Семинар 13-14

1. Модель поиска работы МакКолла

Рассмотрим следующую задачу. В текущий момент времени некий индивид безработен. Он существует в следующем упрощенном мире:

- последовательность заработных плат $\{w_t\}_{t \geqslant 0}$ независима и одинаково распределена;
- $w_t \in \mathbb{W}, \ \forall t$, вероятность наблюдать значение w равно $q(w)$;
- индивид наблюдает w_t в начале момента времени t;
- агент **знает** параметры распределения $\{w_t\}_{t \geqslant 0}$.

В момент времени t агент может

- принять предложение о работе и получать постоянную заработную плату w_t;
- отказаться от предложения, получить пособие по безработице c и перейти в следующий период.

Агент живет вечно и стремится максимизировать ожидаемую дисконтированную сумму доходов

$$\mathrm{E} \sum_{t=0}^{\infty} \beta^t y_t \tag{1}$$

где

$$y_t = \begin{cases} w_t & \text{при занятости} \\ c & \text{при безработности} \end{cases}$$

Агент имеет два источника потерь:

- если ждать слишком долго, то потери возникнут из-за дисконтирования;
- если согласиться слишком быстро, то можно упустить более выгодные предложения.

Для нахождения оптимального выбора применяется динамическое программирование. Оно может рассматриваться как двухшаговая процедура:

1. «состояниям» мира присваивается некоторая ценность;
2. оптимальная стратегия строится на основе этих ценностей.

1.1. Функция ценности

Введем функцию $v^*(w)$, равную значению целевой функции (1) при условии, что агент принимает **оптимальные** решения. Также мы предполагаем, что агент входит в текущий момент безработным и получает предложение $w \in \mathbb{W}$.

Нетрудно понять, что, хотя мы не знаем вид функции $v^*(w)$, она должна удовлетворять следующему равенству:

$$v^*(w) = \max\left\{\frac{w}{1-\beta},\ c + \beta \sum_{w' \in \mathbb{W}} v^*(w')q(w')\right\},\ \forall w \in \mathbb{W} \tag{2}$$

Поиск v^* достаточно прост (те, кто внимательно слушал материал предыдущей лекции и/или семинара, могут что-то заподозрить). Обозначим, для удобства, $v(i) \coloneqq v(w_i)$.

1. Возьмите произвольное начальное значение v.
2. Посчитайте новый вектор v' как

$$v'(i) = \max\left\{\frac{w_i}{1-\beta}, c + \beta \sum_{1 \leqslant j \leqslant n} v(j)q(j)\right\},\ \forall i = 1, \dots, n$$

3. Посчитайте какую-либо меру расхождения между векторами v' и v. Например $\max_i |v'(i) - v(i)|$.
4. Продолжайте выполнять шаги 2–3 до тех пор, пока расхождение не будет меньше интересующего вас порога.

1.2. Оптимальная стратегия

Предположим, что мы можем решить (2) для неизвестной функции v^*. Зная значения v^*, мы можем найти оптимальный выбор агента для любого значения w. Проще всего представить его как отображение $\sigma : \mathbb{R} \to \{0, 1\}$.

Пусть

$$\mathbf{1}\{P\} \coloneqq \begin{cases} 1 & P \text{ истинно} \\ 0 & P \text{ ложно} \end{cases}$$

Тогда оптимальный выбор можно определить как

$$\sigma(w) \coloneqq \mathbf{1}\left\{\frac{w}{1-\beta} \geqslant c + \beta \sum_{w' \in \mathbb{W}} v^*(w')q(w')\right\}$$

или

$$\sigma(w) \coloneqq \mathbf{1}\{w \geqslant \bar{w}\}$$
$$\bar{w} \coloneqq (1-\beta)\left(c + \beta \sum_{w' \in \mathbb{W}} v^*(w')q(w')\right)$$

1.3. Другой вид уравнения Беллмана

Уравнение Беллмана (2) можно переписать в виде, в котором вычисление v не требуется. Это удобно, особенно в случае высокой размерности v.

Пусть h — значение ценности продолжения быть безработным (отказа от найма)

$$h = c + \beta \sum_{w \in \mathbb{W}} v^*(w)q(w) \tag{3}$$

или, если перейти к терминам состояний мира,

$$h = c + \beta \sum_{s \in \mathbb{S}} v^*(s) q(s)$$

Уравнение (2) тогда примет вид

$$v^*(w) = \max\left\{\frac{w}{1-\beta}, h\right\} \tag{4}$$

Подставив (4) в (3), получим

$$h = c + \beta \sum_{w \in \mathbb{W}} \max\left\{\frac{w}{1-\beta}, h\right\} q(w) \tag{5}$$

Нахождение оптимального значения h^* аналогично нахождению v^*:

1. Выберите начальное значение h.
2. Посчитайте следующее значение h' по формуле (5).
3. Посчитайте отклонение $|h' - h|$.
4. Продолжайте выполнять шаги 2–3 до тех пор, пока расхождение не будет меньше интересующего вас порога.

1.4. Эксперимент

```
# %%capture
# %pip install -U quantecon
# %pip install -U numba
```

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numba import float64, jit
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial

plt.rcParams["figure.figsize"] = (11, 5)
```

```
n, a, b = 50, 200, 100
q_default = BetaBinomial(n, a, b).pdf()

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

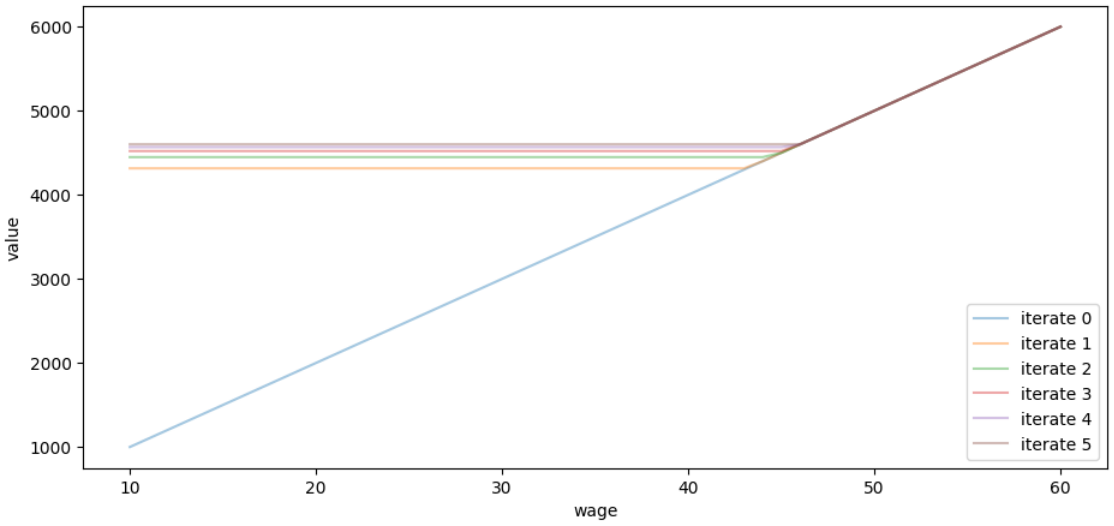
```
mccall_data = [
    ("c", float64),  # unemployment compensation
    ("beta", float64),  # discount factor
    ("w", float64[:]),  # array of wage values, w[i] = wage at␣
  ↪state i
    ("q", float64[:]),  # array of probabilities
]
```

```
@jitclass(mccall_data)
class McCallModel:
    def __init__(self, c=25, beta=0.99, w=w_default,␣
  ↪q=q_default):
        self.c, self.beta = c, beta
        self.w, self.q = w_default, q_default

    def state_action_values(self, i, v):
        c, beta, w, q = self.c, self.beta, self.w, self.q

        accept = w[i] / (1 - beta)
        reject = c + beta * np.sum(v * q)

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

Функция для демонстрации сходимости v к v^*.

