Arellano08_Cetin

November 28, 2022

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import random
from IPython.display import Image
from numba import njit, int64, float64, prange
from numba.experimental import jitclass
%matplotlib inline
```

Hasan Cetin Github Link

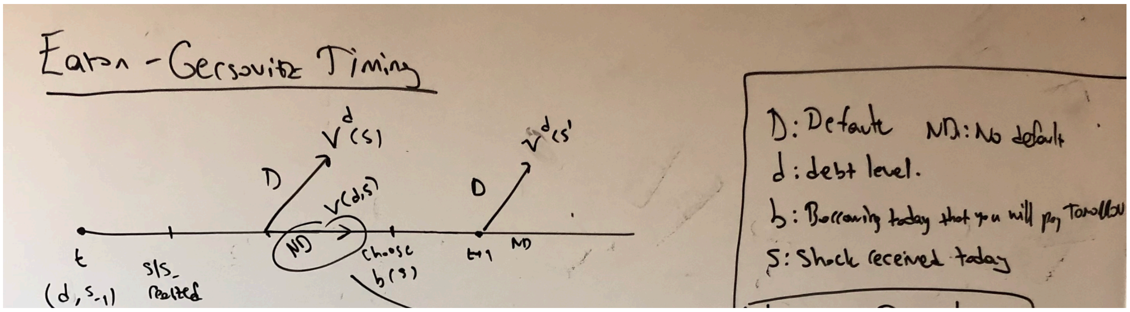
Goal In this program, we are going to replicate Arellano (08) model economy and simulate the process.

Model

- identical agents/citizens in a country (assume there is a representitative agent), having CRRA utility, having time discounting.
- One benevolent government (maximizes representitative agent's expected utility), who has the sole access to foreign assets.
- It is a stochastic endowment economy. We assume that it is an AR(1) process.
- Foreign creditors are risk neutral and have deep pocket (i.e. can buy or sell as much asset as they want). Their time discounting is $\frac{1}{(1+r)}$.
- One risk free (exogenously risk free) foreign asset. (**Exogenously incomplete market**)
- The government can default on its debt at the beginning of the period after seeing its endowment shock. (**Endogenously incomplete market**)
- If government decides to default, he will be penalized by a default output $h(y)$ until he gets forgiven by the international market.
- The defaulted government has θ probability of being forgiven each default period.

[ ]:
```
Image("EG_timing.png")
```

[ ]:



Timing and the Value functions of the Government

- It is an Eaton Gersovitz type of model.
- After seeing today's endowment y_t, at the beginning of the period, government will decide whether default on its debt or not. So, the state variables are:
  - Debt that the government has to pay today
  - Today's endowment

We can write down the government's at the beginning of the period default or no defualt decision as the following:

$$V(B, y) = \max_{D,C} \{V_C(B, y), V_D(y)\}$$

where,

$$V_D(y) = u(h(y)) + \beta \int [\theta V(0, y') + (1 - \theta)V_D(y')]\pi(y'|y)dy'$$

$$V_C(B, y) = \max_{B'} \left\{ u(y + B - q(B', y)B') + \beta \int V(B', y')\pi(y'|y)dy' \right\}$$

- V_C's C is for representing the "continuation value", i.e. not defaulting.
- If you want to borrow today, you will pay B' (< 0) tomorrow and get $q(B', y)B'$ today.
- Here, in order not to get confused, note the following: The government can save by purchasing one period foreign asset as well. i.e. $B \leq, \geq 0$ if B > 0, that means the government saved yesterday for today and he enjoys positive B, if it is negative then he borrowed yesterday and he needs to repay if he wants to continue having an access to international asset market.
- From this, we can infer that government won't default if B > 0.

Why $q(B', y)$? This is where the endogenous incompleteness comes in. Now, government can default and creditors know this, (they also know the government's endowment process). The price depends on the default probability tomorrow of the government.

The default next period can be computed as:

$$\delta(B', y) = \int \mathbb{I}\left\{V_C(B', y') < V_D(y')\right\} \pi(y'|y) dy'$$

- So, price depends on:
  - tomorrow's endowment y'
  - B'

Since B'(B,y) and since tomorrow's endowment depends on today's endowment because of $\pi(y', y)$, price depends on y as well.

Thus $q(B', y)$.

In equilibrium, our risk neutral beloved creditors will yield zero profits and this makes the price function in equilibrium as:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r}$$

0.0.1 Computation of the Equilibrium

We will define the following class and functions in order:

- Arellano (class): To store the parameters of the model.
- u: Utility function of the representative agent.
- computing_q: To compute the bond price at each state (B,y), given V_D, V_C
- T_d: To compute RHS of V_D
- T_c: To compute RHS of V_C
- update: To update values V_D, V_C and price q
- solve: Put everything that we defined above to compute the equilibrium
- simulate: After getting the policy functions from solve(), we will use them to simulate Arellano (08) Model

Parameters and boundaries Finally, we are going to use the following parameters:

- $\beta = 0.953$: Time discounting of the agent
- $\gamma = 2.000$: Risk aversion parameter of the agent
- $r = 0.017$: International risk free interest rate
- $\rho = 0.945$: Persistence of AR(1) income process
- $\mu = 0.025$: Standard deviation of AR(1) income process
- $\theta = 0.282$: Forgiveness probability of the government after default

We assume that $B \in [-0.45, 0.45]$

[ ]:
```
class Arellano:
    def __init__(self, =0.953, =2, r=0.017, =0.945, =0.025, =0.282,␣
 ↪B_grid_size=251, y_grid_size=20,
     B_min=-0.45, B_max=0.45, default_cost = 0.969, symmetric_default_cost =␣
 ↪False):
        """
        This class sets up the parameters, Bond grid, income grid and the␣
 ↪transition matrix
```

```
        One note: np.empty_like(v_c) creates an empty array that has the same dimensions as v_c

        """
        #Parameters
        self. =    #time discounting
        self. =    #risk aversion rate
        self.r = r  #risk free international interest rate
        self. =    #persistence level of income shock
        self. =    #standard deviation of income shock
        self. =    #forgiveness probability

        #Bond grid
        self.B = np.linspace(B_min,B_max,B_grid_size)  #B grid
        self.B_0_index = np.searchsorted(self.B,0)      #0 bond index, it is useful for defining after forgiveness value

        #Income grid and Transition matrix
        self.markov = qe.markov.tauchen( ,   , 0, 3, y_grid_size)
        self.Π = self.markov.P                                  #Transition probability
        self.y_grid = np.exp(self.markov.state_values) #y_grid

        #Finding Stationary Transition Matrix
        dist = 10
        x = self.Π
        while dist > 10e-7:
            y = np.dot(self.Π, x)
            x = y.copy()
            dist = np.max(np.abs(x - np.dot(x,self.Π)))
        self.Π_stationary = x.copy()

        #Adjusted y_grid after default
        if symmetric_default_cost == True:
            self.def_y_grid = self.y_grid.copy() * default_cost
        else:
            self.def_y_grid = np.minimum(default_cost * np.mean(self.y_grid), self.y_grid)

