Job Search I: The McCall Search Model

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 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.

2 Overview

The McCall search model [1] 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.

3 The McCall Model

An unemployed worker receives in each period a job offer at wage W_t.

At time t, our worker 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 wage sequence $\{W_t\}$ is assumed to be iid with probability mass function $p_1, \dots, p_n$.

Here p_i is the probability of observing wage offer $W_t = w_i$ in the set $w_1, \dots, w_n$.

The worker 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 worker discounts future utility relative to current utility.

The variable Y_t is income, equal to

- his wage W_t when employed
- unemployment compensation c when unemployed

3.1 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.

3.2 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 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 $V(w)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed but with wage offer w in hand.

More precisely, $V(w)$ 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 $V(w)$ 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 wage w 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

$$V(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{i=1}^{n} V(w_i) p_i \right\} \tag{1}$$

for every possible w_i in $w_1, \ldots, w_n$.

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 w, since

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

- 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 offer w.

But this is precisely $V(w)$, which is the l.h.s. of (1).

3.3 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.

In our case, the state is the current wage offer w.

Given *any* w, 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 zero meaning reject.

We can write the policy as follows

$$\sigma(w) := \mathbf{1}\left\{\frac{w}{1-\beta} \geq c + \beta \sum_{i=1}^{n} V(w_i) p_i\right\}$$

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

We can also write this as

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

where

$$\bar{w} := (1-\beta)\left\{c + \beta \sum_{i=1}^{n} V(w_i) p_i\right\} \tag{2}$$

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

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

Clearly, we can compute this reservation wage if we can compute the value function.

4 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at points $w_1, \ldots, w_n$.

In doing so, we can identify these values with the vector $v = (v_i)$ where $v_i := V(w_i)$.

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

$$v_i = \max\left\{\frac{w_i}{1-\beta},\, c + \beta \sum_{i=1}^{n} v_i p_i\right\} \quad \text{for } i = 1, \ldots, n \tag{3}$$

It turns out that there is exactly one vector $v := (v_i)_{i=1}^n$ in $\mathbb{R}^n$ that satisfies this equation.

4.1 The Algorithm

To compute this vector, we proceed as follows:

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

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

$$v'_i = \max\left\{\frac{w_i}{1-\beta},\, c + \beta \sum_{i=1}^{n} v_i p_i\right\} \quad \text{for } i = 1, \ldots, n \tag{4}$$

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.

This algorithm returns an arbitrarily good approximation to the true solution to (3), which represents the value function.

(Arbitrarily good means here that the approximation converges to the true solution as the tolerance goes to zero)

4.2 The Fixed Point Theory

What's the math behind these ideas?

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

$$Tv_i = \max \left\{ \frac{w_i}{1-\beta}, \, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \tag{5}$$

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

One can show that the conditions of the Banach contraction mapping theorem are satisfied by T as a self-mapping on $\mathbb{R}^n$.

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

Moreover, it's immediate from the definition of T that this fixed point is precisely the value function.

The iterative algorithm presented above corresponds to iterating with T from some initial guess v.

The Banach contraction mapping theorem tells us that this iterative process generates a sequence that converges to the fixed point.

4.3 Implementation

4.4 Setup

```
In [1]: using InstantiateFromURL
        # optionally add arguments to force installation: instantiate = true,
 ↪precompile = true
        github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```
In [2]: using LinearAlgebra, Statistics
       using Distributions, Expectations, NLsolve, Roots, Random, Plots, Parameters
```

```
In [3]: gr(fmt = :png);
```