```
def plot_value_function_seq(mcm, ax, num_plots=6):
    n = len(mcm.w)
    v = mcm.w / (1 - mcm.beta)
    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 j in range(n):
            v_next[j] = np.max(mcm.state_action_values(j, v))
        v[:] = v_next  # copy contents into v

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

```
mcm = McCallModel()

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




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

    n = len(w)
    v = w / (1 - beta)  # initial guess
    v_next = np.empty_like(v)

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

        if np.max(np.abs(v_next - v)) <= tol:
            break

        v[:] = v_next

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

```
compute_reservation_wage(mcm)
```

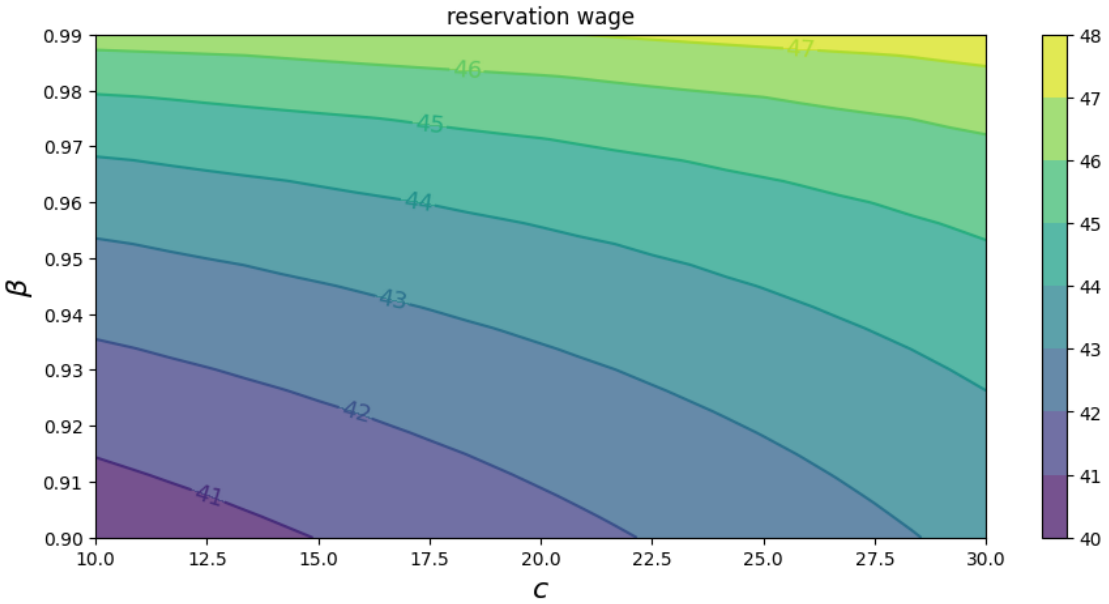
```
47.31649970153045
```

Посмотрим на изменение резервной заработной платы при изменении параметров модели β и c.

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

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

for i, c in enumerate(c_vals):
```

```
    for j, beta in enumerate(beta_vals):
        mcm = McCallModel(c=c, beta=beta)
        R[i, j] = compute_reservation_wage(mcm)
```

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

cs1 = ax.contourf(c_vals, beta_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, beta_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()
```



2. Модель поиска работы МакКолла с сепарацией (увольнением)

Одним из допущений базовой модели является то, что если уж агент работает, то он делает это на протяжении бесконечного горизонта.

Рассмотрим несколько более реалистичную задачу. Агент по-прежнему живет вечно. Распределение предложений по заработной плате такое же, как раньше.

Агент стремится максимизировать ожидаемую дисконтированную полезность своих доходов

$$\mathrm{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{6}$$

где

$$y_t = \begin{cases} w_t & \text{при занятости} \\ c & \text{при безработности} \end{cases}$$

Функция полезности строго монотонная ($u' > 0$) и вогнутая ($u'' < 0$).

В момент времени t **безработный** агент может

- принять предложение о работе и получать постоянную заработную плату w_t;
- отказаться от предложения, получить пособие по безработице c и перейти в следующий период.

В момент времени t **работающий** агент может

- продолжать работать, получая заработную плату w_e;
- быть уволенным с некоторой вероятностью α.

2.1. Функции ценности

Аналогично базовой задаче введем функции ценности для агента. Отличие состоит в том, что нам потребуются **две** функции: для работающего и безработного агента.

Пусть

- $v(w_e)$ — совокупная ценность (значение целевой функции 6) агента, работающего на начало текущего периода;
- $h(w)$ — совокупная ценность агента, безработного на начало текущего периода.

Тогда можно показать, что v и h должны удовлетворять

$$v(w_e) = u(w_e) + \beta \left[(1-\alpha)\, v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w') q(w') \right] \tag{7}$$

$$h(w) = \max \left\{ v(w), u(c) + \beta \sum_{w' \in \mathbb{W}} h(w') q(w') \right\} \tag{8}$$

Уравнения (7) и (8) также можно представить в более простом виде. Пусть d — ожидаемая ценность «завтрашней безработности»

$$d \coloneqq \sum_{w' \in \mathbb{W}} h(w') q(w')$$

Тогда (8) можно записать как

$$\begin{aligned} h(w) &= \max \{ v(w), u(c) + \beta d \} \\ \sum_{w' \in \mathbb{W}} h(w') q(w') &= \sum_{w' \in \mathbb{W}} \max \{ v(w'), u(c) + \beta d \}\, q(w') \\ d &= \sum_{w' \in \mathbb{W}} \max \{ v(w'), u(c) + \beta d \}\, q(w') \end{aligned} \tag{9}$$

(7) же примет вид

$$v(w) = u(w) + \beta \left[(1 - \alpha) v(w) + \alpha d \right] \tag{10}$$

Оптимальные значения d и v также получаются методом послеовательных итераций:

1. Выберите начальные значения d и v.
2. Подставьте их в (9) и (10) и получите следующую итерацию.
3. Рассчитайте меру отклонения текущей итерации от предыдущей.
4. Продолжайте выполнять шаги 2–3 до тех пор, пока расхождение не будет меньше интересующего вас порога.

2.2. Оптимальная стратегия

Предположим, что мы можем решить (9) и (10) для неизвестного значения d и функции v. Зная значения d и v, мы можем найти оптимальный выбор агента для любого значения w.

Оптимальный выбор можно определить как

$$\sigma(w) := \mathbf{1} \{ v(w) \geqslant u(c) + \beta d \}$$

или

$$\begin{aligned} \sigma(w) &:= \mathbf{1} \{ w \geqslant \bar{w} \} \\ \bar{w} &: v(\bar{w}) = u(c) + \beta d \end{aligned}$$

2.3. Эксперимент

