Asset Price Models Review and their application in pricing of land [*]

Suraj Kumar

November 12, 2019

Abstract

This paper reviews the two asset pricing models such as Lucas Asset Price Model and Harrison and Kreps Asset Price Model. It then applies the framework of these models in the context of pricing of land in the presence of land use regulations and land rents. It then presents the python code to solve these models as *Two State and One Control Bellman Equation.*

Keywords: Asset Pricing, Dynamic Programming, Two State and One Control Bellman Equation, Land Pricing

JEL Codes: E31, C63, C68

[*]I am grateful to Prof. Abhijit Banerji and Prof. Ram Singh for their guidance in the course of this work.

1. Asset Price Models Review

1.1. Lucas Asset Price Model

The asset price model of Lucas [1]attempts to find the correct price of asset with stream of prospective payments in an equilibrium setting with risk averse agents.

1.1.1. Basic Structure

The basic structure of the model can be expressed as

1. **Economy:** Lucas economy is a pure exchange economy with a representative consumer(or household). Thus all endowments are exogenous and all consumers have identical endowments and preferences.
2. **Asset:** There is single *productive unit* that costlessly generates a sequence of consumption goods(endowments) $\{y_t\}_{t=0}^{\infty}$. The endowment is Markovian, following the exogenous process $y_{t+1} = G(y_t, \xi_{t+1})$. Here $\{\xi_t\}$ is an iid shock sequence with known distribution ϕ and $y_t \geq 0$.. The asset is a claim to all or part of this endowment. The consumption goods $\{y_t\}_{t=0}^{\infty}$ are not storable thus by holding assets only can the wealth be transfered into futures
3. **Consumers:** The consumer ranks consumption streams $\{c_t\}$ according to time separable utility function $\mathbb{E}\sum_{t=0}^{\infty}\beta^t u(c_t)$. Here $\beta \in (0,1)$ is fixed discount factor, u is strictly increasing, strictly concave, continuously differentiable period utility function and $\mathbb{E}$ is mathematical expectation. The consumer takes the price function $p(y)$ as given.

1.1.2. Solution and Results

Lets us assume that asset is of type *ex dividend* claim meaning that

- seller has right to current period dividend
- the buyer pays p_t in the current period to have claim on y_{t+1} and the right to sell the asset in the next period at price p_{t+1}

The consumer behaviour is to control the share π_t of the asset in each period. Thus the budget constraint can be written as

$$c_t + \pi_{t+1} * p_t \leq \pi_t * y_t + \pi_t * p_t \tag{1.1}$$

here $c_t \geq 0$ and $0 \leq \pi_t \leq 1$ for all t. Thus the decision to hold share π_{t+1} is made at time t.
Using the assumption that price is given function p of y, the value function and constraint can be written as

$$v(\pi, y) = \max_{c,\pi'}\left\{u(c) + \beta\int v(\pi', G(y,z))\phi(dz)\right\} \tag{1.2}$$

$$\text{subject to } c + \pi' p(y) \leq \pi y + \pi p(y) \tag{1.3}$$

Under the assumption of non storable consumption goods, so $c_t = y_t$ for all t. As there is only one representative consumer, there should be no trade in equilibrium and consumer owns entire asset in each period. Thus $\pi_t = 1$ for all t. Thus the equilibrium price function can be derived by first order condition of eq[1.2] as

$$u'(c)p(y) = \beta\int v_1'(\pi', G(y,z))\phi(dz) \tag{1.4}$$

[1]Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*

$$v_1'(\pi, y) = u'(c)(y + p(y)) \tag{1.5}$$
$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)}[G(y,z) + p(G(y,z))]\phi(dz) \tag{1.6}$$

The eq[1.6] can be solved by substituting $f(y) = u'(y)p(y)$

$$f(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{1.7}$$
$$\text{where } h(y) = \beta \int u'[G(y,z)]G(y,z)\phi(dz) \tag{1.8}$$

Lets define operator T mapping f into Tf as

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)]\phi(dz) \tag{1.9}$$

Thus the solution of eq[1.7] corresponds to a function f^*(fixed point) of T such that $(Tf^*)(y) = f^*(y)$ for all y.

1.1.3. Application of Lucas Asset Model to Land

The y_t in the context of land can be considered as rent in time period t generated by a unit of land. Let Z_t be the index of Land Use Regulation at time t. Let the agent own share π_t at time t.

1.1.3.1. Representative Landlord and $Z_t = Z$ for all t In this case eq[1.2] and eq[1.3] can be transformed as

$$v(\pi, y) = \max_{c,\pi'} \left\{ u(c) + \beta \int v(\pi', G(y,z))\phi(dz) \right\} \tag{1.10}$$
$$\text{subject to } c + \pi' p(y) \leq \pi y Z + \pi p(y) \tag{1.11}$$

Now under the assumptions of nonstorable rents (y_t) and one representative land owner, $c = \pi y Z$ and $\pi_t = 1$ for all t. Since the representative land owner takes price as given function, the equation similar to eq[1.6] can be constructed to obtain the price as function of current period y(and implicit function of land use regulation Z). This new obtained equation can be solved as fixed point of operator T as defined in eq[1.9].

