Economics Homework #1: Overlapping Generations Model

Eric C. Miller

June 27, 2017

Note: I used the git.tutorial code as a basis, and built off of there. I also apologize for the late turn-in; I initially messed up with GitHub, and elected to make sure everything looked good before turning in again.

1 Question #1

$\{b_i\}_{i=2}^{3} = [0.0193, 0.0584]$

$\{c_i\}_{i=1}^{3} = [0.18252, 0.2096, 0.2408]$

$w = 0.20172$

$r = 2.4330$

2 Question #2

$\{b_i\}_{i=2}^{3} = [0.0281, 0.0768]$

$\{c_i\}_{i=1}^{3} = [0.1959, 0.2286, 0.2666]$

$w = 0.22415$

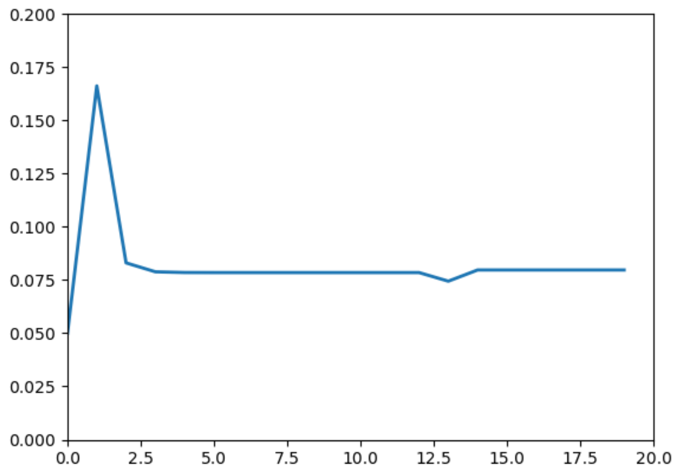
$r = 1.88637$

We observe that both savings rates increase, all consumptions increase by about 0.02, wages slightly increase, and interest rates slightly decrease. Wages slightly increase, as there is more of an incentive to put away money earlier, to consume later. As more money is saved, however, interest rates fall.

3 Question #4

See output for Question#3.