```
%reset -f
```

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

plt.rcParams["figure.figsize"] = (11, 5)
```

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

```
n = 60
w_default = np.linspace(10, 20, n)

a, b = 600, 400
q_default = BetaBinomial(n - 1, a, b).pdf()
```

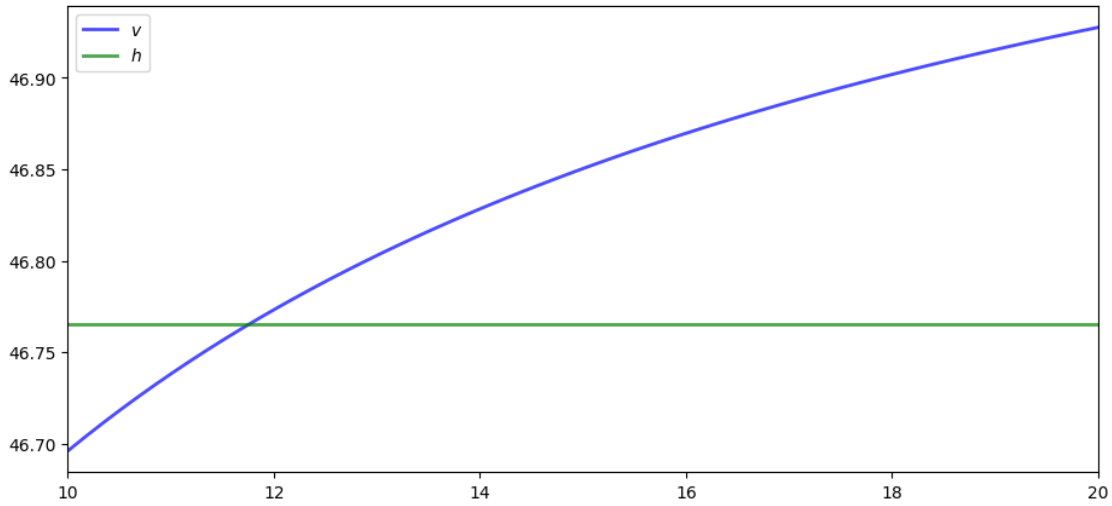
```
mccall_data = [
    ("alpha", float64),  # job separation rate
    ("beta", float64),  # discount factor
```

```
    ("c", float64),  # unemployment compensation
    ("w", float64[:]),  # list of wage values
    ("q", float64[:]),  # pmf of random variable w
]


@jitclass(mccall_data)
class McCallModel:
    def __init__(self, alpha=0.2, beta=0.98, c=6.0,␣
  ↪w=w_default, q=q_default):
        self.alpha, self.beta, self.c, self.w, self.q = alpha,␣
  ↪beta, c, w, q

    def update(self, v, d):
        alpha, beta, c, w, q = self.alpha, self.beta, self.c,␣
  ↪self.w, self.q

        v_new = np.empty_like(v)

        for i in range(len(w)):
            v_new[i] = u(w[i]) + beta * ((1 - alpha) * v[i] +␣
  ↪alpha * d)

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

        return v_new, d_new
```

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    v = np.ones_like(mcm.w)  # Initial guess of v
    d = 1  # Initial guess of d
    error = tol + 1

    for _ in range(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)

        if error <= tol:
            break

        v[:], d = v_new[:], d_new

    return v, d
```

```
mcm = McCallModel()
v, d = solve_model(mcm)
```

```
h = u(mcm.c) + mcm.beta * d

fig, ax = plt.subplots()

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()
```




```
@njit
def compute_reservation_wage(mcm):
    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.beta * d

    i = np.searchsorted(v, h, side="right")
    w_bar = mcm.w[i]

    return w_bar
```

```
compute_reservation_wage(mcm)
```

```
11.864406779661017
```

3. Модель поиска работы МакКолла с сепарацией (непрерывная версия)

Предыдущая версия модели была построена для дискретного набора предложений зарплат. Опишем модель для **непрерывного** распределения зарплат.

Уравнения (9) и (10) примут следующий вид

$$d = \int \max\{v(w), u(c) + \beta d\}\, q(w)\, dw \tag{11}$$

$$v(w) = u(w) + \beta\left[(1-\alpha)\, v(w) + \alpha d\right] \tag{12}$$

где $q(w)$ — плотность распределения w.

Как и раньше, оптимальные значения d и v также получаются методом последовательных итераций:

1. Выберите начальные значения d и v.
2. Подставьте их в (11) и (12) и получите следующую итерацию.
3. Рассчитайте меру отклонения текущей итерации от предыдущей.
4. Продолжайте выполнять шаги 2–3 до тех пор, пока расхождение не будет меньше интересующего вас порога.

Проблема в том, что v теперь непредставима в виде какого-либо вектора, ведь его длина должна быть бесконечной.

Способ решения этой проблемы состоит в применении подхода **Value Function Iteration**. Данный подход состоит в том. что нам не нужно хранить **все** значения v, нам достаточно хранить значения на **конечном** числе промежуточных точек. Значения функции v между этими точками получаются интерполяцией тем или иным методом.

С учетом этого, алгоритм примет следующий вид:

1. Выберите начальные значения скаляра d и функции v на некоторой сетке значений $\{w_i\}$.
2. Проинтерполируйте функцию и посчитайте новое значение d. Так как интеграл представляет собой математическое ожидание, то **примерное** интегрирование можно провести так:
   - Выберите достаточно большую случайную выборку значений w.
   - Найдите проинтерполированные значения функции v на этой выборке.
   - Посчитайте выборочное среднее. По ЗБЧ это среднее приблизительно равно математическому ожиданию.
3. Подставьте полученное значение d' в (12) и получите следующую итерацию значений v на сетке $\{w_i\}$.
4. Рассчитайте меру отклонения текущей итерации от предыдущей.
5. Продолжайте выполнять шаги 2–3 до тех пор, пока расхождение не будет меньше интересующего вас порога.

3.1. Эксперимент

```
%reset -f
```

```
import matplotlib.pyplot as plt
import numpy as np
from numba import float64, njit
from numba.experimental import jitclass

```

```
plt.rcParams["figure.figsize"] = (11, 5)
```

Функция для создания случайной выборки из зарплат.

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

```
mccall_data_continuous = [
    ("c", float64),  # unemployment compensation
    ("alpha", float64),  # job separation rate
    ("beta", float64),  # discount factor
    ("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,
        alpha=0.1,
        beta=0.96,
        grid_min=1e-10,
        grid_max=5,
        grid_size=100,
        w_draws=lognormal_draws(),
    ):
        self.c, self.alpha, self.beta = c, alpha, beta

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

    def update(self, v, d):
        # Simplify names
        c, alpha, beta = self.c, self.alpha, self.beta
        w = self.w_grid

        def u(x):
            return np.log(x)

        # Interpolate array represented value function
        def vf(x):
            return np.interp(x, w, v)

        # Update d using Monte Carlo to evaluate integral
```

```
        d_new = np.mean(np.maximum(vf(self.w_draws), u(c) +␣
  ↪beta * d))

        # Update v
        v_new = u(w) + beta * ((1 - alpha) * v + alpha * d)

        return v_new, d_new
```

```
@njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    v = np.ones_like(mcm.w_grid)
    d = 1
    error = tol + 1

    for _ in range(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)

        if error <= tol:
            break

        v[:], d = v_new[:], d_new

    return v, d
```

```
@njit
def compute_reservation_wage(mcm):
    def u(x):
        return np.log(x)

