate from 0.1 to 0.9. In general, the interest rate has opposite effect with that of the discount rate. Higher interest rates lead the households to consume less today and save more. However, since the interest rate enters the model linearly while the discount rate is on the power, the effects of the former has much smaller magnitudes than that of the later.

Figure 2: Optimal Saving Problem $r = 0.1 \sim 0.9$

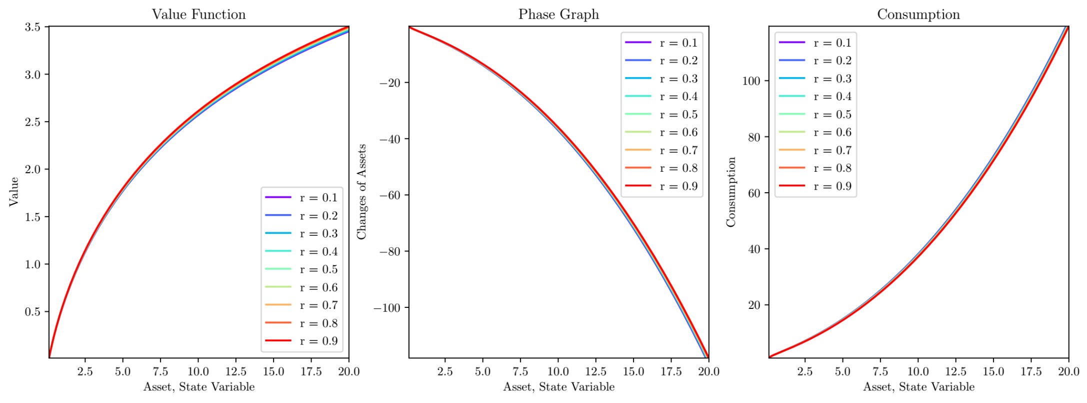
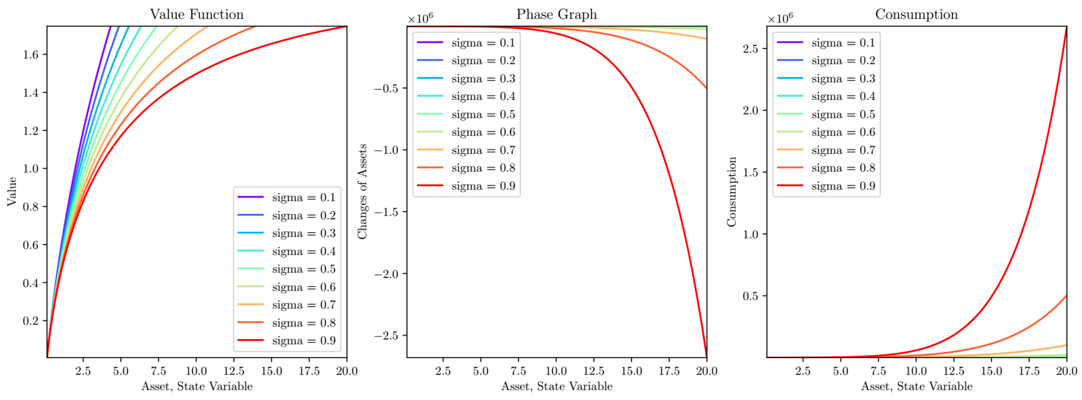


Figure 3 the three functions when the risk aversion of the households increases from 0.1 to 0.9. Higher risk aversion leads to a more concave utility function, causing a lower marginal utility of savings. Thus, a higher risk aversion results in a higher consumption today and lower accumulation of assets.

Figure 3: Optimal Saving Problem $\sigma = 0.1 \sim 0.9$



3.2 Comparing with Exact Solutions

3.3 Accuracy Analysis

4 An Optimal Saving Problem with Poisson Process

4.1 The Setup

Under the continuous-time setting, we include the uncertainty mainly through the Brownian motion or the Poisson process. The former one often represents the fluctuations of asset prices, and the late one represents the innovations process. Since researchers have gone through the Brownian Motion deeply, this paper will focus on the Poisson Process and the derivative in this section follows Wälde(2012).

The household's asset follows a Poisson Process with innovation frequency λ:

$$da = \{ra(t) + w - c(t)\}dt + \beta a(t)dq(t) \tag{17}$$

When there is no innovation, the asset grows with the speed of $r(t)a(t) + w(t) - pc(t)$. If innovation happens, asset's value jumps from $a(t)$ to $(1 + \beta)a(t)$.