1.1.3.2. Non representative Landlords and $Z_t = Z$ for all t Let suppose an agent owns π_t units of land. In this case eq[1.10] and eq[1.11] can be transformed as follows

$$v(\pi, y) = \max_{\pi'} \left\{ u\left[\pi(yZ + p(y)) - \pi' p(y)\right] + \beta \int v(\pi', G(y,z))\phi(dz) \right\} \tag{1.12}$$

The solution to this dynamic programming problem is an optima policy expressing π' or c as function of state(π, y). Given π'(or c) can be obtained from eq[1.11]. The first order condition for eq[1.12] can be written as

$$u'(c)p(y) = \beta \int v_1'(\pi', G(y,z))\phi(dz) \tag{1.13}$$
$$v_1'(\pi', G(y,z)) = u'[\pi' G(y,z)Z][G(y,z)Z + p(G(y,z))] \tag{1.14}$$
$$p(y) = \beta \int \frac{u'[\pi' G(y,z)Z]}{u'(\pi y Z)}[G(y,z)Z + p(G(y,z))]\phi(dz) \tag{1.15}$$

To solve eq[1.15], the assumption can be made that for the purpose of determination of equilibrium price function, the representative consumer can be considered to represent the heterogeneous agents (with different π_t), thus for representative consumer $\pi = \pi'$. The eq[1.15] thus becomes similar to eq[1.6], thus can be solved on the similar lines. Now given the equilibrium price function $p(y)$, eq[1.12] becomes *Two State and One Control Bellman Equation* which can be solved according to Section 2.

1.1.3.3. Non representative Landlords and Z_t Let suppose an agent owns π_t units of land. Let $Z_t = Z$ and $Z_{t+1} = Z'$. In this case eq[1.12] can be transformed as follows

$$v(\pi, y, Z) = \max_{\pi'} \left\{ u\left[\pi(yZ + p(y,Z)) - \pi' p(y,Z)\right] + \beta \int v(\pi', G(y,z), Z')\phi(dz) \right\} \tag{1.16}$$

The solution to this dynamic programming problem is an optima policy expressing π' or c as function of state(π, y). Given π'(or c) can be obtained from eq[1.11]. The first order condition for eq[1.12] can be written as

$$u'(c)p(y,Z) = \beta \int v_1'(\pi', G(y,z))\phi(dz) \tag{1.17}$$

$$v_1'(\pi', G(y,z), Z') = u'[\pi' G(y,z)Z'][G(y,z)Z' + p(G(y,z), Z')] \tag{1.18}$$

$$p(y,Z) = \beta \int \frac{u'[\pi' G(y,z)Z']}{u'(\pi y Z)}[G(y,z)Z' + p(G(y,z), Z')]\phi(dz) \tag{1.19}$$

Let $x = yZ$, thus, p(y,Z) = p(x) where

$$p(x) = \beta \int \frac{u'[\pi' x']}{u'(\pi x)}[x' + p(x')]\phi(dz) \tag{1.20}$$

Once again the similar assumption can be made that for the purpose of determination of equilibrium price function, the representative consumer can be considered to represent the heterogeneous agents (with different π_t), thus for representative consumer $\pi = \pi'$.The eq[1.20] thus becomes similar to eq[1.6], thus can be solved on the similar lines. Now given the equilibrium price function $p(y,Z)$ and replacing $x = yZ$, eq[1.16] becomes $v(\pi, x)$ *Two State and One Control Bellman Equation* which can be solved according to Section 2.

1.2. Harrison and Kreps Asset Price Model

The model of this section has been adapted from Harrison and Kreps Model[2]. This model of asset pricing has heterogeneous expectations within the community of potential investors. The Harrison-Kreps model illustrates following concepts of economic bubble:

A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds that they believe that asset's underlying dividend stream justifies

1.2.1. Basic Structure and Assumptions

The basic structure of the model can be expressed as

1. **Asset:** Let there is fixed number A of shares of an asset. Each share entitles its owner to stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$. The dividend obeys

$$d_t = \begin{Bmatrix} 0, & s_t = 0 \\ 1, & s_t = 1 \end{Bmatrix} \tag{1.21}$$

The owner of the asset at the end of time t is entitled to the dividend at time $t+1$ and also has the right to sell the asset at time $t+1$.

2. **Investors:** Investors are partitioned into finite number of internally homogeneous classes. Each class of investors has infinite amount of wealth. All investors have access to same economic information but they may arrive different probability assessments of returns from the asset. Investors are risk-neutral. Discount factor(β) is known and constant. Short sales of stock are forbidden.

Let there are two types $h = a, b$ of investors differ only in their beliefs about a Markovian transition matrix P with typical element

$$P(i, j) = \mathbb{P}\{s_{t+1} = j | s_t = i\} \tag{1.22}$$

Let the transition matrix for type a and type b investor is as follows

$$P_a = \begin{bmatrix} 1/2 & 1/2 \\ 2/3 & 1/3 \end{bmatrix} \tag{1.23}$$

$$P_b = \begin{bmatrix} 2/3 & 1/3 \\ 1/4 & 3/4 \end{bmatrix} \tag{1.24}$$

Given P_a, P_b, the stationary distributions can calculated as $\pi_A = [.57.43]$ and $\pi_B = [.43.57]$. It implies that type B person is more optimistic about the dividend process in the long run than is type A person. But given state 0 (low dividend state), type a agent is more optimistic about next period's dividend than type b agent. And in state 1, type b agent is more optimistic about the next period's dividend.

