Tracht_PS3

January 30, 2019

1 MACS 30150 - Problem Set 3

1.0.1 by Daniel Tracht, January 2019

The code in this Jupyter notebook was written using Python 3.7. This problem set follows exercises 5.1 through 5.22 in Dr. Evan's chapter on Dynamic Programming

1.1 Exercise 5.1

Our problem, in terms of consumption is

$$\max_{c_t \in [0, W_t]} \sum_{t=1}^{T} \beta^{t-1} u\left(c_t\right) \quad \text{s.t. } W_{t+1} = W_t - c_t$$

where W_t is non-negative and taken as given by the agent in period t. If $T = 1$, our problem becomes

$$\max_{c_1 \in [0, W_1]} u\left(c_1\right) \quad \text{s.t. } W_2 = W_1 - c_1$$

At the optimum,

$$W_1 = c_1$$
$$W_2 = 0$$

as $u\left(c_1\right)$ is increasing in its argument. Written in the equivalent way, our problem is

$$\max_{W_{t+1} \in [0, W_t]} \sum_{t=1}^{T} \beta^{t-1} u\left(W_t - W_{t+1}\right)$$

In the case that $T = 1$, our problem becomes

$$\max_{W_2 \in [0, W_1]} u\left(W_1 - W_2\right)$$

At the optimum,

$$W_2 = 0$$

as utility is increasing in W_1 and decreasing in W_2. This can be written as

$$W_{T+1} = \Psi_T\left(W_T\right) = 0$$

We wish to find the condition that characterizes the optimal amount of cake to eat in period 1, if the individual lives for one period $T = 1$. The individual would choose W_1 such that

$$\max_{w_2 \in [0, W_1]} u(W_1 - W_2)$$

1.2 Exercise 5.2

In the case that the agent lives for two periods, that is $T = 2$, our problem in period 2 becomes

$$\max_{W_3 \in [0, W_2]} u\left(W_2 - W_3\right)$$

and we find that

$$W_3 = 0$$

In period 1, our problem becomes

$$\max_{W_2 \in [0, W_1]} u\left(W_1 - W_2\right) + \beta u\left(W_2\right)$$

subject to our law of motion. At the optimum, we have

$$\frac{\delta u\left(W_1 - W_2\right)}{\delta W_2} = \beta \frac{\delta u\left(W_2\right)}{\delta W_2}$$

where the agent chooses W_2 such that marginal utility today equals the discounted marginal utility in the next period. This can be written as

$$W_T = \Psi_{T-1} = \left(W_{T-1}\right) : \frac{\delta u\left(W_{T-1} - W_T\right)}{\delta W_T} = \beta \frac{\delta V_T\left(W_T\right)}{\delta W_T}$$

where we've replaced the utility next period with the value function next period

1.3 Exercise 5.3

In the case that the age lives for three periods, that is $T = 3$, our problem becomes easier to write using the policy functions from the previous two exercises and the notation used in the lectures:

$$\max_{W_{T-1}} u\left(W_{T-2} - W_{T-1}\right) + \beta u\left(W_{T-1} - \Psi_{T-1}\left(W_{T-1}\right)\right) + \beta^2 u\left(\Psi_{T-1}\left(W_{T-1}\right)\right)$$

where W_{T-2} is given. Taking a derivative with respect to W_{T-1}, we get

$$\frac{\delta u\left(W_{T-2} - W_{T-1}\right)}{\delta W_{T-1}} = \beta \frac{\delta u\left(W_{T-1} - W_T\right)}{\delta W_{T-1}}$$

For $T = 3$ we get

$$\frac{\delta u\left(W_1 - W_2\right)}{\delta W_2} = \beta \frac{\delta u\left(W_2 - W_3\right)}{\delta W_2} = \beta \frac{\delta V_T\left(W_T\right)}{\delta W_T}$$

Combining our information, we have

$$
\begin{aligned}
\{W_2, W_3, W_4\} &= \{\} \\
W_4 &= 0 \\
W_3 &= \Psi_2 (W_2) \\
W_2 &=
\end{aligned}
$$