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

    i = np.searchsorted(v, h, side="right")

    return mcm.w_grid[i]
```

```
mcm = McCallModelContinuous()
compute_reservation_wage(mcm)
```

```
4.040404040423232
```

4. Коррелированные заработные платы

Рассмотрим постановку **базовой** задачи, в которой предложения заработных плат непрерывны и представляют собой сумму двух процессов

$$\begin{aligned} w_t &= \exp(z_t) + y_t \\ z_t &= d + \rho z_{t-1} + \sigma \varepsilon_t \\ y_t &\sim \exp(\mu + s\zeta_t) \\ \varepsilon_t,\ \zeta_t &\sim iid\ N(0,1) \end{aligned}$$

z_t является устойчивой авторегрессионной компонентой, а y_t — возмущением.

Условие для функции ценности v^* примет следующий вид

$$v^*(w,z) = \max\left\{\frac{u(w)}{1-\beta}, u(c) + \beta\, \mathrm{E}_z\, v^*(w',z')\right\} \tag{13}$$
$$\mathrm{E}_z\, x \coloneqq \mathrm{E}(x \mid z)$$

Если обозначить ценность отказа от найма как

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

то (13) примет вид

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

что приводит нас к выражению

$$f(z) = u(c) + \beta\, \mathrm{E}_z \max\left\{\frac{u(w')}{1-\beta}, f(z')\right\} \tag{14}$$

Соответственно, решив уравнение (14) относительно $f(z)$ с помощью последовательных аппроксимаций, можем определить оптимальный выбор агента как

$$\sigma(w,z) \coloneqq \mathbf{1}\left\{\frac{u(w)}{1-\beta} \geqslant f(z)\right\}$$

а резервную заработную плату как

$$\bar{w}(z) \coloneqq u^{-1}(f(z)(1-\beta))$$

4.1. Эксперимент

```
%reset -f
```

```
import matplotlib.pyplot as plt
import numpy as np
from numba import float64, njit, prange
from numba.experimental import jitclass
import quantecon as qe

plt.rcParams["figure.figsize"] = (11, 5)
```

```
job_search_data = [
    ("mu", float64),
    ("s", float64),
    ("d", float64),
    ("rho", float64),
    ("sigma", float64),
    ("beta", float64),
    ("c", float64),
    ("z_grid", float64[:]),
    ("e_draws", float64[:, :]),
]
```

```
@jitclass(job_search_data)
class JobSearch:
    def __init__(
        self,
        mu=0.0,
        s=1.0,
        d=0.0,
        rho=0.9,
        sigma=0.1,
        beta=0.98,
        c=5,
        mc_size=1000,
        grid_size=100,
    ):
        self.mu, self.s, self.d = mu, s, d

        self.rho, self.sigma, self.beta, self.c = rho, sigma,␣
 ↪beta, c

        # Строим сетку для z как 3 отклонения от среднего
        z_mean = d / (1 - rho)
        z_sd = sigma / np.sqrt(1 - rho**2)
        k = 3
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        np.random.seed(1234)
        self.e_draws = np.random.randn(2, mc_size)

    def parameters(self):
        return self.mu, self.s, self.d, self.rho, self.sigma,␣
 ↪self.beta, self.c
```

Определим оператор Q : $Qf_t := Qf(z_t) = f_{t+1}$.

```
@njit(parallel=True)
def Q(js, f_in, f_out):
    mu, s, d, rho, sigma, beta, 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 + rho * z + sigma * e1

            # Ценность безработицы
            go_val = np.interp(z_next, js.z_grid, f_in)

            # Предложение заработной платы
            y_next = np.exp(mu + s * e2)
            w_next = np.exp(z_next) + y_next

            # Ценность найма
            stop_val = np.log(w_next) / (1 - beta)

            # Сумма для усреднения
            expectation += max(stop_val, go_val)
        expectation = expectation / M
        f_out[i] = np.log(c) + beta * expectation
```

Функция для приблизительного вычисления неподвижной точки Q.

```
def compute_fixed_point(js, tol=1e-4, max_iter=1000,␣
  ↪verbose=True, print_skip=25):
    f_init = np.full(len(js.z_grid), np.log(js.c))
    f_out = np.empty_like(f_init)

    # Начальное значение для цикла
    f_in = f_init
    error = np.inf

    for i in range(max_iter):
        Q(js, f_in, f_out)
        error = np.max(np.abs(f_in - f_out))

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        f_in[:] = f_out

        if error <= tol:
            break

    if error > tol:
        print("Failed to converge!")
    elif verbose:
```

```
        print(f"\nConverged in {i} iterations.")

    return f_out
```

```
%%time
js = JobSearch()
f_star = compute_fixed_point(js, verbose=True)
```

```
Error at iteration 0 is 57.39139771207811.
Error at iteration 25 is 0.5362345562527935.
Error at iteration 50 is 0.11142701301950808.
Error at iteration 75 is 0.027023284242943646.
Error at iteration 100 is 0.006773730534447964.
Error at iteration 125 is 0.0017060647048054989.
Error at iteration 150 is 0.0004305088477565276.
Error at iteration 175 is 0.0001086401849477169.

Converged in 177 iterations.
CPU times: total: 25.7 s
Wall time: 8.06 s
```

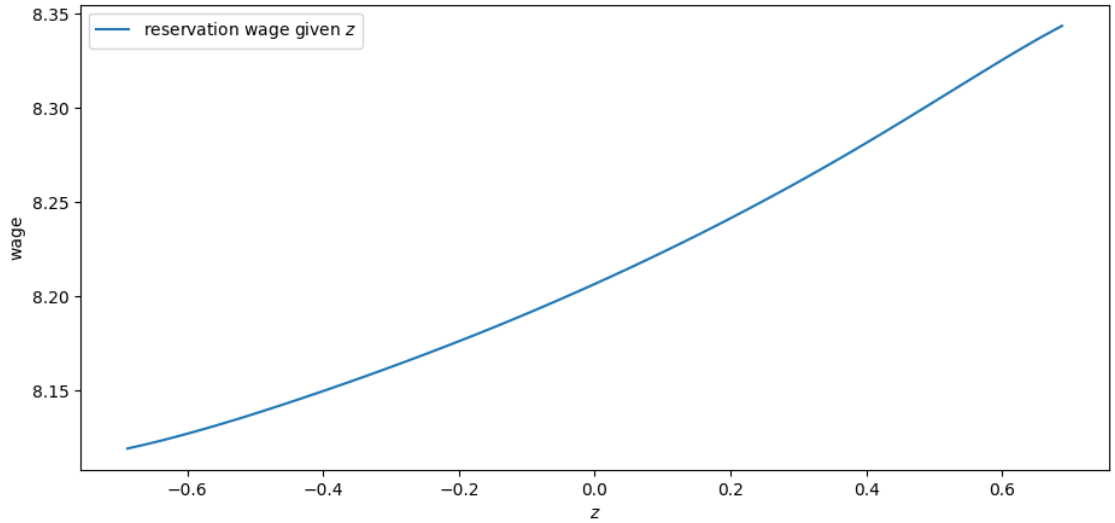
Для подсчета времени исполнения можно воспользоваться функциями `quantecon.tic()` и `quantecon.toc()`.

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

```
Error at iteration 0 is 57.39139771207811.
Error at iteration 25 is 0.5362345562527935.
Error at iteration 50 is 0.11142701301950808.
Error at iteration 75 is 0.027023284242943646.
Error at iteration 100 is 0.006773730534447964.
Error at iteration 125 is 0.0017060647048054989.
Error at iteration 150 is 0.0004305088477565276.
Error at iteration 175 is 0.0001086401849477169.

