Analysis of the McCall (1970) search model

Theory

The worker in the McCall (1970) receives a sequence of wage offers w_t at times $t = 0, 1, 2, \ldots$, which are iid and drawn from a distribution with cdf $F(w)$, with a bounded support $[0, B]$. After seeing the current wage offer w_t, the worker either accepts it, in which case the worker leaves the labor market and receives income $y_{t+j} = w_t$ in all future periods, $j = 0, 1, 2, \ldots$, or rejects it, in which case the worker receives an unemployment benefit $y_t = c \in [0, B]$, and moves on to the next period to receive a new offer w_{t+1}.

The worker hence solves the sequence problem

$$V_0^* = \max_{\{a_t\}_{t=0}^{\infty}} E_0\left[\sum_{t=0}^{\infty} \beta^t y_t\right]$$

where $a_t \in \{\text{accept}, \text{reject}\}$ if the worker has not yet accepted any earlier offer, and $a_t \in \{\}$ otherwise. The quantity V_0^* is the **value function**, and we assume that the expectations operator $E_0[\cdot]$ conditions on information available at time 0, including the current offer w_0.

The recursive formulation of the problem relies on the **principle of optimality** (Bellman, 1952, 1957). We can rewrite the problem as:

$$V_0^* = \max_{\{a_t\}_{t=0}^{\infty}} \left\{ y_0 + \beta E_0\left[\sum_{t=1}^{\infty} \beta^{t-1} y_t\right] \right\} = \max_{a_0} \left\{ y_0 + \beta \max_{\{a_t\}_{t=1}^{\infty}} E_0\left[\sum_{t=1}^{\infty} \beta^{t-1} y_t\right] \right\}$$

$$= \max_{a_0} \left\{ y_0 + \beta E_0\left[\max_{\{a_t\}_{t=1}^{\infty}} \left\{ y_1 + \beta E_1\left[\sum_{t=2}^{\infty} \beta^{t-2} y_t\right] \right\} \right] \right\} = \max_{a_0} \left\{ y_0 + \beta E_0\left[V_1^*\right] \right\}.$$

Recursive formulation

In order to make the problem tractable, we need to find a representation in which V_0^* and V_1^* have the same structure. This requires finding the **state** for the problem that encodes all relevant information for worker's time-t decision problem.

Finding the state is an art. Thomas Sargent

Introspection of the problem reveals that the relevant information for a worker who has not yet accepted an offer is summarized in the current wage offer w_t. This implies that we can write the value function V_t^* of worker who is still searching at time t as $V_t^* = V(w_t)$.

This invites the following **recursive representation** of the problem in the form of a **Bellman equation**:

$$V(w) = \max_{\{\text{accept, reject}\}} \left\{ V^a(w), c + \beta \int_0^B V\left(w'\right) dF\left(w'\right) \right\}.$$

where $V^a(w)$ is the value of accepting the current offer w:

$$V^a(w) = \sum_{t=0}^{\infty} \beta^t w = \frac{w}{1-\beta}.$$

and

$$Q = c + \beta \int_0^B V\left(w'\right) dF\left(w'\right).$$

is the value of rejecting the offer, collecting the unemployment benefit c and drawing again next period from the offer distribution $F(w)$.

Reservation wage

Since the value of accepting the offer is linearly increasing in w and the value of rejecting the offer is constant, and the two functions cross exactly once on $[0, B]$, the optimal decision must be in the form of a **reservation wage** $\bar{w}$ such that

- worker accepts if $w > \bar{w}$,
- worker rejects if $w < \bar{w}$,
- worker is indifferent between accepting and rejecting at $w = \bar{w}$.

This implies that the function $V(w)$ is given by the following piecewise linear form:

$$V(w) = \begin{cases} c + \beta \int_0^B V\left(w'\right) dF\left(w'\right) = \dfrac{\bar{w}}{1-\beta} & \text{if } w \leq \bar{w} \\ \dfrac{w}{1-\beta} & \text{if } w \geq \bar{w} \end{cases}$$

The key insight is that the only unknown in the characterization is the reservation wage $\bar{w}$, or, equivalently, the present value of receiving reservation wage forever,

$$Q = c + \beta\int_0^B V\left(w'\right)dF\left(w'\right) = \frac{\bar{w}}{1-\beta}.$$