Assuming that $W_1 = 1$, $\beta = 0.9$, and that the period utility function is $\ln (c_t)$, we wish to find the vectors of consumption and cake size over our time periods. We can solve this and practice our Python at the same time:

```
In [6]: # Solving the functions analytically using sympy
        import sympy as sy
        # Given parameters
        w_1 = 1.0
        beta = 0.9
        # Want to solve for w_2 through w_4 and c_1 through c_3
        w_4 = 0 # by construction of the problem
        w_2 = sy.symbols('w_2')
        w_3 = sy.symbols('w_3')
        # a system of our two equations from above
        # returns a list of solution mappings
        solutions = sy.solve([sy.log(w_2 - w_3).diff(w_3) + beta*sy.log(w_3).diff(w_3),
                              sy.log(w_1 - w_2).diff(w_2) + beta*sy.log(w_2 - w_3).diff(w_2)],
                             dict=True)
        # Recall the relations between cake size and consumption
        c_1 = w_1 - solutions[0][w_2]
        c_2 = solutions[0][w_2] - solutions[0][w_3]
        c_3 = solutions[0][w_3]

        # w_1 defined as given parameter
        w_2 = solutions[0][w_2]
        w_3 = solutions[0][w_3]
        w_4 = 0 # no need to solve for this one

        # make dictionaries of the solutions
        w_vec = [w_1, w_2, w_3, w_4]
        c_vec = [c_1, c_2, c_3]
        key_w = ['w_1', 'w_2', 'w_3', 'w_4']
        key_c = ['c_1', 'c_2', 'c_3']

        dict_w = dict(zip(key_w, w_vec))
        dict_c = dict(zip(key_c, c_vec))

        print(dict_w)
        print(dict_c)

{'w_1': 1.0, 'w_2': 0.630996309963100, 'w_3': 0.298892988929889, 'w_4': 0}
```

```
{'c_1': 0.369003690036900, 'c_2': 0.332103321033210, 'c_3': 0.298892988929889}
```

1.4 Exercise 5.4

We have rewritten our problem as

$$V_{T-1}\left(W_{T-1}\right) = \max_{W_T} u\left(W_{T-1} - W_T\right) + \beta V_T\left(W_T\right)$$

We want to show the policy function in period $T-1$ for

$$W_T = \psi_{T-1}\left(W_{T-1}\right)$$

From the work in excercise 5.2, this is

$$W_T = \psi_{T-1}\left(W_{T-1}\right) : \frac{\delta u\left(W_{T-1} - W_T\right)}{\delta W_T} = \beta \frac{\delta V\left(W_T\right)}{\delta W_T}$$

Writing the value function V_{T-1} in terms of the policy function, we have

$$V_{T-1}\left(W_{T-1}\right) = \max_{W_T} u\left(W_{T-1} - \psi_{T-1}\left(W_{T-1}\right)\right) + \beta V_T\left(\psi_{T-1}\left(W_{T-1}\right)\right)$$

1.5 Exercise 5.5

Let us assume that $u\left(c\right) = \ln\left(c\right)$. We want to show that

$$V_{T-1}\left(\bar{W}\right) \neq V_T\left(\bar{W}\right)$$

and that

$$\psi_{T-1}\left(\bar{W}\right) \neq \psi_T\left(\bar{W}\right)$$

for a fixed cake size $\bar{W}$ and $T < \infty$ is the last period of the agent's life. Since the agent is finitely lived, we know that $\psi_T\left(\bar{W}\right) = W_{T+1} = 0$. Intuitively, the agent can't eat cake after death. Furthermore, we know that

$$\psi_{T-1}\left(\bar{W}\right) = W_T = \bar{W}$$

Thus,

$$\psi_{T-1}\left(\bar{W}\right) \neq \psi_T\left(\bar{W}\right)$$

Futhermore, we know that

$$\begin{aligned}
V_T\left(\bar{W}\right) &= \max_{W_{T+1}} u\left(\bar{W} - \psi_T\left(\bar{W}\right)\right) + \beta V_{T+1}\left(\psi_T\left(\bar{W}\right)\right) \\
&= \max_{W_{T+1}} u\left(\bar{W}\right) \\
&= u\left(\bar{W}\right) \\
&= \ln\left(\bar{W}\right)
\end{aligned}$$

and that

$$V_{T-1}\left(\bar{W}\right) = \max_{W_T} u\left(\bar{W} - \psi_{T-1}\left(W_T\right)\right) + \beta V_T\left(\psi_{T-1}\left(W_T\right)\right)$$