Converged in 177 iterations.
TOC: Elapsed: 0:00:2.68

2.684863805770874
```

Выведем график резервной заработной платы.

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

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()
```



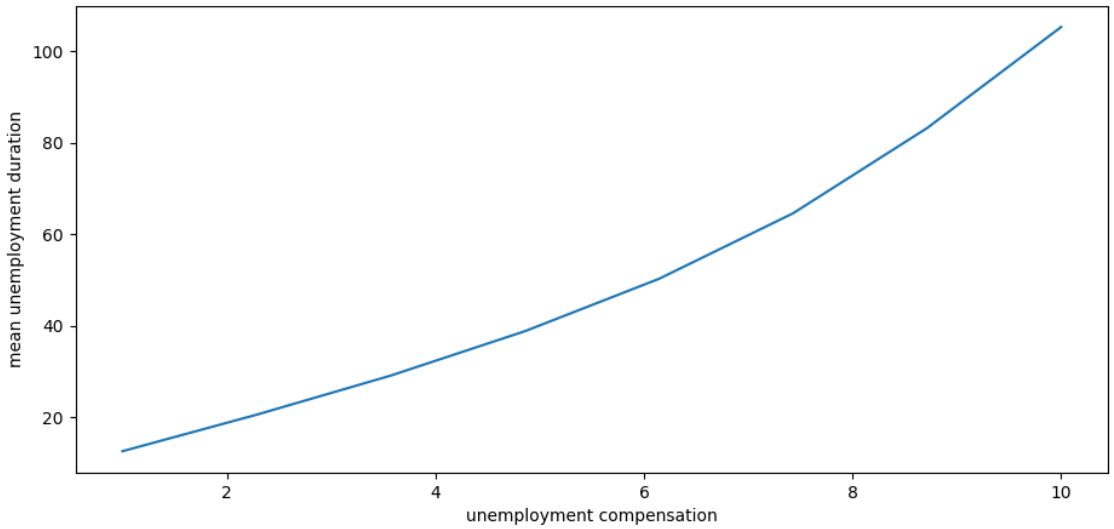
Оценим длину периода безработицы в зависимости от величины компенсации c. Сначала определим функцию для определения продолжительности периода безработицы для конкретной величины пособия.

```
def compute_unemployment_duration(js, seed=1234):
    f_star = compute_fixed_point(js, verbose=False)
    mu, s, d, rho, sigma, beta, c = js.parameters()
    z_grid = js.z_grid

    np.random.seed(seed)

    # Локальная функция, интерполирующая оптимальную функцию f
    @njit
    def f_star_function(z):
        return np.interp(z, z_grid, f_star)

    # Находим длину периода безработицы.
    # Это одна итерация.
    # Для простоты начальное значение z = 0.
    @njit
    def draw_tau(t_max=10_000):
        z = 0
        t = 0
```

```
        for t in range(t_max):
            # Генерируем случайную зарплату
            y = np.exp(mu + s * np.random.randn())
            w = np.exp(z) + y

            # Находим резервную зарплату
            res_wage = np.exp(f_star_function(z) * (1 - beta))

            # Возвращаем t, если агент согласен работать
            if w >= res_wage:
                return t
            # иначе переходим к следующему шагу
            else:
                z = rho * z + d + sigma * np.random.randn()
        return t_max

    @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()
```

```
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)
    durations[i] = compute_unemployment_duration(js)
```

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



5. Смена карьеры

Рассмотрим еще одну модификацию **базовой** модели, в которой предполагается, что агент может менять работу и «карьеру». Что имеется в виду?

В данной постановке считается, что предложение заработной платы состоит из двух частей

$$
\begin{aligned}
w_t &= \theta_t + \varepsilon_t \\
\theta_t &\sim F \\
\varepsilon_t &\sim G
\end{aligned}
$$

где

- θ_t — компонента карьеры, то есть области деятельности, в рамках которой есть несколько вакансий.
- ε_t — компонента конкретной вакансии.

θ_t и ε_t независимы.

В каждый момент времени занятый агент (а в данной постановке он всегда занят) может

- сохранить текущую карьеру и работу,
- поменять работу, но сохранить карьеру,
- поменять и работу, и карьеру.

Пусть агент решает задачу максимизации (1). Его функция ценности имеет вид

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

где

$$
\begin{aligned}
I &= \theta + \varepsilon + \beta v(\theta, \varepsilon) \\
II &= \theta + \int \varepsilon' G(d\varepsilon') + \beta \int v(\theta, \varepsilon') G(d\varepsilon') \\
III &= \int \theta' F(d\theta') + \int \varepsilon' G(d\varepsilon') + \beta \iint v(\theta', \varepsilon') G(d\varepsilon') F(d\theta')
\end{aligned}
\tag{15}
$$

Задача решается аналогично предыдущим: при помощи последовательных итераций ищется оптимальная функция v на сетке значений θ и ε, с учетом условий (15).

5.1. Эксперимент

```
%reset -f
```

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from matplotlib import cm
from numba import njit, prange
from quantecon.distributions import BetaBinomial

plt.rcParams["figure.figsize"] = (11, 5)
```

В качестве распределений F и G используется бета-биномиальное распределение с параметрами $a = 1, b = 1$, что эквивалентно равномерному распределению.

```
class CareerWorkerProblem:
    def __init__(self, B=5.0, beta=0.95, grid_size=50, F_a=1,␣
  ↪F_b=1, G_a=1, G_b=1):
        self.beta, self.grid_size, self.B = beta, grid_size, B
        self.theta = np.linspace(0, B, grid_size)
        self.epsilon = np.linspace(0, B, grid_size)
        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.theta * self.F_probs)
        self.G_mean = np.sum(self.epsilon * self.G_probs)
```

Следующая функция возвращает скомпилированные функции для оператора Q и функции оптимального выбора $\sigma(\theta, \varepsilon)$.

