Part IV

Search

CHAPTER

TWENTYSEVEN

JOB SEARCH I: THE MCCALL SEARCH MODEL

Contents

"Questioning a McCall worker is like having a conversation with an out-of-work friend: 'Maybe you are setting your sights too high', or 'Why did you quit your old job before you had a new one lined up?' This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them." – Robert E. Lucas, Jr.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
```

27.1 Overview

The McCall search model [McC70] helped transform economists' way of thinking about labor markets.

To clarify vague notions such as "involuntary" unemployment, McCall modeled the decision problem of unemployed agents directly, in terms of factors such as

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem he used dynamic programming.

Here we set up McCall's model and adopt the same solution method.

As we'll see, McCall's model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import jit, float64
from numba.experimental import jitclass
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

27.2 The McCall Model

An unemployed agent receives in each period a job offer at wage w_t.

The wage offer is a nonnegative function of some underlying state:

$$w_t = w(s_t) \quad \text{where} \quad s_t \in \mathbb{S}$$

Here you should think of state process $\{s_t\}$ as some underlying, unspecified random factor that impacts on wages.

(Introducing an exogenous stochastic state process is a standard way for economists to inject randomness into their models.)

In this lecture, we adopt the following simple environment:

- $\{s_t\}$ is IID, with $q(s)$ being the probability of observing state s in $\mathbb{S}$ at each point in time, and
- the agent observes s_t at the start of t and hence knows $w_t = w(s_t)$,
- the set $\mathbb{S}$ is finite.

(In later lectures, we will relax all of these assumptions.)

At time t, our agent has two choices:

1. Accept the offer and work permanently at constant wage w_t.
2. Reject the offer, receive unemployment compensation c, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future utility relative to current utility.

The variable y_t is income, equal to

- his/her wage w_t when employed
- unemployment compensation c when unemployed

The agent is assumed to know that $\{s_t\}$ is IID with common distribution q and can use this when computing expectations.

27.2.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide optimally in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to "states" and
2. then deduces optimal actions given those values

We'll go through these steps in turn.

27.2.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period (in this case, either employment or unemployment)

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(s)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed when the state is $s \in \mathbb{S}$.

In particular, the agent has wage offer $w(s)$ in hand.

More precisely, $v^*(s)$ denotes the value of the objective function (1) when an agent in this situation makes *optimal* decisions now and at all future points in time.

Of course $v^*(s)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

But think of v^* as a function that assigns to each possible state s the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy the recursion

$$v^*(s) = \max \left\{ \frac{w(s)}{1-\beta}, \, c + \beta \sum_{s' \in \mathbb{S}} v^*(s') q(s') \right\} \tag{1}$$

for every possible s in $\mathbb{S}$.

This important equation is a version of the **Bellman equation**, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since

$$\frac{w(s)}{1-\beta} = w(s) + \beta w(s) + \beta^2 w(s) + \cdots$$

- the second term inside the max operation is the **continuation value**, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current state s.

But this is precisely $v^*(s)$, which is the l.h.s. of (1).

27.2.3 The Optimal Policy

Suppose for now that we are able to solve (1) for the unknown function v^*.

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the r.h.s. of (1).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given *any* s, we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of (1).

Thus, we have a map from $\mathbb{R}$ to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(s) := \mathbf{1} \left\{ \frac{w(s)}{1-\beta} \geq c + \beta \sum_{s' \in \mathbb{S}} v^*(s') q(s') \right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(s) := \mathbf{1}\{w(s) \geq \bar{w}\}$$

where

$$\bar{w} := (1-\beta) \left\{ c + \beta \sum_{s'} v^*(s') q(s') \right\} \tag{2}$$

Here $\bar{w}$ (called the *reservation wage*) is a constant depending on β, c and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (2), we can compute this reservation wage if we can compute the value function.

27.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $s \in \mathbb{S}$.

Let's suppose that $\mathbb{S} = \{1, \dots, n\}$.

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v^*(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{3}$$

27.3.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{4}$$

Step 3: calculate a measure of the deviation between v and v', such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return v.

For small tolerance, the returned function v is a close approximation to the value function v^*.

The theory below elaborates on this point.

27.3.2 The Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \ldots, n \tag{5}$$

(A new vector Tv is obtained from given vector v by evaluating the r.h.s. at each i.)

The element v_k in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is T applied k times, starting at the initial guess v

One can show that the conditions of the Banach fixed point theorem are satisfied by T on $\mathbb{R}^n$.

One implication is that T has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it's immediate from the definition of T that this fixed point is v^*.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point v^* regardless of v.

27.3.3 Implementation

Our default for q, the distribution of the state process, will be Beta-binomial.

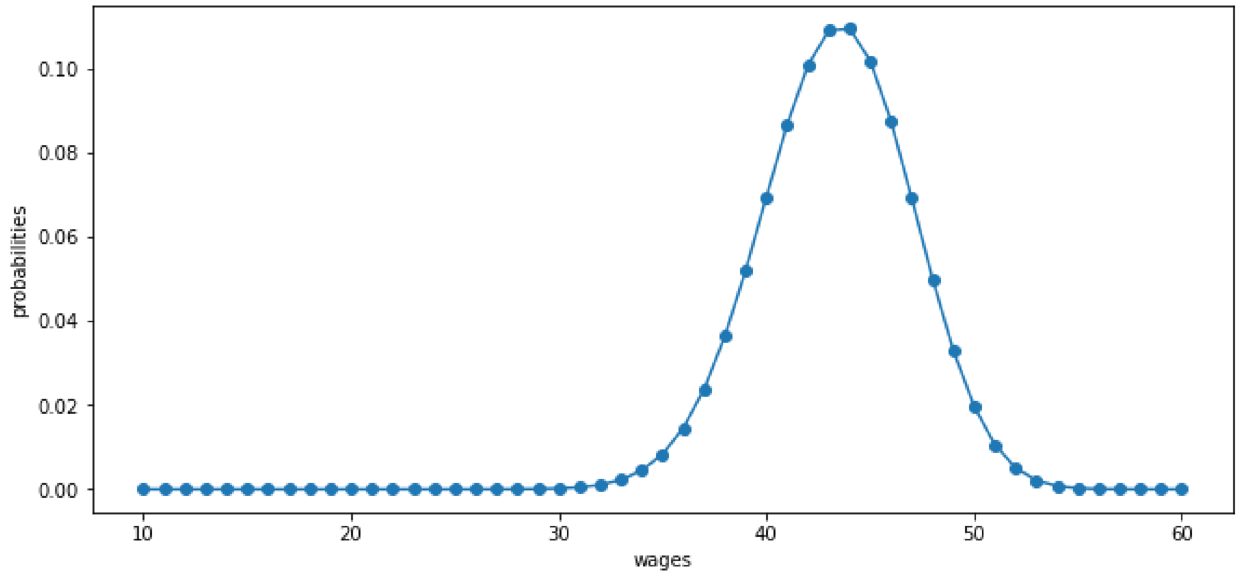
```python
n, a, b = 50, 200, 100                        # default parameters
q_default = BetaBinomial(n, a, b).pdf()       # default choice of q
```

Our default set of values for wages will be

```python
w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```python
fig, ax = plt.subplots()
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



We are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in our lecture on Numba.

The following helps Numba by providing some type

```
mccall_data = [
    ('c', float64),      # unemployment compensation
    ('β', float64),      # discount factor
    ('w', float64[:]),   # array of wage values, w[i] = wage at state i
    ('q', float64[:])    # array of probabilities
]
```

Here's a class that stores the data and computes the values of state-action pairs, i.e. the value in the maximum bracket on the right hand side of the Bellman equation (4), given the current state and an arbitrary feasible action.

Default parameter values are embedded in the class.

```
@jitclass(mccall_data)
class McCallModel:

    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

        self.c, self.β = c, β
        self.w, self.q = w_default, q_default

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
        """
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate value for each state-action pair
        # Consider action = accept or reject the current offer
        accept = w[i] / (1 - β)
        reject = c + β * np.sum(v * q)

        return np.array([accept, reject])
```

Based on these defaults, let's try plotting the first few approximate value functions in the sequence $\{T^k v\}$.

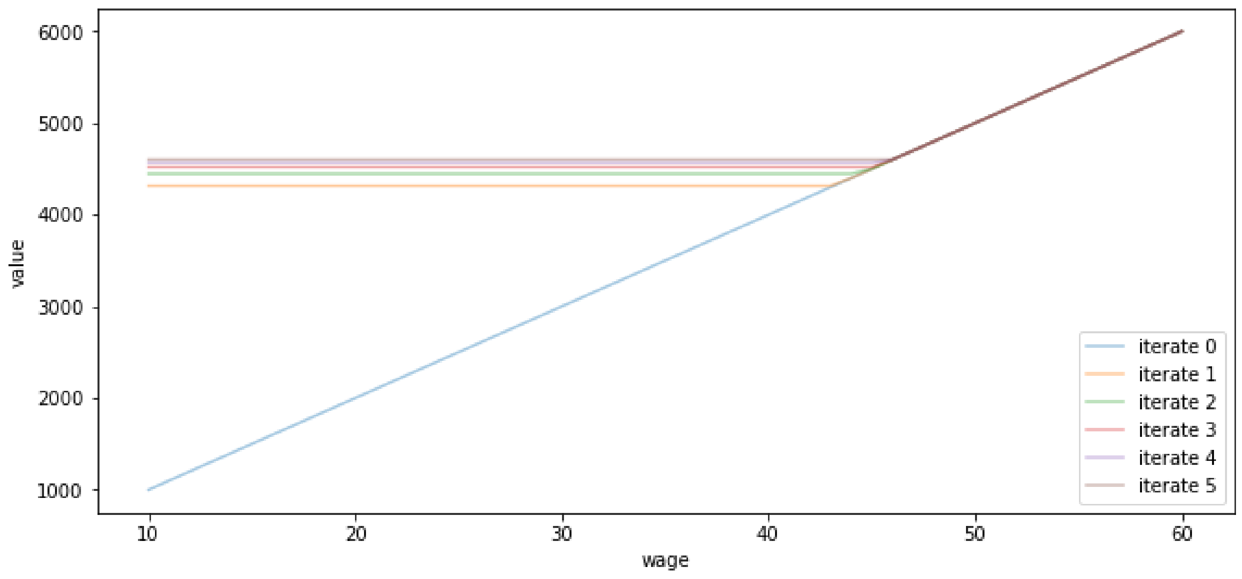
We will start from guess v given by $v(i) = w(i)/(1 - \beta)$, which is the value of accepting at every given wage.

Here's a function to implement this:

```python
def plot_value_function_seq(mcm, ax, num_plots=6):
    """
    Plot a sequence of value functions.

        * mcm is an instance of McCallModel
        * ax is an axes object that implements a plot method.

    """

    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

Now let's create an instance of `McCallModel` and call the function:

```python
mcm = McCallModel()

fig, ax = plt.subplots()
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
plt.show()
```



You can see that convergence is occuring: successive iterates are getting closer together.

Here's a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below tol.

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

We'll be using JIT compilation via Numba to turbocharge our loops.

```
@jit(nopython=True)
def compute_reservation_wage(mcm,
                             max_iter=500,
                             tol=1e-6):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #

    n = len(w)
    v = w / (1 - β)          # initial guess
    v_next = np.empty_like(v)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))

        error = np.max(np.abs(v_next - v))
        i += 1

        v[:] = v_next  # copy contents into v

    # == Now compute the reservation wage == #

    return (1 - β) * (c + β * np.sum(v * q))
```

The next line computes the reservation wage at the default parameters

```
compute_reservation_wage(mcm)
```

```
47.316499710024964
```

27.3.4 Comparative Statics

Now we know how to compute the reservation wage, let's see how it varies with parameters.

In particular, let's look at what happens when we change β and c.

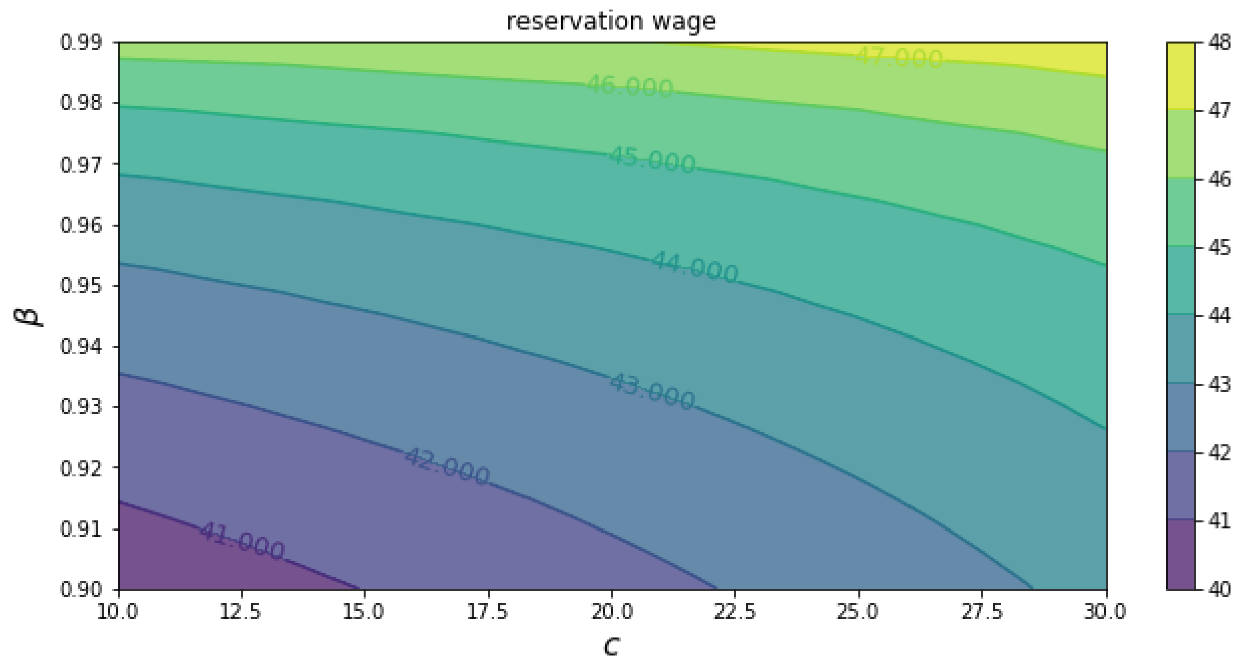
```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
```

(continues on next page)

(continued from previous page)

```
        mcm = McCallModel(c=c, β=β)
        R[i, j] = compute_reservation_wage(mcm)
```

```
fig, ax = plt.subplots()

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```



As expected, the reservation wage increases both with patience and with unemployment compensation.

27.4 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there's also an easier way, which circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{s'} v^*(s') q(s') \tag{6}$$

The Bellman equation can now be written as

$$v^*(s') = \max \left\{ \frac{w(s')}{1-\beta}, \, h \right\}$$

Substituting this last equation into (6) gives

$$h = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{7}$$

This is a nonlinear equation that we can solve for h.

As before, we will use successive approximations:

Step 1: pick an initial guess h.

Step 2: compute the update h' via

$$h' = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{8}$$

Step 3: calculate the deviation $|h - h'|$.

Step 4: if the deviation is larger than some fixed tolerance, set $h = h'$ and go to step 2, else return h.

Once again, one can use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we're iterating on a single number, rather than an n-vector.

Here's an implementation:

```python
@jit(nopython=True)
def compute_reservation_wage_two(mcm,
                                 max_iter=500,
                                 tol=1e-5):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute h == #

    h = np.sum(w * q) / (1 - β)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        s = np.maximum(w / (1 - β), h)
        h_next = c + β * np.sum(s * q)

```

(continues on next page)

(continued from previous page)

```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

You can use this code to solve the exercise below.

27.5 Exercises

27.5.1 Exercise 1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 25)
```

That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of c in `c_vals`.

27.5.2 Exercise 2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that h in (6) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods:

To shift to a continuous offer distribution, we can replace (6) by

$$h = c + \beta \int v^*(s')q(s')ds'. \tag{9}$$

Equation (7) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s')ds' \tag{10}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use `c=25, β=0.99, σ=0.5, μ=2.5`.