At optimality with our assumed form for utility, we have

1.6 Exercise 5.6

Assuming that $u(c) = \ln(c)$, we want to write the finite horizon Bellman equation for the value function at time $T-2$. This is

$$V_{T-2}(W_{T-2}) = \max_{W_{T-1}} \ln(W_{T-2} - W_{T-1}) + \beta V_T(W_{T-1})$$

which is a very pretty recursive equation. We want the analytical solution for

$$W_{T-1} = \psi_{T-2}(W_{T-2})$$

using the envelope theorem. This is

$$W_{t-1}$$

Furthermore, we want the analytical solution for V_{T-2}. This is

$$V_{T-2}$$

1.7 Exercise 5.7

Assuming that $u(c) = \ln(c)$, and with the answers to the previous two exercises, we want to write down the expressions for the analytical solutions for

$$\psi_{T-s}(W_{T-s})$$

and

$$V_{T-s}(W_{T-s})$$

for the general integer $s \geq 1$ using induction.

This is

$$\psi_{T-s}(W_{T-s}) = \frac{\sum_{i=1}^{s} \beta^i}{1 + \sum_{i=1}^{s} \beta^i} W_{T-s}$$

which intuitively means that the cake saved for the next s periods until death is the cake that keeps the discounted stream of log utility stable. Furthermore, we have

$$V_{T-s}(W_{T-s}) = \sum_{i=1}^{s} \beta^i \ln\left(\frac{\beta^i W_{T-s}}{1 + \sum_{i=1}^{s} \beta^i}\right)$$

which intuitively means that the value We want to show that

$$\lim_{s \to \infty} V_{T-s}(W_{T-s}) = V(W_{T-s})$$

and that

$$\lim_{s \to \infty} \psi_{T-s}(W_{T-s}) = \psi(W_{T-s})$$

1.8 Exercise 5.8

We want to write the Bellman equation for the cake eating problem with a general utility function with $T = \infty$. This is

$$V(W) = \max_{W' \in [0,W]} u\left(W - W'\right) + \beta V\left(W'\right)$$

1.9 Exercise 5.9

For this we can work in Python

```
In [34]: import numpy as np
         w_min = 1e-2
         w_max = 1.0
         N = 100
         w_vec = np.linspace(w_min, w_max, N)
```

1.10 Exercise 5.10

Continuing in Python

```
In [35]: # defining the utility function
         def utility(consumption):
             utils = np.log(consumption)
             return utils
         # given parameter
         beta = 0.9

         # following the code as covered in lecture
         w = np.tile(w_vec.reshape((N, 1)), (1, N))
         w_prime = np.tile(w_vec.reshape((1, N)), (N, 1))
         c_mat = w - w_prime
         c_pos = c_mat > 0
         c_mat[~c_pos] = 1e-7
         u_mat = utility(c_mat)
         # initial guess of zeros
         v_init = np.zeros(N).reshape((N, 1))
         vt_w = u_mat + beta * v_init

         # this is the value function
         v_vec = vt_w.max(axis=1)

         #print(v_vec)

         index = np.argmax(vt_w, axis=1)

         # this is the policy function
         w_prime_opt = w_vec[index]
         #print(w_prime_opt)
```

1.11 Exercise 5.11

Defining the distance metric in Python:

```
In [36]: def distance(vf_1, vf_2):
             norm = np.sum((vf_1 - vf_2)**2)
             return norm
```

1.12 Exercise 5.12

Generating V_{T-1} and ψ_{T-1} in Python:

```
In [37]: # taking v_vec from 5.10
         v_prime = np.tile(v_vec.reshape((1, N)), (N, 1))
         # punish the values for negative consumption
         v_prime[~c_pos] = -9e+5
         v_w_1 = u_mat + beta * v_prime
         # new value function
         v_max_1 = v_w_1.max(axis=1)
         #print(v_max_1)

         index = np.argmax(v_w_1, axis=1)
         # new policy function
         w_prime_opt = w_vec[index]
         #print(w_prime_opt)

         # compare the distances
         dist_T = distance(v_vec, v_init)
         print("Distance at time T:", dist_T)
         dist_T_1 = distance(v_max_1, v_vec)
         print("Distance at time T-1:", dist_T_1)
```

```
Distance at time T: 43871.91180731695
Distance at time T-1: 656100000726.7605
```