4.2 The Bellman Equation

With the Poisson jump, the HJB is an expectation form of that under deterministic situation:

$$\rho V(a(t)) = \max_{c(t)}\{u(c(t)) + \frac{1}{dt}\mathbb{E}dV(a(t))\} \tag{18}$$

The key is to deal with the term $\mathbb{E}dV(a(t))$. The differential term includes two parts, the 'normal term' and the 'jump term'. The former presents the changes without innovations and the later captures the jumps.

$$dV(a(t)) = V_a\{ra(t) + w - c(t)\}dt + \{V((1 + \beta)a(t)) - V(a(t))\}dq \tag{19}$$

The expectation operator is linear. It maps the deterministic terms to deterministic ones, and the Poisson terms to its frequency terms.

$$\begin{aligned}\mathbb{E}dV(a(t)) &= V_a\{ra(t) + w - c(t)\}dt + \{V((1 + \beta)a(t)) - V(a(t))\}\mathbb{E}dq \\ &= V_a\{ra(t) + w - c(t)\}dt + \lambda\{V((1 + \beta)a(t)) - V(a(t))\}dt \end{aligned} \tag{20}$$

Utimately, the HJB under Poisson process has the following form:

$$\rho V(a(t)) = \max_{c(t)}\{u(c(t)) + V_a\{ra(t) + w - c(t)\} + \lambda\{V((1 + \beta)a(t)) - V(a(t))\}\} \tag{21}$$

4.3 The Exact Solution

Similar to the process of solving the deterministic Bellman Equation, solving the HJB under Poisson process requires the first-order condition and the Envelop

Theorem:

$$u'(c) = V'(a)$$
$$\rho V'(a) = rV'(a) + V''(a)(ra + w - c) + \lambda[(1+\beta)V'((1+\beta)a) - V'(a)]$$

Dealing with the $V''(a)$ term requires the Change of Variable Formula (CVF). This formula, which aims to deal with the stochastic calculus of stochastic variables, bridges the $V''(a)$ term and the derivative of $V'(a)$:

$$dV'(a) = V''(a)(ra + w - c)dt + [V'((1+\beta)a) - V'(a)]dq$$

Combining the above three equations gives out:

$$du'(c) = \{(\rho - r)u'(c) - \lambda[(1+\beta)u'((1+\beta)c) - u'(c)]\}dt + [u'((1+\beta)c) - u'(c)]dq$$

With the help of inverse function $f(u'(c)) = c$ and the CVF, the Keynes-Ramsey rule

$$-\frac{u''(c)}{u'(c)}dc = \left\{r - \rho + \lambda\left[\frac{u'((1+\beta)c)}{u'(c)}(1+\beta) - 1\right]\right\}dt - \frac{u''(c)}{u'(c)}\{(1+\beta)c - c\}dq \tag{22}$$

5 Apply to the Stochastic Optimal Saving Problem

5.1 Results Discussion

5.2 Accuracy Analysis

Part III

The Novel Method and its Applications

6 Improvement: XXXXX Methods

Part IV

Appendix

7 Appendix A: Codes

In this appendix, I list the Python codes for solving the optimal growth models. The specific method is Finite Difference Method.