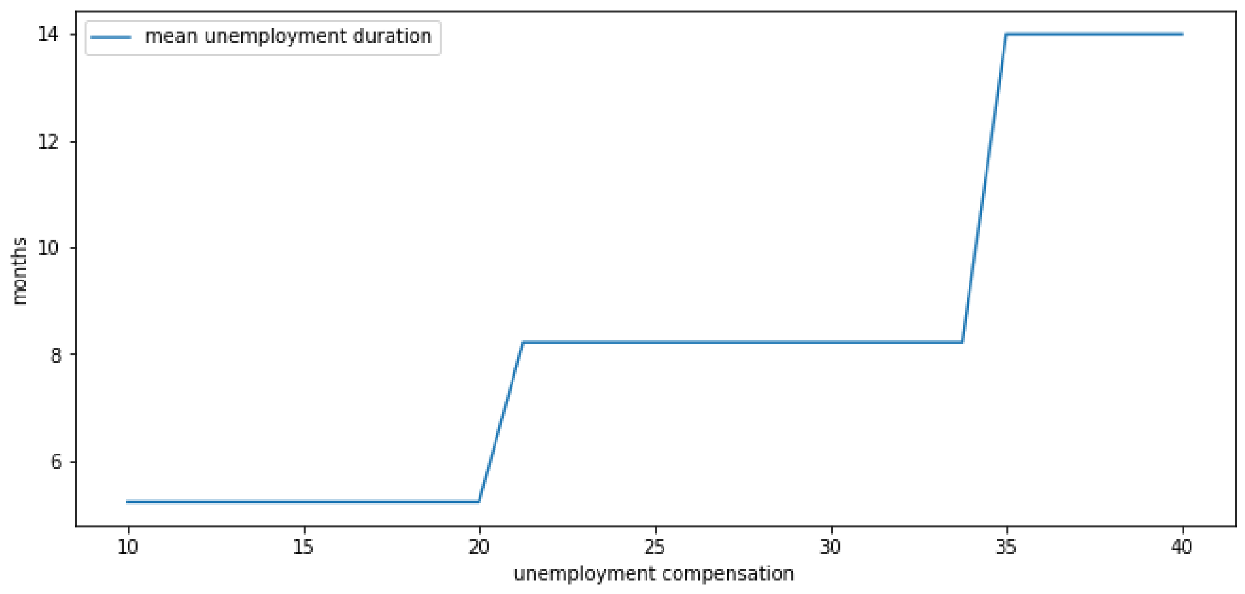
Once your code is working, investigate how the reservation wage changes with c and β.

27.6 Solutions

27.6.1 Exercise 1

Here's one solution

```
cdf = np.cumsum(q_default)

@jit(nopython=True)
def compute_stopping_time(w_bar, seed=1234):

    np.random.seed(seed)
    t = 1
    while True:
        # Generate a wage draw
        w = w_default[qe.random.draw(cdf)]
        # Stop when the draw is above the reservation wage
        if w >= w_bar:
            stopping_time = t
            break
        else:
            t += 1
    return stopping_time

@jit(nopython=True)
def compute_mean_stopping_time(w_bar, num_reps=100000):
    obs = np.empty(num_reps)
    for i in range(num_reps):
        obs[i] = compute_stopping_time(w_bar, seed=i)
    return obs.mean()

c_vals = np.linspace(10, 40, 25)
stop_times = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    mcm = McCallModel(c=c)
    w_bar = compute_reservation_wage_two(mcm)
    stop_times[i] = compute_mean_stopping_time(w_bar)

fig, ax = plt.subplots()

ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()

plt.show()
```



27.6.2 Exercise 2

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('β', float64),          # discount factor
    ('σ', float64),          # scale parameter in lognormal distribution
    ('μ', float64),          # location parameter in lognormal distribution
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):

        self.c, self.β, self.σ, self.μ = c, β, σ, μ

        # Draw and store shocks
        np.random.seed(1234)
        s = np.random.randn(mc_size)
        self.w_draws = np.exp(μ+ σ * s)


@jit(nopython=True)
def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):

    c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws

    h = np.mean(w_draws) / (1 - β)  # initial guess
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        integral = np.mean(np.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral
```

(continues on next page)

(continued from previous page)

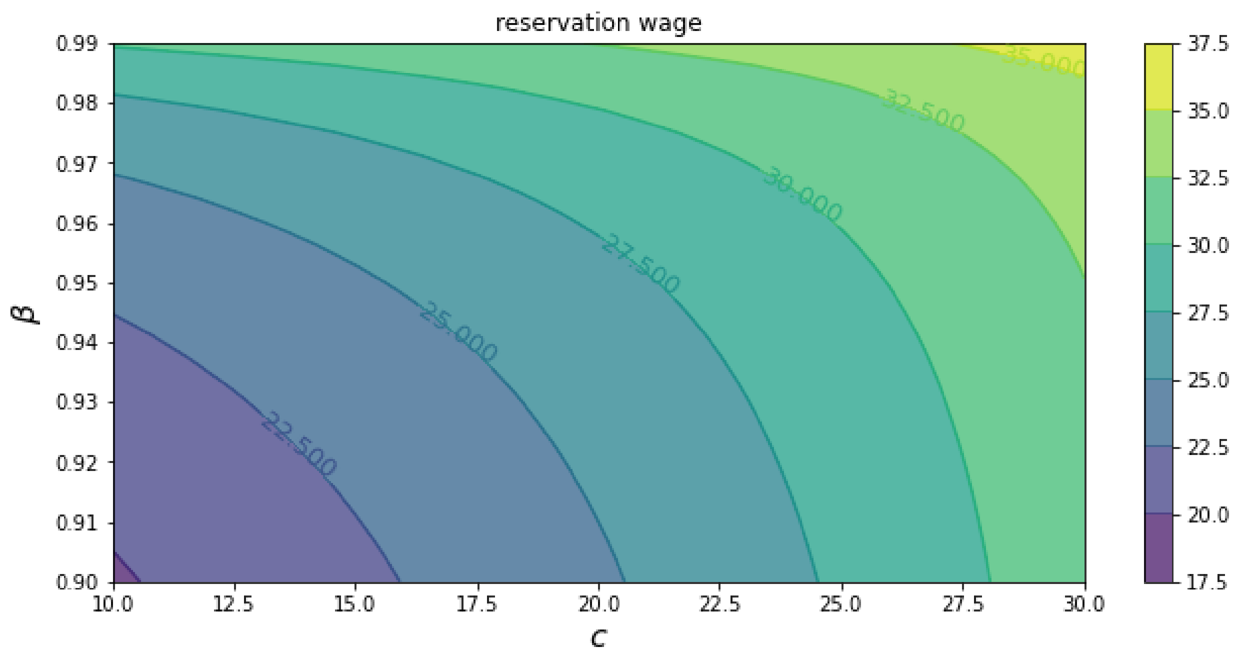
```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

Now we investigate how the reservation wage changes with $c$ and β.

We will do this using a contour plot.

```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcmc = McCallModelContinuous(c=c, β=β)
        R[i, j] = compute_reservation_wage_continuous(mcmc)
```

```
fig, ax = plt.subplots()

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```


reservation wage
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.90
β
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
c
22.500
25.000
27.500
30.000
32.500
35.000
37.5
35.0
32.5
30.0
27.5
25.0
22.5
20.0
17.5

CHAPTER

TWENTYEIGHT

JOB SEARCH II: SEARCH AND SEPARATION

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
```

28.1 Overview

Previously *we looked* at the McCall job search model [McC70] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture, we extend the McCall model by introducing job separation.

Once separation enters the picture, the agent comes to view

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

The other minor addition is that a utility function will be included to make worker preferences slightly more sophisticated.

We'll need the following imports

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
```

(continues on next page)

(continued from previous page)

```python
import numpy as np
from numba import njit, float64
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial
```

28.2 The Model

The model is similar to the *baseline McCall job search model*.

It concerns the life of an infinitely lived worker and

- the opportunities he or she (let's say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{1}$$

At this stage the only difference from the *baseline model* is that we've added some flexibility to preferences by introducing a utility function u.

It satisfies $u' > 0$ and $u'' < 0$.

28.2.1 The Wage Process

For now we will drop the separation of state process and wage process that we maintained for the *baseline model*.

In particular, we simply suppose that wage offers $\{w_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

(Later we will go back to having a separate state process $\{s_t\}$ driving random outcomes, since this formulation is usually convenient in more sophisticated models.)

28.2.2 Timing and Decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w_e.

At the start of a given period, the current wage offer w_t is observed.

If currently *employed*, the worker

1. receives utility $u(w_e)$ and
2. is fired with some (small) probability α.

If currently *unemployed*, the worker either accepts or rejects the current offer w_t.

If he accepts, then he begins work immediately at wage w_t.

If he rejects, then he receives unemployment compensation c.

The process then repeats.

(Note: we do not allow for job search while employed—this topic is taken up in a *later lecture*.)

28.3 Solving the Model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v(w_e)$ be total lifetime value accruing to a worker who enters the current period *employed* with existing wage w_e
- $h(w)$ be total lifetime value accruing to a worker who who enters the current period *unemployed* and receives wage offer w.

Here *value* means the value of the objective function (1) when the worker makes optimal decisions at all future points in time.

Our first aim is to obtain these functions.

28.3.1 The Bellman Equations

Suppose for now that the worker can calculate the functions v and h and use them in his decision making.

Then v and h should satisfy

$$v(w_e) = u(w_e) + \beta \left[(1-\alpha) v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w') q(w') \right] \tag{2}$$

and

$$h(w) = \max \left\{ v(w), \, u(c) + \beta \sum_{w' \in \mathbb{W}} h(w') q(w') \right\} \tag{3}$$

Equation (2) expresses the value of being employed at wage w_e in terms of

- current reward $u(w_e)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

Equation (3) expresses the value of being unemployed with offer w in hand as a maximum over the value of two options: accept or reject the current offer.

Accepting transitions the worker to employment and hence yields reward $v(w)$.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equations (2) and (3) are the Bellman equations for this model.

They provide enough information to solve for both v and h.

28.3.2 A Simplifying Transformation

Rather than jumping straight into solving these equations, let's see if we can simplify them somewhat.

(This process will be analogous to our *second pass* at the plain vanilla McCall model, where we simplified the Bellman equation.)

First, let

$$d := \sum_{w' \in \mathbb{W}} h(w') q(w') \tag{4}$$

be the expected value of unemployment tomorrow.

We can now write (3) as

$$h(w) = \max \left\{ v(w), \, u(c) + \beta d \right\}$$

or, shifting time forward one period

$$\sum_{w' \in \mathbb{W}} h(w') q(w') = \sum_{w' \in \mathbb{W}} \max \left\{ v(w'), \, u(c) + \beta d \right\} q(w')$$

Using (4) again now gives

$$d = \sum_{w' \in \mathbb{W}} \max \left\{ v(w'), \, u(c) + \beta d \right\} q(w') \tag{5}$$

Finally, (2) can now be rewritten as

$$v(w) = u(w) + \beta \left[(1 - \alpha) v(w) + \alpha d \right] \tag{6}$$

In the last expression, we wrote w_e as w to make the notation simpler.

28.3.3 The Reservation Wage

Suppose we can use (5) and (6) to solve for d and v.

(We will do this soon.)

We can then determine optimal behavior for the worker.

From (3), we see that an unemployed agent accepts current offer w if $v(w) \geq u(c) + \beta d$.

This means precisely that the value of accepting is higher than the expected value of rejecting.

It is clear that v is (at least weakly) increasing in w, since the agent is never made worse off by a higher wage offer.

Hence, we can express the optimal choice as accepting wage offer w if and only if

$$w \geq \bar{w} \quad \text{where} \quad \bar{w} \text{ solves } v(\bar{w}) = u(c) + \beta d$$

28.3.4 Solving the Bellman Equations

We'll use the same iterative approach to solving the Bellman equations that we adopted in the *first job search lecture*.

Here this amounts to

1. make guesses for d and v
2. plug these guesses into the right-hand sides of (5) and (6)

3. update the left-hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$d_{n+1} = \sum_{w' \in \mathbb{W}} \max \left\{ v_n(w'), \, u(c) + \beta d_n \right\} q(w') \tag{7}$$

$$v_{n+1}(w) = u(w) + \beta \left[(1 - \alpha) v_n(w) + \alpha d_n \right] \tag{8}$$

starting from some initial conditions d_0, v_0.

As before, the system always converges to the true solutions—in this case, the v and d that solve (5) and (6).

(A proof can be obtained via the Banach contraction mapping theorem.)

28.4 Implementation

Let's implement this iterative process.

In the code, you'll see that we use a class to store the various parameters and other objects associated with a given model.

This helps to tidy up the code and provides an object that's easy to pass to functions.

The default utility function is a CRRA utility function

```
@njit
def u(c, σ=2.0):
    return (c**(1 - σ) - 1) / (1 - σ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
n = 60                                     # n possible outcomes for w
w_default = np.linspace(10, 20, n)         # wages between 10 and 20
a, b = 600, 400                            # shape parameters
dist = BetaBinomial(n-1, a, b)
q_default = dist.pdf()
```

Here's our jitted class for the McCall model with separation.

```
mccall_data = [
    ('α', float64),       # job separation rate
    ('β', float64),       # discount factor
    ('c', float64),       # unemployment compensation
    ('w', float64[:]),    # list of wage values
    ('q', float64[:])     # pmf of random variable w
]

@jitclass(mccall_data)
class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, α=0.2, β=0.98, c=6.0, w=w_default, q=q_default):

        self.α, self.β, self.c, self.w, self.q = α, β, c, w, q


```

(continues on next page)

(continued from previous page)

```
    def update(self, v, d):

        α, β, c, w, q = self.α, self.β, self.c, self.w, self.q

        v_new = np.empty_like(v)

        for i in range(len(w)):
            v_new[i] = u(w[i]) + β * ((1 - α) * v[i] + α * d)

        d_new = np.sum(np.maximum(v, u(c) + β * d) * q)

        return v_new, d_new
```

Now we iterate until successive realizations are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w)    # Initial guess of v
    d = 1                      # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

28.4.1 The Reservation Wage: First Pass

The optimal choice of the agent is summarized by the reservation wage.

As discussed above, the reservation wage is the $\bar{w}$ that solves $v(\bar{w}) = h$ where $h := u(c) + \beta d$ is the continuation value.

Let's compare v and h to see what they look like.

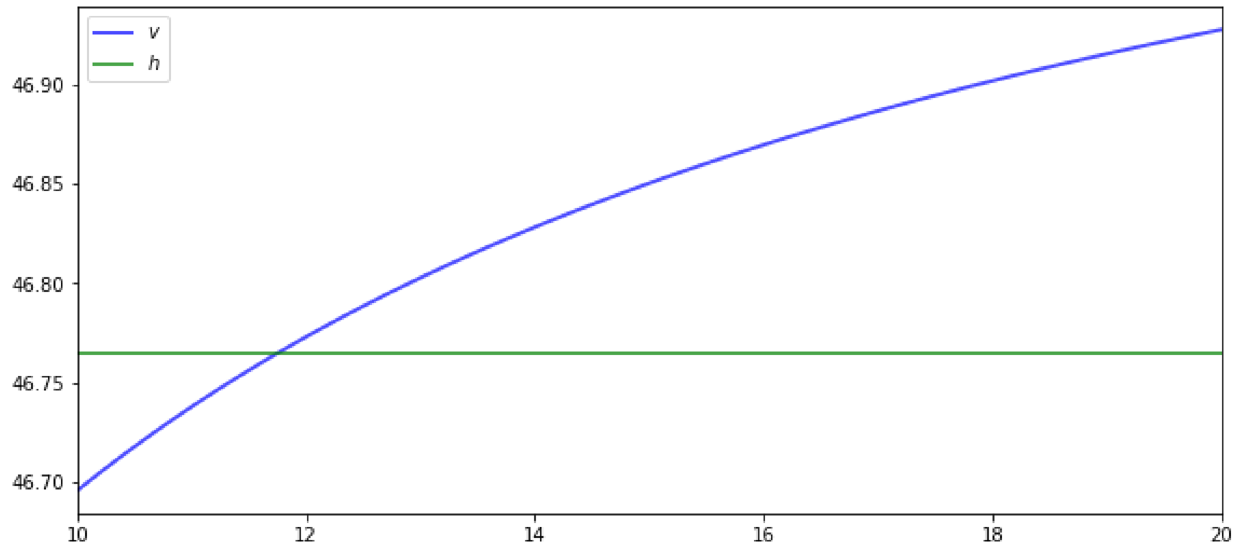
We'll use the default parameterizations found in the code above.