The distance for $T-1$ is much larger than for T

1.13 Exercise 5.13

Repeating the takse for $T-2$

```
In [39]: # taking v_max_1 from 5.12
         v_prime = np.tile(v_max_1.reshape((1, N)), (N, 1))
         # punish the values for negative consumption
         v_prime[~c_pos] = -9e+5
         v_w_2 = u_mat + beta * v_prime
         # new value function
         v_max_2 = v_w_2.max(axis=1)
         #print(v_max_1)

         index = np.argmax(v_w_2, axis=1)
         # new policy function
         w_prime_opt = w_vec[index]
         #print(w_prime_opt)

         # compare the distances
```

```
         dist_T = distance(v_vec, v_init)
         print("Distance at time T:", dist_T)
         dist_T_1 = distance(v_max_1, v_vec)
         print("Distance at time T-1:", dist_T_1)
         dist_T_2 = distance(v_max_2, v_max_1)
         print("Distance at time T-2 compared to T-1:", dist_T_2)
         dist_T_2_alt = distance(v_max_2, v_init)
         print("Distance at time T-2 compared to T:", dist_T_2_alt)
```

```
Distance at time T: 43871.91180731695
Distance at time T-1: 656100000726.7605
Distance at time T-2 compared to T-1: 531441000766.63214
Distance at time T-2 compared to T: 118759498024501.34
```

Again, the distances at $T-2$ are greater than for time T.

1.14 Exercise 5.14

A first exercise of value function iteration in Python:

```
In [32]: max_iters = 500
         tolerance = 1e-10
         dist = 10.0
         vf_iter = 0
         while dist > tolerance and vf_iter < max_iters:
             vf_iter += 1
             # takes an initial guess and calculates the new value function
             v_prime = np.tile(v_init.reshape((1, N)), (N, 1))
             v_prime[~c_pos] = -9e+4
             v_new = (u_mat + beta * v_prime).max(axis=1)
             # measures the distance between the new value function and the old
             dist = distance(v_new, v_init)
             print('Iteration =', vf_iter, '; distance =', dist)
             v_init = v_new

         print("Convergence!")
         print("It takes %d iterations to converge" % vf_iter)
```

```
Iteration = 1 ; distance = 656361157021.4574
Iteration = 2 ; distance = 5316525743.271798
Iteration = 3 ; distance = 4306386030.006323
Iteration = 4 ; distance = 3488172794.5714226
Iteration = 5 ; distance = 2825420037.630621
Iteration = 6 ; distance = 2288590282.5590844
Iteration = 7 ; distance = 1853758166.5375955
Iteration = 8 ; distance = 1501544142.7426846
Iteration = 9 ; distance = 1216250776.642259
Iteration = 10 ; distance = 985163145.1419044
```

```
Iteration = 11 ; distance = 797982160.0373642
Iteration = 12 ; distance = 646365559.4266962
Iteration = 13 ; distance = 523556110.9935629
Iteration = 14 ; distance = 424080456.20615387
Iteration = 15 ; distance = 343505174.74340343
Iteration = 16 ; distance = 278239195.8852569
Iteration = 17 ; distance = 225373752.31221965
Iteration = 18 ; distance = 182552742.55996224
Iteration = 19 ; distance = 147867724.27005062
Iteration = 20 ; distance = 119772859.10622115
Iteration = 21 ; distance = 97016018.0385953
Iteration = 22 ; distance = 78582976.63315375
Iteration = 23 ; distance = 63652212.98545916
Iteration = 24 ; distance = 51558294.32776982
Iteration = 25 ; distance = 41762220.1200653
Iteration = 26 ; distance = 33827399.92370578
Iteration = 27 ; distance = 27400195.48398699
Iteration = 28 ; distance = 22194159.813998673
Iteration = 29 ; distance = 17977270.853317253
Iteration = 30 ; distance = 14561590.732883928
Iteration = 31 ; distance = 11794889.778709196
Iteration = 32 ; distance = 9553861.95285677
Iteration = 33 ; distance = 7738629.36668709
Iteration = 34 ; distance = 6268290.928397754
Iteration = 35 ; distance = 5077316.754233369
Iteration = 36 ; distance = 4112627.637429302
Iteration = 37 ; distance = 3331229.420618224
Iteration = 38 ; distance = 2698296.8357309587
Iteration = 39 ; distance = 2185621.415306089
Iteration = 40 ; distance = 1770354.30067605
Iteration = 41 ; distance = 1433987.914944458
Iteration = 42 ; distance = 1161531.1223381404
Iteration = 43 ; distance = 940841.1015289264
Iteration = 44 ; distance = 762082.1677779292
Iteration = 45 ; distance = 617287.4156153646
Iteration = 46 ; distance = 500003.6522140527
Iteration = 47 ; distance = 405003.7907942261
Iteration = 48 ; distance = 328053.89114983805
Iteration = 49 ; distance = 265724.4605370663
Iteration = 50 ; distance = 215237.6113282474
Iteration = 51 ; distance = 174343.25337503717
Iteration = 52 ; distance = 141218.81426884216
Iteration = 53 ; distance = 114388.00935499243
Iteration = 54 ; distance = 92655.04917523317
Iteration = 55 ; distance = 75051.34382720977
Iteration = 56 ; distance = 60792.33427200552
Iteration = 57 ; distance = 49242.52936902187
Iteration = 58 ; distance = 39887.180071882685
```