The temporarily optimistic investors transition matrix (,i.e. the investor with the most optimistic beliefs in each state) and temporarily pessimistic transition matrix (,i.e. the investor with the most pessimistic beliefs in each state) can be constructed as

$$P_o = \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & 3/4 \end{bmatrix} \tag{1.25}$$

[2] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics, 1978*

$$P_p = \begin{bmatrix} 2/3 & 1/3 \\ 2/3 & 1/3 \end{bmatrix} \tag{1.26}$$

$$\tag{1.27}$$

Let both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset.

Investor know a price function mapping the state s_t at t into the equilibrium $p(s_t)$ that prevails in that state. The price function is endogenous. The investors know s_t when they decide to whether purchase or sell the asset at t.

1.2.2. Equilibrium Price Function

1.2.2.1. Equilibrium Price Function under only type of agent either a or b: Let there is only one type of investor, either of type a or b. This type of investor prices the asset. Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all the investors are of type h. The current period price is expected discounted value of tomorrows's divided and tomorrow's price of the asset:

$$p_h(s) = \beta \left[P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right], \; s = 0, 1 \tag{1.28}$$

The above equations can be solved to derived the equilibrium price vector as

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} * P_h * \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.29}$$

Here $p_h(s)$ can be interpreted as what the investor h thinks is the *"fundamental value"* of the asset.It is equal to expected discounted value of future dividends.

1.2.2.2. Equilibrium Price Function under both types of agents with sufficient funds: It is assumed that both type of agents have sufficient wealth to purchase all of the asset themselves. In this scenario, according to Harrison and Kreps, the marginal investor who prices the asset is the more optimistic type. the equilibrium price $p(s)$ satsifies Harrison and Kreps(*Proposition 1*) as follows:

$$p(s) = \beta \max \Big\{ P_a(s,0)p(0) + P_a(s,1)(1+p(1)), P_b(s,0)p(0) + P_b(s,1)(1+p(1)) \Big\} \tag{1.30}$$

This implies that marginal investor who prices the asset in state s is of type i if

$$i = a \text{ if } P_a(s,0)p(0) + P_a(s,1)(1+p(1)) > P_b(s,0)p(0) + P_b(s,1)(1+p(1)) \tag{1.31}$$

$$i = b \text{ if } P_a(s,0)p(0) + P_a(s,1)(1+p(1)) < P_b(s,0)p(0) + P_b(s,1)(1+p(1)) \tag{1.32}$$

The eq[1.30] can be written in matrix form as

$$p = \begin{bmatrix} p(0) \\ p(1) \end{bmatrix} = \beta * \max \left\{ P_a \left[p + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right], P_b \left[p + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right] \right\} \tag{1.33}$$

The eq[1.33] is like functional equation. Lets define operator T mapping p to Tp as

$$Tp = \beta * \max \left\{ P_a \left[p + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right], P_b \left[p + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right] \right\} \tag{1.34}$$

Let $X = \{0,1\}$, $\mathscr{T}_X = 2^X, Y = \{p(0), p(1)\}$ and $\mathscr{T}_Y = 2^Y$. Thus $(X, \mathscr{T}_X)$ and $(Y, \mathscr{T}_Y)$ are topological spaces and $p : X \to Y$. p is continuous function as for every $E \in \mathscr{T}_Y$, $p^{-1}(E) \in \mathscr{T}_X$. p is bounded as X is compact and p is continous. Let $C(X)$ denotes the space of bounded continuous function. It is easy to show that $T : C(X) \to C(X)$. Now the existence of fixed point of eq[1.34] can be interpreted as solution to eq[1.33]. Let us check whether T satisfy *Blackwell's sufficient conditions for a contraction.*

- **Monotonicity**: Let $p, p' \in C(X)$ and $p(x) \leq p'(x)$ for $x \in X$. It is trivial to show that $Tp(x) \leq Tp'(x)$.
- **Discounting**:

$$T(p+c) = \beta * \max\left\{P_a\left[(p+c)+\begin{bmatrix}0\\1\end{bmatrix}\right], P_b\left[(p+c)+\begin{bmatrix}0\\1\end{bmatrix}\right]\right\} \tag{1.35}$$

$$= \beta * \max\left\{P_a\left[p+\begin{bmatrix}0\\1\end{bmatrix}\right]+P_a c, P_b\left[p+\begin{bmatrix}0\\1\end{bmatrix}\right]+P_b c\right\} \tag{1.36}$$

$$\leq Tp + \beta c \tag{1.37}$$

Thus T is contraction with module β, by *Banach Fixed Point Theorem*, there exist a unique bounded continonus function p^* which is solution to eq[1.33]. Thus eq[1.33] can be solved like bellman iterations as

- start with guess for the price vector p
- iterate to convergence on the operator T that maps guess p^j into an updated guess p^{j+1} as $T(p^j) = p^{j+1}$

1.2.2.3. Results: Under the assumed transition matrices in eq[1.23] and eq[1.24], the equilibrium price function are given in Table(1). p_a and p_b are equilibrium price function under homogeneous beliefs P_a and P_b respectively. p_o is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors. It is been observed that equilibrium price under heterogeneous beliefs economy, p_o, exceed what any prospective investor regards as the fundamental value of the asset in each possible state. Thus economic bubble situation arises under marginal optimistic investor behaviour.