```
mcm = McCallModel()
v, d = solve_model(mcm)
h = u(mcm.c) + mcm.β * d

fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.plot(mcm.w, v, 'b-', lw=2, alpha=0.7, label='$v$')
ax.plot(mcm.w, [h] * len(mcm.w),
        'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(mcm.w), max(mcm.w))
ax.legend()

plt.show()
```



The value v is increasing because higher w generates a higher wage flow conditional on staying employed.

28.4.2 The Reservation Wage: Computation

Here's a function `compute_reservation_wage` that takes an instance of `McCallModel` and returns the associated reservation wage.

```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

Next we will investigate how the reservation wage varies with parameters.

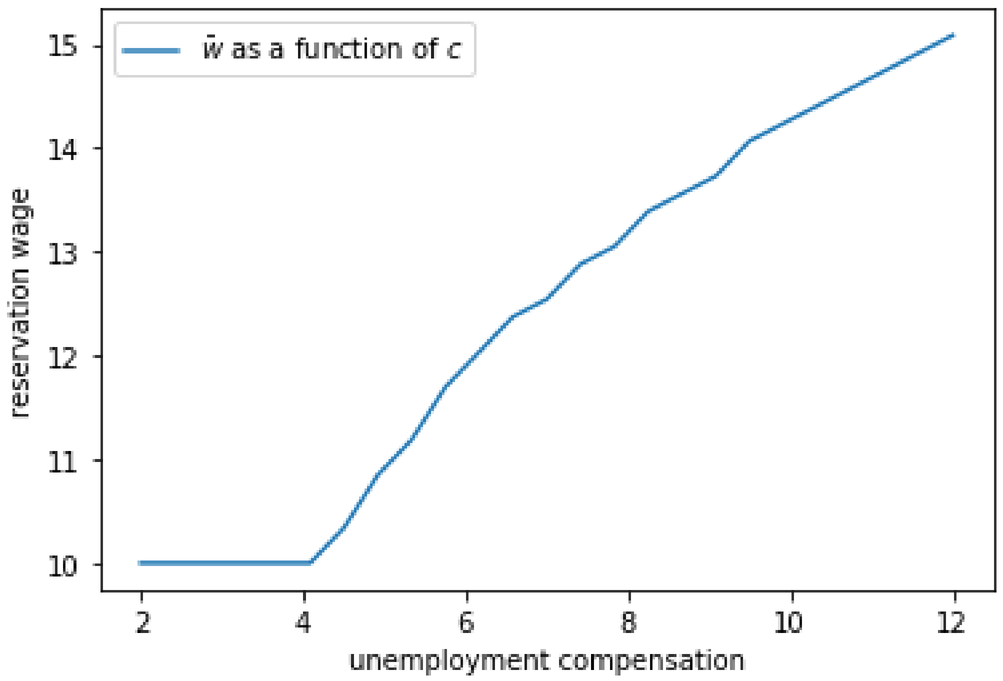
28.5 Impact of Parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

28.5.1 The Reservation Wage and Unemployment Compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation.

In the figure below, we use the default parameters in the `McCallModel` class, apart from c (which takes the values given on the horizontal axis)



As expected, higher unemployment compensation causes the worker to hold out for higher wages.

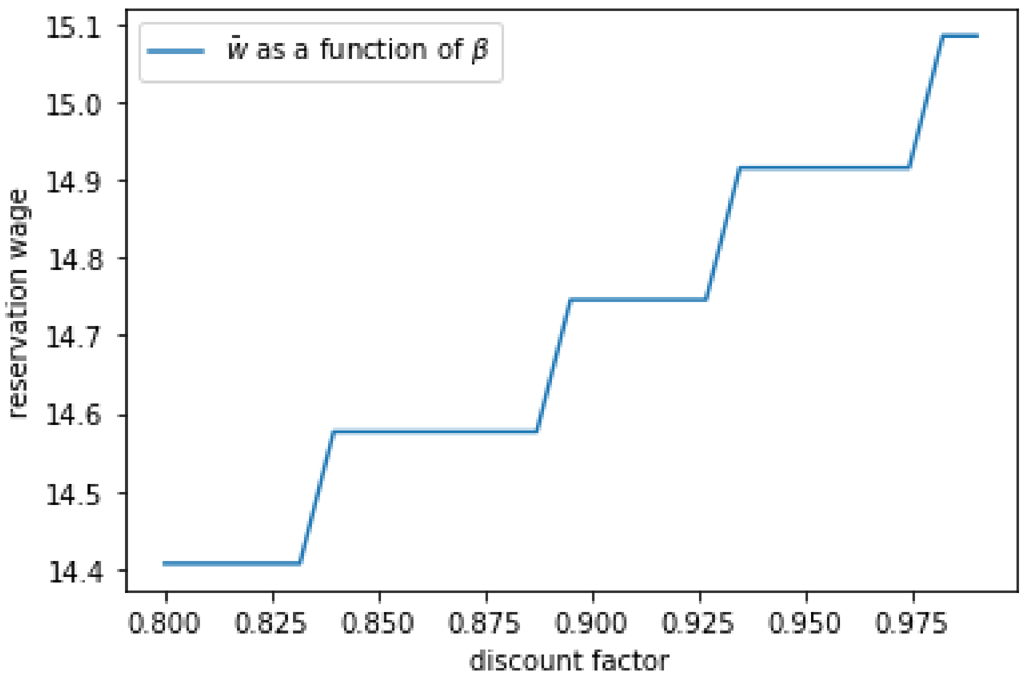
In effect, the cost of continuing job search is reduced.

28.5.2 The Reservation Wage and Discounting

Next, let's investigate how $\bar{w}$ varies with the discount factor.

The next figure plots the reservation wage associated with different values of β

Again, the results are intuitive: More patient workers will hold out for higher wages.



28.5.3 The Reservation Wage and Job Destruction

Finally, let's look at how $\bar{w}$ varies with the job separation rate α.

Higher α translates to a greater chance that a worker will face termination in each period once employed.

Once more, the results are in line with our intuition.

If the separation rate is high, then the benefit of holding out for a higher wage falls.

Hence the reservation wage is lower.

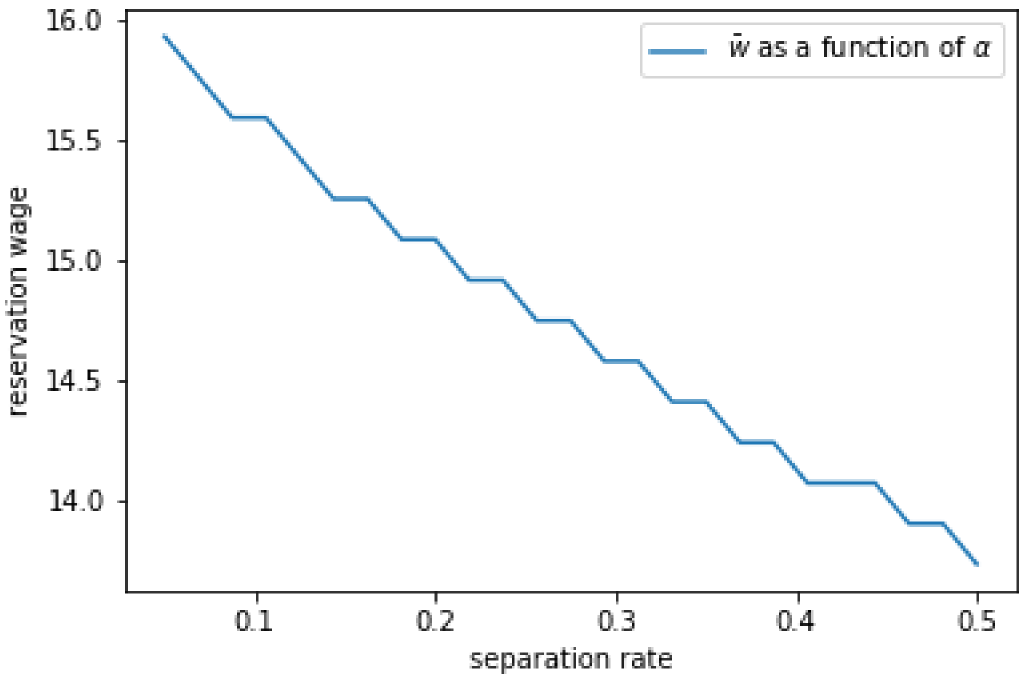
28.6 Exercises

28.6.1 Exercise 1

Reproduce all the reservation wage figures shown above.

Regarding the values on the horizontal axis, use

```
grid_size = 25
c_vals = np.linspace(2, 12, grid_size)         # unemployment compensation
beta_vals = np.linspace(0.8, 0.99, grid_size)  # discount factors
alpha_vals = np.linspace(0.05, 0.5, grid_size) # separation rate
```

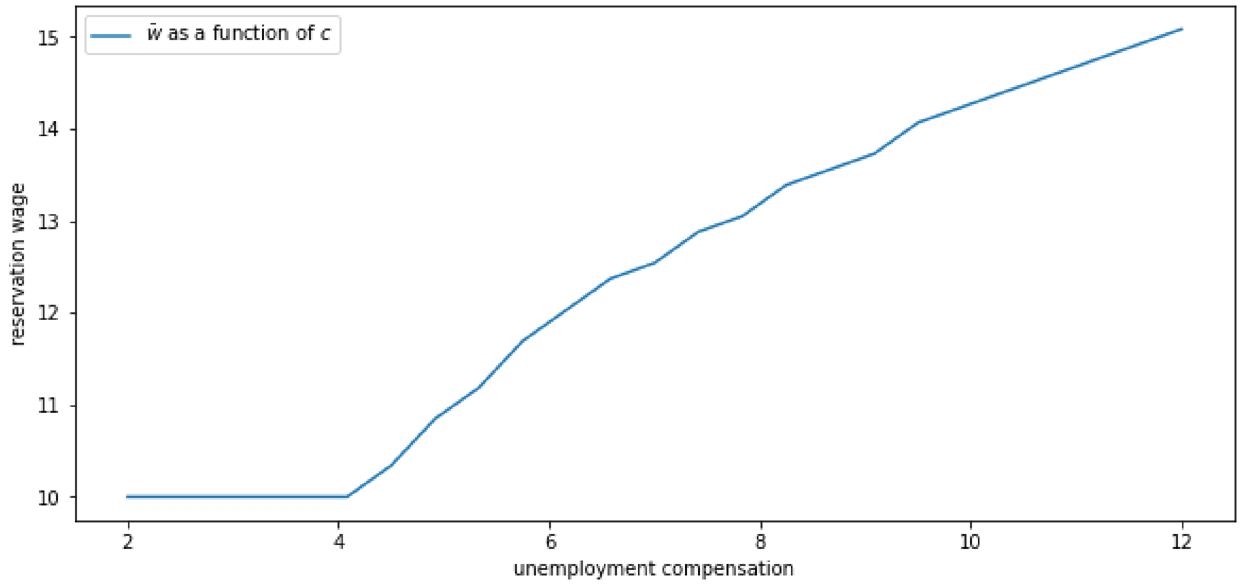


28.7 Solutions

28.7.1 Exercise 1

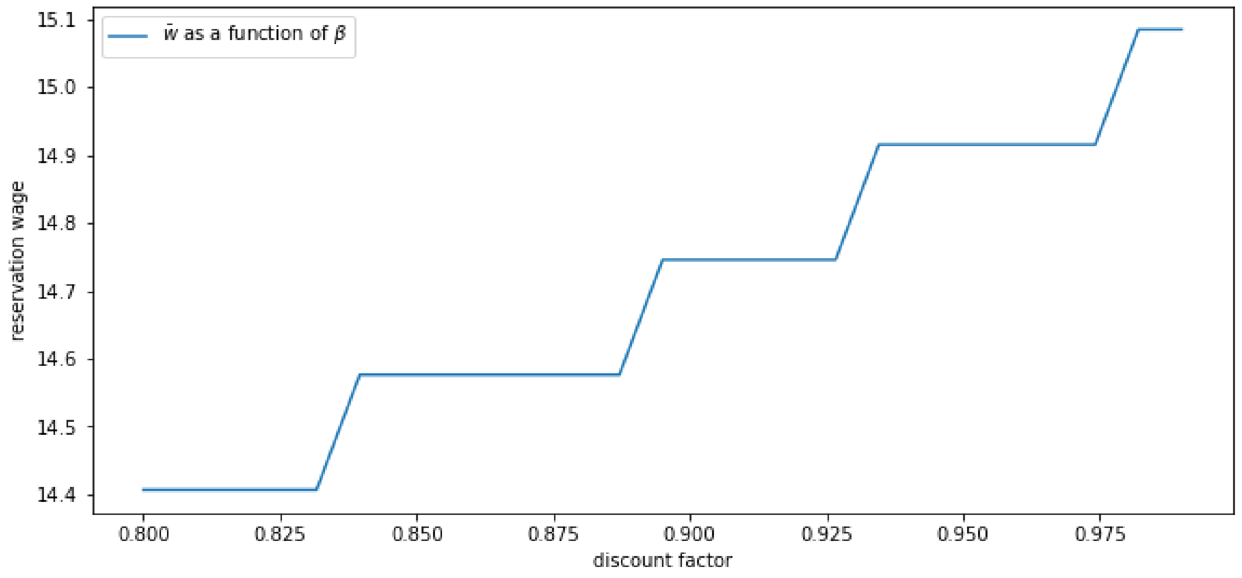
Here's the first figure.

```
mcm = McCallModel()

w_bar_vals = np.empty_like(c_vals)

fig, ax = plt.subplots()

for i, c in enumerate(c_vals):
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='unemployment compensation',
       ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, label=r'$\bar w$ as a function of $c$')
ax.legend()

plt.show()
```



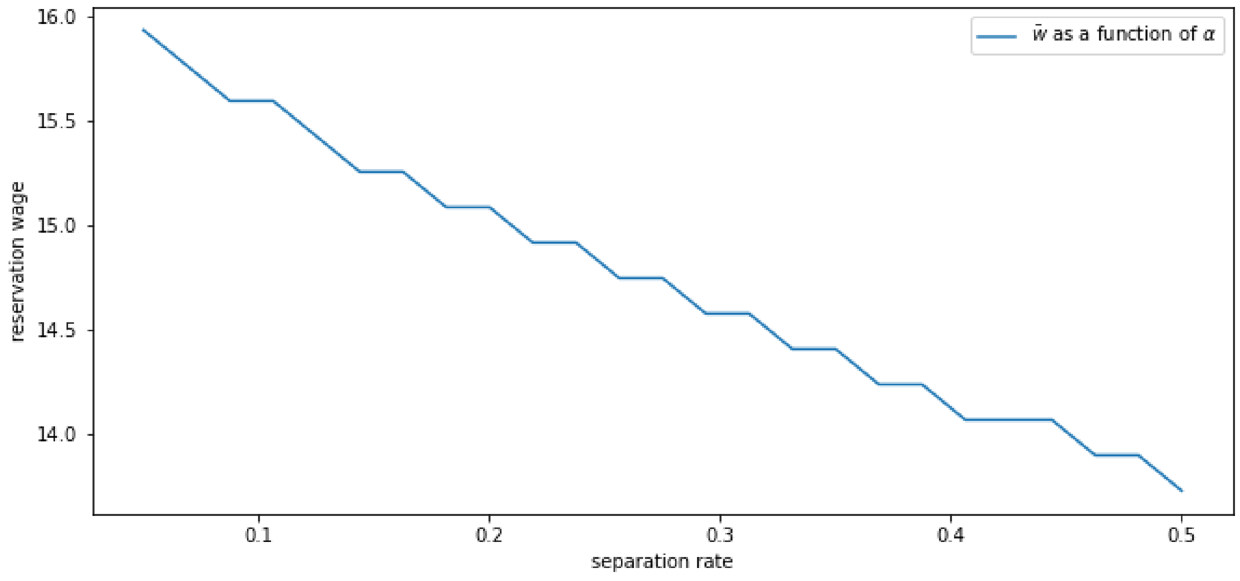
Here's the second one.

```
fig, ax = plt.subplots()

for i, β in enumerate(beta_vals):
    mcm.β = β
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(beta_vals, w_bar_vals, label=r'$\bar w$ as a function of $\beta$')
ax.legend()

plt.show()
```



Here's the third.

```
fig, ax = plt.subplots()

for i, α in enumerate(alpha_vals):
    mcm.α = α
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(alpha_vals, w_bar_vals, label=r'$\bar w$ as a function of $\alpha$')
ax.legend()

plt.show()
```



CHAPTER

TWENTYNINE

JOB SEARCH III: FITTED VALUE FUNCTION ITERATION

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
!pip install interpolation
```

29.1 Overview

In this lecture we again study the *McCall job search model with separation*, but now with a continuous wage distribution.

While we already considered continuous wage distributions briefly in the exercises of the *first job search lecture*, the change was relatively trivial in that case.

This is because we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, with separation, the change is less trivial, since a continuous wage distribution leads to an uncountably infinite state space.

The infinite state space leads to additional challenges, particularly when it comes to applying value function iteration (VFI).

These challenges will lead us to modify VFI by adding an interpolation step.

The combination of VFI and this interpolation step is called **fitted value function iteration** (fitted VFI).

Fitted VFI is very common in practice, so we will take some time to work through the details.

We will use the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
```

(continues on next page)

(continued from previous page)

```python
import numpy as np
import quantecon as qe
from interpolation import interp
from numpy.random import randn
from numba import njit, prange, float64, int32
from numba.experimental import jitclass
```

29.2 The Algorithm

The model is the same as the McCall model with job separation we *studied before*, except that the wage offer distribution is continuous.