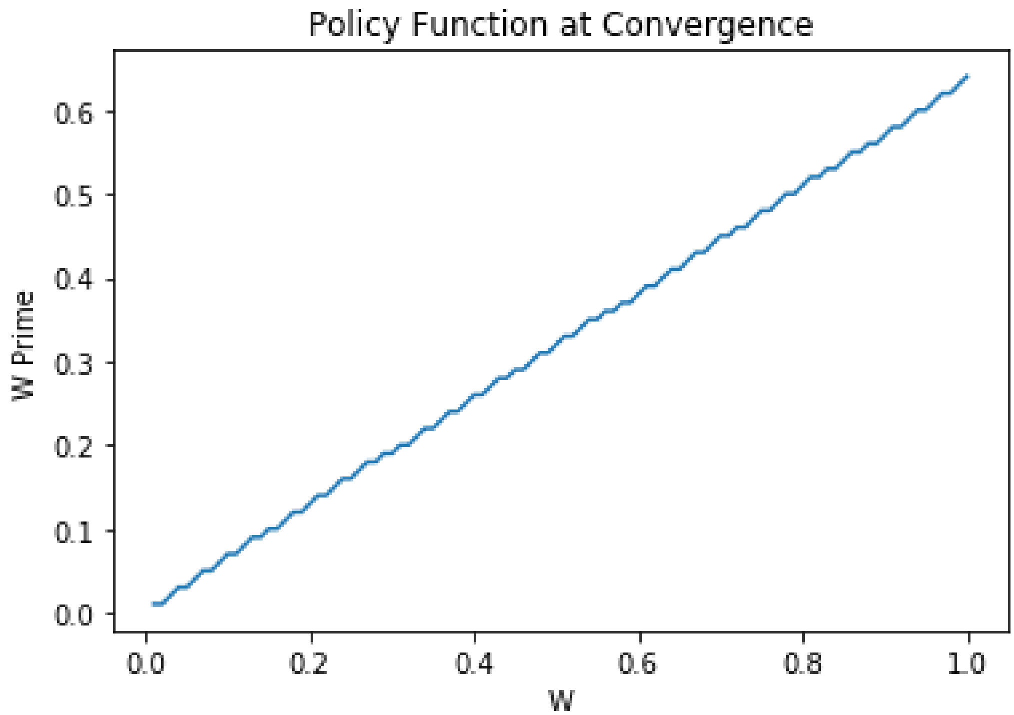
```
Iteration = 59 ; distance = 32309.340384184605
Iteration = 60 ; distance = 26171.282961856
Iteration = 61 ; distance = 21199.44996270106
Iteration = 62 ; distance = 17172.25765403326
Iteration = 63 ; distance = 13910.225445246653
Iteration = 64 ; distance = 11267.97247896265
Iteration = 65 ; distance = 9127.741149256373
Iteration = 66 ; distance = 7394.146525049693
Iteration = 67 ; distance = 5989.928390551506
Iteration = 68 ; distance = 4852.503735570291
Iteration = 69 ; distance = 3931.1818695174757
Iteration = 70 ; distance = 3184.904097831992
Iteration = 71 ; distance = 2580.410948027704
Iteration = 72 ; distance = 2090.7641764207274
Iteration = 73 ; distance = 1694.1410904514291
Iteration = 74 ; distance = 1372.8672014517142
Iteration = 75 ; distance = 1112.627119310548
Iteration = 76 ; distance = 901.8230243640993
Iteration = 77 ; distance = 731.0600844858621
Iteration = 78 ; distance = 592.7310334744678
Iteration = 79 ; distance = 480.6745763286175
Iteration = 80 ; distance = 389.89717447236984
Iteration = 81 ; distance = 316.3533259689117
Iteration = 82 ; distance = 256.76931150496404
Iteration = 83 ; distance = 208.4941502491576
Iteration = 84 ; distance = 169.37141215479707
Iteration = 85 ; distance = 137.665420034222
Iteration = 86 ; distance = 111.96869169941148
Iteration = 87 ; distance = 91.12990180160644
Iteration = 88 ; distance = 74.23008056632061
Iteration = 89 ; distance = 60.52291228596271
Iteration = 90 ; distance = 49.38428817377354
Iteration = 91 ; distance = 40.24315102906038
Iteration = 92 ; distance = 32.126682064026454
Iteration = 93 ; distance = 25.541799163125216
Iteration = 94 ; distance = 19.767115537050348
Iteration = 95 ; distance = 15.155025726039097
Iteration = 96 ; distance = 11.174263919022518
Iteration = 97 ; distance = 8.02000552946439
Iteration = 98 ; distance = 5.341082016591021
Iteration = 99 ; distance = 3.2347254868217576
Iteration = 100 ; distance = 1.4634529907462
Iteration = 101 ; distance = 0.0
Convergence!
It takes 101 iterations to converge
```

