Aiyagari_diffusion

November 18, 2021

[1]:
```
globals().clear()

import numpy as np
from numpy.core.fromnumeric import transpose
from scipy import sparse
import matplotlib.pyplot as plt
from scipy.linalg import block_diag
#np.set_printoptions(formatter={'float': lambda x: "{0:0.4f}".format(x)})
#np.set_printoptions(precision=3)


# ------------------------------------------------------------------------ #
#                                  Parameters                              #
# ------------------------------------------------------------------------ #
alpha = 0.35#1/3,
beta  = 1.05**(-1/4)
rho   = 0.05#1-
delta = 0.1
gamma = 2
phi   = -1
Delta = 1000
Na=100 #100
Nz=40 #40

# Grid of assets
a_max  = 30
a_min  = phi
a_grid = np.array(np.linspace(a_min,a_max,num=Na) )
da = a_grid[2]-a_grid[1]

AA = np.tile(a_grid, (Nz,1)).T
#A = np.tile(a_grid.reshape(-1,1), (1,Nz))


# Grid of productivity shock
z_min = 0.5   # Range z
z_max = 1.5
```

```
z_grid = np.array(np.linspace(z_min,z_max,num=Nz) )
dz = z_grid[2] - z_grid[1]
dz2 =  dz**2
ZZ = np.tile(z_grid, (Na,1))


# ------------------------------------------------------------------------ #
#            Ornstein-Uhlebeck process: dX_t = (X - X_t)dt + dW_t          #
# ------------------------------------------------------------------------ #
muz_bar = 1;      # mean O-U process (in levels). This parameter has to be␣
 ↪adjusted to ensure that the mean of z (truncated gaussian) is 1.
sigmaz = 0.10;  # sigma O-U
thetaz = 0.3

muz  = thetaz*(muz_bar - z_grid) #DRIFT (FROM ITO'S LEMMA)
varz = np.repeat(sigmaz**2,Nz)    #VARIANCE (FROM ITO'S LEMMA)


# ------------------------------------------------------------------------ #
#                                  Derivatives                             #
# ------------------------------------------------------------------------ #
# Convert          V = u(c) + a V_a +  _z V_z + (1/2) 2 V_zz
# into             V = u(c) + a D_a*V +  _z D_z*V + (1/2) 2 D_zz*V
# and iterate over V_{n+1} = u(c_n(V_n))[  - aD_a -  _zD_z - (1/2) 2*D_zz] -1

#Derivatives on z: ( _z*D_z + (1/2)* 2*D_zz) * V

#First Derivative
D_z = np.zeros((Nz,Nz))
for j in range(1,Nz-1):
        if muz[j] >= 0: #OJO con el >=, pq no estoy seguro
            D_z[j,j]   = muz[j] * (-1)/dz
            D_z[j,j+1] = muz[j] * 1/dz
        elif muz[j] < 0:
            D_z[j,j]   = muz[j] * 1/dz
            D_z[j,j-1] = muz[j] * (-1)/dz

if muz[0] >= 0:
        D_z[0,0]   = muz[0] * (-1)/dz
        D_z[0,0+1] = muz[0] * 1/dz
elif muz[0] < 0:
        D_z[0,0]   = muz[0] * 1/dz

if muz[Nz-1] >= 0:
        D_z[Nz-1,Nz-1] = muz[0] * (-1)/dz
elif muz[Nz-1] < 0:
        D_z[Nz-1,Nz-1] = muz[Nz-1] * 1/dz
```

```
        D_z[Nz-1,Nz-2] = muz[Nz-1] * (-1)/dz

#Second Derivative
D_zz = np.zeros((Nz,Nz))
for j in range(1,Nz-1):
        D_zz[j,j]   = (varz[j]/2)* (-2)/dz**2
        D_zz[j,j+1] = (varz[j]/2)* 1/dz**2
        D_zz[j,j-1] = (varz[j]/2)* 1/dz**2

D_zz[0,0]   = (varz[0]/2)* (-2/dz**2) /2
D_zz[0,0+1] = (varz[0]/2)* (1/dz**2)

D_zz[Nz-1,Nz-1] = (varz[Nz-1]/2)* (-2/dz**2)  /2
D_zz[Nz-1,Nz-2] = (varz[Nz-1]/2)* (1/dz**2)
DZ = D_z + D_zz

#expanding on a-dimension
Aswitch = np.kron(DZ,np.identity(Na))

#Derivative on a
#First Derivative
D_af = np.zeros((Na,Na))
D_ab = np.zeros((Na,Na))

for j in range(0,Na-1):
        D_af[j,j]   = -1/da
        D_af[j,j+1] = 1/da
D_af[Na-1,Na-1] = -1

for j in range(1,Na):
        D_ab[j,j]   = 1/da
        D_ab[j,j-1] = -1/da
D_ab[0,0] = 1


# ------------------------------------------------------------------------ #
#                                Initial Values                            #
# ------------------------------------------------------------------------ #
def utility(c):
    if gamma!=1:
        u = (c**(1-gamma)) / (1-gamma)
    else:
        u = np.log(c)
    return u

def price_r(K):
    r = alpha * K**( alpha-1) - delta
```

```
    return r

def price_w(K):
    w = (1-alpha) * K**alpha
    return w

def consumption(dV):
    '''
        FOC:
            du/dc = dv/da
            c^(1/gamma) = dv/da
            c=  (dv/da)^(-1/gamma)
    '''
    c = dV**(-1/gamma)
    return c

c = np.zeros((Na,Nz))
K = 3.8
r = price_r(K)
w = price_w(K)

V0 = utility( w*ZZ + r*AA) /rho
```