Table 1: Equilibrium Price

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
$\overline{p}_a$	1.85	1.69
$\overline{p}_b$	1.69	2.08

The economy repeatedly visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth. The investor expects to have the option to sell the asset latter to another investor who will value the asset more likely than he will.

- Investors of type a are willing to pay the following price for the asset

$$\overline{p}_a(s) = \left\{\begin{matrix} p(0), & s_t = 0 \\ \beta[P_a(1,0)p(0) + P_a(1,1)(1+p(1))], & s_t = 1 \end{matrix}\right\} \tag{1.38}$$

- Investors of type b are willing to pay the following price for the asset

$$\overline{p}_b(s) = \left\{\begin{matrix} \beta[P_b(0,0)p(0) + P_b(0,1)(1+p(1))], & s_t = 0 \\ p(1), & s_t = 1 \end{matrix}\right\} \tag{1.39}$$

It is clear from eq[1.30], eq[1.38] and eq[1.39], that $\overline{p}_a(1) < p(1)$ and $\overline{p}_b(0) < p(0)$. Thus investors of type a want to sell the asset in state 1 while investors of type b want to sell it in state 0. Thus asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.

1.2.3. Application to Harrison-Kreps Model to Land

The Harrison-Kreps model can be used to explain the bubble in Land Asset Markets. It is essentially because the pricing of asset is being done by marginal optimistic investor. In context of land, dividends can be interpreted as rents which depends on state of nature such as government investment in nearby area, natural calamity or index of Land Use Regulation (Z_t). The model can be extended to include finite dimension states of nature, say N, with correspoding rents to be d_s where $s \in \{1, 2, .., N\} = \mathcal{N}$. Let there are M number of investors with transition matrix beliefs as P_i where $i \in \{1, 2, ..., M\} = \mathcal{M}$.

1.2.3.1. Equilibrium Price Function under only one type of investors($h \in \mathcal{M}$): The equilibrium price function under homogeneous beliefs h where $h \in \mathcal{M}$ can be calculated using eq[1.29].

1.2.3.2. Equilibrium Price Funtion under M types of agents with sufficient funds: The equilibrium price function under M investors with heterogeneous beliefs can written similarly to eq[1.30] as follows

$$p_t(s) = \max_{j \in \mathcal{M}} \mathbb{E}^j \left[\beta d_{t+1}(x_{t+1}) + \beta p_{t+1}(x_{t+1}) | x_t = s \right] \tag{1.40}$$

where x_t denote the state of nature at time t.
The eq[1.40] can be solved similarly to eq[1.33] using bellman iterations. The Equilibrium price function table similar to Table[1] can be obtained by first solving for eq[1.40] and then using equations similar to eq[1.38] and eq[1.39].

2. Computational Solution

This section present the solution of pricing models considered in this paper.

2 - Computation -Two State and One Control Bellman Equation Model

November 11, 2019

0.0.1 2.2. Code

[3]:
```
### Importing relevent packages

# To showcase the outputs of all cells
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# Numerican Packages
import numpy as np
from numpy import random
from scipy.stats import lognorm
from scipy.integrate import fixed_quad
from scipy import interpolate
from scipy import optimize

# Plotting Packages
%matplotlib notebook
import matplotlib.pyplot as plt

# 3D Plotting Packages
from mpl_toolkits import mplot3d

# Impornting
from numba import jit,njit,vectorize
```