```
def operator_factory(cw, parallel_flag=True):
    theta, epsilon, beta = cw.theta, cw.epsilon, cw.beta
    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 Q(v):
        v_new = np.empty_like(v)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = theta[i] + epsilon[j] + beta * v[i, j]
                v2 = theta[i] + G_mean + beta * v[i, :] @ G_probs
                v3 = G_mean + F_mean + beta * F_probs @ v @ G_probs
                v_new[i, j] = max(v1, v2, v3)
```

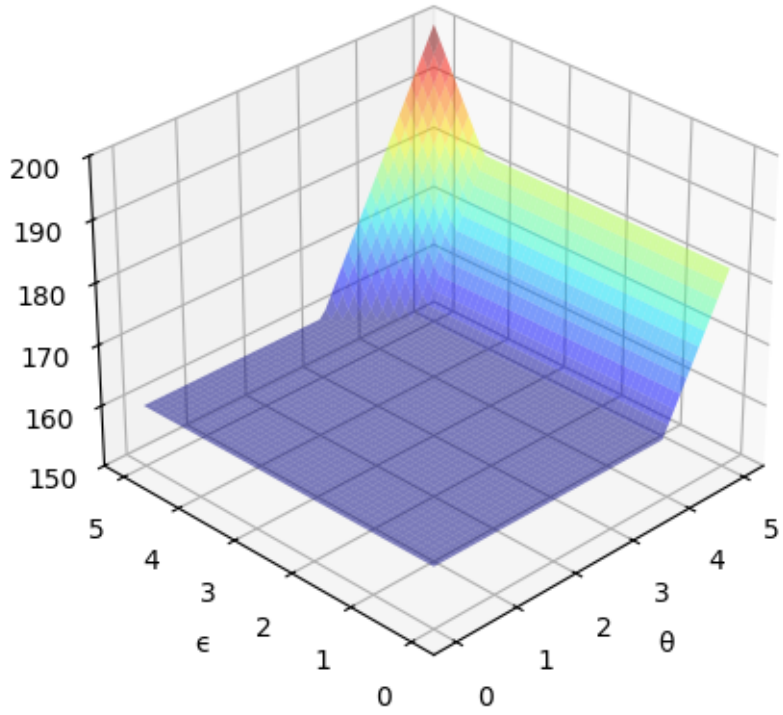
```
        return v_new

    @njit(parallel=parallel_flag)
    def get_greedy(v):
        sigma = np.empty(v.shape)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = theta[i] + epsilon[j] + beta * v[i, j]
               v2 = theta[i] + G_mean + beta * v[i, :] @ G_probs
              v3 = G_mean + F_mean + beta * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                sigma[i, j] = action

        return sigma

    return Q, get_greedy
```

```
def solve_model(
    cw, use_parallel=True, tol=1e-4, max_iter=1000,␣
  ↪verbose=True, print_skip=25
):
    Q, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.full((cw.grid_size, cw.grid_size), cw.F_mean + cw.
  ↪G_mean)
    error = np.inf

    for i in range(max_iter):
        v_new = Q(v)
        error = np.max(np.abs(v - v_new))

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        if error <= tol:
            break

        v = v_new

    if error > tol:
        print("Failed to converge!")
```

```
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v
```

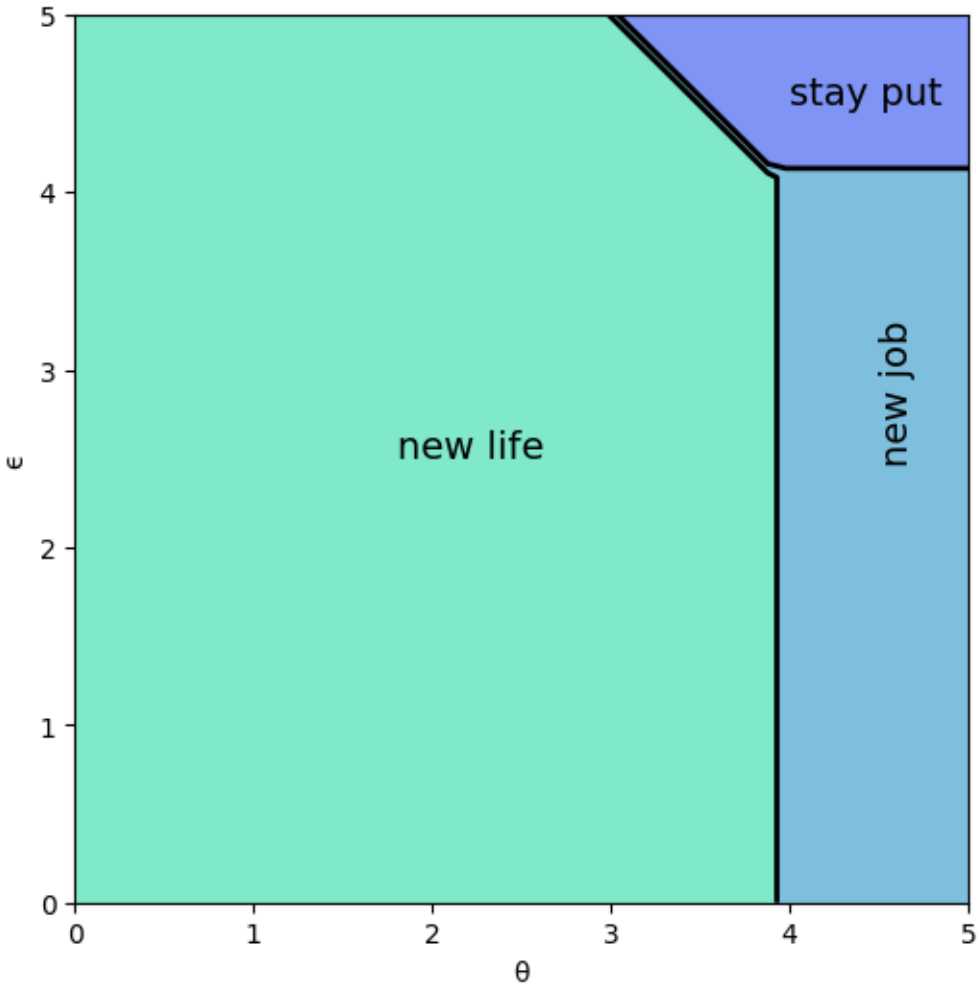
```
cw = CareerWorkerProblem()
_, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

tg, eg = np.meshgrid(cw.theta, cw.epsilon)
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()
```




```
fig, ax = plt.subplots(figsize=(6, 6))

tg, eg = np.meshgrid(cw.theta, cw.epsilon)
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()
```



6. Принятие решения о смене работы

Пусть

- x_t — специфичная для данного места работы величина человечесткого капитала в момент времени t.
- $w_t = x_t(1 - s_t - \phi_t)$ — текущая зарплата, где

– s_t — усилия по поиску работы, получение предложений;
– ϕ_t — инвестиции в текущий человеческий капитал.

Пока работник сохраняет свое рабочее место, его человеческий капитал подчиняется закону $x_{t+1} = g(x_t, \phi_t)$.

Работник получает предложение с вероятностью $\pi(s_t) \in [0, 1]$. Каждое предложение имеет полезность u_{t+1}, причем $\{u_t\} \sim \text{iid } f$.

Так как работник может отвергнуть предложение, то

$$x_{t+1} = \begin{cases} g(x_t, \phi_t) & \text{не принимает} \\ u_{t+1} & \text{принимает} \end{cases}$$

Если $b_{t+1} \in \{0, 1\}$ — бинарная случайная величина, показывающая принятие работником предложения, то

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

Если взять математическое ожидание от целевой функции, то можно получить следующее уравнение Беллмана

$$v(x) = \max_{0 \leqslant s+\phi \leqslant 1} \left\{ x(1 - s - \phi) + \beta[1 - \pi(s)]v[g(x, \phi)] + \beta\pi(s) \int v[\max\{g(x, \phi), u\}] f(du) \right\}$$

Далее рассмотрим следующую параметризацию модели

- $g(x, \phi) = A(x\phi)^\alpha, \quad A = 1.4,\ \alpha = 0.6$,
- $\pi(s) = \sqrt{s}$,
- $f = \text{Beta}(2, 2)$,
- $\beta = 0.96$.

Несложно увидеть, что предельные издержки инвестирования в человеческий капитал и в поиск работы одинаков. Поэтому работник будет выбирать «инвестпродукт» в зависимости от того, что ему даст большую прибыль.

Например, если $x = 0.05$, то

- $s = 1,\ \phi = 0$, тогда $g(x, \phi) = 0$ и $\mathbb{E}x_{t+1} = \pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- $s = 0,\ \phi = 1$, тогда $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Или, если $x = 0.4$, то

- $s = 1,\ \phi = 0$, тогда $g(x, \phi) = 0$ и $\mathbb{E}x_{t+1} = \pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- $s = 0,\ \phi = 1$, тогда $g(x, \phi) = g(0.4, 1) \approx 0.8$.

Также можно показать, что для **бесконечно** терпеливого работника $s = 0$, и можно получить равновесную зарплату

$$w^*(\phi) = x^*(\phi)(1 - \phi)$$

где x^* — неподвижная точка $g(x, \phi)$.

6.1. Эксперимент