7.1 A Deterministic Optimal Saving Problem

Listing 1: Numerical Solution of A Deterministic Optimal Saving Model with Finite Difference Methods

```
%reset -f
%clear

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import scipy.sparse as sparse
import math
from matplotlib.pyplot import cm

## Utility Function
def Utility(c):
    utility = np.zeros(c.size)
    utility = (c**(1-sigma)-1)/(1-sigma)
    utility = np.mat(utility).T

    return utility

###——— Upwind Derivative
def upwind_d(x_grid,v):
    V = v
```

```
# Initialize the forward derivative,
# Dv_f, the backward derivative, Dv_b,
# the consumption level, c,

 Dv_f = np.zeros(num_x)
 Dv_b = np.zeros(num_x)
 cons = np.zeros(num_x)

 # Update the forward, backward,
   # stable state derivative of value function
   # ———> dV/dx_f,b,s,

 for i in range(0, num_x-1):
     Dv_f[i] = (V[i+1] - V[i])/x_h
           # calculate the forward derivative by linearization

 Dv_f[num_x-1] = 1/ (r*x_max + w)**sigma
                    # calculate the endpoint of forward derivative,
         # use the F.O.C.
         # assume the value is the steady state value

 for i in range(1, num_x):
     Dv_b[i] = (V[i] - V[i-1])/x_h
                    # calculate the backward derivative by linearization

 Dv_b[0] = 1/(r*x_min + w)**sigma
                    # calculate the endpoint of backward derivative,
         # use the F.O.C.
         # assume the value is the steady state value

 # Calculate the consumption, the critic condition,
   # the corresponding f(x)

 cons_f = np.zeros(num_x)
 cons_f = np.sign(Dv_f)* abs(1/Dv_f)**(1/sigma)
         # use the F.O.C to calculate the consumption
```

```
                  # under the forward situation
f_x_f = r * x_grid + w - cons_f
                  # calculate the f(x) under the forward situation

cons_b = np.zeros(num_x)
cons_b = np.sign(Dv_f)* abs(1/Dv_b)**(1/sigma)
          # use the F.O.C to calculate the consumption
                  # under the backward situation
f_x_b = r * x_grid + w - cons_b
                  # calculate the f(x) under the backward situation

cons_ss = np.zeros(num_x)
cons_ss = r * x_grid + w
                  # calculate the consumption under steady state
Dv_ss = 1/(cons_ss**sigma)
                  # calculate the dV/dx under steady state

# combining the dV/dx_f, _b, _ss to upwind dV/dx

Dv_upwind = np.zeros(num_x)
for i in range(0,num_x):
    weight = np.array([0,0,0])
    if f_x_f[i] > 0:
        weight[0] = 1

    if f_x_b[i] < 0:
        weight[1] = 1

    if weight[0] == 0 and weight[1]==0:
        weight[2] = 1

    Dv_upwind[i] = weight[0] * Dv_f[i] +weight[1] * Dv_b[i] \\
          + weight[2] * Dv_ss[i]

# Calculate the consumption and f(x) with
  # upwind derivative of value function
```

```
    cons = np.sign(Dv_upwind) * abs(1/ Dv_upwind)**(1/sigma)

    u_c = np.zeros(num_x)
    u_c = (cons**(1-sigma)-1)/(1-sigma)

    return [u_c, f_x_f, f_x_b,cons, Dv_upwind]

###----- Update Step:

def update_v(x_grid, v):
    #Calculate the A matrix

    f_x_b = np.zeros((num_x,2))
    f_x_b[:,0] = upwind_d(x_grid, v)[2]
    f_x_f = np.zeros((num_x,2))
    f_x_f[:,0] = upwind_d(x_grid, v)[1]

    X = -np.min(f_x_b,axis = 1 )/x_h
                    # 1st elements in the row vector
    Y = -np.max(f_x_f,axis = 1)/x_h + np.min(f_x_b, axis = 1)/x_h
                    # 2nd element in the row vector
    Z = np.max(f_x_f,axis = 1)/x_h
                    # 3rd element in the row vector

    A = np.zeros((num_x, num_x))
    for i in range(1,num_x-1):
        A[i,i-1] = X[i]
        A[i,i] = Y[i]
        A[i,i+1] = Z[i]

    A[0,0]=Y[0]
    A[0,1]=Z[0]
    A[num_x-1,num_x-1] = Y[num_x-1]
    A[num_x-1,num_x-2] = X[num_x-1]

    # Set the length of step
    Delta = 1000
    # Calculate the LHS of (*), the coefficient of v^{n+1}
    L = np.identity(num_x)
    L = (rho + 1/Delta) * L - A

    # Calculate the RHS of (*)
    R = np.zeros(num_x)
    R = upwind_d(x_grid, v)[0] + v/Delta;

    # update the new value function
    v_new = np.zeros(num_x)
    v_new = np.linalg.inv(L).dot(R)

    v_delta = v_new - v
    v_delta = np.absolute(v_delta)
    v = v_new
    return [v_delta, v, upwind_d(x_grid, v)[3], upwind_d(x_grid, v)[4]]