[2]:
```
# ------------------------------------------------------------------------ #
#                                   MAIN LOOP                              #
# ------------------------------------------------------------------------ #

#HJB iter
iterHJB=0
maxiterHJB=1000
tolHJB = 1e-6
convHJB = 1000

#FKE iter
iterFKE=0
maxiterFKE=120
tolFKE= 1e-5
convFKE = 100
relax=0.99

for iterFKE in range(0,maxiterFKE):
    print("Iteration Num " + str(iterFKE))
    # '''
    # HAMILTON-JACOBI-BELLMAN EQUATION
    # '''
    for iterHJB in range(0,maxiterFKE):
            V1 = V0
```

```
            V_af = np.matmul(D_af,V1) ; V_af[Na-1,:] = (w*z_grid +␣
↪r*a_grid[Na-1])**(- gamma)
            V_ab = np.matmul(D_ab,V1) ; V_ab[0   ,:] = (w*z_grid +␣
↪r*a_grid[0])**(- gamma)

            Ind_concave = V_ab > V_af #indicator whether value function is␣
↪concave (problems arise if this is not the case)

            #forward
            c_f = consumption(V_af)
            s_f = w*ZZ + r*AA - c_f
            I_f = s_f>0

            #backward
            c_b = consumption(V_ab)
            s_b = w*ZZ + r*AA - c_b
            I_b = s_b<0

            #complement (equals zero in steady state)
            c_0 = w*ZZ + r*AA
            V_a0= c_0**(-gamma)
            I_0 = (1 - I_f - I_b)

            #Upwind scheme
            V_a = V_af*I_f + V_ab*I_b + V_a0*I_0 #important to include third␣
↪term (complement)

            c = V_a**(-1/gamma)
            u = utility(c)
            s = s_f*I_f + s_b*I_b #Drift

            # Derivative on a-dimension (it is a matrix for every z-state)
            DA_aux = np.zeros((Nz,Na,Na))
            for j in range(0,Nz):
                for i in range(0,Na):

                    #FIRST VALUE
                    if i == 0:
                        if s[0,j] >= 0:
                            DA_aux[j,0,0]       = s[0,j] * (-1)/da
                            DA_aux[j,0,1]       = s[0,j] * 1/da
                        elif s[0,j] < 0:
                            DA_aux[j,0,0]       = s[0,j] * 1/da

                    #LAST VALUE
                    elif i == Na-1:
```

```
                if s[Na-1,j] >= 0:
                    DA_aux[j,Na-1,Na-1] = s[Na-1,j] * (-1)/da
                elif s[Na-1,j] < 0:
                    DA_aux[j,Na-1,Na-1] = s[Na-1,j] * (1)/da
                    DA_aux[j,Na-1,Na-2] = s[Na-1,j] * (-1)/da
            #CENTER
            else:
                if s[i,j] >= 0:
                    DA_aux[j,i,i]       = s[i,j] * (-1)/da
                    DA_aux[j,i,i+1]     = s[i,j] * 1/da
                elif s[i,j] < 0:
                    DA_aux[j,i,i]       = s[i,j] * 1/da
                    DA_aux[j,i,i-1]     = s[i,j] * (-1)/da


        #np.set_printoptions(precision=5)
        #DA_aux
        DA=block_diag(*DA_aux)

        #Solve the system:
        # '''
        # (v_{n+1} - v_{n})/Delta + rho * v_{n+1} = u_{n} + A_{n}*v_{n+1}
        # can be expressed as
        # X_{n} * v_{n+1}  = y_{n}
        # where
        # X_{n} = (1/Delta + rho)*I - A_{n}
        # y_{n} = u_{n} + (1/Delta)*v_{n}
        # then
        # v_{n+1} = X_{n}\y_{n}
        # '''
        A = DA + Aswitch
        X = (1/Delta+rho) * np.eye(Na*Nz) - A

        u_stacked =u.reshape((Na*Nz,1),order='F')
        V1_stacked=V1.reshape((Na*Nz,1),order='F')
        y = u_stacked + V1_stacked/Delta

        V_stacked = np.matmul(np.linalg.inv(X),y)

        V1  = V_stacked.reshape((Na,Nz),order='F')

        #convergence criterionwhile
        convHJB = np.amax(np.abs(V1-V0))
        #print(convHJB)

        #update
        V0 = V1
```