We are going to start with the two Bellman equations we obtained for the model with job separation after *a simplifying transformation*.

Modified to accommodate continuous wage draws, they take the following form:

$$d = \int \max \left\{ v(w'), \, u(c) + \beta d \right\} q(w') dw' \tag{1}$$

and

$$v(w) = u(w) + \beta \left[(1 - \alpha) v(w) + \alpha d \right] \tag{2}$$

The unknowns here are the function v and the scalar d.

The difference between these and the pair of Bellman equations we previously worked on are

1. in (1), what used to be a sum over a finite number of wage values is an integral over an infinite set.
2. The function v in (2) is defined over all $w \in \mathbb{R}_+$.

The function q in (1) is the density of the wage offer distribution.

Its support is taken as equal to $\mathbb{R}_+$.

29.2.1 Value Function Iteration

In theory, we should now proceed as follows:

1. Begin with a guess v, d for the solutions to (1)–(2).
2. Plug v, d into the right hand side of (1)–(2) and compute the left hand side to obtain updates v', d'
3. Unless some stopping condition is satisfied, set $(v, d) = (v', d')$ and go to step 2.

However, there is a problem we must confront before we implement this procedure: The iterates of the value function can neither be calculated exactly nor stored on a computer.

To see the issue, consider (2).

Even if v is a known function, the only way to store its update v' is to record its value $v'(w)$ for every $w \in \mathbb{R}_+$.

Clearly, this is impossible.

29.2.2 Fitted Value Function Iteration

What we will do instead is use **fitted value function iteration**.

The procedure is as follows:

Let a current guess v be given.

Now we record the value of the function v' at only finitely many "grid" points $w_1 < w_2 < \cdots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array **v** representing the values of an initial guess of the value function on some grid points $\{w_i\}$.
2. Build a function v on the state space $\mathbb{R}_+$ by interpolation or approximation, based on **v** and $\{w_i\}$.
3. Obtain and record the samples of the updated function $v'(w_i)$ on each grid point w_i.
4. Unless some stopping condition is satisfied, take this as the new array and go to step 1.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to each v, but also that it combines well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation.

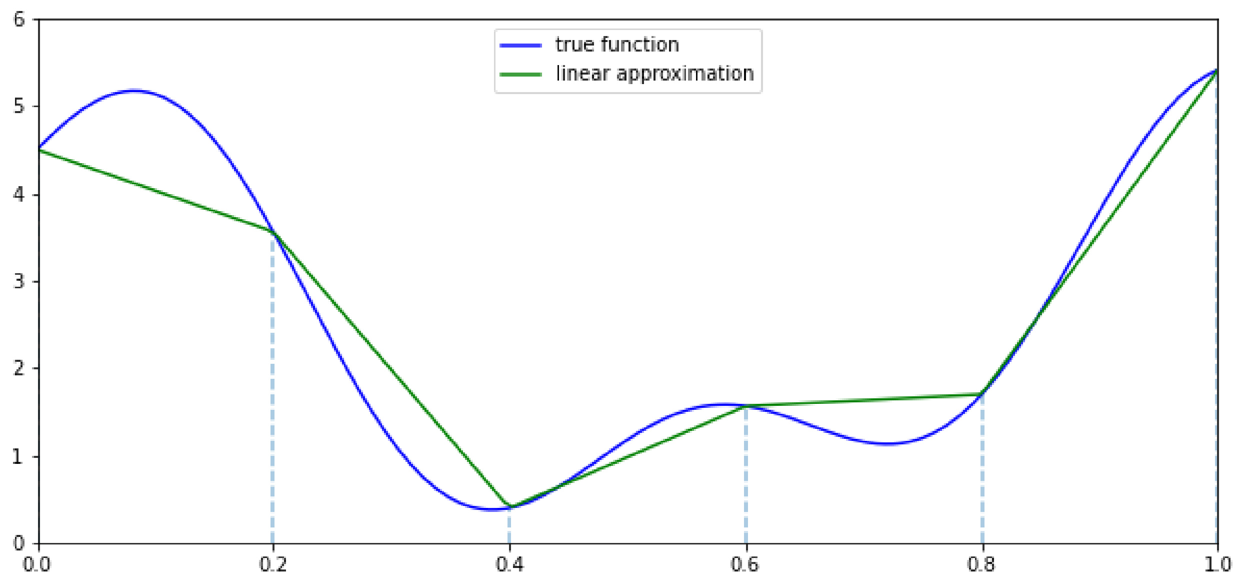
This method

1. combines well with value function iteration (see., e.g., [Gor95] or [Sta08]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using a JIT-aware Python interpolation library called interpolation.py.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

```python
def f(x):
    y1 = 2 * np.cos(6 * x) + np.sin(14 * x)
    return y1 + 2.5

c_grid = np.linspace(0, 1, 6)
f_grid = np.linspace(0, 1, 150)

def Af(x):
    return interp(c_grid, f(c_grid), x)

fig, ax = plt.subplots()

ax.plot(f_grid, f(f_grid), 'b-', label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)

ax.legend(loc="upper center")

ax.set(xlim=(0, 1), ylim=(0, 6))
plt.show()
```



29.3 Implementation

The first step is to build a jitted class for the McCall model with separation and a continuous wage offer distribution.

We will take the utility function to be the log function for this application, with $u(c) = \ln c$.

We will adopt the lognormal distribution for wages, with $w = \exp(\mu + \sigma z)$ when z is standard normal and μ, σ are parameters.

```
@njit
def lognormal_draws(n=1000, μ=2.5, σ=0.5, seed=1234):
    np.random.seed(seed)
    z = np.random.randn(n)
    w_draws = np.exp(μ + σ * z)
    return w_draws
```

Here's our class.

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('α', float64),          # job separation rate
    ('β', float64),          # discount factor
    ('σ', float64),          # scale parameter in lognormal distribution
    ('μ', float64),          # location parameter in lognormal distribution
    ('w_grid', float64[:]),  # grid of points for fitted VFI
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self,
                 c=1,
                 α=0.1,
                 β=0.96,
```

(continues on next page)

(continued from previous page)

```
                 grid_min=1e-10,
                 grid_max=5,
                 grid_size=100,
                 w_draws=lognormal_draws()):

        self.c, self.α, self.β = c, α, β

        self.w_grid = np.linspace(grid_min, grid_max, grid_size)
        self.w_draws = w_draws

    def update(self, v, d):

        # Simplify names
        c, α, β, σ, μ = self.c, self.α, self.β, self.σ, self.μ
        w = self.w_grid
        u = lambda x: np.log(x)

        # Interpolate array represented value function
        vf = lambda x: interp(w, v, x)

        # Update d using Monte Carlo to evaluate integral
        d_new = np.mean(np.maximum(vf(self.w_draws), u(c) + β * d))

        # Update v
        v_new = u(w) + β * ((1 - α) * v + α * d)

        return v_new, d_new
```

We then return the current iterate as an approximate solution.

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w_grid)    # Initial guess of v
    d = 1                           # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

Here's a function `compute_reservation_wage` that takes an instance of `McCallModelContinuous` and returns the associated reservation wage.

If $v(w) < h$ for all w, then the function returns np.inf

```python
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """
    u = lambda x: np.log(x)

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w_grid):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

The exercises ask you to explore the solution and how it changes with parameters.

29.4 Exercises

29.4.1 Exercise 1

Use the code above to explore what happens to the reservation wage when the wage parameter μ changes.

Use the default parameters and μ in `mu_vals = np.linspace(0.0, 2.0, 15)`.

Is the impact on the reservation wage as you expected?

29.4.2 Exercise 2

Let us now consider how the agent responds to an increase in volatility.

To try to understand this, compute the reservation wage when the wage offer distribution is uniform on $(m - s, m + s)$ and s varies.

The idea here is that we are holding the mean constant and spreading the support.

(This is a form of *mean-preserving spread*.)

Use `s_vals = np.linspace(1.0, 2.0, 15)` and `m = 2.0`.

State how you expect the reservation wage to vary with s.

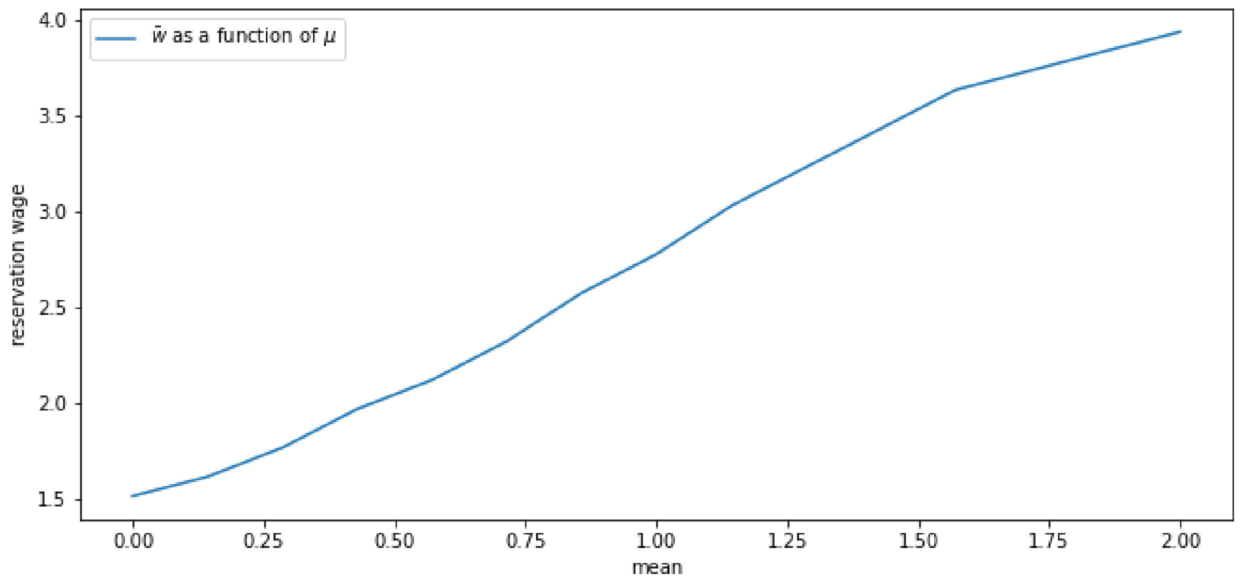
Now compute it. Is this as you expected?

29.5 Solutions

29.5.1 Exercise 1

Here is one solution.

```python
mcm = McCallModelContinuous()
mu_vals = np.linspace(0.0, 2.0, 15)
w_bar_vals = np.empty_like(mu_vals)

fig, ax = plt.subplots()

for i, m in enumerate(mu_vals):
    mcm.w_draws = lognormal_draws(μ=m)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='mean', ylabel='reservation wage')
ax.plot(mu_vals, w_bar_vals, label=r'$\bar w$ as a function of $\mu$')
ax.legend()

plt.show()
```

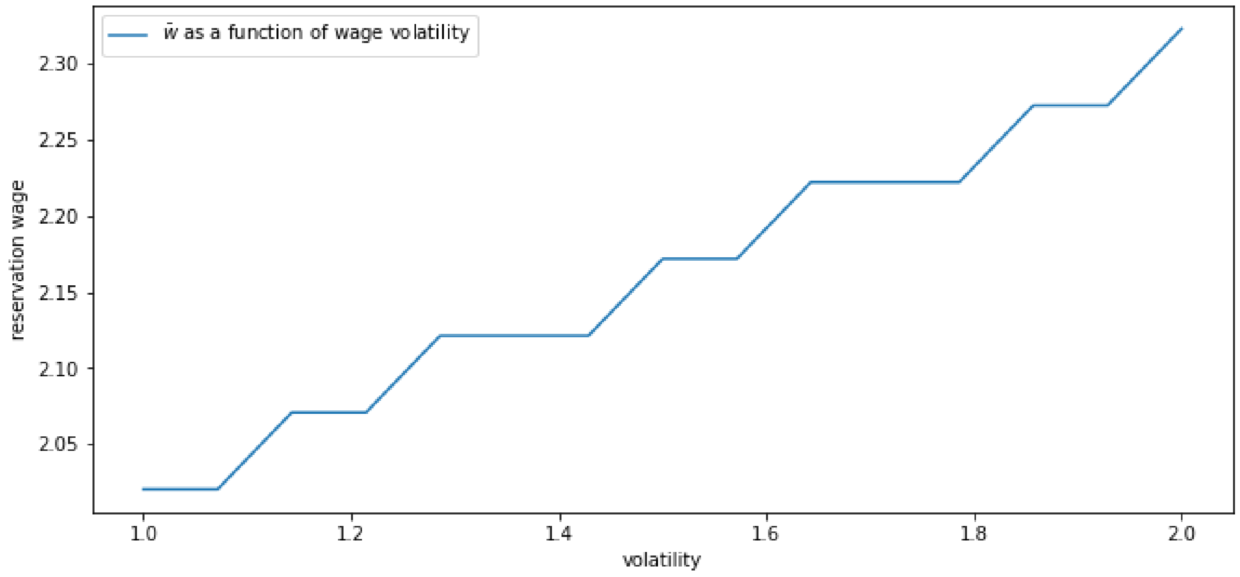


Not surprisingly, the agent is more inclined to wait when the distribution of offers shifts to the right.

29.5.2 Exercise 2

Here is one solution.

```python
mcm = McCallModelContinuous()
s_vals = np.linspace(1.0, 2.0, 15)
m = 2.0
w_bar_vals = np.empty_like(s_vals)

fig, ax = plt.subplots()

for i, s in enumerate(s_vals):
    a, b = m - s, m + s
    mcm.w_draws = np.random.uniform(low=a, high=b, size=10_000)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='volatility', ylabel='reservation wage')
ax.plot(s_vals, w_bar_vals, label=r'$\bar w$ as a function of wage volatility')
ax.legend()

plt.show()
```



The reservation wage increases with volatility.

One might think that higher volatility would make the agent more inclined to take a given offer, since doing so represents certainty and waiting represents risk.

But job search is like holding an option: the worker is only exposed to upside risk (since, in a free market, no one can force them to take a bad offer).

More volatility means higher upside potential, which encourages the agent to wait.

CHAPTER

THIRTY

JOB SEARCH IV: CORRELATED WAGE OFFERS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
!pip install interpolation
```

30.1 Overview

In this lecture we solve a *McCall style job search model* with persistent and transitory components to wages.

In other words, we relax the unrealistic assumption that randomness in wages is independent over time.

At the same time, we will go back to assuming that jobs are permanent and no separation occurs.

This is to keep the model relatively simple as we study the impact of correlation.

We will use the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from interpolation import interp
from numpy.random import randn
from numba import njit, prange, float64
from numba.experimental import jitclass
```

30.2 The Model

Wages at each point in time are given by

$$w_t = \exp(z_t) + y_t$$

where

$$y_t \sim \exp(\mu + s\zeta_t) \quad \text{and} \quad z_{t+1} = d + \rho z_t + \sigma \epsilon_{t+1}$$

Here $\{\zeta_t\}$ and $\{\epsilon_t\}$ are both IID and standard normal.