###—— Calculate the value function by iteration
def FDM_OG(x_grid, rho, r, w, sigma, num_x):

    ## Set the initial value function:

    v = np.ones(num_x)
    for i in range(0,num_x):
        v[i] = ((r*x_grid[i]+w)**(1-sigma)-1)/(1-sigma)
             # assume the initial state is the steady state

    ## initial iteratin times and critical value
    itera = 0                # The iteration times
    metric_v = 100           # The metric between new and old value function

    # iteration
    while metric_v > 0.001:
        if itera < 5000:
            metric_v = np.max(update_v(x_grid, v)[0])
            v = update_v(x_grid, v)[1]
            itera = itera + 1
            #print(itera, metric_v)
        else:
            break

    Dv_Dx = np.zeros(num_x)
    Dv_Dx = update_v(x_grid, v)[3]

    cons = 1/Dv_Dx ** sigma
    return [v, cons]



# plot the value function
def plot_result(I, text):
    color = cm.rainbow(np.linspace(0, 1, I))

    fig = plt.figure(figsize = (15,5))
    ax1 = plt.subplot(1,3,1)
    ax1.set_xlim(x_min, x_max)
    ax1.set_ylim(min(v), max(v))

    plt.xlabel('Asset, State Variable')
    plt.ylabel('Value')
    plt.title('Value Function')



    # plot the phase graph

    ax2 = plt.subplot(1,3,2)
    ax2.set_xlim(x_min, x_max)
    ax2.set_ylim(min(x_dot), max(x_dot))

    plt.xlabel('Asset, State Variable')
    plt.ylabel('Changes of Assets')
    plt.title('Phase Graph')
    # plot the consumption

    ax3 = plt.subplot(1,3,3)
    ax3.set_xlim(x_min, x_max)
    ax3.set_ylim(min(cons), max(cons))

    plt.xlabel('Asset, State Variable')
    plt.ylabel('Consumption')
    plt.title('Consumption')

    for i in range(I):
        ax1.plot(x_grid, v_store[:,i], color = color[i], label = text[i])
        ax1.legend()
        ax2.plot(x_grid, x_dot_store[:,i], color = color[i], label = text[i])
        ax2.legend()
        ax3.plot(x_grid, cons_store[:,i], color = color[i], label = text[i])
        ax3.legend()
    plt.show()

###——— Initialization Part:
## Parameters
I = 9
Rho = np.linspace(0.1,0.9,I)
R = np.linspace(0.01,0.02,I)
w = 1
Sigma = np.linspace(1.5,3.5,I)

## Assest Grid
x_min = 0.1          # The minimum value of asset
x_max = 20           # The maximum (allowed) value of asset
x_h = 0.1            # The width of each cell of the asset grid
num_x = int((x_max - x_min)/x_h) + 1
                                        # The number of the asset grid's nodes
x_grid = np.linspace(x_min, x_max, num_x, endpoint = True)
                                        # the gird is [0.1,0.2 ... 200], all 2001

###——— Results:
```

```
# for rho from 0.1 to 0.9

#rho = Rho[1]
r = R[1]
sigma = Sigma[1]

v_store = np.zeros([num_x, I])
cons_store = np.zeros([num_x, I])
x_dot_store = np.zeros([num_x, I])

for i in range(0,I):
    rho = Rho[i]
    cons = FDM_LOG(x_grid=x_grid, rho = rho, \\
      r = r, w = w, sigma = sigma, num_x = num_x)[1]
    v = FDM_LOG(x_grid=x_grid, rho = Rho[i], \\
      r = r, w = w, sigma = sigma, num_x = num_x)[0]
    x_dot = r * x_grid + w - cons

    x_dot_store[:,i] = x_dot
    v_store[:,i] = v
    cons_store[:, i] = cons

# for r from 0.01 to 0.02
rho = Rho[1]
#r = R[1]
sigma = Sigma[1]

v_store = np.zeros([num_x, I])
cons_store = np.zeros([num_x, I])
x_dot_store = np.zeros([num_x, I])

for i in range(0,I):
    r = R[i]
    cons = FDM_LOG(x_grid=x_grid, rho = rho, r = r,\\
      w = w, sigma = sigma, num_x = num_x)[1]
    v = FDM_LOG(x_grid=x_grid, rho = Rho[i], r = r,\\
      w = w, sigma = sigma, num_x = num_x)[0]
    x_dot = r * x_grid + w - cons

    x_dot_store[:,i] = x_dot
    v_store[:,i] = v
    cons_store[:, i] = cons

#for sigma from 0.1 to 0.9

rho = Rho[1]
r = R[1]
#sigma = Sigma[1]

v_store = np.zeros([num_x, I])
cons_store = np.zeros([num_x, I])
x_dot_store = np.zeros([num_x, I])

for i in range(0,I):
    sigma = Sigma[i]
    cons = FDM_OG(x_grid=x_grid, rho = rho, r = r,\\
      w = w, sigma = sigma, num_x = num_x)[1]
    v = FDM_OG(x_grid=x_grid, rho = Rho[i], r = r, \\
      w = w, sigma = sigma, num_x = num_x)[0]
    x_dot = r * x_grid + w - cons

    x_dot_store[:,i] = x_dot
    v_store[:,i] = v
    cons_store[:, i] = cons
```