        #Initializing price and value functions
        self.v_c = np.zeros((B_grid_size,y_grid_size)) #Initial guess for V_C
        self.v_d = np.zeros(y_grid_size) #Initial guess for V_D
        self.q = np.empty_like(self.v_c) #Initial guess for prices
        self.opt_b = np.empty_like(self.v_c) #To store optimal policy function for each (B,y) pair
```

```
    def params(self):
        """A shortcut for the parameters"""
        return self. , self. , self.r, self.

    def grids(self, stationary=False):
        """A shortcut for the static grids i.e. these grids won't get any␣
↪update"""
        if stationary == False:
            return self.B, self.B_0_index, self.y_grid, self.Π, self.def_y_grid
        else:
            return self.B, self.B_0_index, self.y_grid, self.Π_stationary, self.
↪def_y_grid
```

[ ]:

```
@njit
def u(c, ):
    """The utility function of the representative agent"""
    return (c**(1- ))/(1- )
```

[ ]:

```
@njit
def computing_q(v_c, v_d, q, params, grids):
    """
    Given V_C, V_D; this function first calculates the default probability,␣
↪then wrt this probability,
    it calculates the price for each pair of (b,y)

    Reminder: Remember, we are using the same grid for B and B', and actually␣
↪we are computing the price for each pair
    of (b',y). Same thing is for the y grid. That's why when we are calculating␣
↪the default probability, when we are
    writing v_d, it is a vector of values for each income level and v_d won't␣
↪change for tomorrow,
    because we are using the same grid.
    """
     , , r,  = params
    B, B_0_index, y_grid, Π, def_y_grid = grids

    for i,b in enumerate(B):
        for j,y in enumerate(y_grid):
            default_states = (1*(v_c[i,:] < v_d)).astype(np.float64) #We␣
↪multiply with 1 to make True's one
            default_probability = (np.dot(default_states, Π[j,:])) # (B',y)
            q[i,j] = (1-default_probability)/(1+r) #Equilibrium price for B' =␣
↪B[i], y = y_grid[j]

    return q
```

```
[ ]: @njit
def T_d(y_idx, v_c, v_d, params, grids, stochastic_default=False):
    """
    This function calculates the RHS of V_D and updates V_D for a given y shock␣
 ↪(y_idx)

    Note on calculating cont_value: first we know that if the government gets␣
 ↪forgiven, then he would start
    maximization of default no-default with the state parameters (0,y'). And␣
 ↪this maximization problem is written
    down by np.max() function.

    After calculating that v, we calculate the expected value by, (\theta * v +␣
 ↪1-\theta * v_d). This is a vector for
    y'. we know that we are in y_idx). to calculate the probability of being in␣
 ↪y', we use \Pi[y_idx, :].

    Now, by * multiplication, we multiply the first vector and this transtion␣
 ↪matrix's vector ELEMENT-WISE.
    This element-wise multiplication gives the value * probability at y' given␣
 ↪y. In order to get the expected
    value, we need to sum them up. Thus, we use np.sum() to get the expected␣
 ↪value.

    """
     ,  , r,   = params
    B, B_0_index, y_grid, Π, def_y_grid = grids

    if stochastic_default == False:
        today_return = u(def_y_grid[y_idx],  )
        v = np.maximum(v_c[B_0_index, :], v_d) #Tomorrow if the government gets␣
 ↪forgiven, he would do original  max problem with 0 debt.
        cont_value = np.sum((  * v + (1- ) * v_d) * Π[y_idx,:])
        return today_return +   * cont_value
    else:
        today_return = u((def_y_grid[y_idx]/y_grid[y_idx])*np.mean(y_grid),  )
        v = np.maximum(v_c[B_0_index, :], v_d) #Tomorrow if the government gets␣
 ↪forgiven, he would do original  max problem with 0 debt.
        cont_value = np.sum((  * v + (1- ) * v_d) * Π[y_idx,:])
        return today_return +   * cont_value
```

```
[ ]: @njit
def T_c(B_idx, y_idx, v_c, v_d, q, params, grids, stochastic_default = False):
    """
    This function calculates RHS of Bellmann of Continuation value and updates␣
 ↪V_C for a given (B_idx,y_idx),
```

```
    given price function q and given value functions v_c and v_d

    Now, we need to find the optimal b'. To do that, we compute the values (current utility + cont_value) for all b'
    and get the maximum of that. The function stores the optimal b' and its value.

    Note that in the original problem, there is non-negativity constraint on consumption. To impose this constraint, we
    added all the maximization code under the if c > 0 condition so that if c <= 0 then the code won't work and those
    b's cannot be optimal.

    """
     ,  , r,   = params
    B, B_0_index, y_grid, Π, def_y_grid = grids

    value = -10e10       #aux value
    optimal_b_index = 0  #aux index

    if stochastic_default == False:
        for bp_index, bp in enumerate(B): #I use bp to remember that this is a for loop in b'
            c = y_grid[y_idx] + B[B_idx] - q[bp_index, y_idx] * B[bp_index] #Consumption when b' = B[bp_index] selected
            if c > 0: # Non-negativity constraint of consumption
                today_return = u(c,  )
                v = np.maximum(v_c[bp_index,:], v_d)
                cont_value =   * np.sum(v * Π[y_idx, :])
                if today_return + cont_value > value:    #Storing optimal value and optimal bond
                    value = today_return + cont_value
                    optimal_b_index = bp_index
        return value, optimal_b_index

    else:
        for bp_index, bp in enumerate(B): #I use bp to remember that this is a for loop in b'
            c = np.mean(y_grid) + B[B_idx] - q[bp_index, y_idx] * B[bp_index] #Consumption when b' = B[bp_index] selected
            if c > 0: # Non-negativity constraint of consumption
                today_return = u(c,  )
                v = np.maximum(v_c[bp_index,:], v_d)
                cont_value =   * np.sum(v * Π[y_idx, :])
                if today_return + cont_value > value:    #Storing optimal value and optimal bond
```

```
                    value = today_return + cont_value
                    optimal_b_index = bp_index
        return value, optimal_b_index
```

[ ]:
```
@njit
def update(v_c, v_d, q, optimal_b, params, grids, stochastic_default=False):
    """
    This function will update: 1) continuation value v_c using T_c(), 2)Default␣
 ↪value v_d using T_d()
    and 3)price function/grid q using computing_q()

    In the beginning of the period, v_c, v_d, q are given.

    We need to find q(B',y) first by using computing_q() function. Then we can␣
 ↪use this updated q to update
    v_c and v_d as well.