Here $\{y_t\}$ is a transitory component and $\{z_t\}$ is persistent.

As before, the worker can either

1. accept an offer and work permanently at that wage, or
2. take unemployment compensation c and wait till next period.

The value function satisfies the Bellman equation

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, u(c) + \beta \, \mathbb{E}_z v^*(w', z') \right\}$$

In this express, u is a utility function and $\mathbb{E}_z$ is expectation of next period variables given current z.

The variable z enters as a state in the Bellman equation because its current value helps predict future wages.

30.2.1 A Simplification

There is a way that we can reduce dimensionality in this problem, which greatly accelerates computation.

To start, let f^* be the continuation value function, defined by

$$f^*(z) := u(c) + \beta \, \mathbb{E}_z v^*(w', z')$$

The Bellman equation can now be written

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, \, f^*(z) \right\}$$

Combining the last two expressions, we see that the continuation value function satisfies

$$f^*(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f^*(z') \right\}$$

We'll solve this functional equation for f^* by introducing the operator

$$Qf(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f(z') \right\}$$

By construction, f^* is a fixed point of Q, in the sense that $Qf^* = f^*$.

Under mild assumptions, it can be shown that Q is a contraction mapping over a suitable space of continuous functions on $\mathbb{R}$.

By Banach's contraction mapping theorem, this means that f^* is the unique fixed point and we can calculate it by iterating with Q from any reasonable initial condition.

Once we have f^*, we can solve the search problem by stopping when the reward for accepting exceeds the continuation value, or

$$\frac{u(w)}{1-\beta} \geq f^*(z)$$

For utility we take $u(c) = \ln(c)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

30.3 Implementation

Let f be our initial guess of f^*.

When we iterate, we use the *fitted value function iteration* algorithm.

In particular, f and all subsequent iterates are stored as a vector of values on a grid.

These points are interpolated into a function as required, using piecewise linear interpolation.

The integral in the definition of Qf is calculated by Monte Carlo.

The following list helps Numba by providing some type information about the data we will work with.

```
job_search_data = [
     ('μ', float64),             # transient shock log mean
     ('s', float64),             # transient shock log variance
     ('d', float64),             # shift coefficient of persistent state
     ('ρ', float64),             # correlation coefficient of persistent state
     ('σ', float64),             # state volatility
     ('β', float64),             # discount factor
     ('c', float64),             # unemployment compensation
     ('z_grid', float64[:]),     # grid over the state space
     ('e_draws', float64[:,:])   # Monte Carlo draws for integration
]
```

Here's a class that stores the data and the right hand side of the Bellman equation.

Default parameter values are embedded in the class.

```
@jitclass(job_search_data)
class JobSearch:

    def __init__(self,
                 μ=0.0,       # transient shock log mean
                 s=1.0,       # transient shock log variance
                 d=0.0,       # shift coefficient of persistent state
                 ρ=0.9,       # correlation coefficient of persistent state
                 σ=0.1,       # state volatility
                 β=0.98,      # discount factor
                 c=5,         # unemployment compensation
                 mc_size=1000,
                 grid_size=100):
```

(continues on next page)

(continued from previous page)

```
        self.μ, self.s, self.d,  = μ, s, d,
        self.ρ, self.σ, self.β, self.c = ρ, σ, β, c

        # Set up grid
        z_mean = d / (1 - ρ)
        z_sd = np.sqrt(σ / (1 - ρ**2))
        k = 3  # std devs from mean
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        # Draw and store shocks
        np.random.seed(1234)
        self.e_draws = randn(2, mc_size)

    def parameters(self):
        """
        Return all parameters as a tuple.
        """
        return self.μ, self.s, self.d, \
                self.ρ, self.σ, self.β, self.c
```

Next we implement the Q operator.

```
@njit(parallel=True)
def Q(js, f_in, f_out):
    """
    Apply the operator Q.

        * js is an instance of JobSearch
        * f_in and f_out are arrays that represent f and Qf respectively

    """

    μ, s, d, ρ, σ, β, c = js.parameters()
    M = js.e_draws.shape[1]

    for i in prange(len(js.z_grid)):
        z = js.z_grid[i]
        expectation = 0.0
        for m in range(M):
            e1, e2 = js.e_draws[:, m]
            z_next = d + ρ * z + σ * e1
            go_val = interp(js.z_grid, f_in, z_next)     # f(z')
            y_next = np.exp(μ + s * e2)                  # y' draw
            w_next = np.exp(z_next) + y_next             # w' draw
            stop_val = np.log(w_next) / (1 - β)
            expectation += max(stop_val, go_val)
        expectation = expectation / M
        f_out[i] = np.log(c) + β * expectation
```

Here's a function to compute an approximation to the fixed point of Q.

```
def compute_fixed_point(js,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
```

(continues on next page)

(continued from previous page)

```
                         verbose=True,
                         print_skip=25):

    f_init = np.full(len(js.z_grid), np.log(js.c))
    f_out = np.empty_like(f_init)

    # Set up loop
    f_in = f_init
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        Q(js, f_in, f_out)
        error = np.max(np.abs(f_in - f_out))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        f_in[:] = f_out

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return f_out
```

Let's try generating an instance and solving the model.

```
js = JobSearch()

qe.tic()
f_star = compute_fixed_point(js, verbose=True)
qe.toc()
```

```
Error at iteration 25 is 0.6540143893175809.
```

```
Error at iteration 50 is 0.12643184012381425.
```

```
Error at iteration 75 is 0.030376323858035903.
```

```
Error at iteration 100 is 0.007581959253982973.
```

```
Error at iteration 125 is 0.0019085682645538782.
```

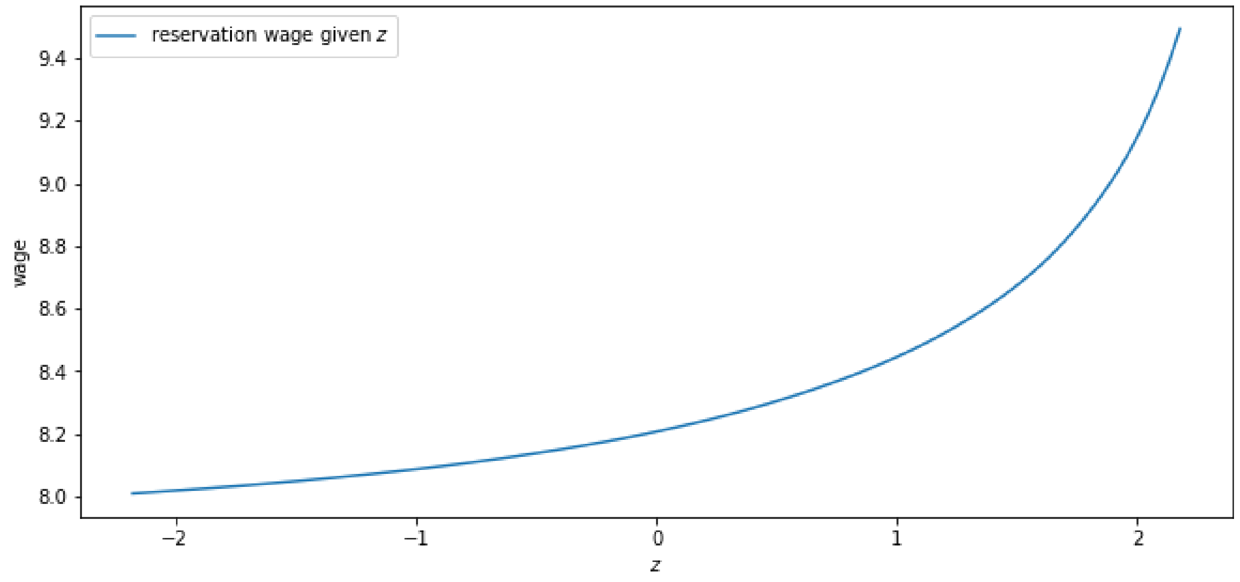
```
Error at iteration 150 is 0.00048173786846916755.
```

```
Error at iteration 175 is 0.000121400125664195.

Converged in 179 iterations.
TOC: Elapsed: 0:00:8.07
```

```
8.079387187957764
```

Next we will compute and plot the reservation wage function defined in (1).

```
res_wage_function = np.exp(f_star * (1 - js.β))

fig, ax = plt.subplots()
ax.plot(js.z_grid, res_wage_function, label="reservation wage given $z$")
ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```

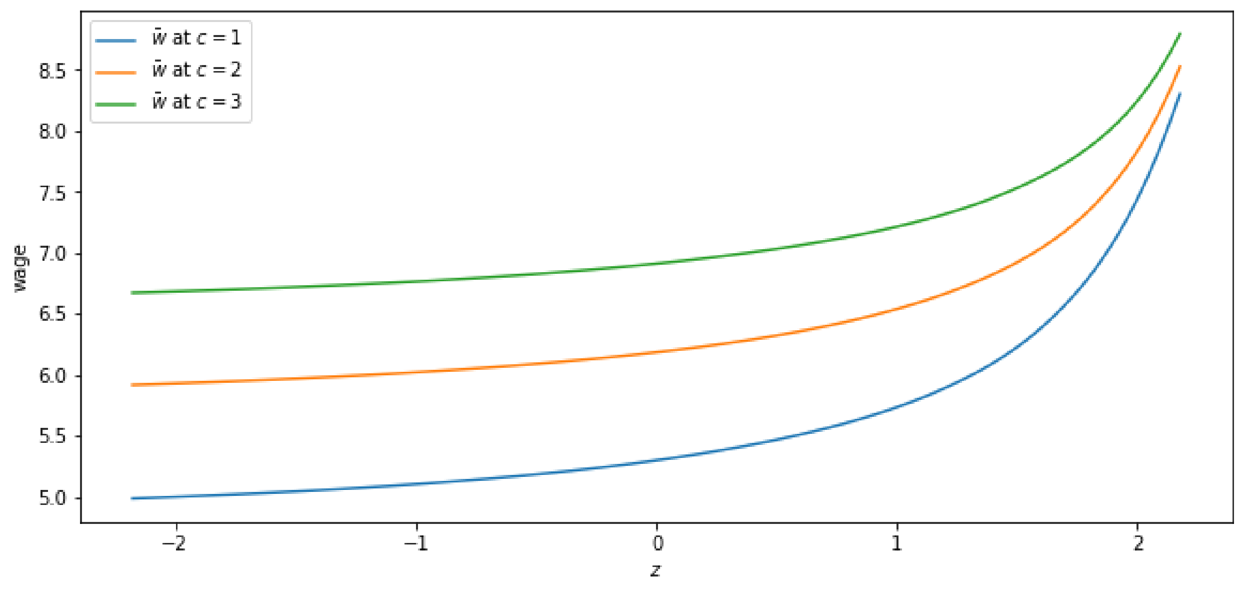


Notice that the reservation wage is increasing in the current state z.

This is because a higher state leads the agent to predict higher future wages, increasing the option value of waiting.

Let's try changing unemployment compensation and look at its impact on the reservation wage:

```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    js = JobSearch(c=c)
    f_star = compute_fixed_point(js, verbose=False)
    res_wage_function = np.exp(f_star * (1 - js.β))
    ax.plot(js.z_grid, res_wage_function, label=rf"$\bar w$ at $c = {c}$")

ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```



As expected, higher unemployment compensation shifts the reservation wage up at all state values.

30.4 Unemployment Duration

Next we study how mean unemployment duration varies with unemployment compensation.

For simplicity we'll fix the initial state at $z_t = 0$.

```python
def compute_unemployment_duration(js, seed=1234):

    f_star = compute_fixed_point(js, verbose=False)
    μ, s, d, ρ, σ, β, c = js.parameters()
    z_grid = js.z_grid
    np.random.seed(seed)

    @njit
    def f_star_function(z):
        return interp(z_grid, f_star, z)

    @njit
    def draw_tau(t_max=10_000):
        z = 0
        t = 0

        unemployed = True
        while unemployed and t < t_max:
            # draw current wage
            y = np.exp(μ + s * np.random.randn())
            w = np.exp(z) + y
            res_wage = np.exp(f_star_function(z) * (1 - β))
            # if optimal to stop, record t
            if w >= res_wage:
                unemployed = False
                τ = t
            # else increment data and state
```

(continues on next page)

(continued from previous page)

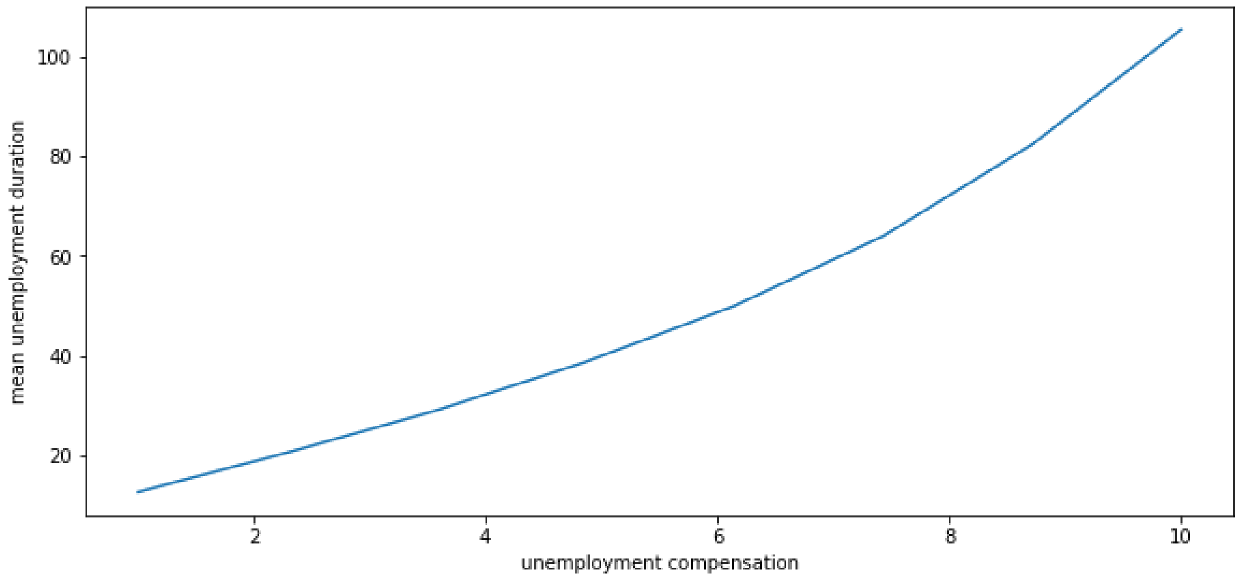
```
            else:
                z = ρ * z + d + σ * np.random.randn()
                t += 1
        return τ

    @njit(parallel=True)
    def compute_expected_tau(num_reps=100_000):
        sum_value = 0
        for i in prange(num_reps):
            sum_value += draw_tau()
        return sum_value / num_reps

    return compute_expected_tau()
```

Let's test this out with some possible values for unemployment compensation.

```
c_vals = np.linspace(1.0, 10.0, 8)
durations = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    js = JobSearch(c=c)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

Here is a plot of the results.

```
fig, ax = plt.subplots()
ax.plot(c_vals, durations)
ax.set_xlabel("unemployment compensation")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



Not surprisingly, unemployment duration increases when unemployment compensation is higher.

This is because the value of waiting increases with unemployment compensation.

30.5 Exercises

30.5.1 Exercise 1

Investigate how mean unemployment duration varies with the discount factor β.

- What is your prior expectation?
- Do your results match up?