Using the characterization of the value function $V(w')$, this expression can be written as

$$\begin{aligned}\frac{\bar{w}}{1-\beta} &= c + \beta\int_0^{\bar{w}}\frac{\bar{w}}{1-\beta}dF\left(w'\right) + \beta\int_{\bar{w}}^B\frac{w'}{1-\beta}dF\left(w'\right)\\ &= c + \beta\int_0^B\frac{\bar{w}}{1-\beta}dF\left(w'\right) + \frac{\beta}{1-\beta}\int_{\bar{w}}^B\left(w'-\bar{w}\right)dF\left(w'\right)\end{aligned}$$

which yields the following expression for the reservation wage:

$$\bar{w} - c = \frac{\beta}{1-\beta}\int_{\bar{w}}^B\left(w'-\bar{w}\right)dF\left(w'\right).$$

The left-hand side is the cost of searching one more time when the current offer is $\bar{w}$, while the right-hand side is the net benefit of searching one more time. The left-hand side is increasing in $\bar{w}$, while the right-hand side is decreasing, and the two functions cross exactly once, hence there is a unique reservation wage $\bar{w}$.

Quadrature methods

Numerical implementation of the model solution will require the evaluation of an integral of a function over the distribution with cdf $F(w)$$. While it is possible to choose distributions for which analytical formulas exist, we implement a numerical approximation by discretizing the continuous distribution to a grid of finitely many nodes. Such an algorithm is called a **quadrature rule**.

Imagine we are given the cdf $F(w)$ with density $f(w)$ and an interval $[a, b] \subseteq [0, B]$. A quadrature rule involves choosing a grid of nodes w^i, $i = 1, \ldots, I$ such that $w^i < w^{i+1}$, and associated weights f^i such that we can approximate the integral of a function $g(w)$ using a sum over the pre-designed nodes:

$$\int_a^b g(w)f(w)dw \approx \sum_{i=1}^I g\left(w^i\right)f^i$$

A simple choice would be an equidistant grid on $[a, b]$ that splits the interval into $I - 1$ subintervals of equal distance, with $w^1 = a$, $w^I = b$, and

$$w^i = a + \frac{i-1}{I-1}(b-a), \qquad i = 1, \ldots, I.$$

We work with a general subinterval $[a, b]$, for example, we could choose $[a, b] = [\bar{w}, B]$.

For a given grid of points, we can choose the weights as follows:

$$f^i = \begin{cases} F\left(\frac{1}{2}\left(w^2 + w^1\right)\right) - F(a) & i = 1 \\ F\left(\frac{1}{2}\left(w^{i+1} + w^i\right)\right) - F\left(\frac{1}{2}\left(w^i + w^{i-1}\right)\right) & 0 < i < I \\ F(b) - F\left(\frac{1}{2}\left(w^I + w^{I-1}\right)\right) & i = I \end{cases}$$

This weighting scheme concentrates the continuous density $f(w)$ into the nearest mass points on the grid. Also observe that, as desired

$$\int_a^b f(w)dw = \int_a^b dF(w) = F(b) - F(a) = \sum_{i=1}^{I} f^i.$$

We now turn to implementation of the simple quadrature method.

```
In [ ]: # # DOWNLOAD the github code as zip, unzip and upload only the course folder
```

```
In [ ]: # import some useful predefined functions from the course package

        from course import *
```

```
Done everything.
```

```
In [ ]: # simple_quadrature constructs an equidistant grid quadrature rule on interv
        # the functions f or F need to be provided as arguments, param is the parame
        def simple_quadrature(r=[0,1],I=10,f="",F="",param=""):
            if F:
                # construct weights from F
                nodes = np.linspace(r[0],r[1],I)
                weights = np.linspace(r[0],r[1],I)
                weights[1:-1] = F((nodes[2:]+nodes[1:-1])/2,param) - F((nodes[1:-1]+
                weights[0] = F((nodes[1]+nodes[0])/2,param) - F(nodes[0],param)
                weights[-1] = F(nodes[-1],param) - F((nodes[-1]+nodes[-2])/2,param)
            elif f:
                nodes = np.linspace(r[0],r[1],I)
                weights = f(nodes,param)
                # nodes at boundaries receive half weight
                weights[0] /= 2
                weights[-1] /= 2
                # renormalize weights to sum up to one
                weights = weights/sum(weights)
            else:
                print('Neither pdf nor cdf were defined.')

            return nodes, weights
```

Test the simple quadrature method.