    We will take for loop for each pair of (B,y) to update v_c

    We need only y loop IN DEFAULT GRID to update v_d. Note that since␣
 ↪defaulted grid and normal y_grid has the same
    length, we don't need to write a seperate for loop to update v_d.
    """
     ,  , r,   = params
    B, B_0_index, y_grid, Π, def_y_grid = grids

    q_prime = computing_q(v_c,v_d,q,params,grids) #Updating q(B',y)

    v_c_prime = v_c.copy()
    v_d_prime = v_d.copy()
    for y_index, y in enumerate(y_grid):
        v_d_prime[y_index] = T_d(y_index, v_c, v_d, params, grids,␣
 ↪stochastic_default) #Updating V_D
        for b_index, b in enumerate(B):
            v_c_prime[b_index, y_index], optimal_b[b_index, y_index] =␣
 ↪T_c(b_index, y_index, v_c, v_d, q_prime, params, grids, stochastic_default)␣
 ↪#Updating V_C

    return q_prime, v_c_prime, v_d_prime, optimal_b
```

[ ]:
```
def solve(Model, tol=10e-8, max_iter=10e5, stat = False,␣
 ↪stochastic_default=False):
    """
    This function finds the equilibrium value function and pricesIt uses all␣
 ↪the classes
```

```
    and functions that we have defined above.

    tol: tolerance level of the convergence
    max_iter: maximum iteration

    First it creates the class object and unpacks all the parameters, initial␣
↪guesses and grids from the class.

    Then it uses update() and computing_q() functions to update v_c and v_d␣
↪until convergence.
    """
    #Initialization of q and optimal_policy_for_B. Initial guesses for v_c and␣
↪v_d
    q = Model.q
    v_c = Model.v_c
    v_d = Model.v_d
    opt_b = Model.opt_b

    params = Model.params()
    grids = Model.grids(stationary = stat)

    iteration = 0
    dist = np.inf
    while dist > tol and iteration < max_iter:
        q_prime, v_c_prime, v_d_prime, opt_b = update(v_c, v_d, q, opt_b,␣
↪params, grids, stochastic_default)
        dist1 = np.max(np.abs(v_c_prime - v_c))
        dist2 = np.max(np.abs(v_d_prime - v_d))
        dist = max(dist1, dist2)
        q, v_c, v_d = q_prime.copy(), v_c_prime.copy(), v_d_prime.copy()

        if iteration % 50 == 0:
            print('iteration: ', iteration)
            print('dist: ', dist)

        iteration = iteration + 1

    return q, v_c, v_d, opt_b, iteration, dist
```

[ ]:
```
Model_non_stat = Arellano()
q_ns, v_c_ns, v_d_ns, opt_b_ns, iteration_ns, dist_ns = solve(Model_non_stat)
```

```
iteration:  0
dist:  1.2697427468366411
iteration:  50
dist:  0.0937750660947394
iteration:  100
```

```
dist:  0.008340745608492739
iteration:  150
dist:  0.0007508246760004056
iteration:  200
dist:  6.763414996058259e-05
iteration:  250
dist:  6.092704030180585e-06
iteration:  300
dist:  5.488517693663653e-07
```

[ ]:
```
iteration_ns, dist_ns
```

[ ]:
```
(337, 9.700242387111757e-08)
```

[ ]:
```
Model_stationary = Arellano()

q_stat, v_c_stat, v_d_stat, opt_b_stat, iteration_stat, dist_stat =␣
 ↪solve(Model_stationary, stat=True)
```

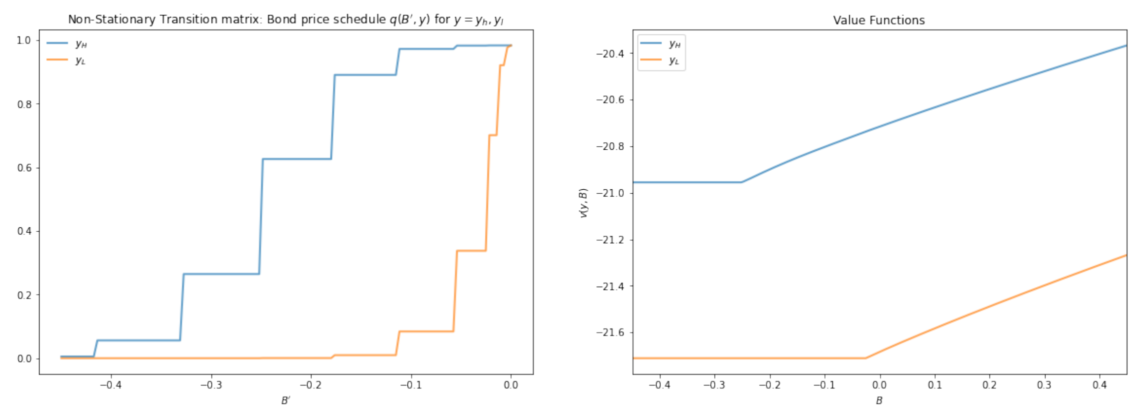
```
iteration:  0
dist:  1.2697427468366411
iteration:  50
dist:  0.0920111795847447
iteration:  100
dist:  0.00828868490611967
iteration:  150
dist:  0.0007466733693810568
iteration:  200
dist:  6.726291649172822e-05
iteration:  250
dist:  6.05927588992472e-06
iteration:  300
dist:  5.458405212266371e-07
```

[ ]:
```
iteration_stat, dist_stat
```

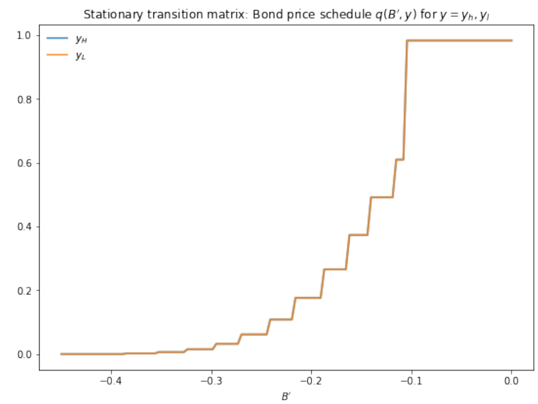
[ ]:
```
(337, 9.647398968858191e-08)
```

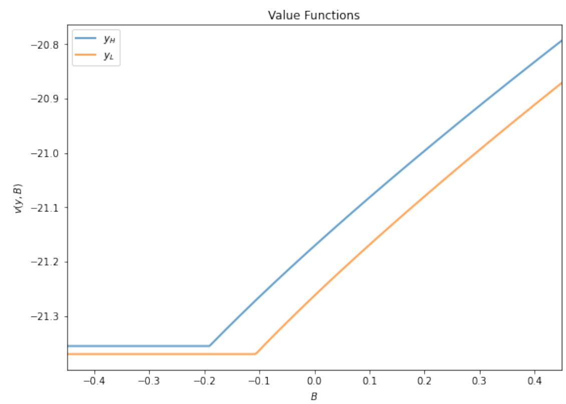
[ ]:
```
# Unpack some useful names
B_grid, y_grid, P = Model_non_stat.B, Model_non_stat.y_grid, Model_non_stat.Π
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = Model_non_stat.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
y_high_index = np.searchsorted(y_grid, high)
y_low_index = np.searchsorted(y_grid, low)
```

```
fig, ax = plt.subplots(1,2,figsize=(20, 6.5))