[5]:
```
### Creating a Class to hold the model paramters
class LucasAssetModel:
    """
    Lucas Asset Price Tree model
    This class creates all model parameters needed for the computation
    """

    # Creating the class Paramters
    def __init__(self,
                  = 0.5,
                 = 0.95,
                 = 0.9,
                 = 0,
                s = 0.1,
                rent_min = 1e-5,
                rent_max = 4,
                rent_grid_size = 100,
                shock_size = 100,
                seed = 100,
                 _min = 1e-5,
                 _max = 1,
                 _grid_size = 100):
        """
        Creates Instance of Lucas Model
        --------------------------------
        Parameters are defined as below
        --------------------------------
        #Model Parameters
         : relates the y(t) with y(t-1)
         : discount factor of agent utility
         : Utiltiy parameter of the agent and 0 <   < 1

        # Shock Parameters
        s,  : Shock Parametes as z_t = e^( + s*_t), _t is iid shock sequence␣
↪with N(0,1)
        shock_size: No of shocks to consider

        ### State Paramters
        # Rent
        rent_min: minimum y (state) value to consider
        rent_max: maximum y (state) value to consider
        rent_grid_size: No of points to select between rent_min and rent_max
        # Share
         _min: minimum  (share) value to consider
         _max: maximum  (share) value to consider
         _grid_size: No. of points to select between _min and _max

        # Randomness
        seed : For getting the same set of random numbers again and again so as␣
↪to reproduce the results

        """
        ### Assinging Parameters

        # Model Parameters
        self. , self. , self.  =  ,  ,
        # Shocks Parameters
        self.s, self. , self.shock_size, self.seed  = s,  , shock_size, seed

        #Rental Grid State Allocation
        self.rent_min, self.rent_max, self.rent_grid_size  = rent_min,␣
↪rent_max, rent_grid_size
        self.rent_grid = np.linspace( self.rent_min, self.rent_max, self.
↪rent_grid_size)

        #Setting up shocks in the models
        np.random.seed(self.seed) # setting seed to generate same random␣
↪numbers,
        #useful for repeating the steps of computation again to see check the␣
↪result,
        self.shocks = np.exp( self.  + self.s * np.random.randn(self.
↪shock_size)  )

        #Setting up grid for share
        self. _min, self. _max, self. _grid_size =  _min,  _max,  _grid_size
        self. _grid = np.linspace(self. _min, self. _max, self. _grid_size)

    # Initial Surface to input into bellman operator

    # First Fuction
    def initial_w(self): #Intialzing the grid with some continous, concave␣
↪function
        pi_s,rent_s = np.meshgrid(self. _grid, self.rent_grid)
        return 5*np.sqrt(pi_s*rent_s)

    # Second Function
    def initial_w2(self): #Intialzing the grid with some continous, concave␣
↪function
        pi_s,rent_s = np.meshgrid(self. _grid, self.rent_grid)
        return np.power(pi_s*rent_s,1/3)

    # Solving the Model
    def solve_optvalue(self,
                       tol = 1e-6,
                       max_iter = 100):
        """
        This function is used to solving the model. Its first extract the␣
↪parameters to be passed on to bellman operator.
        Then given the modelling preferences i.e tol and max_iter, it solves␣
↪the model.

        """
        ### Extracting The Parameters from the class object self to be passed␣
↪on to bellman operator
        # Parameters Extractions
        model_parameters = np.array([self. , self. , self. ])
        self.model_parameters = model_parameters

        #Share Grid Extraction
         _max,  _min, _grid = self. _max, self. _min, self. _grid

        #Rent Grid Extraction
        rent_grid = self.rent_grid

        #Shocks Grid Extract
        shocks_sequence = self.shocks

        #### Initial Function
        w = self.initial_w()
        # w = self.initial_w2() # as second choice for the surface

        # Creating Dictionary to see the convergence of surface later on
        d={}

        # Loop paramters
        error = tol + 1
        i = 0

        # Storing the first intial 3D surface in dictionary object
        d[0] = w

        #Create Storage for bellman Operator
        Tw = np.empty_like(w)

        #Iterate to find Solution
        while error > tol and i < max_iter:
            w_new = bellman_operator(w,
                                     _grid,
                                    rent_grid,
                                    model_vars = model_parameters,
                                    shocks = shocks_sequence,
                                    Tw = None,
                                    compute_policy = 0) #computes the bellman
            error = np.max(np.abs(w_new - w)) # Find the maximum absolute␣
↪distance between the
            ################################### two surfaces(w and w_new) over␣
↪the state space
            w = w_new
            i += 1
            d[i] = w_new # Stores the computed surface in dictionary object,
            #it can be used later on to see the convergence of the surfaces

        return w,i,d # Returns converges surface, iterations it took and␣
↪Surface history
```

[6]:
```
#####NOTE####
#Computation function have been written separtely from class object so as to␣
↪make code faster
#later on with the use of numba. At presently numba support for classes i.e␣
↪@jitclass, is still at
#early stages.


# Writingn the Utility Function
@njit
def u( _prime,  , y, p,  ):
    cons = ( *(y + p)) - ( _prime*p)
    return cons**

# State Function
@njit
def g(y,z, ):
    return (y** )*z

# Price Function
@njit
def p(y, ):
    return y/(1- )


# Writing my own Minimum Function to compare against fminbound

def min_fun(objective, _grid, ans = None):
    if ans is None:
        ans = np.empty_like( _grid)
    for i,   in enumerate( _grid):
        ans[i] = objective( )
    return  _grid[ans == min(ans)]


# Bellman Operator
def bellman_operator(w,
                      _grid,
                     rent_grid,
                     model_vars,
                     shocks,
                    Tw = None,
                    compute_policy = 0):
    """
    Parameters
    ------------
    w = is mesh of different   values and rent(y) values
     _grid = share grid
    rent_grid = rent grids or y grid
    model_vars = [ ,  ,  ]
    shocks = shocks value to be used
    Tw = array to output values, Tw is just for computation optimization
    compute_policy = to decide whether to ouput policy function or not

    """
     ,  ,   = model_vars

    # === Appyling Linear interpolation to a two dimensional surface === #
    interpolation_func = interpolate.interp2d( _grid, rent_grid, w,␣
↪kind='linear')


    # === Intitiate Tw if required === #
    if Tw is None:
        Tw = np.empty_like(w)

    # === Intitate compute policy if asked === #

    if compute_policy:
         _opt_policy = np.empty_like(w)

    for i, y in enumerate(rent_grid):
        for j,   in enumerate( _grid):
            def objective( _prime):
                return - u( _prime,  , y, p(y, ),  ) -   * np.mean(␣
↪interpolation_func(  _prime,g(y,shocks, ) ) )
            #_star = optimize.fminbound(objective,  _grid[0],  _grid[-1])
             _star = min_fun(objective, _grid)
            if compute_policy:
                 _opt_policy[i,j] =  _star
            Tw[i,j] = - objective( _star)

    if compute_policy:
        return Tw,  _opt_policy
    else:
        return Tw
```