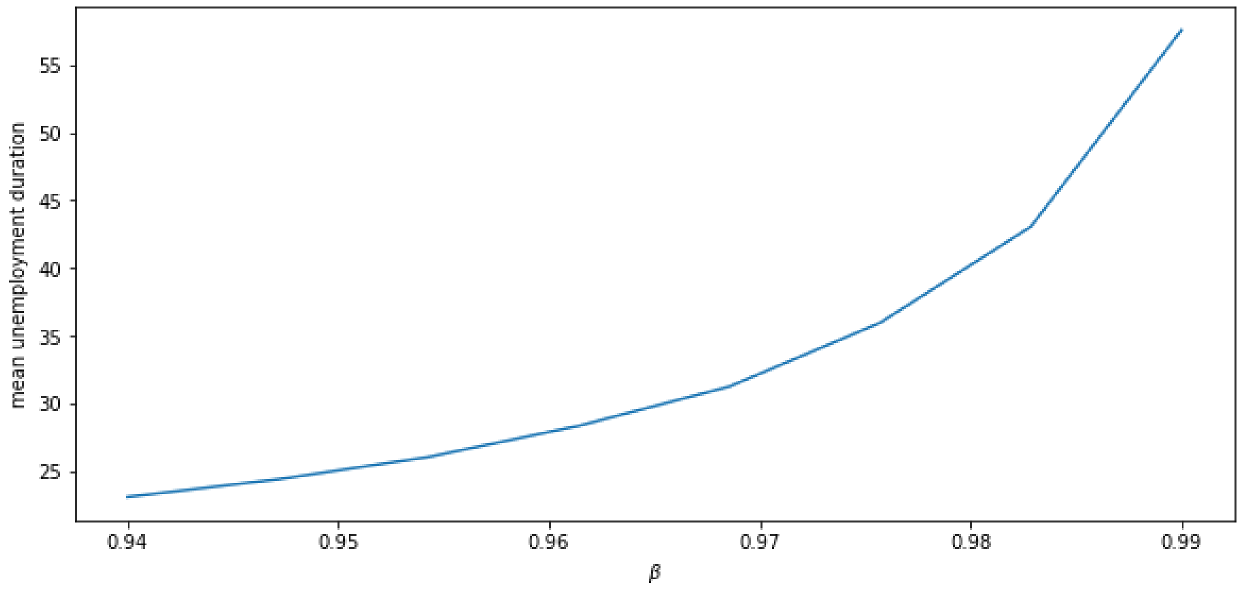
30.6 Solutions

30.6.1 Exercise 1

Here is one solution.

```
beta_vals = np.linspace(0.94, 0.99, 8)
durations = np.empty_like(beta_vals)
for i, β in enumerate(beta_vals):
    js = JobSearch(β=β)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

```
fig, ax = plt.subplots()
ax.plot(beta_vals, durations)
ax.set_xlabel(r"$\beta$")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



The figure shows that more patient individuals tend to wait longer before accepting an offer.

CHAPTER

THIRTYONE

JOB SEARCH V: MODELING CAREER CHOICE

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
```

31.1 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Nea99].

This exposition draws on the presentation in [LS18], section 6.5.

We begin with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numba import njit, prange
from quantecon.distributions import BetaBinomial
from scipy.special import binom, beta
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
```

31.1.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

31.2 Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is the contribution of career at time t
  - ϵ_t is the contribution of the job at time t

At the start of time t, a worker has the following options

- retain a current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned}
I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')
\end{aligned}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

31.2.1 Parameterization

As in [LS18], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, grid_size)` — an even grid of points between 0 and B inclusive
- `grid_size = 50`
- `B = 5`
- `β = 0.95`

The distributions F and G are discrete distributions generating draws from the grid points `np.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \mid n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \qquad k = 0, \ldots, n$$

Interpretation:

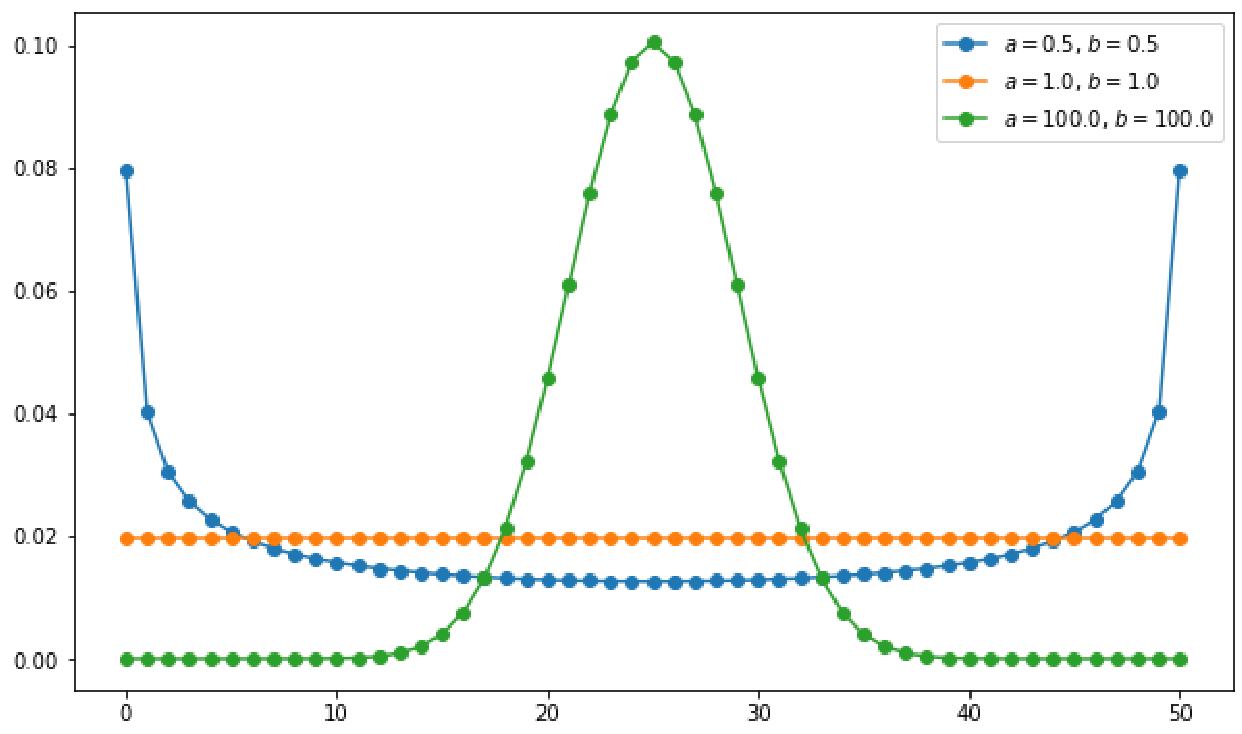
- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \mid n, a, b)$ is the probability of k successes in these n trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect on the pmf of different shape parameters when $n = 50$.

```python
def gen_probs(n, a, b):
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]
fig, ax = plt.subplots(figsize=(10, 6))
for a, b in zip(a_vals, b_vals):
    ab_label = f'$a = {a:.1f}$, $b = {b:.1f}$'
    ax.plot(list(range(0, n+1)), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
plt.show()
```



31.3 Implementation

We will first create a class `CareerWorkerProblem` which will hold the default parameterizations of the model and an initial guess for the value function.

```python
class CareerWorkerProblem:

    def __init__(self,
                 B=5.0,          # Upper bound
                 β=0.95,         # Discount factor
                 grid_size=50,   # Grid size
                 F_a=1,
                 F_b=1,
                 G_a=1,
                 G_b=1):

        self.β, self.grid_size, self.B = β, grid_size, B

        self.θ = np.linspace(0, B, grid_size)     # Set of θ values
        self.ϵ = np.linspace(0, B, grid_size)     # Set of ϵ values

        self.F_probs = BetaBinomial(grid_size - 1, F_a, F_b).pdf()
        self.G_probs = BetaBinomial(grid_size - 1, G_a, G_b).pdf()
        self.F_mean = np.sum(self.θ * self.F_probs)
        self.G_mean = np.sum(self.ϵ * self.G_probs)

        # Store these parameters for str and repr methods
        self._F_a, self._F_b = F_a, F_b
        self._G_a, self._G_b = G_a, G_b
```

The following function takes an instance of `CareerWorkerProblem` and returns the corresponding Bellman operator T and the greedy policy function.

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (2).

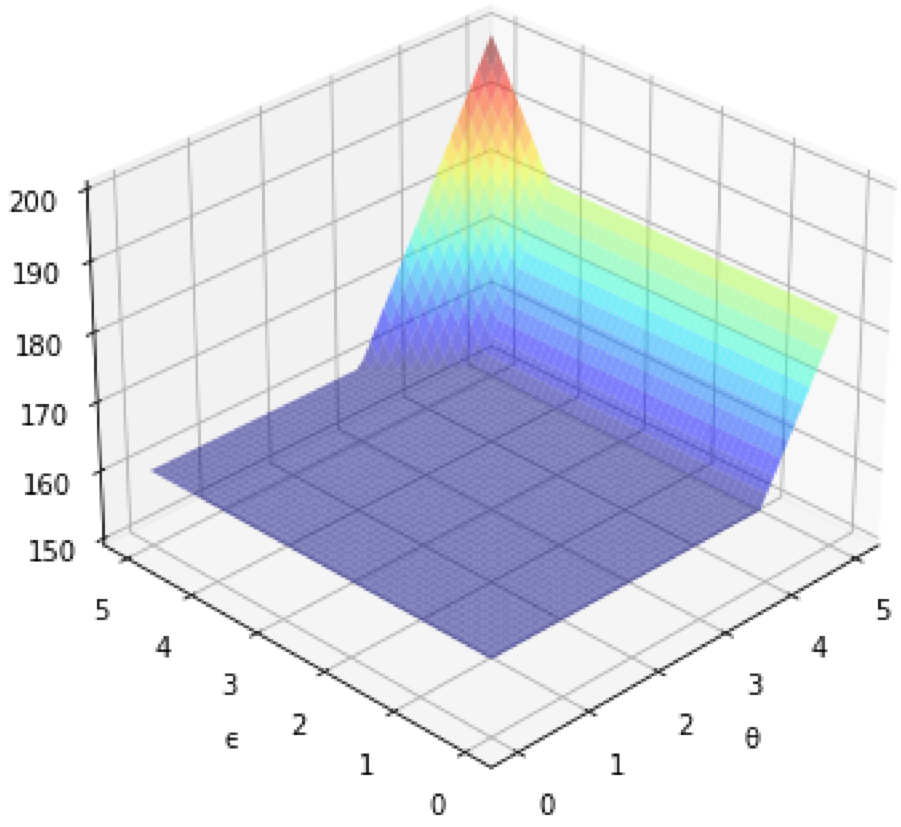
```python
def operator_factory(cw, parallel_flag=True):

    """
    Returns jitted versions of the Bellman operator and the
    greedy policy function

    cw is an instance of ``CareerWorkerProblem``
    """

    θ, ϵ, β = cw.θ, cw.ϵ, cw.β
    F_probs, G_probs = cw.F_probs, cw.G_probs
    F_mean, G_mean = cw.F_mean, cw.G_mean

    @njit(parallel=parallel_flag)
    def T(v):
        "The Bellman operator"

        v_new = np.empty_like(v)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]                    # Stay put
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs        # New job
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs  # New life
                v_new[i, j] = max(v1, v2, v3)

        return v_new

    @njit
    def get_greedy(v):
        "Computes the v-greedy policy"

        σ = np.empty(v.shape)

        for i in range(len(v)):
            for j in range(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                σ[i, j] = action

        return σ

    return T, get_greedy
```

Lastly, `solve_model` will take an instance of `CareerWorkerProblem` and iterate using the Bellman operator to find the fixed point of the value function.

```python
def solve_model(cw,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    T, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.full((cw.grid_size, cw.grid_size), 100.)  # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter and error > tol:
        print("Failed to converge!")

    else:
        if verbose:
            print(f"\nConverged in {i} iterations.")

    return v_new
```

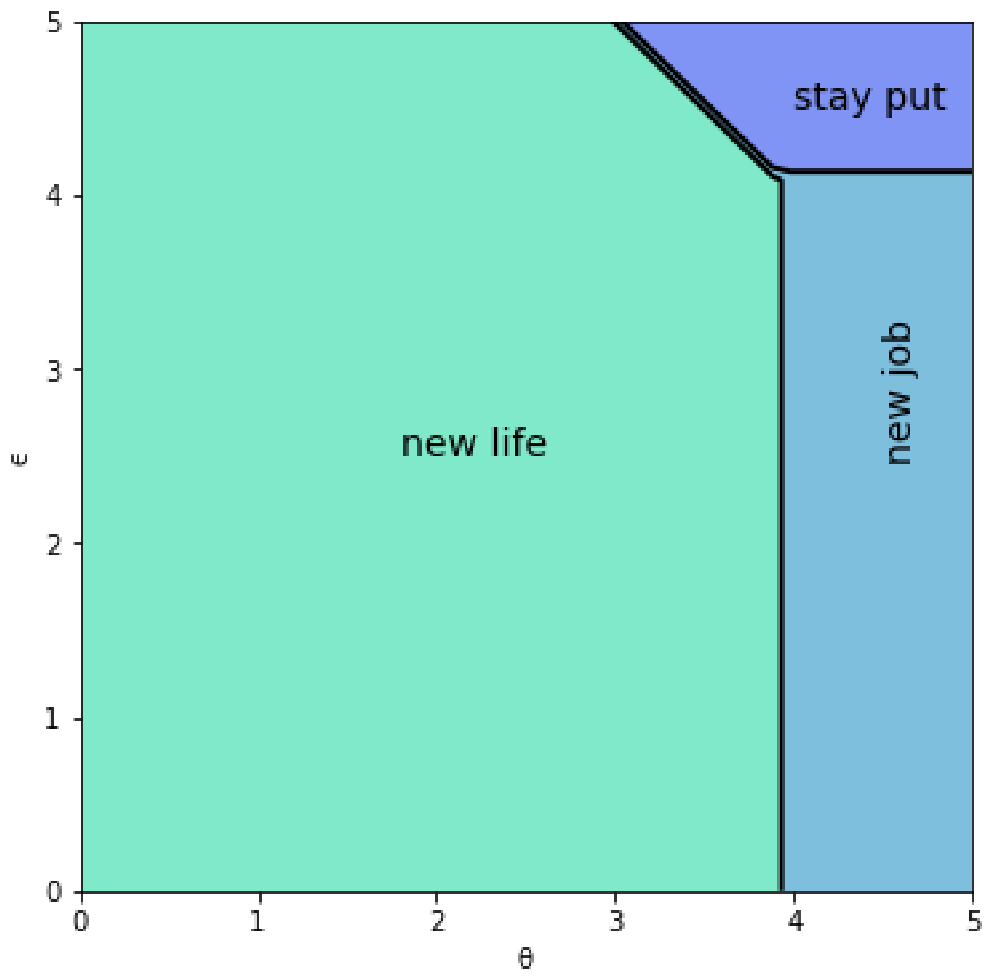
Here's the solution to the model – an approximate value function

```python
cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
ax.plot_surface(tg,
                eg,
                v_star.T,
                cmap=cm.jet,
                alpha=0.5,
                linewidth=0.25)
ax.set(xlabel='θ', ylabel='ϵ', zlim=(150, 200))
ax.view_init(ax.elev, 225)
plt.show()
```