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.45 <= B <= 0:           #We know that if B>0 then q = 1/(1+r), not interesting.
        x.append(B)
        q_low.append(q_ns[i, y_low_index])
        q_high.append(q_ns[i, y_high_index])
ax[0].set_title("Non-Stationary Transition matrix: Bond price schedule $q(B', y)$ for $y = y_h, y_l$")
ax[0].plot(x, q_high, label="$y_H$", lw=2, alpha=0.7)
ax[0].plot(x, q_low, label="$y_L$", lw=2, alpha=0.7)
ax[0].set_xlabel("$B'$")
ax[0].legend(loc='upper left', frameon=False)


v = np.maximum(v_c_ns, np.reshape(v_d_ns, (1, y_grid_size)))

ax[1].set_title("Value Functions")
ax[1].plot(B_grid, v[:, y_high_index], label="$y_H$", lw=2, alpha=0.7)
ax[1].plot(B_grid, v[:, y_low_index], label="$y_L$", lw=2, alpha=0.7)
ax[1].legend(loc='upper left')
ax[1].set(xlabel="$B$", ylabel="$v(y, B)$")
ax[1].set_xlim(min(B_grid), max(B_grid))

plt.show()
```



[ ]:
```
# Unpack some useful names
B_grid, y_grid, P = Model_stationary.B, Model_stationary.y_grid,␣
 ↪Model_stationary.Π
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = Model_non_stat.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
y_high_index = np.searchsorted(y_grid, high)
y_low_index = np.searchsorted(y_grid, low)

fig, ax = plt.subplots(1,2, figsize=(20, 6.5))

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.45 <= B <= 0:          #We know that if B>0 then q = 1/(1+r), not␣
 ↪interesting.
        x.append(B)
        q_low.append(q_stat[i, y_low_index])
        q_high.append(q_stat[i, y_high_index])
ax[0].set_title("Stationary transition matrix: Bond price schedule $q(B', y)$␣
 ↪for $y = y_h, y_l$")
ax[0].plot(x, q_high, label="$y_H$", lw=2, alpha=0.7)
ax[0].plot(x, q_low, label="$y_L$", lw=2, alpha=0.7)
ax[0].set_xlabel("$B'$")
ax[0].legend(loc='upper left', frameon=False)

v = np.maximum(v_c_stat, np.reshape(v_d_stat, (1, y_grid_size)))

ax[1].set_title("Value Functions")
ax[1].plot(B_grid, v[:, y_high_index], label="$y_H$", lw=2, alpha=0.7)
ax[1].plot(B_grid, v[:, y_low_index], label="$y_L$", lw=2, alpha=0.7)
ax[1].legend(loc='upper left')
ax[1].set(xlabel="$B$", ylabel="$v(y, B)$")
ax[1].set_xlim(min(B_grid), max(B_grid))

plt.show()
```





In stationary transition matrix we have the same price function for both y_h and y_l because in stationary process it becomes iid process (i.e. every row of the stationary transition matrix is the same).

Simulation Let's simulate the model.

```
[ ]: #@njit
def Simulation(Model,q, v_c, v_d, opt_b, T = 100000, inherit_debt=False):
    """
    This function simulates output realization, default decision according to this realization,
    bond selection if not defaulted, price function, and will find how long the economy will remain in its default state.

    T: Simulation length
    Model is needed to have parameters and grids
    q, v_c, v_d, opt_b are obtained after solving the model


    mc = qe.MarkovChain(Π, y_grid) creates a class of Markov chain with transition matrix Π and state values y_grid
    Then, mc.simulate_indices(T, init_idx) simulates of the process that started from y_grid[init_idx] and returns the
    simulation INDICES from y_grid.

    Since y_grid and def_y_grid have the same dimensions, even if the economy is in default phase, we can use this
    simulated indices to find the stochastic income that the government has.

    The economy starts with middle shock y_0 and zero debt B[B_0_index]

    Our variables, y_sim_indices stores the indices instead of values.

```

```
    But y_sim and b_sim store the values.

    d_sim will show the default periods of the economy; if d_sim[t] == 1, then␣
↪it means at period t, the country is in
    default phase, no foreign asset access.

    Note that since in default phase the government cannot access foreign␣
↪bonds, B' = 0 in default cases.
    """

    #Getting the necessary parameters, grids and transition matrix
    y_grid = Model.y_grid          #y_grid
    def_y_grid = Model.def_y_grid #default_y_grid
    Π = Model.Π                   #Transition Matrix
    zero_debt_index = Model.B_0_index #zero debt index
    B = Model.B                   #Bond grid
      = Model.                    #Forgiveness probability

    #Simulation of income shocks
    mc = qe.MarkovChain(Π, y_grid)
    y_sim_indices = mc.simulate_indices(T, init = np.searchsorted(y_grid, np.
↪mean(y_grid)))  #We assumed that
                                                                            #the␣
↪simulation started from the middle shock

    #Creating simulation arrays
    b_sim_indices = np.empty(T)  #Bond selection simulation
    b_sim_indices[0] = zero_debt_index #We start with zero debt (assumption)

    y_sim = np.empty(T)  #endowment simulation
    b_sim = np.empty(T)  #bond simulation
    b_sim[0] = B[zero_debt_index] #Starts with zero debt
    if inherit_debt == True:
        b_sim[0] = B[50] #Starts with a debt
    q_sim = np.empty(T)  #Price simulation
    d_sim = np.zeros(T)  #Default phase simulation (if 1 then economy is in␣
↪default)
    def_decision = np.zeros(T)

    t = 0              #Initial period
    in_default = False #We start with no default environment

    while t<T-1:
        #At time t, we are at (b_idx,y_idx) and at t=0, we are at␣
↪(zero_debt_index, y_sim[0])
        y_idx = y_sim_indices[t].astype(np.int32)
```

```
        b_idx = b_sim_indices[t].astype(np.int32)

        #Price simulation
        #q_sim[t] = q[b_idx, y_idx]

        #Default case:
        if v_c[b_idx, y_idx] < v_d[y_idx] or in_default:
            if in_default == False:
                def_decision[t] = 1
            in_default = True
            b_sim_indices[t+1] = zero_debt_index
            b_sim[t+1] = B[b_sim_indices[t+1].astype(np.int32)]
            y_sim[t] = def_y_grid[y_idx]
            d_sim[t] = 1

            if np.random.uniform() <  :
                in_default = False
        #No-Default case
        else:
            b_sim_indices[t+1] = opt_b[b_idx, y_idx]
            b_sim[t+1] = B[b_sim_indices[t+1].astype(np.int32)]
            y_sim[t] = y_grid[y_idx]
            d_sim[t] = 0


        t = t + 1

    return q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices,␣
↪b_sim_indices
```

0.1 Question 1: Ergodic distribution of endowment and assets

Let's first find our equilibrium price and value functions

[ ]:
```
Model_regular = Arellano()