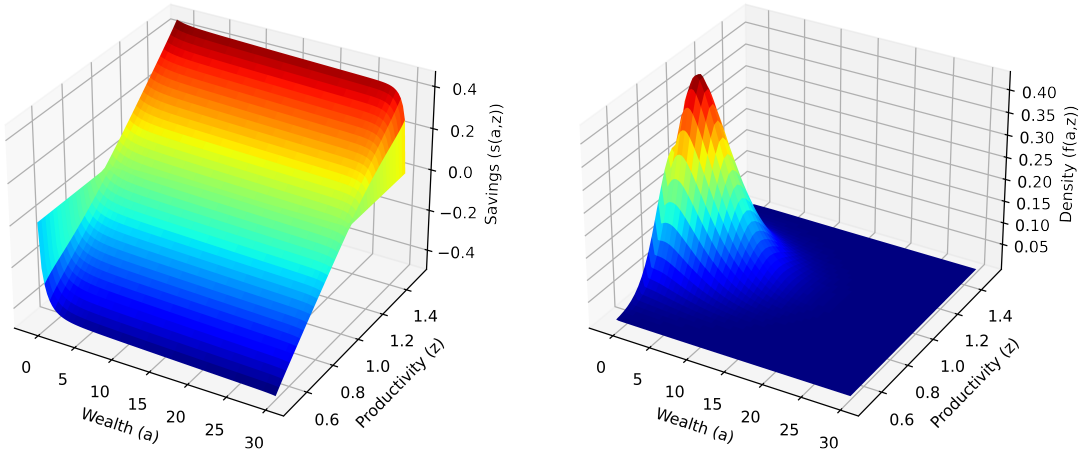
```
            #iterHJB += 1
            if convHJB < tolHJB:
                break

    # '''
    # FOKKER - PLANCK EQUATION
    # '''

    # A'g = 0
    np.set_printoptions(precision=5)
    TA= np.transpose(A)

    #OPT1: use built-in eigenvalues
    # eigvals, eigvecs =np.linalg.eig( TA )
    # eigvals.real
    # eigvecs.real
    # TA @ eigvecs[:,0]
    # np.matmul(TA,eigvecs[:,0] )
    # gg = eigvecs.real[:,0] / np.sum(eigvecs.real[:,0])
    # g=gg.reshape((Na,Nz))


    #OPT2: fix one value
    i_fix = 0
    b_aux = np.zeros((Na*Nz,1))
    b_aux[i_fix,0] = .1

    row_aux= np.zeros((1,Na*Nz))
    row_aux[0,i_fix] = 1

    TA[i_fix,:] = row_aux
    gg = np.linalg.inv(TA) @ b_aux

    #normalization
    g_sum= np.sum(gg)*da*dz
    g= (gg/g_sum).reshape((Na,Nz),order='F')

    #Update Agg Capital
    S = np.sum(g.T @ a_grid *da*dz)

    #Update prices
    K = relax*K + (1-relax)*S
    r=price_r(K)
    w=price_w(K)

    convFKE = np.abs(K-S)
    print(convFKE)
```

```
    if convFKE < tolFKE:
        break
```

```
Iteration Num 0
2.316079316403469
Iteration Num 1
1.7382314294225973
Iteration Num 2
1.0878035475202692
Iteration Num 3
0.5169197418657103
Iteration Num 4
0.17920023513989491
Iteration Num 5
0.04906847189125374
Iteration Num 6
0.012152953007054101
Iteration Num 7
0.0029058364321081775
Iteration Num 8
0.0006891112046956138
Iteration Num 9
0.00017119673531551527
Iteration Num 10
3.442318822832746e-05
Iteration Num 11
2.8148219677781583e-05
Iteration Num 12
3.0434289546299453e-06
```

[47]:
```
# ------------------------------------------------------------------------ #
#                                PLOT DISTRIBUTION                         #
# ------------------------------------------------------------------------ #
from matplotlib import cm
from matplotlib.ticker import LinearLocator
from matplotlib.pyplot import figure

#figure(figsize=(8, 6), dpi=80)

#Savings policy function
SS = w*ZZ + r*AA - c

fig = plt.figure(figsize=(12, 8), dpi=1200)

ax1=fig.add_subplot(121,projection='3d')
ax1.plot_surface(AA, ZZ , SS, cmap=cm.jet, linewidth=0, antialiased=False)
```

```
ax1.set_xlabel('Wealth (a)')
ax1.set_ylabel('Productivity (z)')
ax1.set_zlabel('Savings (s(a,z))')
#ax1.view_init(elev=45., azim=25)}

#Wealth distribution
ax2=fig.add_subplot(122,projection='3d')
ax2.plot_surface(AA, ZZ , g, cmap=cm.jet, linewidth=0, antialiased=False)
ax2.set_xlabel('Wealth (a)')
ax2.set_ylabel('Productivity (z)')
ax2.set_zlabel('Density (f(a,z))')
#ax2.view_init(elev=45., azim=25)

plt.show()
plt.tight_layout()
fig.savefig('aiyagari_difussion.svg', format='svg', dpi=1200)
```



<Figure size 432x288 with 0 Axes>