```
%reset -f
```

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
from numba import jit, prange
```

Создадим класс, в котором будет храниться вся параметризация нашей модели.

```
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,
        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 = jit(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)
```

Следующая функция создает две функции:

- оператор $Tv(x) = \max_{0 \leqslant s+\phi \leqslant 1} w(s, \phi)$;
- функцию `get_greedy`, возвращающую оптимальные значения параметров модели x, s, ϕ.

```
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

    @jit
    def state_action_values(z, x, v):
        s, φ = z

        def v_func(x):
            return np.interp(x, x_grid, v)

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

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

        search_grid = np.linspace(ε, 1, 15)
        v_new = np.empty_like(v)
        for i in prange(len(x_grid)):
            x = x_grid[i]

            # Search on a grid
            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

    @jit
    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)

        search_grid = np.linspace(ε, 1, 15)
        for i in range(len(x_grid)):
            x = x_grid[i]
            # Search on a grid
            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
```

Функция для решения модели.

```
def solve_model(
    jv, use_parallel=True, 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 = np.inf

    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 error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

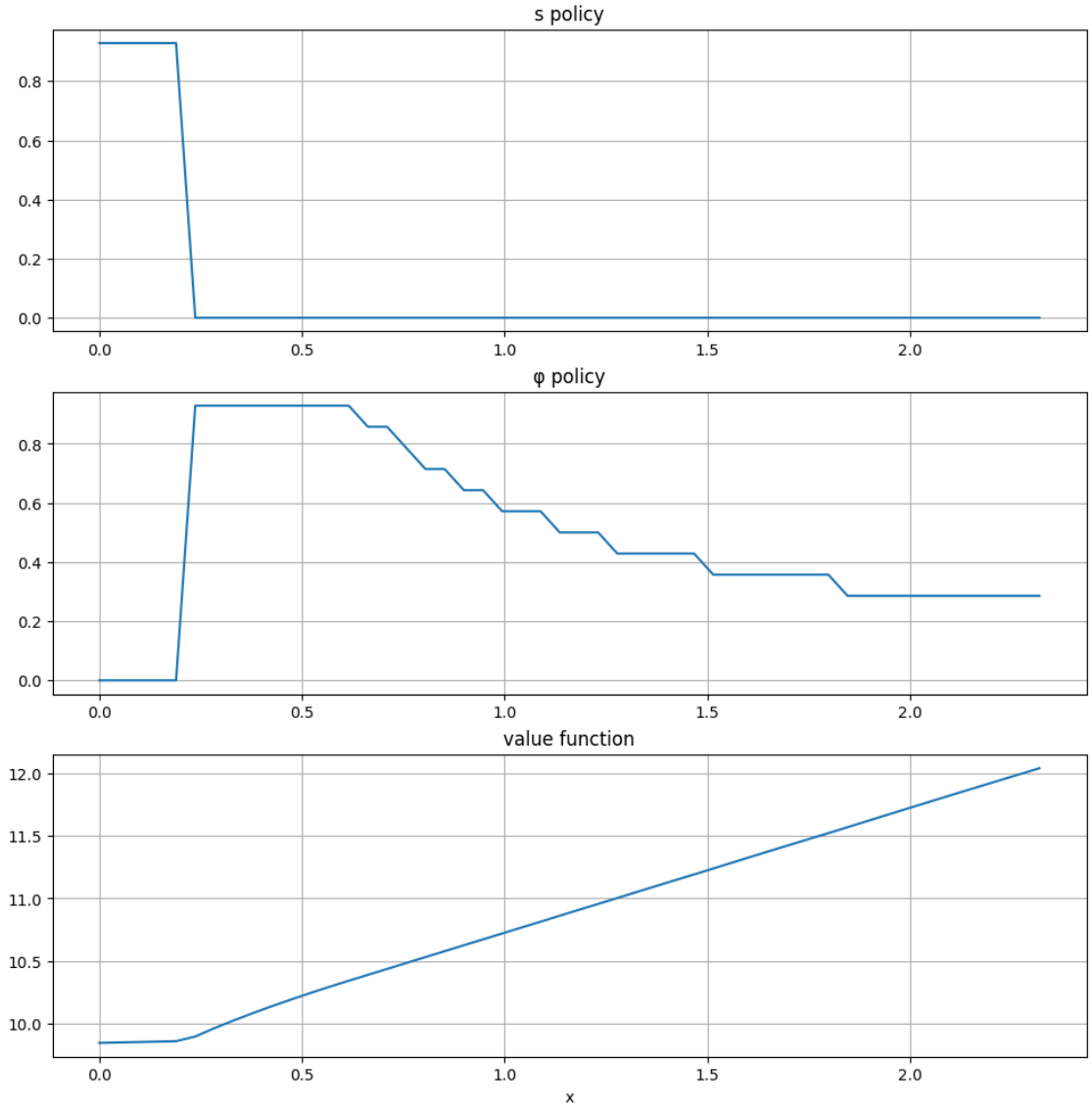
Посмотрим на оптимальные траектории.

```
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 0.1511072077890594.

Error at iteration 50 is 0.054458541555273854.

Error at iteration 75 is 0.019626679581397966.

Error at iteration 100 is 0.00707339088396175.

Error at iteration 125 is 0.0025492268516345007.

Error at iteration 150 is 0.0009187329878574957.

Error at iteration 175 is 0.0003311083524959457.

Error at iteration 200 is 0.00011933036316591483.