8 Appendix B: Graphs

Figure 4: Optimal Saving Problem $\rho = 0.1 \sim 0.9$

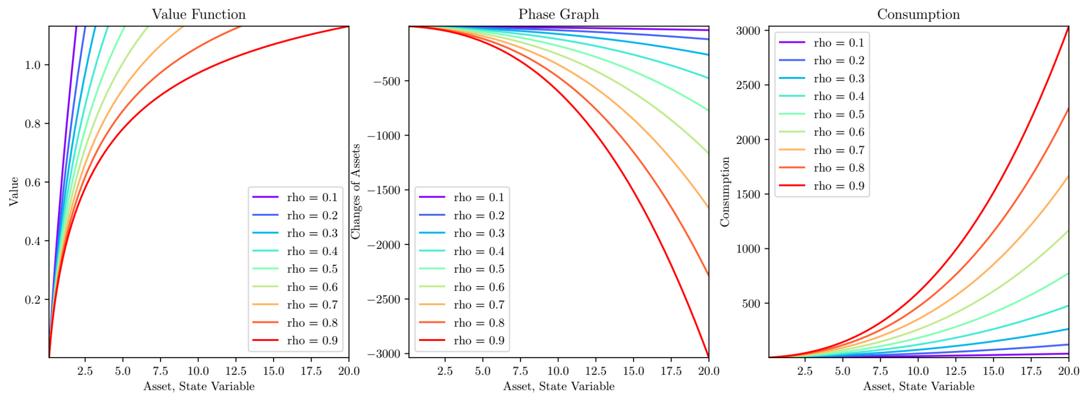


Figure 5: Optimal Saving Problem $r = 0.01 \sim 0.02$

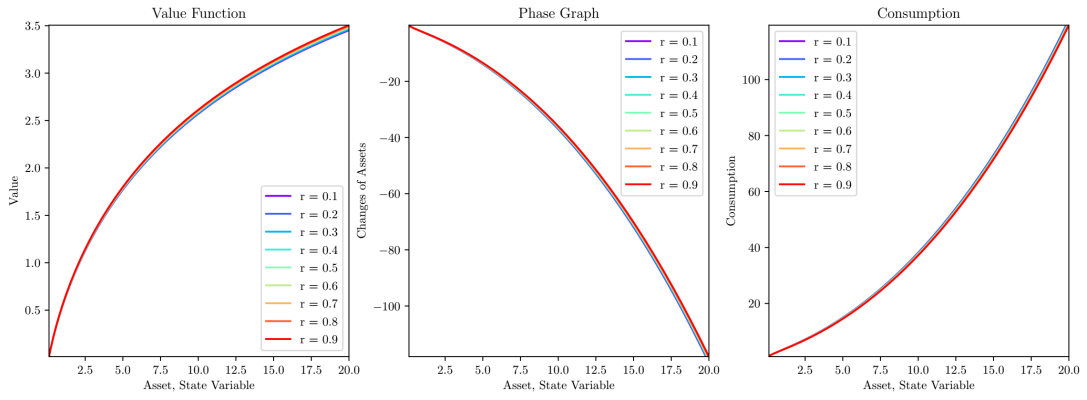
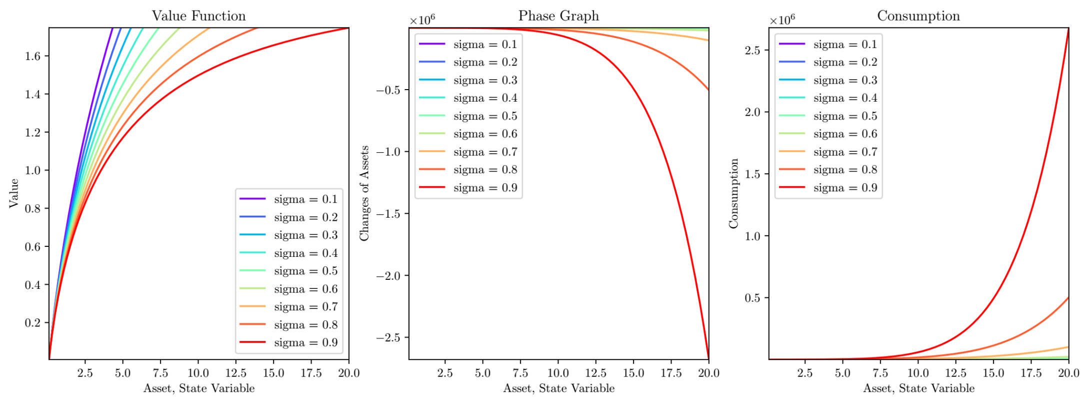


Figure 6: Optimal Saving Problem $\sigma = 0.1 \sim 0.9$


Value Function
Value
Asset, State Variable
Phase Graph
Changes of Assets
$\times 10^6$
Consumption
sigma = 0.1
sigma = 0.2
sigma = 0.3
sigma = 0.4
sigma = 0.5
sigma = 0.6
sigma = 0.7
sigma = 0.8
sigma = 0.9