Bond_grid = Model_regular.B
y_grid = Model_regular.y_grid
def_y_grid = Model_regular.def_y_grid
```

[ ]:
```
q_regular, v_c_regular, v_d_regular, opt_b_regular, iteration_regular,␣
↪dist_regular = solve(Model_regular)
```

```
iteration:  0
dist:  1.2697427468366411
iteration:  50
dist:  0.0937750660947394
iteration:  100
```

```
dist:  0.008340745608492739
iteration:  150
dist:  0.0007508246760004056
iteration:  200
dist:  6.763414996058259e-05
iteration:  250
dist:  6.092704030180585e-06
iteration:  300
dist:  5.488517693663653e-07
```

Let's simulate the process. We set T = 5 * 10e5 as in paper to see the ergodic behaivor.

[ ]:
```python
q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices =␣
↪Simulation(Model_regular,q_regular, v_c_regular, v_d_regular, opt_b_regular,␣
↪T=500000)
```

[ ]:
```python
y_sim_indices, b_sim_indices = y_sim_indices.astype(int), b_sim_indices.
↪astype(int)
```

Let's find the probability of each (b,y) pair:

[ ]:
```python
#@njit
def find_asset_end_dist(B_grid, y_grid, T, y_sim_indices, b_sim_indices):
    Λ = np.zeros((len(B_grid),len(y_grid)))

    for i in range(len(b_sim_indices)):
            Λ[b_sim_indices[i],y_sim_indices[i]] =␣
↪Λ[b_sim_indices[i],y_sim_indices[i]]+1
    return Λ/T
```

[ ]:
```python
Λ_regular =  find_asset_end_dist(Bond_grid, y_grid, 500000,y_sim_indices,␣
↪b_sim_indices)
Λ_regular.shape
```

[ ]:
```
(251, 20)
```

[ ]:
```python
Λ_regular
```

[ ]:
```
array([[0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.]])
```

[ ]:
```python
np.max(Λ_regular)
```

[ ]: 0.061036

[ ]:
```python
np.argmax(Λ_regular)

np.unravel_index(np.argmax(Λ_regular), Λ_regular.shape)
```

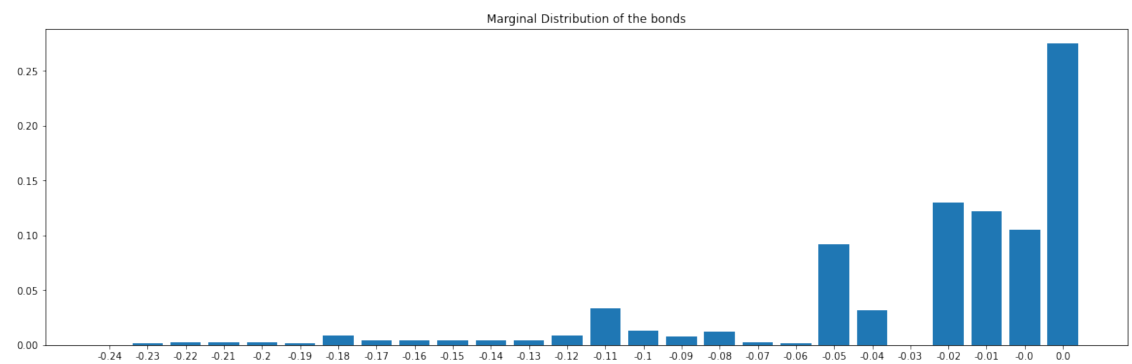
[ ]: (125, 6)

The most frequent (b,y) pair happended was B = 0, y = 0.91. It happened 6.1% of the time. And we can see that the marginal cases when B = -0.45 or B = 0.45 never happened.

0.2 Question 2

[ ]:
```python
unique, counts = np.unique(b_sim, return_counts=True)
```

[ ]:
```python
fig, ax = plt.subplots(figsize=(20,6))

ax.bar(list(np.round(unique,2).astype(str)), list(counts/500000))
ax.set_title('Marginal Distribution of the bonds')
plt.show()
```



As you can see, the government did not save even once over time.

0.3 Question 3

[ ]:
```python
print("The fraction of time the economy stayed in default state: ",np.
↪sum(d_sim)/500000)
print("The fraction of time that the government decided to default while he is␣
↪in no-default state: ", np.sum(def_decision)/(500000 - np.sum(d_sim) + np.
↪sum(def_decision)))
```

```
The fraction of time the economy stayed in default state:  0.02107
The fraction of time that the government decided to default while he is in no-
default state:  0.006087766999754296
```

Economy stayed around 2.1% of the time in default state. While the government was in no-default state, he chose .6% time to default.

```
[ ]: print("Average level of debt: ",np.mean(b_sim))
     print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  -0.03869674560000002
Maximum level of debt:  -0.2376
```

0.4 Question 4

To calculate average output loss, we will do the following:

- Calculate the average output in no-default states, take log of it
- Subtract it from the logged output of each default state
- Take an average.

We are taking log-dif to calculate the average loss percentage.

```
[ ]: avg_output_no_def = np.mean(y_grid[y_sim_indices[d_sim!=1]])

     def_output = def_y_grid[y_sim_indices[d_sim!=0]]

     avg_loss = np.mean(np.log(def_output) - np.log(avg_output_no_def))

     avg_loss
```

```
[ ]: -0.06820515473426426
```

There is 6.8% output loss in average.

0.5 Question 5: Symmetric default cost case

```
[ ]: symmetric_cost_model = Arellano(symmetric_default_cost=True, default_cost=0.98)

     q_sym, v_c_sym, v_d_sym, opt_b_sym, iteration_sym, dist_sym = 
      ↪solve(symmetric_cost_model)
```

```
iteration:  0
dist:  1.2833979223252077
iteration:  50
dist:  0.09364797713606521
iteration:  100
dist:  0.008329886054259106
iteration:  150
dist:  0.0007498494960l0313
iteration:  200
dist:  6.754631804994915e-05
iteration:  250
dist:  6.084791905891507e-06
```

```
iteration:  300
dist:  5.481390132899833e-07
```

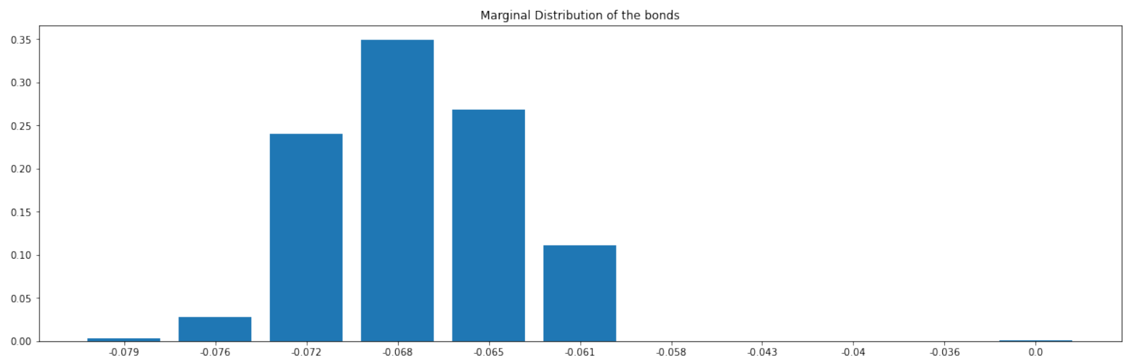
```
[ ]: q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices =␣
      ↪Simulation(symmetric_cost_model,q_sym, v_c_sym, v_d_sym, opt_b_sym, T=500000)
```

```
[ ]: print("Average level of debt: ",np.mean(b_sim))
     print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  -0.0677112984
Maximum level of debt:  -0.07920000000000005
```

The average debt increased but the maximum level of debt decreased substantially, i.e. the government borrowed less but frequent asset comparing to the non-symmetric default cost case.