1.15 Exercise 5.15

Plotting in Python

```
In [42]: import matplotlib.pyplot as plt

         index = np.argmax(u_mat + beta * v_prime, axis=1)
         w_prime_opt = w_vec[index]
         plt.plot(w_vec, w_prime_opt)
         plt.title('Policy Function at Convergence')
         plt.xlabel('W')
         plt.ylabel("W Prime")
         plt.show()
```



1.16 Exercise 5.16

With i.i.d. stochastic shocks:

```
In [46]: import scipy.stats as sts
         # parameters from the exercise
         sigma = np.sqrt(0.25)
         mu = 4 * sigma
         e_max = mu + 3 * sigma
         e_min = mu - 3 * sigma
```

```
M = 7
e_vec = np.linspace(e_min, e_max, M)
Gamma = sts.norm.pdf(e_vec, loc = mu, scale = sigma)
```

```
Out[46]: array([0.0088637 , 0.10798193, 0.48394145, 0.79788456, 0.48394145,
                0.10798193, 0.0088637 ])
```

1.17 Exercise 5.17

Getting the policy function in the stochastic case:

```
In [59]: # as from above
         w = np.tile(w_vec.reshape((N, 1)), (1, N))
         w_prime = np.tile(w_vec.reshape((1, N)), (N, 1))
         c_mat = w - w_prime
         c_pos = c_mat > 0
         c_mat[~c_pos] = 1e-7
         u_mat = utility(c_mat)
         # adding the shocks
         u_prism = np.array([u_mat*e for e in Gamma])

         # following from lecture
         v_init = np.zeros((N, M))
         v_expected = v_init @ Gamma.reshape((M,1))
         v_expected_mat = np.tile(v_expected.reshape((1, N)), (N, 1))
         v_expected_mat[~c_pos] = -9e+5
         v_expected_prism = np.array([v_expected_mat for i in range(M)])

         # get the new value
         v_t = u_prism + beta * v_expected_prism
         # maximizing the axis to conform dimensions
         v_new = np.zeros((N, M))
         w_prime = np.zeros((N, M))
         for i in range(N):
             v_w = v_t[:, i, :]
             v_new[i] = v_w.max(axis=1)
             index = np.argmax(v_w, axis=1)
             w_prime[i] = w_vec[index]
```

1.18 Exercise 5.18

The norm revisted

```
In [60]: def distance(vf_1, vf_2):
             norm = np.sum((vf_1 - vf_2)**2)
             return norm
         dist = distance(v_new, v_init)
         print("The distance between T and init:", dist)
```

```
The distance between T and init: 4592752208991.432
```