And here is the optimal policy

```
fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with a new job and new career.
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found.
- If both job and career are good, the worker will stay put.

Notice that the worker will always hold on to a sufficiently good career, but not necessarily hold on to even the best paying job.

The reason is that high lifetime wages require both variables to be large, and the worker cannot change careers without changing jobs.

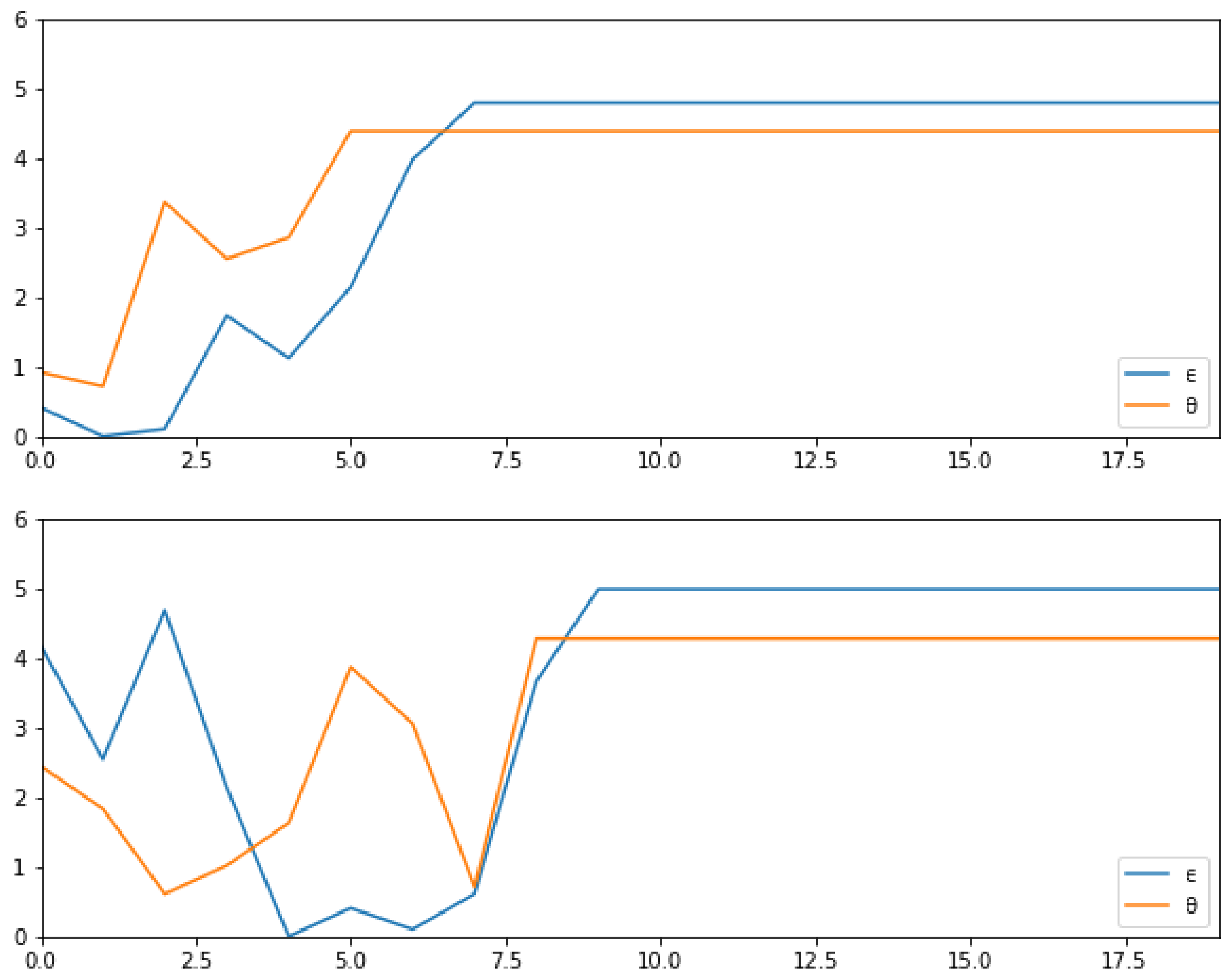
- Sometimes a good job must be sacrificed in order to change to a better career.

31.4 Exercises

31.4.1 Exercise 1

Using the default parameterization in the class `CareerWorkerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)



Hint: To generate the draws from the distributions F and G, use `quantecon.random.draw()`.

31.4.2 Exercise 2

Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space.

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

31.4.3 Exercise 3

Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

31.5 Solutions

31.5.1 Exercise 1

Simulate job/career paths.

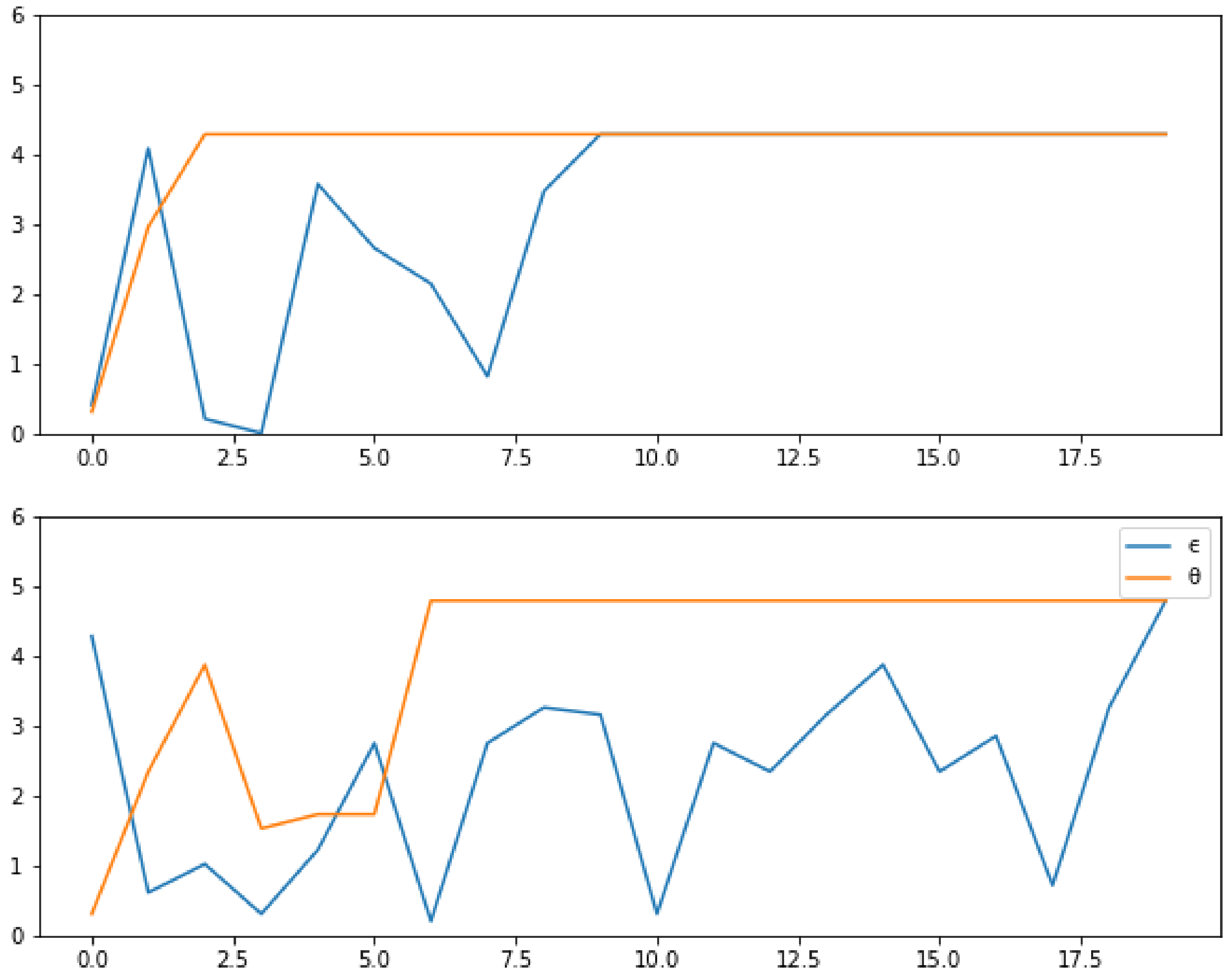
In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

```python
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
v_star = solve_model(cw, verbose=False)
T, get_greedy = operator_factory(cw)
greedy_star = get_greedy(v_star)

def gen_path(optimal_policy, F, G, t=20):
    i = j = 0
    θ_index = []
    ϵ_index = []
    for t in range(t):
        if optimal_policy[i, j] == 1:       # Stay put
            pass

        elif greedy_star[i, j] == 2:      # New job
            j = qe.random.draw(G)

        else:                             # New life
            i, j = qe.random.draw(F), qe.random.draw(G)
        θ_index.append(i)
        ϵ_index.append(j)
    return cw.θ[θ_index], cw.ϵ[ϵ_index]
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    θ_path, ϵ_path = gen_path(greedy_star, F, G)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)

plt.legend()
plt.show()
```



31.5.2 Exercise 2

The median for the original parameterization can be computed as follows

```
cw = CareerWorkerProblem()
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

@njit
def passage_time(optimal_policy, F, G):
    t = 0
    i = j = 0
    while True:
        if optimal_policy[i, j] == 1:    # Stay put
            return t
        elif optimal_policy[i, j] == 2:  # New job
            j = qe.random.draw(G)
        else:                            # New life
            i, j  = qe.random.draw(F), qe.random.draw(G)
        t += 1

@njit(parallel=True)
def median_time(optimal_policy, F, G, M=25000):
    samples = np.empty(M)
    for i in prange(M):
        samples[i] = passage_time(optimal_policy, F, G)
    return np.median(samples)

median_time(greedy_star, F, G)
```

```
7.0
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem(β=0.99)`.

The medians are subject to randomness but should be about 7 and 14 respectively.

Not surprisingly, more patient workers will wait longer to settle down to their final job.

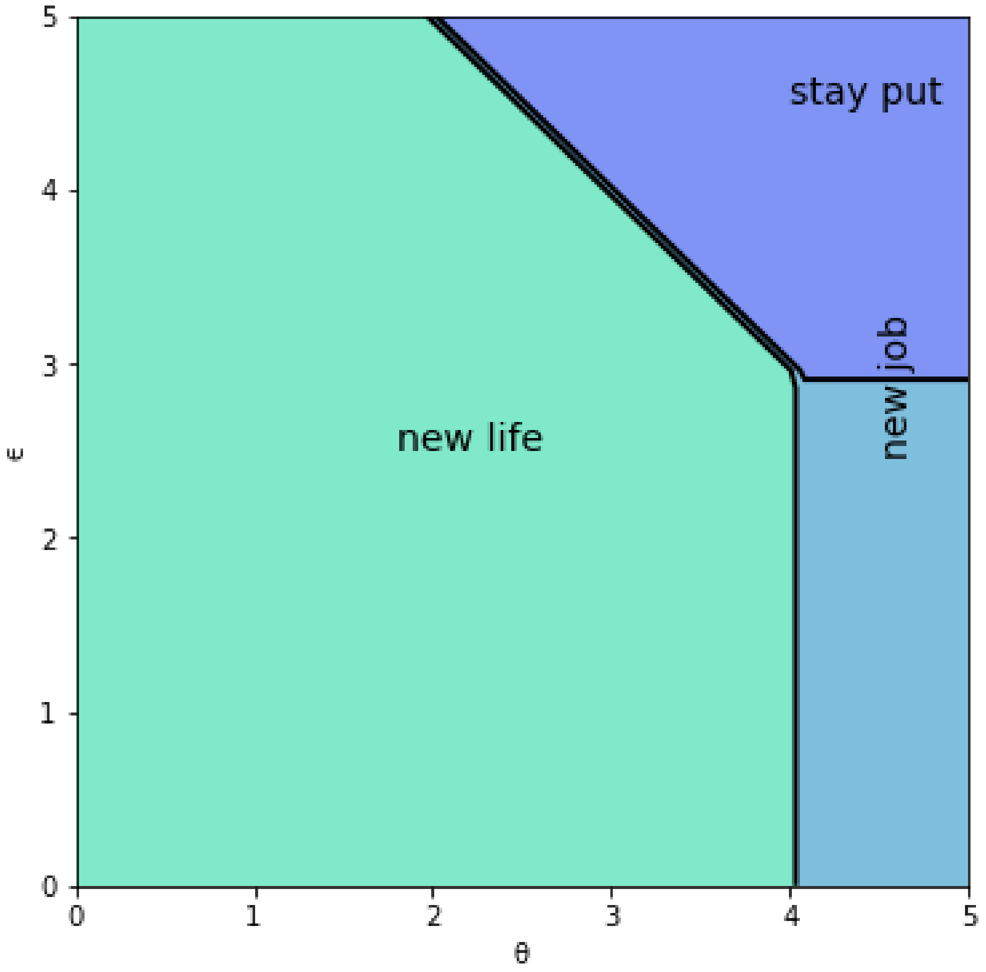
31.5.3 Exercise 3

```
cw = CareerWorkerProblem(G_a=100, G_b=100)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
```

(continues on next page)

(continued from previous page)

```
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



In the new figure, you see that the region for which the worker stays put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic.

CHAPTER

THIRTYTWO

JOB SEARCH VI: ON-THE-JOB SEARCH

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!conda install -y quantecon
!pip install interpolation
```

32.1 Overview

In this section, we solve a simple on-the-job search model

- based on [LS18], exercise 6.18, and [Jov79]

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import scipy.stats as stats
from interpolation import interp
from numba import njit, prange
from math import gamma
```

32.1.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

32.2 Model

Let x_t denote the time-t job-specific human capital of a worker employed at a given firm and let w_t denote current wages.

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role and
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = g(x_t, \phi_t)$.

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$.

The value of the offer, measured in job-specific human capital, is u_{t+1}, where $\{u_t\}$ is IID with common distribution f.

The worker can reject the current offer and continue with existing job.

Hence $x_{t+1} = u_{t+1}$ if he/she accepts and $x_{t+1} = g(x_t, \phi_t)$ otherwise.

Let $b_{t+1} \in \{0, 1\}$ be a binary random variable, where $b_{t+1} = 1$ indicates that the worker receives an offer at the end of time t.

We can write

$$x_{t+1} = (1 - b_{t+1}) g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{1}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$.

Taking the expectation of $v(x_{t+1})$ and using (1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s)) v[g(x, \phi)] + \beta \pi(s) \int v[g(x, \phi) \vee u] f(du) \right\} \tag{2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

32.2.1 Parameterization

In the implementation below, we will focus on the parameterization

$$g(x, \phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad f = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The $\text{Beta}(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

32.2.2 Back-of-the-Envelope Calculations

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1 - s - \phi)$, marginal cost of investment via either ϕ or s is identical.

Our risk-neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on x.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $g(x, \phi) = 0$, taking expectations of (1) gives expected next period capital equal to $\pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next, suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $g(x, \phi) = g(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

32.3 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```python
class JVWorker:
    r"""
    A Jovanovic-type model of employment with on-the-job search.

    """

    def __init__(self,
                 A=1.4,
                 α=0.6,
                 β=0.96,          # Discount factor
                 π=np.sqrt,       # Search effort function
                 a=2,             # Parameter of f
                 b=2,             # Parameter of f
                 grid_size=50,
                 mc_size=100,
```

(continues on next page)

(continued from previous page)

```
                 ε=1e-4):

        self.A, self.α, self.β, self.π = A, α, β, π
        self.mc_size, self.ε = mc_size, ε

        self.g = njit(lambda x, ϕ: A * (x * ϕ)**α)     # Transition function
        self.f_rvs = np.random.beta(a, b, mc_size)

        # Max of grid is the max of a large quantile value for f and the
        # fixed point y = g(y, 1)
        ε = 1e-4
        grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - ε))

        # Human capital
        self.x_grid = np.linspace(ε, grid_max, grid_size)
```

The function `operator_factory` takes an instance of this class and returns a jitted version of the Bellman operator `T`, i.e.

$$Tv(x) = \max_{s+\phi\leq 1} w(s,\phi)$$

where

$$w(s,\phi) := x(1-s-\phi) + \beta(1-\pi(s))v[g(x,\phi)] + \beta\pi(s)\int v[g(x,\phi)\vee u]f(du) \tag{3}$$

When we represent v, it will be with a NumPy array `v` giving values on grid `x_grid`.

But to evaluate the right-hand side of (3), we need a function, so we replace the arrays `v` and `x_grid` with a function `v_func` that gives linear interpolation of `v` on `x_grid`.

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function $w(z) = w(s,\phi)$ defined in (3).

The function is maximized over all feasible (s,ϕ) pairs.