```
[ ]: unique, counts = np.unique(b_sim, return_counts=True)
     fig, ax = plt.subplots(figsize=(20,6))

     ax.bar(list(np.round(unique,3).astype(str)), list(counts/500000))
     ax.set_title('Marginal Distribution of the bonds')
     plt.show()
```




```
[ ]: print("The fraction of time the economy stayed in default state: ",np.
      ↪sum(d_sim)/500000)
     print("The fraction of time that the government decided to default while he is␣
      ↪in no-default state: ", np.sum(def_decision)/(500000 - np.sum(d_sim) + np.
      ↪sum(def_decision)))
```

```
The fraction of time the economy stayed in default state:  0.000398
The fraction of time that the government decided to default while he is in no-
default state:  0.00010203020093947808
```

0.6 Question 6

The main problem of the symmetric default cost scheme is that the default state almost does not appear in equilibrium. That's why Arellano used non-symmetric default cost scheme to make default states appear more frequently in equilibrium.

0.7 Question 7: $\gamma = 10$ case

[ ]:
```
model_10 = Arellano( = 10)
q_10, v_c_10, v_d_10, opt_b_10, iteration_10, dist_10 = solve(model_10)
```

```
iteration:  0
dist:  0.9532331929817328
iteration:  50
dist:  0.01851523060992122
iteration:  100
dist:  0.001412102889331912
iteration:  150
dist:  0.00012589361176651437
iteration:  200
dist:  1.1334182204336685e-05
iteration:  250
dist:  1.0209880656475434e-06
```

[ ]:
```
q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices =␣
↪Simulation(model_10,q_10, v_c_10, v_d_10, opt_b_10, T=500000)
```

[ ]:
```
print("Average level of debt: ",np.mean(b_sim))
print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  0.12270805199999996
Maximum level of debt:  0.0
```

As you can see, the govenment did not even borrow once over time. When the govenment (actually the representitative agent) becomes too risk averse, the government starts to become paranoid about future consumption and he is willing to consume less today by saving more for tomorrow.

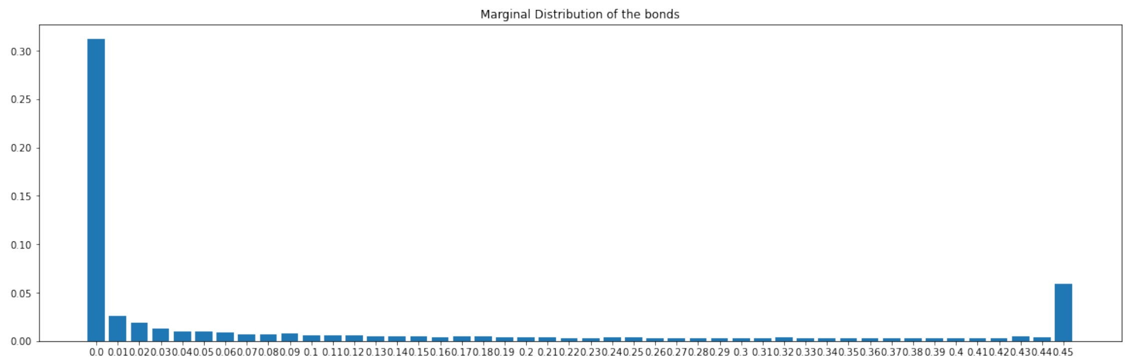
[ ]:
```
print("The fraction of time the economy stayed in default state: ",np.
↪sum(d_sim)/500000)
print("The fraction of time that the government decided to default while he is␣
↪in no-default state: ", np.sum(def_decision)/(500000 - np.sum(d_sim) + np.
↪sum(def_decision)))
```

```
The fraction of time the economy stayed in default state:  0.0
The fraction of time that the government decided to default while he is in no-
default state:  0.0
```

It is obvious that the government did not choose to default even once, because he did not even borrow once.

[ ]:
```
unique, counts = np.unique(b_sim, return_counts=True)
fig, ax = plt.subplots(figsize=(20,6))
```

```python
ax.bar(list(np.round(unique,2).astype(str)), list(counts/500000))
ax.set_title('Marginal Distribution of the bonds')
plt.show()
```



0.8 Question 8: $\beta\frac{1}{1+r} = 1$ case

[ ]:
```python
1/(1+model_10.r)
```

[ ]:
```
0.9832841691248771
```

We will choose β be above such that both lenders and the government discount future at the same rate.

[ ]:
```python
model_same_discount = Arellano( =0.983)

q_disc, v_c_disc, v_d_disc, opt_b_disc, iteration_disc, dist_disc = solve(model_same_discount)
```

```
iteration:  0
dist:  1.2697427468366411
iteration:  50
dist:  0.4341993222090821
iteration:  100
dist:  0.18185959451324152
iteration:  150
dist:  0.0771048502170828
iteration:  200
dist:  0.03271439381623509
iteration:  250
dist:  0.013880774996898992
iteration:  300
dist:  0.005889650809251634
iteration:  350
dist:  0.002498995275040272
```