For the TPI method of solving for the functional equilibrium, we observe the following trend according to the graph (see next page). The algorithm took approximately 5 periods to achieve the necessary tolerance level with a $\zeta = 0.5$.




```
import numpy as np
import scipy.optimize as opt
from numpy import linalg as la
import matplotlib.pyplot as plt

class Steady(object):
    def __init__(self):
        self.A = 1.0
        self.beta = 0.96 ** (60 / 3)
        self.sigma = 3.0
        self.alpha = 0.35
        self.delta = 0.6415
        self.b2_init = 0.14
        self.b3_init = 0.05
        self.b_init = np.array([self.b2_init, self.b3_init])
        self.b_args = (self.A, self.alpha, self.beta, self.delta, self.sigma)
        self.b_result = opt.root(self.get_EulErrs, self.b_init, args=(self.b_args))
        self.b2bar = self.b_result.x[0]
        self.b3bar = self.b_result.x[1]
        self.Time = 13
        self.Final = [0,0]

    def get_ct(self, bt, btp1, rt, wt):
        ct = wt + (1 + rt)*bt - btp1
        return ct

    def MU_stitch(self, ct, sigma):
        epsilon = 0.0001
        c_cnstr = ct < epsilon
        if c_cnstr:
            b2 = (-sigma * (epsilon ** (-sigma - 1))) / 2
            b1 = (epsilon ** (-sigma)) - 2 * b2 * epsilon
            MU_c = 2 * b2 * ct + b1
        else:
```

```
            MU_c = ct ** (-sigma)
        return MU_c

    def get_EulErrs(self, bvec, *args):
        b2, b3 = bvec
        A, alpha, beta, delta, sigma = args
        self.w = (1-alpha)*A*((b2+b3)/2.2)**alpha
        self.r = alpha*A*(2.2/(b2+b3))**(1-alpha) - delta
        c1 = self.get_ct(0.0, b2, 0.0, self.w)
        c2 = self.get_ct(b2, b3, self.r, self.w)
        c3 = self.get_ct(b3, 0.0, self.r, 0.2*self.w)
        MU_c1 = self.MU_stitch(c1, sigma)
        MU_c2 = self.MU_stitch(c2, sigma)
        MU_c3 = self.MU_stitch(c3, sigma)
        err1 = MU_c1 - beta * (1 + self.r) * MU_c2
        err2 = MU_c2 - beta * (1 + self.r) * MU_c3
        err_vec = np.array([err1, err2])
        return err_vec

    def Kpathmaker(self, T, K):
        K1 = K[0]
        KT = K[-1]
        R = [self.alpha*self.A*(2.2/(x))**(1-self.alpha) - self.delta for x in K]
        W = [(1-self.alpha)*self.A*((KT)/2.2)**self.alpha for x in K]
        path_vec = [K,W,R]
        return path_vec

    def Pathfinder(self, T, K):
        K1 = K
        KT = 0.8*self.b2bar + 1.1*self.b3bar
        RT = self.alpha*self.A*(2.2/(KT))**(1-self.alpha) - self.delta
        WT = (1-self.alpha)*self.A*((KT)/2.2)**self.alpha
        Kdiff = (KT - K1)/T
        Kpath = []
        Rpath = []
        Wpath = []
        t = 0
        while t < T:
            k = K1 + t*Kdiff
            Kpath.append(K1 + t*Kdiff)
            Rpath.append(self.alpha*self.A*(2.2/(k))**(1-self.alpha) - self.delta)
            Wpath.append((1-self.alpha)*self.A*((k)/2.2)**self.alpha)
            t = t + 1
        t = T
        while t < T + 6:
            Kpath.append(KT)
            t = t + 1
        Kpath.append(KT)
        Rpath.append(RT)
        Wpath.append(WT)
        path_vec = [Kpath,Wpath,Rpath]
        return path_vec
```

```
    def s_temp_errors(self, bvec, *args):
        b2 = bvec
        w,r,A,alpha,beta,delta,sigma = args
        c2 = self.get_ct(self.b2_init, b2, r, w)
        c3 = self.get_ct(b2, 0.0, r, 0.2*w)
        MU_c2 = self.MU_stitch(c2, sigma)
        MU_c3 = self.MU_stitch(c3, sigma)
        err = MU_c2 - beta * (1 + r) * MU_c3
        err_vec = np.array([err])
        return err_vec

    def m_temp_errors(self, bvec, *args):
        b2, b3 = bvec
        w,r,A,alpha,beta,delta,sigma = args
        c1 = self.get_ct(0.0, b2, 0.0, w)
        c2 = self.get_ct(b2, b3, r, w)
        c3 = self.get_ct(b3, 0.0, r, 0.2*w)
        MU_c1 = self.MU_stitch(c1, sigma)
        MU_c2 = self.MU_stitch(c2, sigma)
        MU_c3 = self.MU_stitch(c3, sigma)
        err1 = MU_c1 - beta * (1 + r) * MU_c2
        err2 = MU_c2 - beta * (1 + r) * MU_c3
        err_vec = np.array([err1, err2])
        return err_vec

    def Kprimebot(self, T, b2, b3, path_vec):
        b2path = [b2]
        b3path = [b3]
        b2_newbar = 0.8*self.b2bar
        b3_newbar = 1.1*self.b3bar
        Kpath, Wpath, Rpath = path_vec
        init_args = (Wpath[1], Rpath[1], self.A, self.alpha, self.beta, self.delta, self.si
        init_Kprime = opt.brentq(self.s_temp_errors,0,1, xtol=np.array([b2]), args=(init_ar
        b3path.append(init_Kprime)
        for i in range(1,T):
            i_args = (Wpath[i], Rpath[i], self.A, self.alpha, self.beta, self.delta, self.s
            i_bvals = np.array([b2path[i-1], b3path[i]])
            i_prime = opt.root(self.m_temp_errors, self.b_init, args=(i_args))
            i_list = list(i_prime.x)
            b2path.append(i_list[0])
            b3path.append(i_list[1])
        b2path.append(b2_newbar)
        t = len(b2path)
        while t < len(Kpath):
            b2path.append(b2_newbar)
            b3path.append(b3_newbar)
            t = t + 1
        Kprimepath = []
        for i in range(0,len(b2path)):
            ki = b2path[i] + b3path[i]
            Kprimepath.append(ki)
        return Kprimepath

    def Kiterator(self,K,C):
```

```
        if C == 0:
            path_vec = self.Pathfinder(self.Time,K)
        if C == 1:
            path_vec = self.Kpathmaker(self.Time,K)
        K_vec, r, w = path_vec
        kprime_vec = self.Kprimebot(self.Time, 0.3*K_vec[0], 0.7*K_vec[0], path_vec)
        K_sub = [x - y for x, y in zip(kprime_vec, K_vec)]
        k = la.norm(np.array(K_sub))
        output = [[k],kprime_vec,K_vec]
        return output

    def Kconverge(self,K,C):
        e = 2.7182818284590452353602874713527
        epsilon2 = 1/e**9
        zeta = 0.5
        k = 1
        K_norm, Kprime_vec, K_vec = self.Kiterator(K,C)
        k = K_norm[0]
        if k > epsilon2:
            K_new = [zeta*x + (1.0 - zeta)*y  for x, y in zip(Kprime_vec, K_vec)]
            self.Kconverge(K_new,1)
        else:
            self.Final = Kprime_vec
            return Kprime_vec

S = Steady()
print('Roots: ', S.b2bar, S.b3bar)
print('Wage, Rate:', S.w, S.r)
print(S.Kconverge(0.05,0))
xaxis = [x for x in range(0,20)]
plt.plot(xaxis, S.Final, linewidth=2.0)
plt.axis([0, 20, 0, 0.2])
plt.show()
print('Final Capital Path:', S.Final)
```