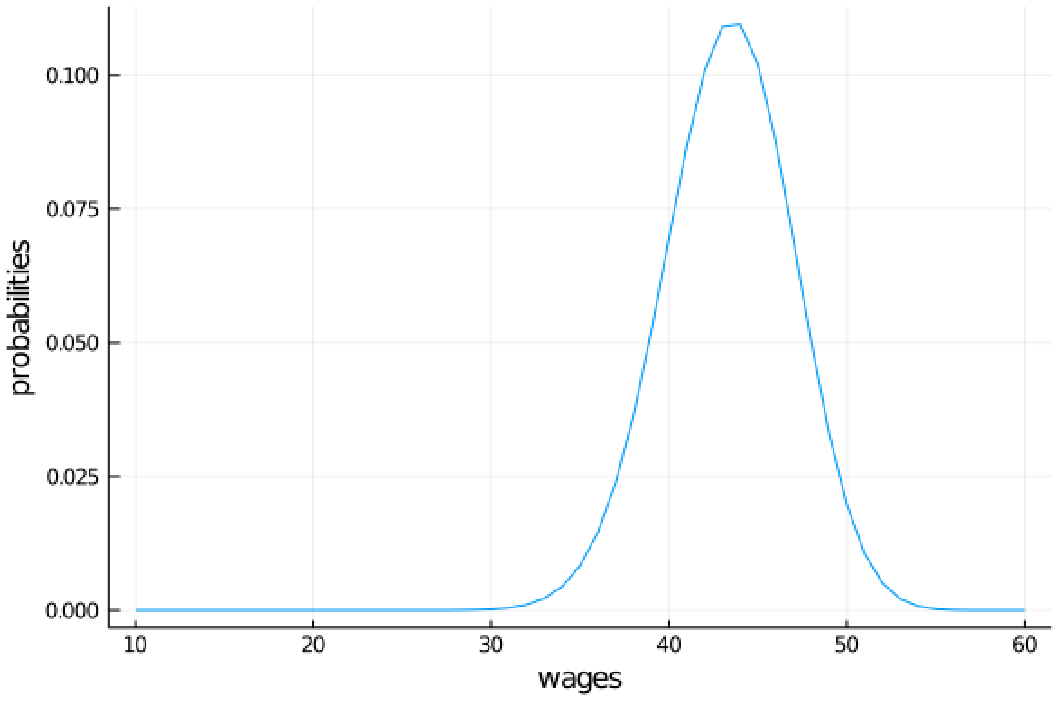
Here's the distribution of wage offers we'll work with

```
In [4]: n = 50
        dist = BetaBinomial(n, 200, 100) # probability distribution
        @show support(dist)
        w = range(10.0, 60.0, length = n+1) # linearly space wages
```

```
using StatsPlots
plt = plot(w, pdf.(dist, support(dist)), xlabel = "wages", ylabel =
↪"probabilities",
legend = false)
```

```
   support(dist) = 0:50
```

Out[4]:



We can explore taking expectations over this distribution

```
In [5]: E = expectation(dist) # expectation operator

        # exploring the properties of the operator
        wage(i) = w[i+1] # +1 to map from support of 0
        E_w = E(wage)
        E_w_2 = E(i -> wage(i)^2) - E_w^2 # variance
        @show E_w, E_w_2

        # use operator with left-multiply
        @show E * w # the `w` are values assigned for the discrete states
        @show dot(pdf.(dist, support(dist)), w); # identical calculation
```

```
   (E_w, E_w_2) = (43.333333333335695, 12.919896640724573)
E * w = 43.3333333333357
dot(pdf.(dist, support(dist)), w) = 43.3333333333357
```

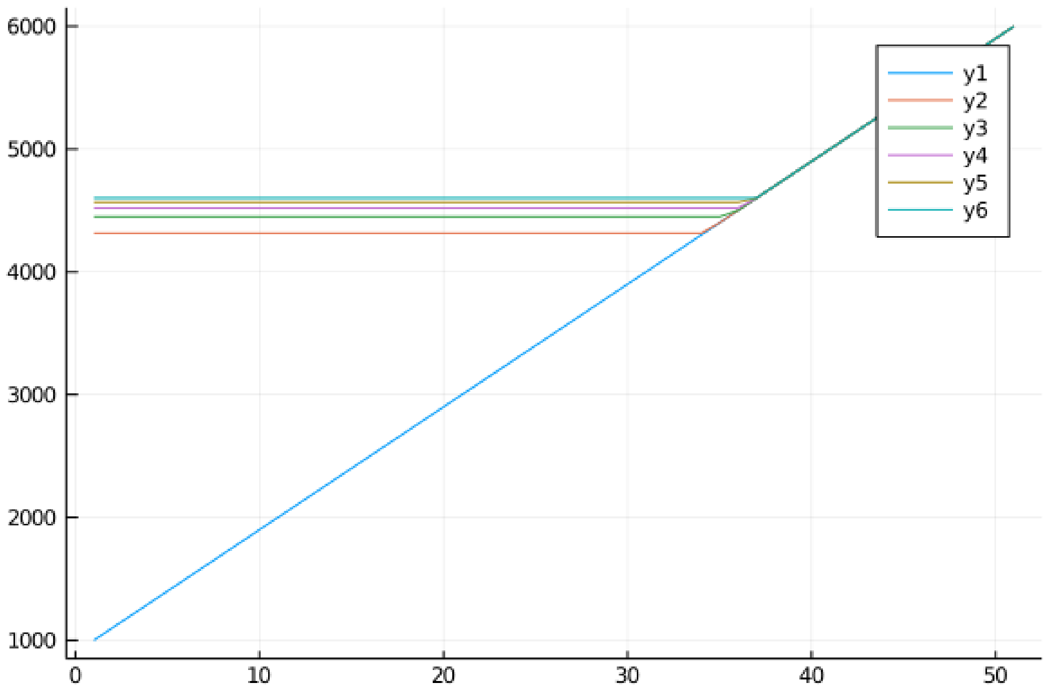
To implement our algorithm, let's have a look at the sequence of approximate value functions that this fixed point algorithm generates.