```
iteration:  400
dist:  0.0010603306770491372
iteration:  450
dist:  0.0004499012688867765
iteration:  500
dist:  0.00019089436544561522
iteration:  550
dist:  8.099701265251724e-05
iteration:  600
dist:  3.436725879168989e-05
iteration:  650
dist:  1.4582123952777692e-05
iteration:  700
dist:  6.187235953802883e-06
iteration:  750
dist:  2.6252615299426907e-06
iteration:  800
dist:  1.1139058173625926e-06
iteration:  850
dist:  4.726333742155475e-07
iteration:  900
dist:  2.0053968086131135e-07
```

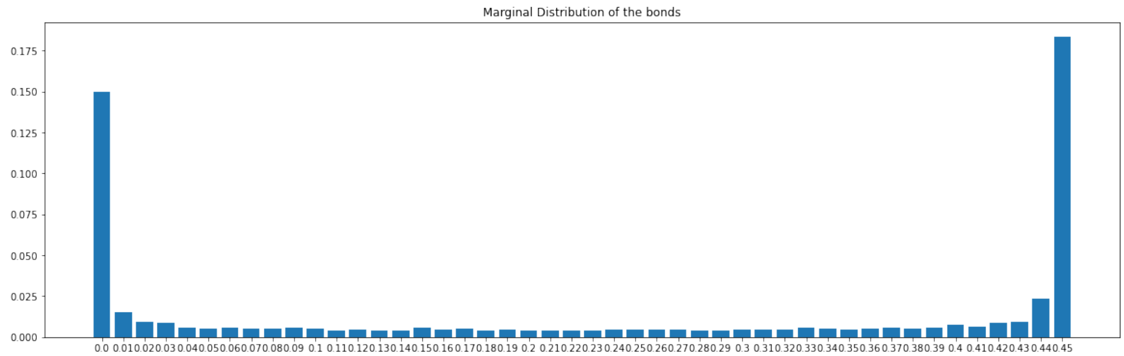
[ ]:
```
q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices =␣
  ↪Simulation(model_same_discount,q_disc, v_c_disc, v_d_disc, opt_b_disc,␣
  ↪T=500000)
```

[ ]:
```
print("Average level of debt: ",np.mean(b_sim))
print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  0.23637183119999997
Maximum level of debt:  0.0
```

Now, the government values future more than before. That's why he started to do savings. The interesting result is that now, he does not borrow even once and the marginal distribution of the bonds graph looks very similar to $\gamma = 10$ case. As you can see, the demand for the maximum bond is too much, showing that our bond upper limit is binding around 17.5% of the time. If we increase the saving limit, it would decrease and in the next section, we increase the upper limit from 0.45 to 5 to see this result.

[ ]:
```
unique, counts = np.unique(b_sim, return_counts=True)
fig, ax = plt.subplots(figsize=(20,6))

ax.bar(list(np.round(unique,2).astype(str)), list(counts/500000))
ax.set_title('Marginal Distribution of the bonds')
plt.show()
```




```
[ ]: print("The fraction of time the economy stayed in default state: ",np.
      ↪sum(d_sim)/500000)
     print("The fraction of time that the government decided to default while he is␣
      ↪in no-default state: ", np.sum(def_decision)/(500000 - np.sum(d_sim) + np.
      ↪sum(def_decision)))
```

```
The fraction of time the economy stayed in default state:  0.0
The fraction of time that the government decided to default while he is in no-
default state:  0.0
```

It is obvious that the government did not choose to default even once, because he did not even borrow once.

Higher saving limit case

```
[ ]: model_same_discount = Arellano( =0.983, B_max=5, B_min=-5)
     q_disc, v_c_disc, v_d_disc, opt_b_disc, iteration_disc, dist_disc =␣
      ↪solve(model_same_discount)
```

```
iteration:  0
dist:  1.405473329893832
iteration:  50
dist:  0.4641937544334809
iteration:  100
dist:  0.19264215877370816
iteration:  150
dist:  0.08143729222249618
iteration:  200
dist:  0.034489760741422515
iteration:  250
dist:  0.014618578960053696
iteration:  300
dist:  0.006199150613632298
iteration:  350
dist:  0.002629521573581428
iteration:  400
```

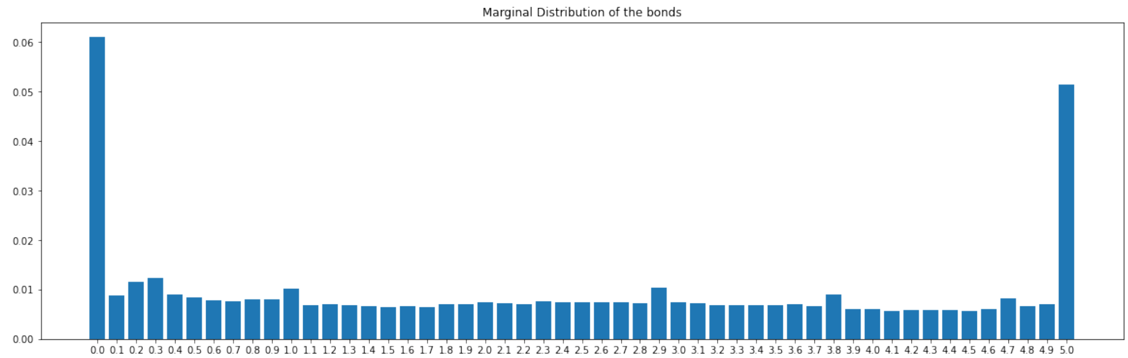
```
dist:  0.0011155367553001838
iteration:  450
dist:  0.00047328627507425836
iteration:  500
dist:  0.00020080803867728036
iteration:  550
dist:  8.520149749102757e-05
iteration:  600
dist:  3.61508097697083e-05
iteration:  650
dist:  1.53387958405915e-05
iteration:  700
dist:  6.508273180827473e-06
iteration:  750
dist:  2.761473908208245e-06
iteration:  800
dist:  1.1717000916178222e-06
iteration:  850
dist:  4.971554119492794e-07
iteration:  900
dist:  2.109443926201493e-07
```

[ ]:
```
q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices =␣
↪Simulation(model_same_discount,q_disc, v_c_disc, v_d_disc, opt_b_disc,␣
↪T=500000)
```

[ ]:
```
print("Average level of debt: ",np.mean(b_sim))
print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  2.3607707199999997
Maximum level of debt:  0.0
```

[ ]:
```
unique, counts = np.unique(b_sim, return_counts=True)
fig, ax = plt.subplots(figsize=(20,6))

ax.bar(list(np.round(unique,1).astype(str)), list(counts/500000))
ax.set_title('Marginal Distribution of the bonds')
plt.show()
```



As you can see, the increase of upper limit decreased the marginal distribution of saving in the upper limit from 17.5% to around 5% but there is still room for improvement by increasing the upper limit. But 0 bond's marginal distribution also decreased and all the other bond's marginal distribution increased. This hints that in limit this will become a uniform distribution.

0.9 Question 9: $y(s) = \bar{y}$ in repayment but $\hat{y}^d(s) = \frac{y^d(s)}{y(s)}\bar{y}$

The reason that government wants to remain in the international market is that in autarky, he is susceptible to income shocks, and since the representitative agent in his country is a risk averse person, the government wants to smooth his citizen's consumption **across states** within each period. And this repayment scheme is doing exactly that. So my guess is that the government will default less in such default cost scheme.

The change is the following: - Now, for each y shock, the V_C will be the same for the same B'. Normally, it is $V_C(B)$ a function of B only, but to make the dimensions work,we still use $V_C(B, y)$ structure but changed T_c and T_d functions accordingly by adding `stochastic_default` variable.