Another function, `get_greedy` returns the optimal choice of s and ϕ at each x, given a value function.

```
def operator_factory(jv, parallel_flag=True):

    """
    Returns a jitted version of the Bellman operator T

    jv is an instance of JVWorker

    """

    π, β = jv.π, jv.β
    x_grid, ε, mc_size = jv.x_grid, jv.ε, jv.mc_size
    f_rvs, g = jv.f_rvs, jv.g

    @njit
    def state_action_values(z, x, v):
        s, ϕ = z
        v_func = lambda x: interp(x_grid, v, x)

        integral = 0
        for m in range(mc_size):
            u = f_rvs[m]
```

(continues on next page)

(continued from previous page)

```
            integral += v_func(max(g(x, ϕ), u))
        integral = integral / mc_size

        q = π(s) * integral + (1 - π(s)) * v_func(g(x, ϕ))
        return x * (1 - ϕ - s) + β * q

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator
        """

        v_new = np.empty_like(v)
        for i in prange(len(x_grid)):
            x = x_grid[i]

            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
            v_new[i] = max_val

        return v_new

    @njit
    def get_greedy(v):
        """
        Computes the v-greedy policy of a given function v
        """
        s_policy, ϕ_policy = np.empty_like(v), np.empty_like(v)

        for i in range(len(x_grid)):
            x = x_grid[i]
            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
                        max_s, max_ϕ = s, ϕ
                        s_policy[i], ϕ_policy[i] = max_s, max_ϕ
        return s_policy, ϕ_policy

    return T, get_greedy
```

To solve the model, we will write a function that uses the Bellman operator and iterates to find a fixed point.

```
def solve_model(jv,
                use_parallel=True,
```

(continues on next page)

(continued from previous page)

```
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    """
    Solves the model by value function iteration

    * jv is an instance of JVWorker

    """

    T, _ = operator_factory(jv, parallel_flag=use_parallel)

    # Set up loop
    v = jv.x_grid * 0.5  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

32.4 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
jv = JVWorker()
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv)
s_star, ϕ_star = get_greedy(v_star)
```

```
Error at iteration 25 is 0.15111180536479818.
```

```
Error at iteration 50 is 0.054460198506204094.
```

```
Error at iteration 75 is 0.01962727674107434.
```

```
Error at iteration 100 is 0.007073606098348506.
Error at iteration 125 is 0.002549304414195319.
```

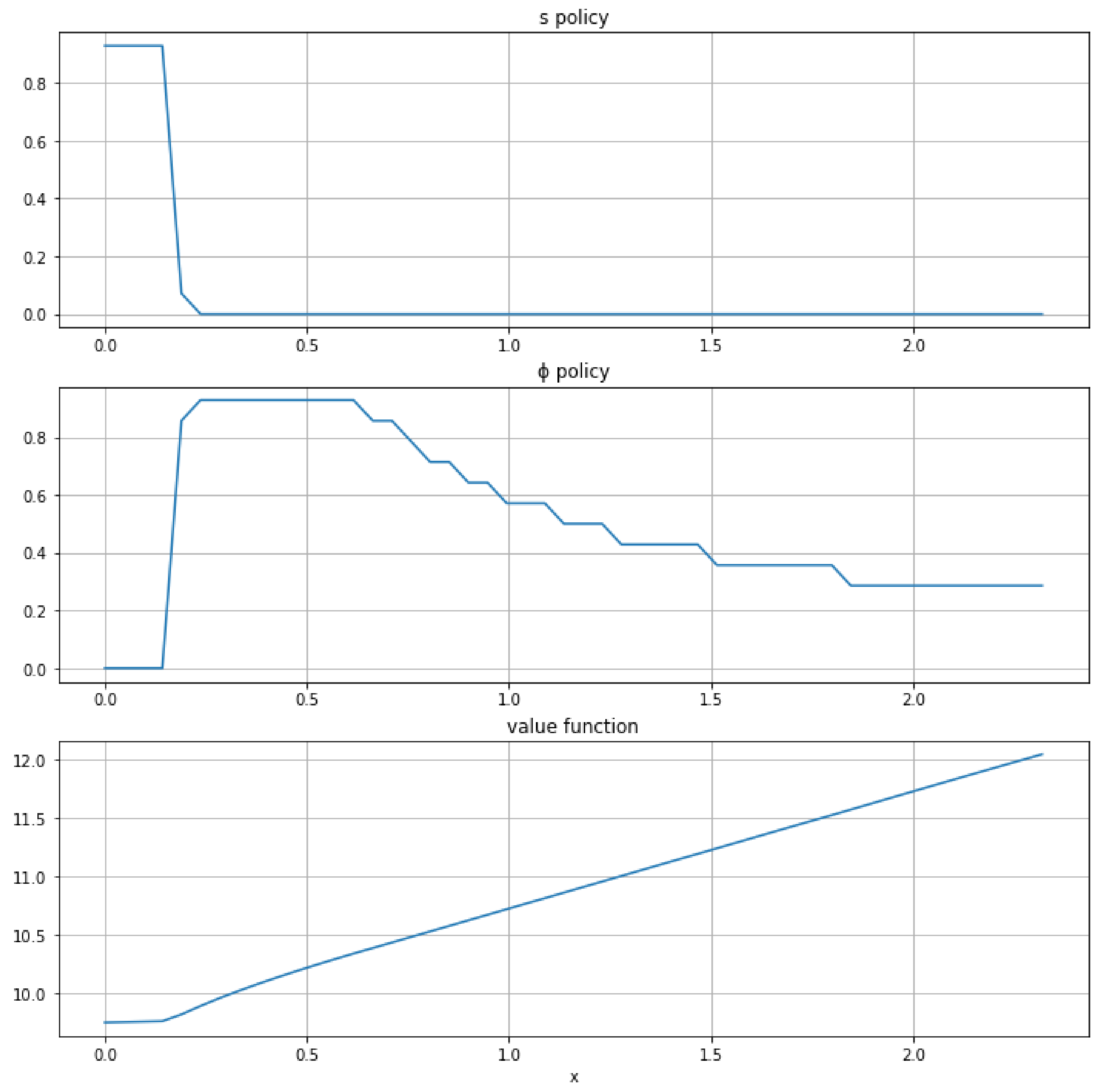
```
Error at iteration 150 is 0.0009187609411522857.
```

```
Error at iteration 175 is 0.00033111842676802894.
```

```
Error at iteration 200 is 0.00011933399390251509.

Converged in 205 iterations.
```

Here are the plots:

```
plots = [s_star, ϕ_star, v_star]
titles = ["s policy", "ϕ policy",  "value function"]

fig, axes = plt.subplots(3, 1, figsize=(12, 12))

for ax, plot, title in zip(axes, plots, titles):
    ax.plot(jv.x_grid, plot)
    ax.set(title=title)
    ax.grid()

axes[-1].set_xlabel("x")
plt.show()
```



The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$.

Overall, the policies match well with our predictions from *above*

- Worker switches from one investment strategy to the other depending on relative return.
- For low values of x, the best option is to search for a new job.
- Once x is larger, worker does better by investing in human capital specific to the current position.

32.5 Exercises

32.5.1 Exercise 1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (1) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$.

Plot this with one dot for each realization, in the form of a 45 degree diagram, setting

```
jv = JVWorker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity.

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

32.5.2 Exercise 2

In the preceding exercise, we found that s_t converges to zero and ϕ_t converges to about 0.6.

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t).

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto g(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ.

Can you give a rough interpretation for the value that you see?

32.6 Solutions

32.6.1 Exercise 1

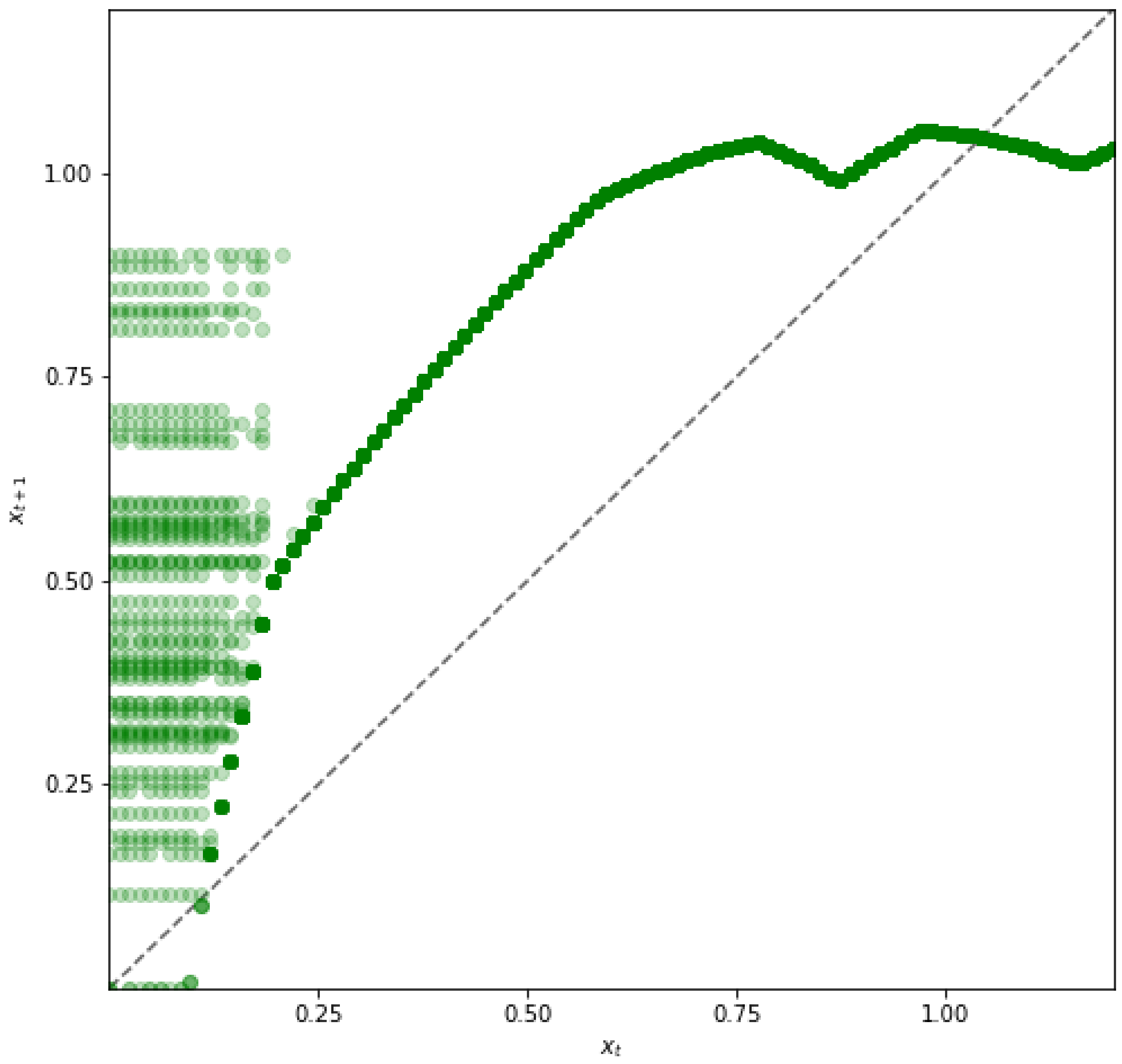
Here's code to produce the 45 degree diagram

```
jv = JVWorker(grid_size=25, mc_size=50)
π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
T, get_greedy = operator_factory(jv)
```

(continues on next page)

(continued from previous page)

```
v_star = solve_model(jv, verbose=False)
s_policy, ϕ_policy = get_greedy(v_star)

# Turn the policy function arrays into actual functions
s = lambda y: interp(x_grid, s_policy, y)
ϕ = lambda y: interp(x_grid, ϕ_policy, y)

def h(x, b, u):
    return (1 - b) * g(x, ϕ(x)) + b * max(g(x, ϕ(x)), u)


plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots(figsize=(8, 8))
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set(xticks=ticks, yticks=ticks,
       xlim=(0, plot_grid_max),
       ylim=(0, plot_grid_max),
       xlabel='$x_t$', ylabel='$x_{t+1}$')

ax.plot(plot_grid, plot_grid, 'k--', alpha=0.6)  # 45 degree line
for x in plot_grid:
    for i in range(jv.mc_size):
        b = 1 if np.random.uniform(0, 1) < π(s(x)) else 0
        u = f_rvs[i]
        y = h(x, b, u)
        ax.plot(x, y, 'go', alpha=0.25)

plt.show()
```



Looking at the dynamics, we can see that

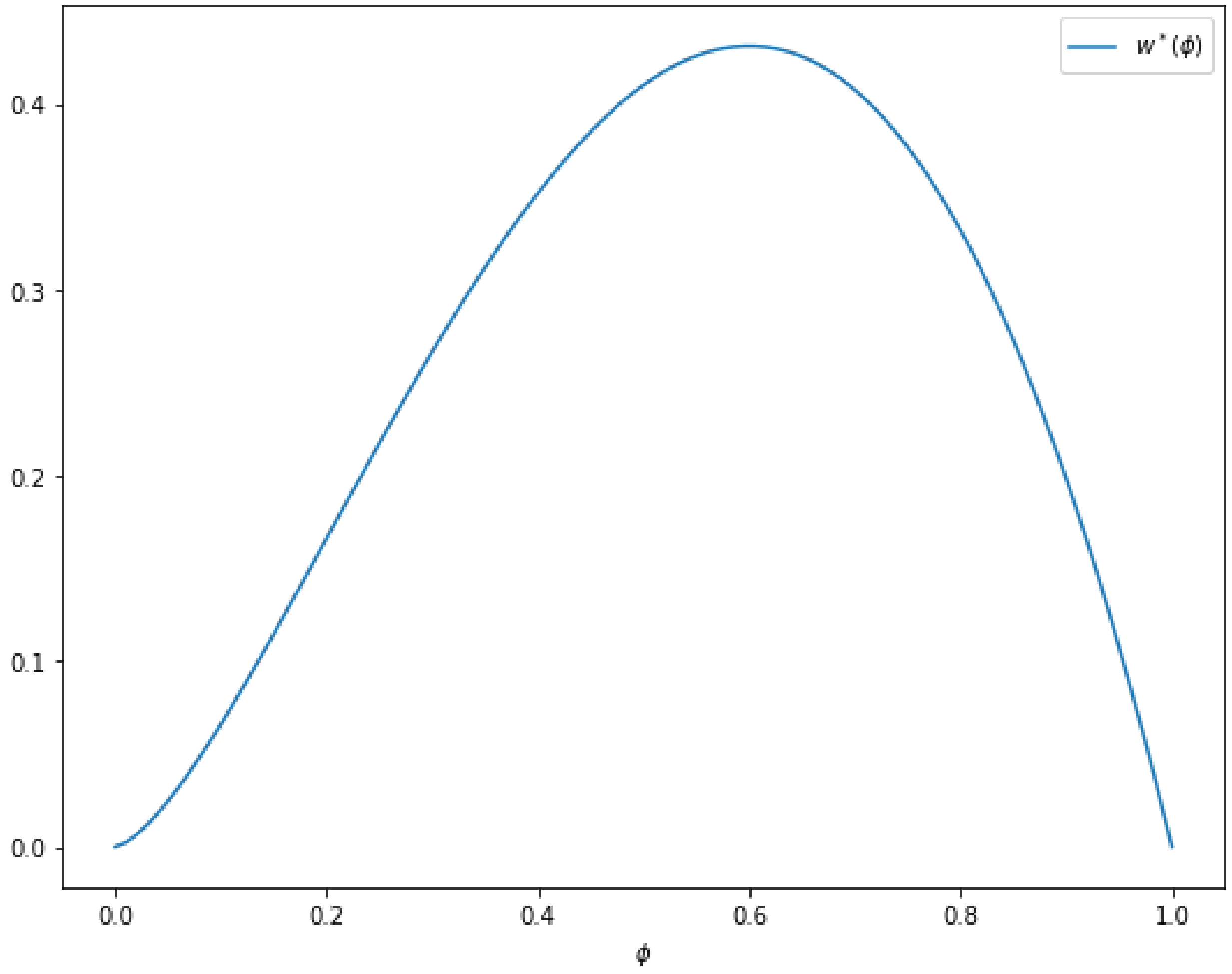
- If x_t is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely.
- As x_t increases the dynamics become deterministic, and x_t converges to a steady state value close to 1.

Referring back to the figure *here* we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

32.6.2 Exercise 2

The figure can be produced as follows

```
jv = JVWorker()

def xbar(ϕ):
    A, α = jv.A, jv.α
    return (A * ϕ**α)**(1 / (1 - α))

ϕ_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel='$\phi$')
ax.plot(ϕ_grid, [xbar(ϕ) * (1 - ϕ) for ϕ in ϕ_grid], label='$w^*(\phi)$')
ax.legend()

plt.show()
```



Observe that the maximizer is around 0.6.

This is similar to the long-run value for ϕ obtained in exercise 1.

Hence the behavior of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$.

This seems reasonable and helps us confirm that our dynamic programming solutions are probably correct.