Default parameter values are embedded in the function.

Our initial guess v is the value of accepting at every given wage

In [6]:
```
# parameters and constant objects

c = 25
β = 0.99
num_plots = 6

# Operator
T(v) = max.(w/(1 - β), c + β * E*v) # (5) broadcasts over the w, fixes the v
# alternatively, T(v) = [max(wval/(1 - β), c + β * E*v) for wval in w]

# fill in  matrix of vs
vs = zeros(n + 1, 6) # data to fill
vs[:, 1] .= w / (1-β) # initial guess of "accept all"

# manually applying operator
for col in 2:num_plots
    v_last = vs[:, col - 1]
    vs[:, col] .= T(v_last)  # apply operator
end
plot(vs)
```

Out[6]:



One approach to solving the model is to directly implement this sort of iteration, and continues until measured deviation between successive iterates is below tol

```
In [7]: function compute_reservation_wage_direct(params; v_iv = collect(w ./(1-β)), max_iter =
        500,
                                                  tol = 1e-6)
            @unpack c, β, w = params

            # create a closure for the T operator
            T(v) = max.(w/(1 - β), c + β * E*v) # (5) fixing the parameter values

            v = copy(v_iv) # start at initial value.  copy to prevent v_iv␣
  ↪modification
            v_next = similar(v)
            i = 0
            error = Inf
            while i < max_iter && error > tol
                v_next .= T(v) # (4)
                error = norm(v_next - v)
                i += 1
                v .= v_next  # copy contents into v.  Also could have used v[:] =␣
  ↪v_next
            end
            # now compute the reservation wage
            return (1 - β) * (c + β * E*v) # (2)
        end
```

```
Out[7]: compute_reservation_wage_direct (generic function with 1 method)
```

In the above, we use `v = copy(v_iv)` rather than just `v_iv = v`.

To understand why, first recall that `v_iv` is a function argument – either defaulting to the given value, or passed into the function

- If we had gone `v = v_iv` instead, then it would have simply created a new name `v` which binds to whatever is located at `v_iv`.

- Since we later use `v .= v_next` later in the algorithm, the values in it would be modified.
- Hence, we would be modifying the `v_iv` vector we were passed in, which may not be what the caller of the function wanted.
- The big issue this creates are "side-effects" where you can call a function and strange things can happen outside of the function that you didn't expect.
- If you intended for the modification to potentially occur, then the Julia style guide says that we should call the function `compute_reservation_wage_direct!` to make the possible side-effects clear.

As usual, we are better off using a package, which may give a better algorithm and is likely to less error prone.

In this case, we can use the `fixedpoint` algorithm discussed in our Julia by Example lecture to find the fixed point of the T operator.

```
In [8]: function compute_reservation_wage(params; v_iv = collect(w ./(1-β)),␣
  ↪iterations = 500,
                                          ftol = 1e-6, m = 6)
            @unpack c, β, w = params
```

```
        T(v) = max.(w/(1 - β), c + β * E*v) # (5) fixing the parameter values

        v_star = fixedpoint(T, v_iv, iterations = iterations, ftol = ftol,
                            m = 0).zero # (5)
        return (1 - β) * (c + β * E*v_star) # (3)
    end
```

```
Out[8]: compute_reservation_wage (generic function with 1 method)
```

Let's compute the reservation wage at the default parameters

```
In [9]: mcm = @with_kw (c=25.0, β=0.99, w=w) # named tuples

        compute_reservation_wage(mcm()) # call with default parameters
```

```
Out[9]: 47.31649970147162
```