In [ ]:
```
# f is uniform density on [0,1]
def f_uniform(w,param=""):
    return w*0 + 1
# F is uniform cdf on [0,1]
def F_uniform(w,param=""):
    return w

# construct quadrature rule using uniform density
what,fhat = simple_quadrature([0,1],6,f = f_uniform)
print('Nodes and weights using the density for approximation.')
print('  Nodes',what)
print('  Weights',fhat)

# construct quadrature rule using cdf of a uniform distribution
what,fhat = simple_quadrature([0,1],6,F = F_uniform)
print('Nodes and weights using the cdf for approximation.')
print('  Nodes',what)
print('  Weights',fhat)
```

```
Nodes and weights using the density for approximation.
  Nodes [0.  0.2 0.4 0.6 0.8 1. ]
  Weights [0.1 0.2 0.2 0.2 0.2 0.1]
Nodes and weights using the cdf for approximation.
  Nodes [0.  0.2 0.4 0.6 0.8 1. ]
  Weights [0.1 0.2 0.2 0.2 0.2 0.1]
```

Gaussian quadrature

Gaussian quadrature is a more sophisticated approach to the approximation of the integral

$$\int_a^b g(w)f(w)dw \approx \sum_{i=1}^{I} g\left(w^i\right)f^i$$

that involves the design of I nodes w^i and weights f^i such that for any polynomial function $g(w)$ of degree up to $2I-1$, the discrete quadrature approximation above is exact. The computation of the nodes and weights can itself involve numerical integration but for some choices of intervals $[a,b]$ and weighting functions $f(w)$, explicit solutions exist. Gaussian approximation will work well when we are confident that the integrated function $g(w)$ is well approximated by a polynomial of a given degree.

We provide an implementation here but skip the outline of the derivation of the quadrature rules. For more details, see notes and the Wikipedia page on Gaussian quadrature.

In [ ]:
```
# The algorithm computes nodes and weights using the Golub-Welsch algorithm
def gaussian_quadrature(J,method):
    if method=='Legendre':
        # interval [-1,1], weighting function f(w)=1
        mu0 = 2
        a_n_n = np.zeros([J,1])
        a_n_nlag = np.linspace(1,J-1,J-1)**2 / (4*np.linspace(1,J-1,J-1)**2
    elif (method=='Hermite') or (method=='Hermite-density'):
        # Hermite: interval [-\infty,\infty], weighting function f(w) = exp(
        # Hermite-density: interval [-\infty,\infty], weighting function f(w
        mu0 = np.pi**0.5
        a_n_n = np.zeros([J,1])
        a_n_nlag = np.linspace(1,J-1,J-1)/2
    else:
        print('Quadrature rule not specified.')

    #
    Lambdahat = np.asmatrix(np.diag(a_n_n,0) + np.diag(a_n_nlag**0.5,1) + np
    eigval,eigvec = np.linalg.eig(Lambdahat)
    idx = eigval.argsort()
    nodes = eigval[idx]
    eigvec = eigvec[:,idx]
    weights = nodes.copy()
    for i in range(0,J):
        weights[i] = mu0*eigvec[0,i]**2 / (eigvec[:,i].transpose()*eigvec[:,

    # normalization for Gauss-Hermite quadrature with weighting function equ
    if (method=='Hermite-density'):
        nodes = nodes * 2**0.5
        weights = weights / np.pi**0.5

    #print(nodes)
    #print(type(nodes))
    #print(type(eigvec))
    #print(eigvec)
    #print(eigvec[0])
    return nodes, weights
    #return nodes,weights
```

Test the Gaussian quadrature function.

In [ ]:
```
nodes,weights = gaussian_quadrature(5,'Hermite-density')
```

Numerical solution of the reservation wage equation

We want to find the solution $\bar{w}$ to the equation

$$\bar{w} - c = \frac{\beta}{1-\beta}\int_{\bar{w}}^{B}\left(w' - \bar{w}\right)dF\left(w'\right).$$

Define the function

$$g(w) = w - c - \frac{\beta}{1-\beta}\int_{w}^{B}\left(w' - w\right)dF\left(w'\right).$$

We already know this function is strictly increasing, with $g(0) < 0 < g(B)$. Hence a unique solution to $g(w) = 0$ exists.