Converged in 205 iterations.
```

Графики...

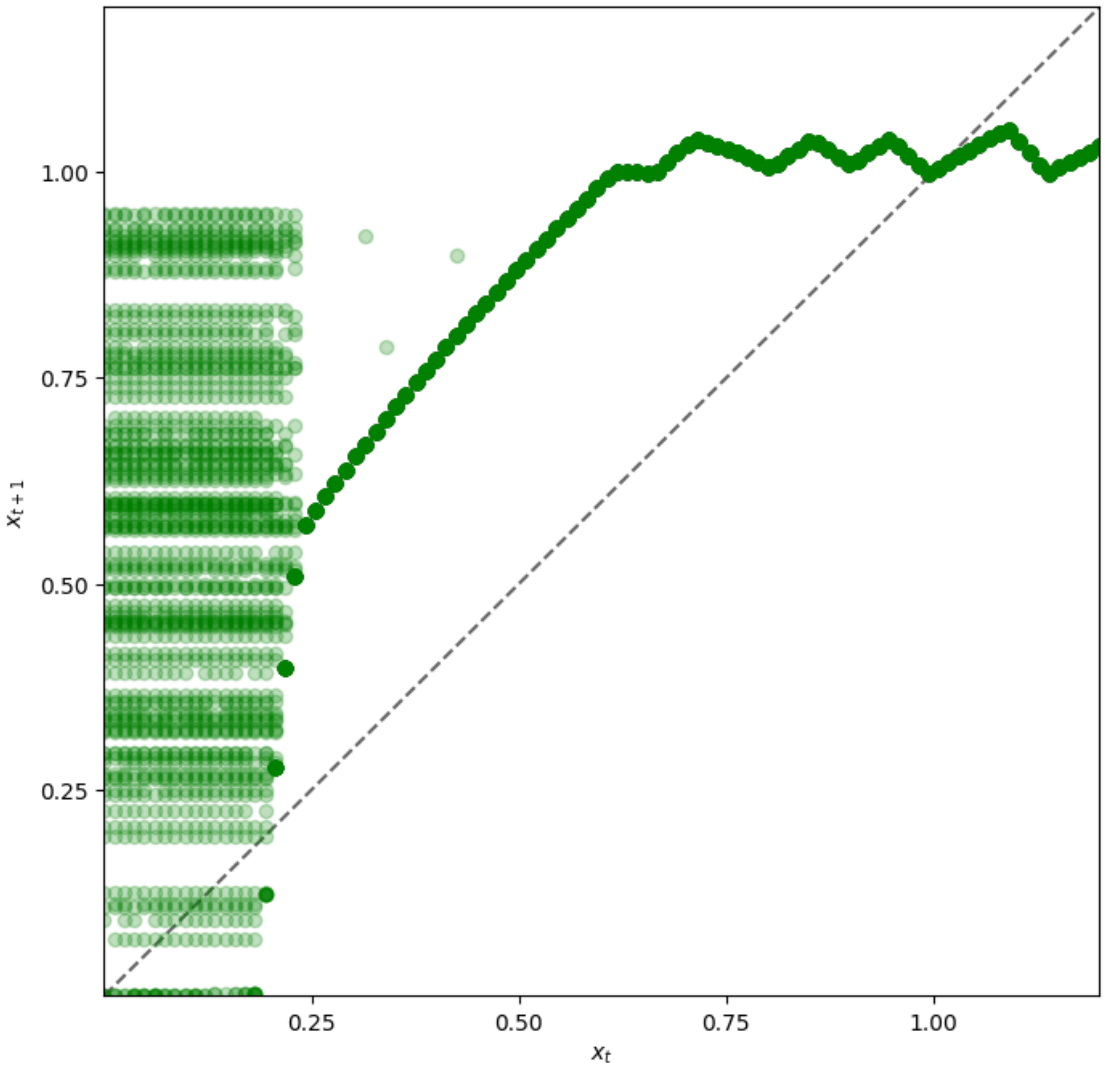
```
plots = [s_star, φ_star, v_star]
titles = ["s policy", "φ policy", "value function"]

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()
```



Посмотрим на 45-градусную диаграмму x_{t+1} от x_t.

```
π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv, verbose=False)
s_policy, φ_policy = get_greedy(v_star)


# Turn the policy function arrays into actual functions
def s(y):
    return np.interp(y, x_grid, s_policy)


def φ(y):
    return np.interp(y, x_grid, φ_policy)
```

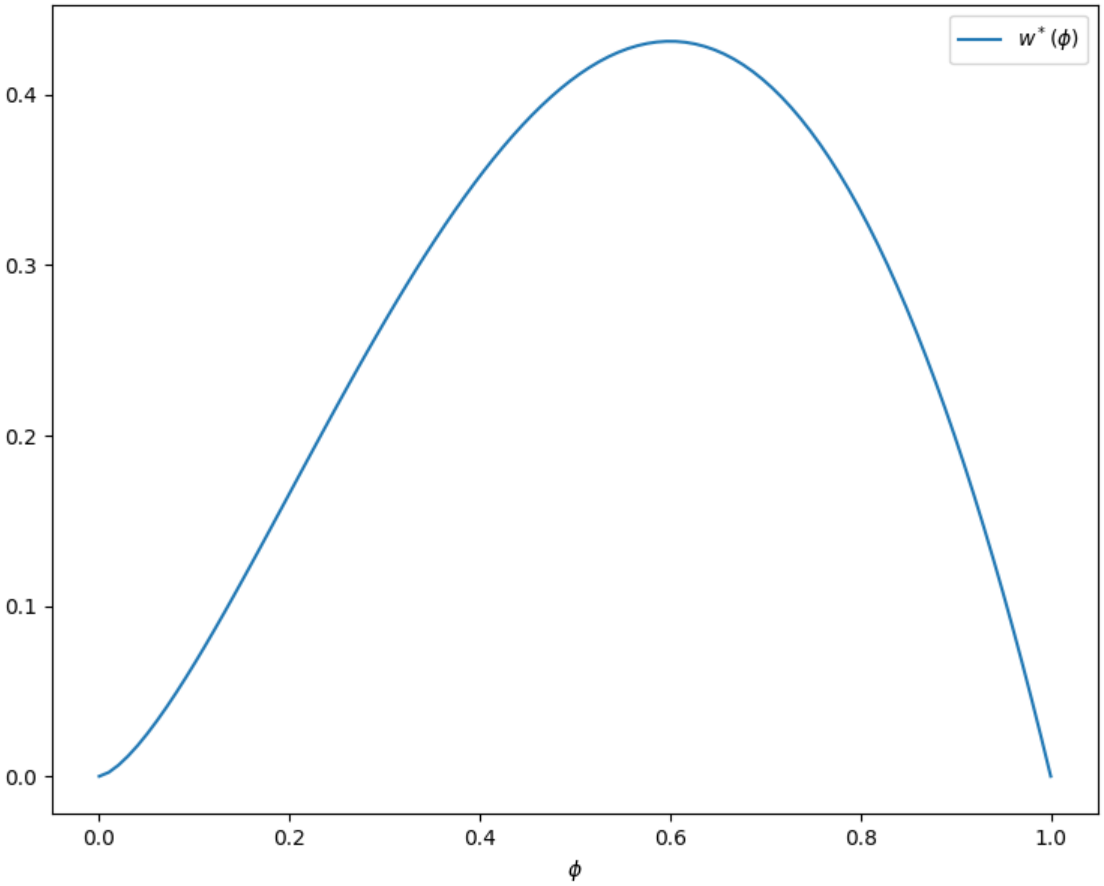
```
def h(x, b, u):
    return (1 - b) * g(x, φ(x)) + b * max(g(x, φ(x)), u)


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()
```



Видно, что все сходится к значению $x = 1$. Это влечет $s = 0,\ \phi = 1$.

Плучим то же самое при помощи стационарного равновесного решения.

```
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=r"$\phi$")
ax.plot(φ_grid, [xbar(φ) * (1 - φ) for φ in φ_grid],␣
  ↪label=r"$w^*(\phi)$")
ax.legend()

plt.show()
```


$w^*(\phi)$
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0
ϕ