4.5 Comparative Statics

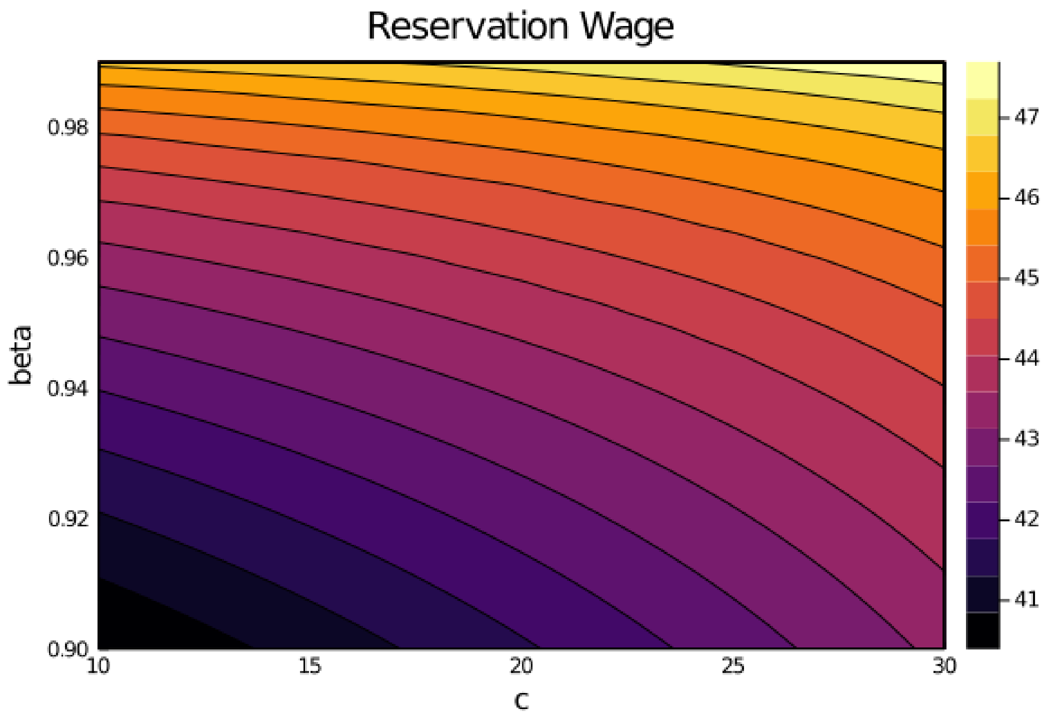
Now we know how to compute the reservation wage, let's see how it varies with parameters.

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

```
In [10]: grid_size = 25
         R = rand(grid_size, grid_size)

         c_vals = range(10.0, 30.0, length = grid_size)
         β_vals = range(0.9, 0.99, length = grid_size)

         for (i, c) in enumerate(c_vals)
             for (j, β) in enumerate(β_vals)
                 R[i, j] = compute_reservation_wage(mcm(c=c, β=β)) # change from
  ↪defaults
             end
         end
```

```
In [11]: contour(c_vals, β_vals, R',
                 title = "Reservation Wage",
                 xlabel = "c",
                 ylabel = "beta",
                 fill = true)
```

```
Out[11]:
```



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

5 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 ψ denote the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

That is,

$$\psi = c + \beta \sum_{i=1}^{n} V(w_i) p_i \tag{6}$$

where V is the value function.

By the Bellman equation, we then have

$$V(w_i) = \max \left\{ \frac{w_i}{1-\beta}, \, \psi \right\}$$

Substituting this last equation into (6) gives

$$\psi = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1-\beta}, \psi \right\} p_i \tag{7}$$

Which we could also write as $\psi = T_\psi(\psi)$ for the appropriate operator.

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

One solution method for this kind of nonlinear equation is iterative.

That is,

Step 1: pick an initial guess ψ.

Step 2: compute the update ψ' via

$$\psi' = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1-\beta}, \psi \right\} p_i \tag{8}$$

Step 3: calculate the deviation $|\psi - \psi'|$.

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

Step 5: return ψ.

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:

```
In [12]: function compute_reservation_wage_ψ(c, β; ψ_iv = E * w ./ (1 - β),
  ↪max_iter = 500,
                                             tol = 1e-5)
             T_ψ(ψ) = [c + β * E*max.((w ./ (1 - β)), ψ[1])] # (7)
             # using vectors since fixedpoint doesn't support scalar
             ψ_star = fixedpoint(T_ψ, [ψ_iv]).zero[1]
             return (1 - β) * ψ_star # (2)
         end
         compute_reservation_wage_ψ(c, β)
```

```
Out[12]: 47.31649976654629
```

You can use this code to solve the exercise below.