0.1 3. Results

0.1.1 3.1. Creating an Instance of Model

[7]:
```
mylam =LucasAssetModel(rent_grid_size=50,
                       shock_size=1000,
                        _grid_size=50,
                       seed= 200,
                      rent_min = 1,
                      rent_max = 500,
                       _min = 1e-2,
                       _max = 1)
```

0.1.2 3.2 Solving the Model

[8]:
```
# Running the Model and Saving the Results
w,i,d = mylam.solve_optvalue(max_iter = 100,tol = 1e-1)
i
```

[8]:
```
34
```

[201]:
```
# Running the Model and Saving the Results
w,i,d = mylam.solve_optvalue(max_iter = 1000,tol = 1e-1)
i
```

[201]:
```
34
```

0.1.3 3.3 Visualizing the Results

3.3.1 Absolute Distance between the subsequent surface after iterations

[9]:
```
# Seeing the Results
distance = [] # setting the distance list which the store the maximum distance␣
 ↪between the two subsequence surface
for j in range(len(d)-1):
    distance.append(np.max(d[j+1]-d[j]))

print('First distance is ',distance[0])
print('Last distance is ',distance[i-1])
dis = distance
```

```
First distance is  18.0386865678
Last distance is  0.0961541522701
```

[10]:
```
fig = plt.figure()  # a new figure window
ax = fig.add_subplot(111)

ax.plot(dis);
ax.set_ylim(0,max(distance)+1);
ax.set_xlim(0,i);
ax.set_ylabel('Maximum Distance Between two subsequent surface');
ax.set_xlabel('No. of Iterations');
```

```
<IPython.core.display.Javascript object>
```

Refer to Figure - 1

```
<IPython.core.display.HTML object>
```

3.3.2 Matplotlib 3D Plots Here, I plot 5 surfaces on using Matplotlib 3D plot, They are as follows 1. Initial Surface 2. 5 Iterations Surface 3. 10 Iterations Surface 4. 20 Iterations Surface 5. 25 Iterations Surface 6. 30 Iterations Surface 7. 34 Iteations Surface

[11]:
```
#### Generating Value Function

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

X,Y = np.meshgrid(mylam. _grid,mylam.rent_grid)
ax = plt.axes(projection='3d');

#Choosen following Surface to plot on Fiure
choosen_iterations = [0,5,10,20,25,30,34]
colors_iterations  = ['black','brown','green', 'blue', 'yellow', 'orange','red']

for iter,c in zip(choosen_iterations,colors_iterations):
     ax.plot_wireframe(X, Y, d[iter], rstride=5, cstride=5,color = c ); #Mesh␣
 ↪Plot

ax.set_title('Two State and One Control Bellman Equation (Convergence)');
ax.set_ylabel('Rent Grid');
ax.set_xlabel('Share  Grid');
ax.set_zlabel('Value Function');
```

```
<IPython.core.display.Javascript object>
```

Refer to Figure - 2

```
<IPython.core.display.HTML object>
```

[ ]:
```
#### Generating the Policy Function

# Last Policy Iteration
w_new, _opt_policy = bellman_operator(w = d[i],
                     _grid = mylam. _grid,
                    rent_grid = mylam.rent_grid,
                    model_vars = mylam.model_parameters,
                    shocks = mylam.shocks,
                    Tw = None,
                    compute_policy = 1)
```

[20]:
```
#### Generating the Policy Function

#First Policy Iterations
w_new2,  _opt_policy2 = bellman_operator(w = d[0],
                     _grid = mylam. _grid,
                    rent_grid = mylam.rent_grid,
                    model_vars = mylam.model_parameters,
                    shocks = mylam.shocks,
                    Tw = None,
                    compute_policy = 1)
```

[21]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

X,Y = np.meshgrid(mylam. _grid,mylam.rent_grid)
ax = plt.axes(projection='3d');
ax.plot_wireframe(X, Y,  _opt_policy, rstride=5, cstride=5,color = 'r' )
ax.plot_wireframe(X, Y,  _opt_policy2, rstride=5, cstride=5,color = 'black' )
ax.set_title('Policy Function');
ax.set_ylabel('Rent Grid');
ax.set_xlabel('Share  Grid');
ax.set_zlabel(' ');
```