Bisection method

The bisection method is a simple algorithm that finds the unique root of a function $g(w)$ on $[a, b]$ with $\text{sgn}(g(a)) \cdot \text{sgn}(g(b)) = -1$.

1. Start with $w_l^0 = a$ and $w_r^0 = b$, choose a precision threshold ϵ.
2. For iteration i, evaluate $w_m = (w_l^i + w_r^i)/2$.
    - if $g(w_m) = 0$, root found, exit.
    - if $\text{sgn}(g(w_m)) \cdot \text{sgn}(g(w_l^i)) = 1$, set $w_l^{i+1} = w_m$ and $w_r^{i+1} = w_r^i$
    - otherwise, set $w_l^{i+1} = w_l^i$ and $w_r^{i+1} = w_m$.
3. Iterate until $w_r^i - w_l^i < \epsilon$, then w_m is the desired root.

The idea of the algorithm is to sequentially remove subintervals in which the function $g(w)$ does not change sign, since the root cannot lie in such subintervals.

In [ ]:
```
def reservation_wage_bisection(model):
    wl,wr = 0, model["B"]
    eps = 10**(-10)
    iters = 0
    while wr-wl > eps:
        w = (wl+wr)/2
        w_nodes,weights = simple_quadrature(r=[w,model["B"]],I = model["I"],
        g = w - model["c"] - model["beta"]/(1-model["beta"])*((w_nodes-w)@we
        if g > 0:
            wr = w
        else:
            wl = w
        iters += 1
    return (wl+wr)/2,iters
```

Test the bisection method.

In [ ]:
```
# first define the structure of the model we are solving
def model_cdf (w,param):
    return w/model["B"]

model = {"beta":0.96, "B": 1, "c": 0.2, "F_cdf": model_cdf, "I" : 1000}

# now call the bisection routine and plot
wbar,i = reservation_wage_bisection(model)
print(f'Reservation wage: {wbar:.5}. Reached in {i} iterations.')
```

```
Reservation wage: 0.78013. Reached in 34 iterations.
```

Newton–Raphson method

The Newton–Raphson method relies on first-order approximation of the function $g(w)$ whose root $\bar{w}$ we are solving for. Consider a guess w_n for the root. Then the first-order

Taylor expansion around w_n implies

$$g\left(\bar{w}\right) - g\left(w_n\right) \approx g'\left(w_n\right)(\bar{w} - w_n).$$

Since $g\left(\bar{w}\right) =$, we can isolate $\bar{w}$ as

$$\bar{w} \approx w_n - \frac{g\left(w_n\right)}{g'\left(w_n\right)}.$$

Hence we obtain an updated guess for the root, which suggests the following algorithm. Start with an initial guess w_0, and then iterate

$$w_{n+1} = w_n - \frac{g\left(w_n\right)}{g'\left(w_n\right)}$$

until $|w_{n+1} - w_n| < \epsilon$, where ϵ is the desired precision.

The method requires the computation of $g'\left(w_n\right)$, which may have to be calculated numerically if an explicit formula is not available.

For the case of the reservation wage equation,

$$g(w) = w - c - \frac{\beta}{1-\beta}\int_w^B \left(w' - w\right) dF\left(w'\right),$$

so that, using the Leibniz rule,

$$g'(w) = \frac{1 - \beta F(w)}{1 - \beta}.$$

An implementation of the Newton–Raphson method is thus readily available when the cdf $F(w)$ is known.

In [ ]:
```
def reservation_wage_newton_raphson(model):

    B = model["B"]
    I = model["I"]
    F_cdf = model["F_cdf"]
    F_param = model["F_param"]
    bet = model["beta"]
    c = model["c"]

    wold,wnew = 0, B/2
    eps = 10**(-10)
    iters = 0
    while abs(wold-wnew) > eps:
        wold = wnew
        w_nodes,weights = simple_quadrature(r=[wold,B],I = I,F=F_cdf,param=F
        g = wold - c - bet/(1-bet)*((w_nodes-wold)@weights)
        dg = (1-bet*F_cdf(wold,F_param))/(1-bet)

        wnew = wold - g/dg
        iters += 1

    return wnew,iters
```

Test the implementation of the Newton–Raphson method.