Another option is to solve for the root of the $T_\psi(\psi) - \psi$ equation

```
In [13]: function compute_reservation_wage_ψ2(c, β; ψ_iv = E * w ./ (1 - β),
  ↪max_iter = 500,
                                              tol = 1e-5)
             root_ψ(ψ) = c + β * E*max.((w ./ (1 - β)), ψ) - ψ # (7)
             ψ_star = find_zero(root_ψ, ψ_iv)
             return (1 - β) * ψ_star # (2)
         end
         compute_reservation_wage_ψ2(c, β)
```

```
Out[13]: 47.316499766546194
```

6 Exercises

6.1 Exercise 1

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

```
c_vals = range(10, 40, length = 25)
```

That is, start the agent off as unemployed, computed 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`.

7 Solutions

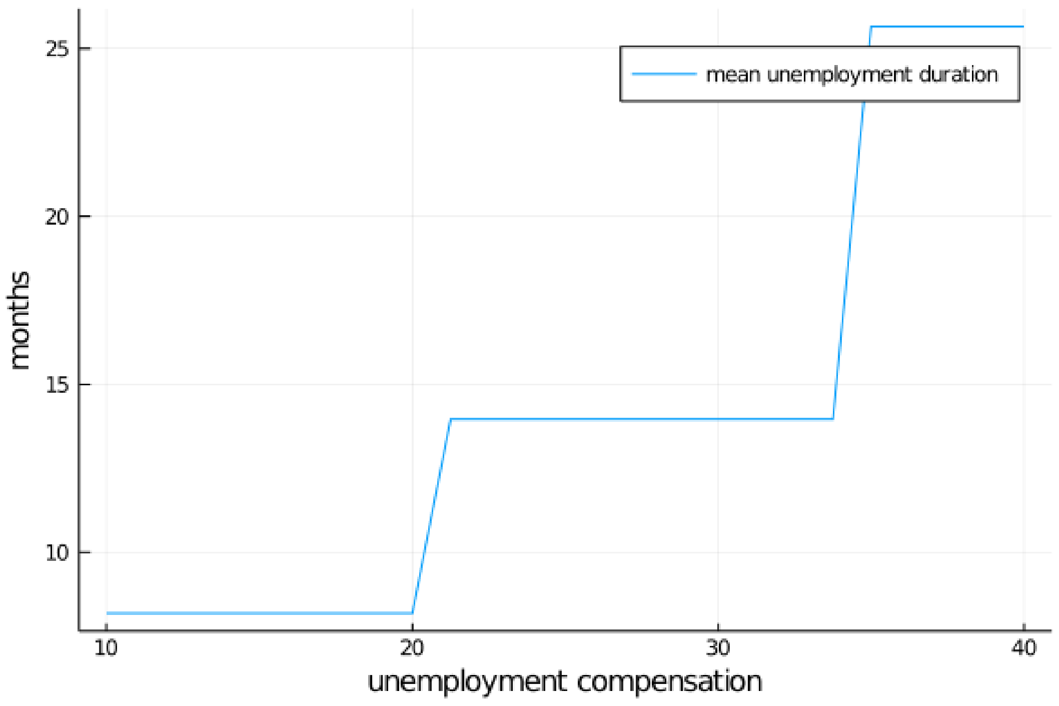
7.1 Exercise 1

Here's one solution

```
In [14]: function compute_stopping_time(w̄; seed=1234)
             Random.seed!(seed)
             stopping_time = 0
             t = 1
             # make sure the constraint is sometimes binding
             @assert length(w) - 1 ∈ support(dist) && w̄ <= w[end]
             while true
                 # Generate a wage draw
                 w_val = w[rand(dist)] # the wage dist set up earlier
                 if w_val ≥ w̄
                     stopping_time = t
                     break
                 else
                     t += 1
                 end
             end
             return stopping_time
         end

         compute_mean_stopping_time(w̄, num_reps=10000) = mean(i ->
                                                                 compute_stopping_time(w̄,
                                                                   seed = i), 1:num_reps)
         c_vals = range(10,  40, length = 25)
         stop_times = similar(c_vals)

         beta = 0.99
         for (i, c) in enumerate(c_vals)
             w̄ = compute_reservation_wage_ψ(c, beta)
             stop_times[i] = compute_mean_stopping_time(w̄)
```

```
    end

    plot(c_vals, stop_times, label = "mean unemployment duration",
         xlabel = "unemployment compensation", ylabel = "months")
```

Out[14]:



References

[1] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.