1.19 Exercise 5.19

Contraction revisited

```
In [61]: # Replacing v_init with v_new from 5.17
         v_init = v_new
         v_expected = v_init @ Gamma.reshape((M,1))
         v_expected_mat = np.tile(v_expected.reshape((1, N)), (N, 1))
         v_expected_mat[~c_pos] = -9e+5
         v_expected_prism = np.array([v_expected_mat for i in range(M)])

         # get the new value
         v_t = u_prism + beta * v_expected_prism
         # maximizing the axis to conform dimensions
         v_new_1 = np.zeros((N, M))
         w_prime_1 = np.zeros((N, M))
         for i in range(N):
             v_w = v_t[:, i, :]
             v_new_1[i] = v_w.max(axis=1)
             index = np.argmax(v_w, axis=1)
             w_prime_1[i] = w_vec[index]

         dist_1 = distance(v_new_1, v_new)
         print("The distance between T-1 and T:", dist_1)
```

```
The distance between T-1 and T: 4592737294212.839
```

This distance has gone down slightly.

1.20 Exercise 5.20

Another contraction

```
In [62]: # Replacing v_init with v_new_1 from 5.19
         v_init = v_new_1
         v_expected = v_init @ Gamma.reshape((M,1))
         v_expected_mat = np.tile(v_expected.reshape((1, N)), (N, 1))
         v_expected_mat[~c_pos] = -9e+5
         v_expected_prism = np.array([v_expected_mat for i in range(M)])

         # get the new value
         v_t = u_prism + beta * v_expected_prism
         # maximizing the axis to conform dimensions
         v_new_2 = np.zeros((N, M))
         w_prime_2 = np.zeros((N, M))
```

```
    for i in range(N):
        v_w = v_t[:, i, :]
        v_new_2[i] = v_w.max(axis=1)
        index = np.argmax(v_w, axis=1)
        w_prime_2[i] = w_vec[index]

    dist_2 = distance(v_new_2, v_new_1)
    print("The distance between T-2 and T:", dist_2)
```

```
The distance between T-2 and T: 4592684263232.5205
```

The distance again decreases.

1.21 Exercise 5.21

First stochastic value function iteration in Python:

```
In [63]: max_iters = 500
         tolerance = 1e-10
         dist = 10.0
         vf_iter = 0
         v_init = np.zeros((N, M))
         while dist > tolerance and vf_iter < max_iters:
             vf_iter += 1
             # takes an initial guess and calculates the new value function
             v_expected = v_init @ Gamma.reshape((M,1))
             v_expected_mat = np.tile(v_expected.reshape((1, N)), (N, 1))
             v_expected_mat[~c_pos] = -9e+5
             v_expected_prism = np.array([v_expected_mat for i in range(M)])
             # get the new value
             v_t = u_prism + beta * v_expected_prism
             # maximizing the axis to conform dimensions
             v_new = np.zeros((N, M))
             w_prime = np.zeros((N, M))
             for i in range(N):
                 v_w = v_t[:, i, :]
                 v_new[i] = v_w.max(axis=1)
                 index = np.argmax(v_w, axis=1)
                 w_prime[i] = w_vec[index]
             dist = distance(v_new, v_init)
             print('Iteration =', vf_iter, '; distance =', dist)
             v_init = v_new

         print("Convergence!")
         print("It takes %d iterations to converge" % vf_iter)
```