```
[ ]: Model_stochastic_default = Arellano()

     q_sd, v_c_sd, v_d_sd, opt_b_sd, iteration_sd, dist_sd =␣
      ↪solve(Model_stochastic_default, stochastic_default=True)
```

```
iteration:  0
dist:  1.2731119660435584
iteration:  50
dist:  0.09124941625193728
iteration:  100
dist:  0.0082187091682151
iteration:  150
dist:  0.0007403633578846325
iteration:  200
dist:  6.669445674489793e-05
iteration:  250
dist:  6.008066904428233e-06
```

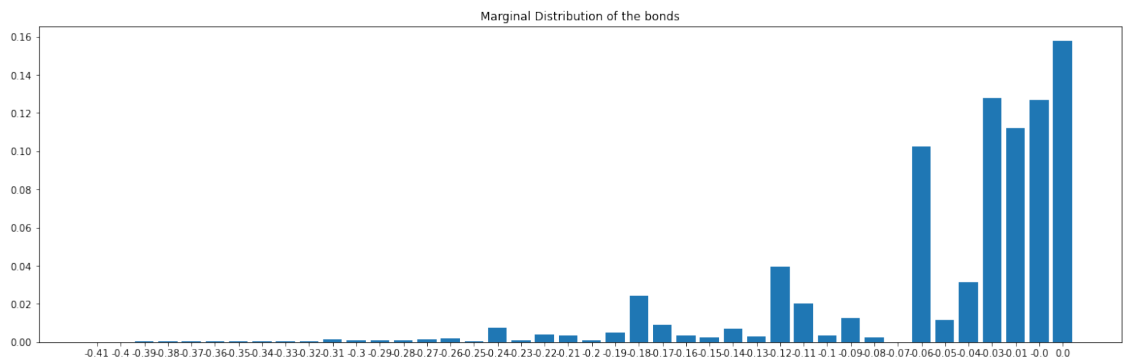
```
iteration:  300
dist:  5.412274362015523e-07
```

[ ]:
```python
q_sim, y_sim, b_sim, d_sim, def_decision, y_sim_indices, b_sim_indices = Simulation(Model_stochastic_default,q_sd, v_c_sd, v_d_sd, opt_b_sd, T=500000)
```

[ ]:
```python
print("Average level of debt: ",np.mean(b_sim))
print("Maximum level of debt: ",np.min(b_sim))
```

```
Average level of debt:  -0.04830910560000003
Maximum level of debt:  -0.4104
```

[ ]:
```python
unique, counts = np.unique(b_sim, return_counts=True)
fig, ax = plt.subplots(figsize=(20,6))

ax.bar(list(np.round(unique,2).astype(str)), list(counts/500000))
ax.set_title('Marginal Distribution of the bonds')
plt.show()
```



[ ]:
```python
print("The fraction of time the economy stayed in default state: ",np.sum(d_sim)/500000)
print("The fraction of time that the government decided to default while he is in no-default state: ", np.sum(def_decision)/(500000 - np.sum(d_sim) + np.sum(def_decision)))
```

```
The fraction of time the economy stayed in default state:  0.027696
The fraction of time that the government decided to default while he is in no-default state:  0.007867234549710514
```

Results:

- There is higher borrowing, and higher default state time fraction rate.
- One reason is that the default cost is not high enough: for lower shocks it will be $y^D(s) = y(s)$ that will make $\hat{y}^D(s) = \frac{y^D(s)}{y(s)}\bar{y} = \bar{y}$, which is what he would also get if he does not default; so

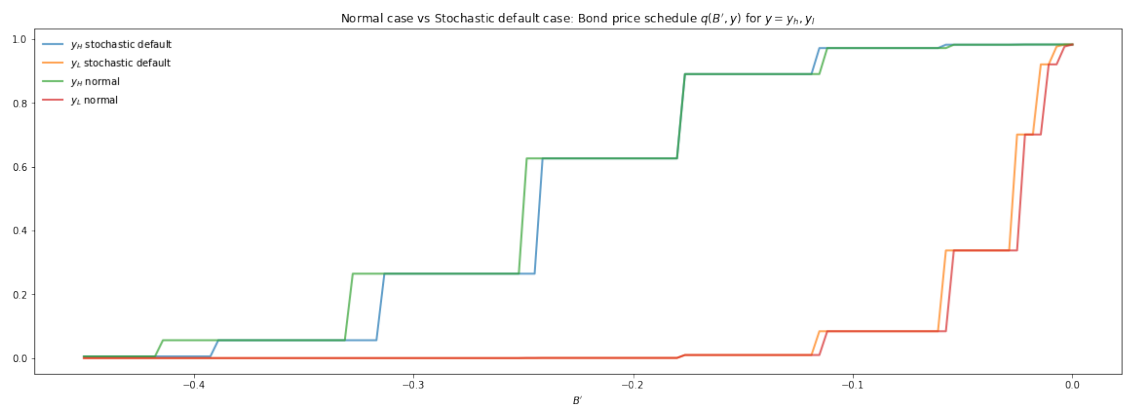
the agent is not being punished to default enough.

- So my initial guess was wrong. In this scheme, the government borrows more because he does not fear to default much.

In order to understand this result a little bit deeper, let's compare the price functions in equilibrium for high and low endowment cases:

[ ]:
```
# Unpack some useful names
B_grid, y_grid, P = Model_stochastic_default.B, Model_stochastic_default.
↪y_grid, Model_stochastic_default.Π
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = Model_stochastic_default.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
y_high_index = np.searchsorted(y_grid, high)
y_low_index = np.searchsorted(y_grid, low)

fig, ax = plt.subplots(figsize=(20, 6.5))

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.45 <= B <= 0:           #We know that if B>0 then q = 1/(1+r), not␣
↪interesting.
        x.append(B)
        q_low.append(q_sd[i, y_low_index])
        q_high.append(q_sd[i, y_high_index])
ax.set_title("Normal case vs Stochastic default case: Bond price schedule␣
↪$q(B', y)$ for $y = y_h, y_l$")
ax.plot(x, q_high, label="$y_H$ stochastic default", lw=2, alpha=0.7)
ax.plot(x, q_low, label="$y_L$ stochastic default", lw=2, alpha=0.7)
ax.set_xlabel("$B'$")



# Unpack some useful names
B_grid, y_grid, P = Model_non_stat.B, Model_non_stat.y_grid, Model_non_stat.Π
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = Model_non_stat.r

# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
y_high_index = np.searchsorted(y_grid, high)
y_low_index = np.searchsorted(y_grid, low)
```

```python
# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.45 <= B <= 0:             #We know that if B>0 then q = 1/(1+r), not␣
 ↪interesting.
        x.append(B)
        q_low.append(q_ns[i, y_low_index])
        q_high.append(q_ns[i, y_high_index])
ax.plot(x, q_high, label="$y_H$ normal", lw=2, alpha=0.7)
ax.plot(x, q_low, label="$y_L$ normal", lw=2, alpha=0.7)
ax.legend(loc='upper left', frameon=False)

plt.show()
```



The graph above shows us some interesting but hard to understand results:

- The price for 1.05 * mean endowment (y_H) mostly shifted to right. That means, lenders think that (after solving default probability) the government tend to default more on its big debts.
- But the prices for 0.95 * mean endowment (y_L) and price of y_H after debt 0.15 shifted to left.
- This shows that the government tends to default on its high debts more than before but at the same time he is willing to pay its lower debt in order not to enter default state. I'm having hard time to understand why this is the case.
- But the shape of prices did not change much because we have almost identical problem apart from the default cost scheme