In [ ]:
```
# first define the structure of the model we are solving
def model_cdf (w,param=""):
    return w/model["B"]

model = {"beta":0.96, "B": 1, "c": 0.2, "F_cdf": model_cdf, "I" : 1000,  "F_

# now call the Newton-Raphson routine and plot
wbar,i = reservation_wage_newton_raphson(model)
print(f'Reservation wage: {wbar:.5}. Reached in {i} iterations.')
```

```
Reservation wage: 0.78013. Reached in 6 iterations.
```

Contraction mapping argument

The contraction mapping argument builds on the equation

$$Q = c + \beta\int_0^B V\left(w'\right)dF\left(w'\right) = c + \beta\int_0^B \max_{\{\text{accept, reject}\}} \left\{\frac{w'}{1-\beta}, Q\right\}dF\left(w'\right).$$

Recall that Q is the present value of the reservation wage,

$$Q = \frac{\bar{w}}{1-\beta} \in \left[0, \frac{B}{1-\beta}\right].$$

We can define the Bellman operator

$$Tq = c + \beta\int_0^B \max_{\{\text{accept, reject}\}} \left\{\frac{w'}{1-\beta}, q\right\}dF\left(w'\right),$$

which can be shown to be a contraction. This means that the equation $Q = TQ$ has a unique solution which can be obtained by successive approximations. Starting from an initial guess $Q_0 \in [0, B/(1-\beta)]$, we can iterate

$$Q_{n+1} = TQ_n,$$

and the contraction mapping argument assures that

$$\lim_{n\to\infty} Q_n = Q.$$

In [ ]:
```
def reservation_wage_Q_iteration(model):
    Qold,Qnew = 0, model["B"]/(1-model["beta"])/2
    eps = 10**(-10)
    iters = 0
    w_nodes,weights = simple_quadrature(r=[0,model["B"]],I = model["I"],F=mc
    while abs(Qold-Qnew) > eps:
        Qold = Qnew
        Qnew = model["c"] + model["beta"]*(np.maximum(w_nodes/(1-model["beta
        iters += 1

    wbar = (1-model["beta"])*Qnew
    return wbar,iters
```

Test the implementation.

In [ ]:
```
# first define the structure of the model we are solving
def model_cdf (w,param=""):
    return w/model["B"]

model = {"beta":0.96, "B": 1, "c": 0.2, "F_cdf": model_cdf, "I" : 1000}

# now call the Newton-Raphson routine and plot
wbar,i = reservation_wage_Q_iteration(model)
print(f'Reservation wage: {wbar:.5}. Reached in {i} iterations.')
```

```
Reservation wage: 0.78013. Reached in 81 iterations.
```

The iteration method that relies on contraction mapping requires substantially more iterations because the rate of convergence is determined by the rate of discounting of future values, given by β, which is close to one.

Value function iteration

We now work with the equation

$$V(w) = \max_{\{\text{accept, reject}\}} \left\{ \frac{w}{1-\beta}, c + \beta \int_0^B V\left(w'\right) dF\left(w'\right) \right\}.$$

Defining the Bellman operator

$$(Tv)(w) = \max_{\{\text{accept, reject}\}} \left\{ \frac{w}{1-\beta}, c + \beta \int_0^B v\left(w'\right) dF\left(w'\right) \right\},$$

we can establish that T is a contraction, which again invites an iterative algorithm. Starting from an initial guess V_0, we have

$$V_{n+1} = TV_n,$$

with the limit $lim_{n\to\infty} V_n = V$.

The substantial change relative to a contraction mapping argument applied to Q is that we are iterating on the whole function V. This is highly inefficient since the only unknown component is the value of Q but we will nevertheless proceed, since the method is useful more broadly.

```
In [ ]: def mccall_value_function_iteration(model):
            I,c,beta = model["I"],model["c"],model["beta"]
            Vnew = np.asmatrix(np.ones([I,1]))
            Vold = np.asmatrix(np.zeros([I,1]))
            w_nodes,weights = simple_quadrature(r=[0,model["B"]],I = model["I"],F=mo
            w_nodes = np.asmatrix(w_nodes).transpose()
            weights = np.asmatrix(weights).transpose()
            eps = 10**(-10)
            iters = 0
            while max(abs(Vold-Vnew)) > eps:
                Vold = Vnew
                Vnew = np.maximum(w_nodes/(1-beta),c + beta*Vnew.transpose()*weights
                iters += 1
            # derive reservation wage from Q = wbar/(1-beta)
            wbar = float((1-model["beta"])*Vnew[0])
            return Vnew,wbar,w_nodes,iters
```

```
In [ ]: # first define the structure of the model we are solving
        def model_cdf(w,param=""):
            return w/model["B"]

        model = {"beta":0.96, "B": 1, "c": 0.2, "F_cdf": model_cdf, "I" : 1000}

        # solve for the value function using value function iteration
        V,wbar,w_nodes,i = mccall_value_function_iteration(model)
        print(f'Reservation wage: {wbar:.5}. Reached in {i} iterations.')
```