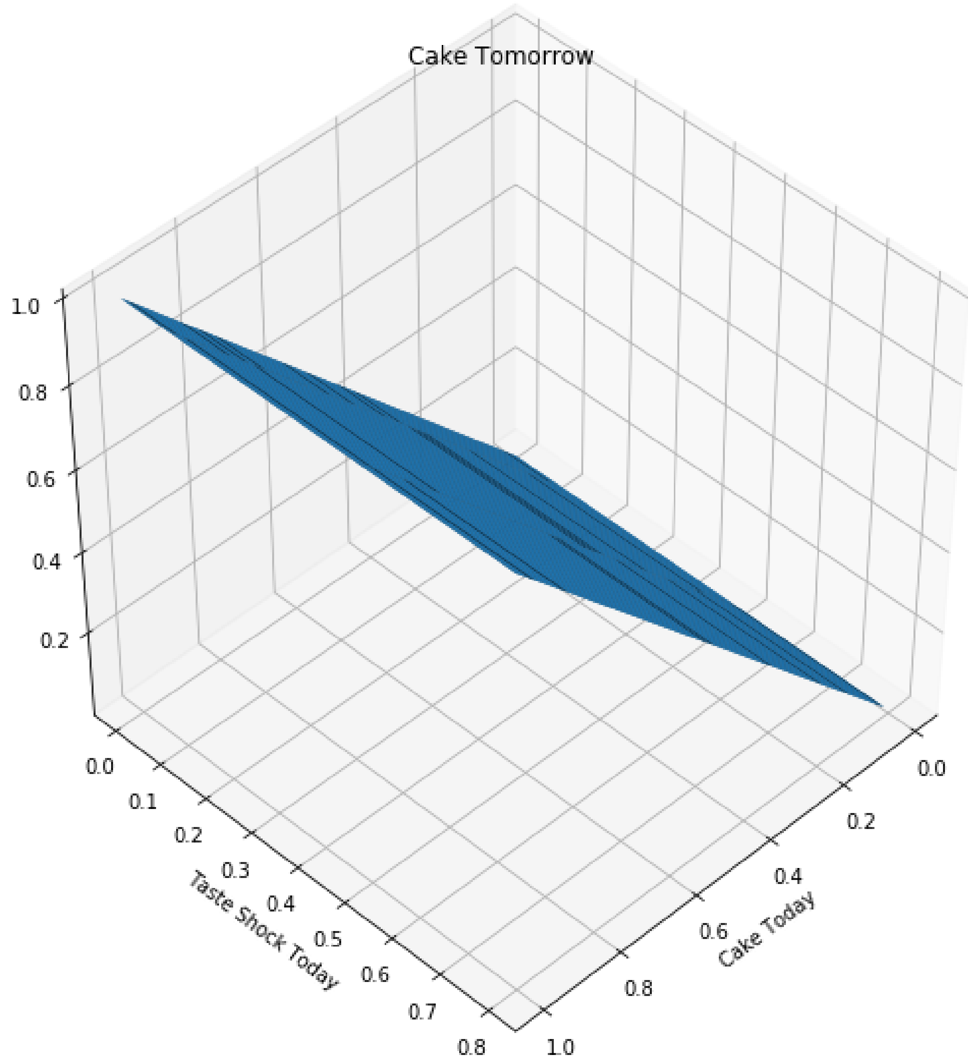
```
Iteration = 1 ; distance = 4592752208991.432
Iteration = 2 ; distance = 4592737294212.839
```

```
Iteration = 3 ; distance = 4592684263232.5205
Iteration = 4 ; distance = 4592588847470.437
Iteration = 5 ; distance = 4592417191307.818
Iteration = 6 ; distance = 4592108440110.062
Iteration = 7 ; distance = 4591553307590.937
Iteration = 8 ; distance = 4590555850423.502
Iteration = 9 ; distance = 4588765787437.956
Iteration = 10 ; distance = 4585560273089.171
Iteration = 11 ; distance = 4579842631902.5625
Iteration = 12 ; distance = 4569717238903.769
Iteration = 13 ; distance = 4552023936574.209
Iteration = 14 ; distance = 4521885388643.65
Iteration = 15 ; distance = 4473134221758.366
Iteration = 16 ; distance = 4403084113955.9795
Iteration = 17 ; distance = 4334015761996.3564
Iteration = 18 ; distance = 4393416684102.841
Iteration = 19 ; distance = 5092352109964.73
Iteration = 20 ; distance = 8258104678352.478
Iteration = 21 ; distance = 20103392134173.99
Iteration = 22 ; distance = 42382137982795.95
Iteration = 23 ; distance = 19775517137988.44
Iteration = 24 ; distance = 0.0
Convergence!
It takes 24 iterations to converge
```

1.22 Exercise 5.22

3-D Plotting in Python:

```
In [66]: from mpl_toolkits.mplot3d import Axes3D
         x,y = np.meshgrid(w_vec, Gamma)
         fig = plt.figure(figsize = (10, 10))
         ax = fig.add_subplot(111, projection='3d')
         ax.plot_surface(x.T, y.T, w_prime)
         ax.set_xlabel('Cake Today')
         ax.set_ylabel('Taste Shock Today')
         ax.set_title("Cake Tomorrow")
         ax.view_init(elev = 45, azim = 45)
         plt.show()
```



Cake Tomorrow
1.0
0.8
0.6
0.4
0.2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Taste Shock Today
0.0
0.2
0.4
0.6
0.8
1.0
Cake Today