```
<IPython.core.display.Javascript object>
```

Refer to Figure - 3

```
<IPython.core.display.HTML object>
```

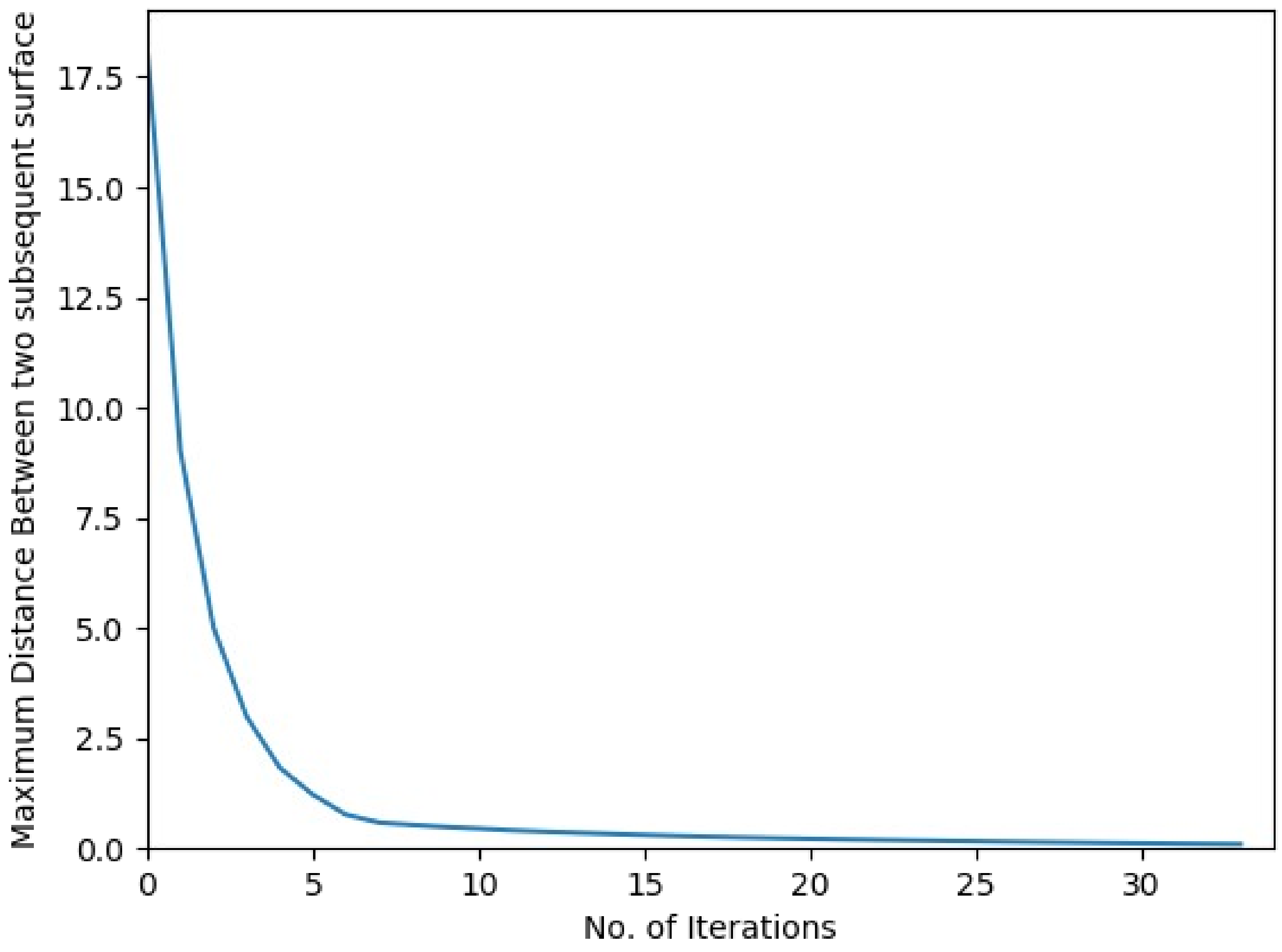

Maximum Distance Between two subsequent surface
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20
25
30
No. of Iterations

Figure - 1

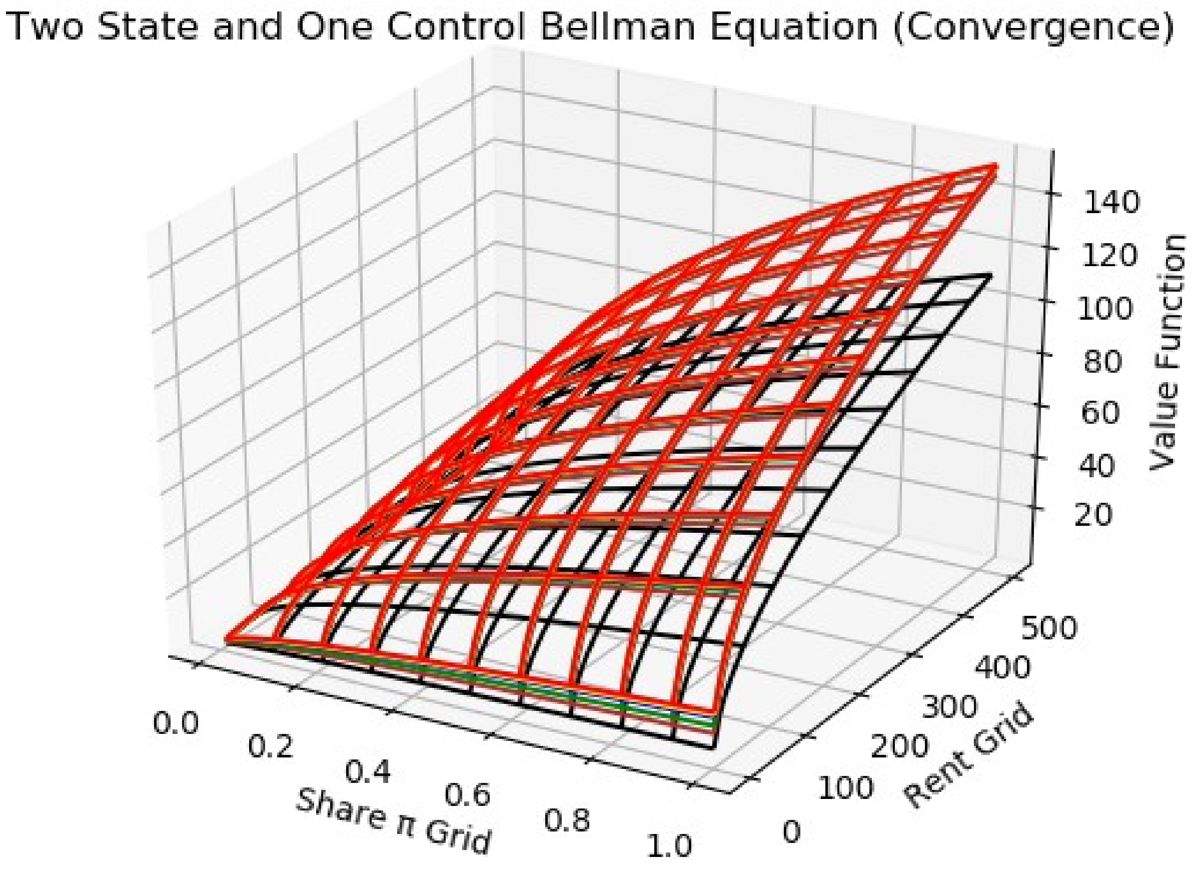