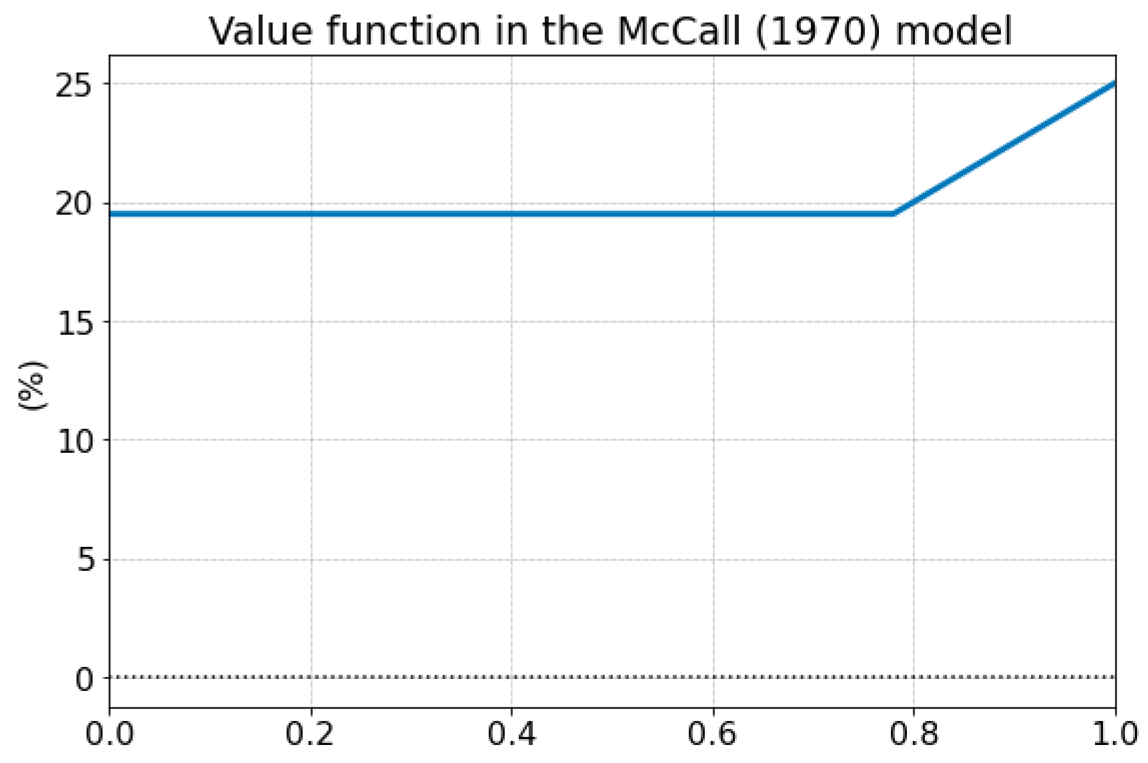
```
Reservation wage: 0.78013. Reached in 83 iterations.
```

We can also plot the value function.

```
In [ ]: param = {'figsize' : [9,6], 'fontsize': 16, 'subplots': [1,1],
                 'title': 'Value function in the McCall (1970) model',
                 'xlim': [0,model["B"]], 'ylim': [0,0],
                 'xlabel': '', 'ylabel': '(%)',
                 'ylogscale': False,
                 'showgrid': True, 'highlightzero': True,
                 'showNBERrecessions' : False, 'showNBERrecessions_y': [0,7]}

        fig,ax = myGenerateTSPlot(param)

        ax.plot(w_nodes,V,linewidth=3,color=myColor['tolVibrantBlue'],linestyle='sol
```


Value function in the McCall (1970) model
(%)
25
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0

```
In [ ]:
```