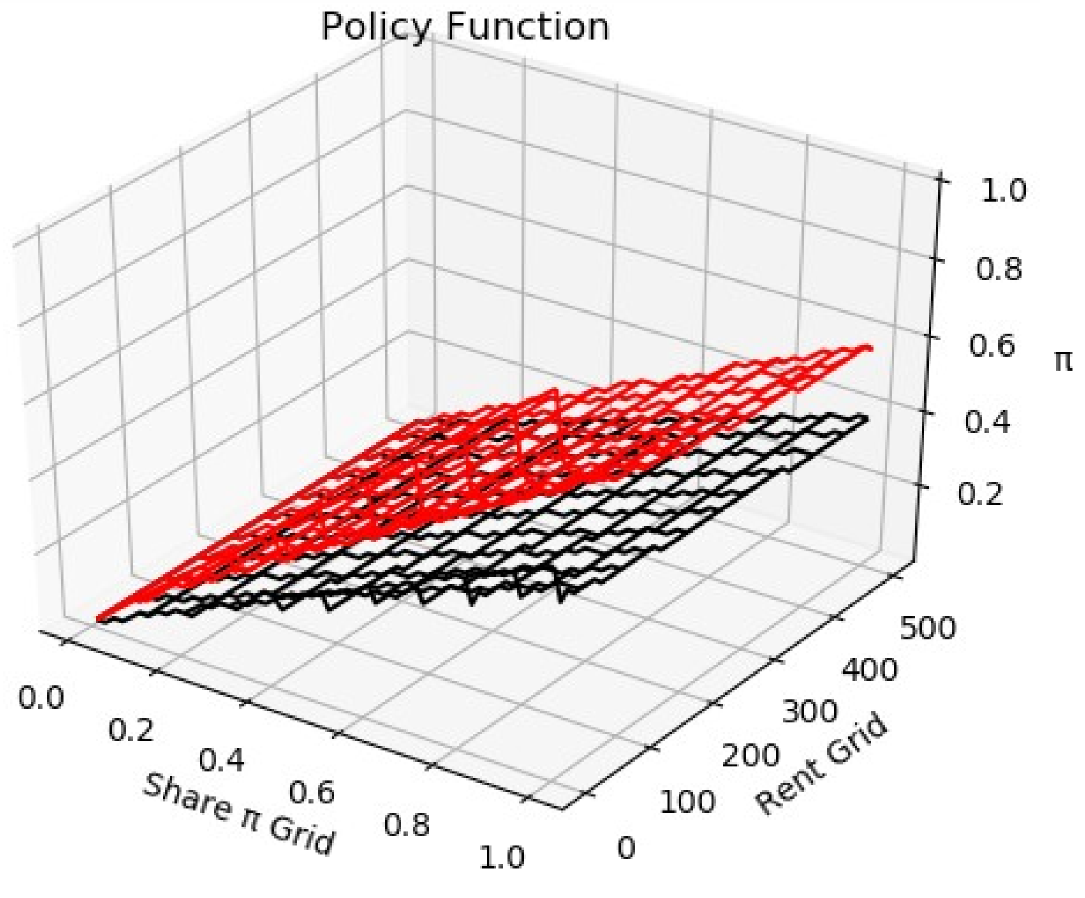

Two State and One Control Bellman Equation (Convergence)
Value Function
20
40
60
80
100
120
140
Share π Grid
0.0
0.2
0.4
0.6
0.8
1.0
Rent Grid
0
100
200
300
400
500

Figure - 2



Policy Function
0.0
0.2
0.4
0.6
0.8
1.0
Share π Grid
0
100
200
300
400
500
Rent Grid
0.2
0.4
0.6
0.8
1.0
π

Figure - 3