Семинар 4

1. Лекция 4: Метод Ньютона

Смелые и отважные могут ознакомиться со следующей книгой

> Лекции по вычислительной математике: Учебное пособие / И.Б. Петров, А. И. Лобанов. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. — 523 с: ил., табл.— (Серия «Основы информационных технологий»)

Нам понадобится пакет **autograd**, который можно установить любым удобным способом. Я пользуюсь **uv**, вы, скорее всего, будете использовать **pip** или **conda**.

[1]:
```
import matplotlib.pyplot as plt
from collections import namedtuple
import scipy.optimize as sco

import numpy as np
import autograd

plt.rcParams["figure.figsize"] = (10, 3)
```

1.1. Неподвижные точки: модель Солоу

Рассмотрим модель Солоу, которую вы должны знать из курса макроэкономики. Вы должны (в теории) помнить, что в данной модели используется производственная функция Кобба-Дугласа

$$Y = AK^{\alpha}L^{1-\alpha}, \quad 0 < \alpha < 1$$

Также вам должно быть известно, что модель Солоу выводится для величин *per capita*

$$y = Ak^{\alpha}$$

И, наконец, в модели считается, что движение капитала описывается следующим выражением

$$k_{t+1} = sAk^{\alpha} + (1 - \delta)k$$

где δ — норма выбытия капитала.

Из этого выражения можно вычислить значение k^*, являющееся неизменным во

времени, то есть стационарным или равновесным. Очевидно, что для него

$$\begin{aligned} k^* &= sAk^{*\alpha} + (1-\delta)k^* =: g(k^*) \\ 0 &= sAk^{*\alpha} - \delta k^* \\ \delta k^{*1-\alpha} &= sA \\ k^{*1-\alpha} &= \frac{sA}{\delta} \\ k^* &= \left(\frac{sA}{\delta}\right)^{1/(1-\alpha)} \end{aligned}$$

1.1.1. Посмотрим на компьютере

Для удобного хранения параметров воспользуемся `namedtuple`.

[2]:
```
SolowParameters = namedtuple(
    "SolowParameters",
    ("A", "s", "alpha", "delta"),
    defaults=(2.0, 0.3, 0.3, 0.4),
)
params = SolowParameters(A=2.0, s=0.3, alpha=0.3, delta=0.4)
```

`namedtuple` — это `tuple`, который `named`. Если серьезно, это кортеж, в котором элементы именованы. Это позволяет

- обращаться к ним по имени,
- указывать их в произвольном порядке при создании объекта.

Также `namedtuple` позволяет указать значения по умолчанию, что также достаточно удобно, что мы увидим ниже.

В остальном это обычный кортеж: итерабельный, фиксированного размера и неизменяемый.

[3]:
```
params.A = 4
```

```
---------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent
  ↪call last)
Cell In[3], line 1
----> 1 params.A = 4

AttributeError: can't set attribute
```

Запишем функции, которые понадобятся нам для решения задачи поиска стационарной точки. Можно и без них, но они позволят нам сделать дальнейший код более универсальным.

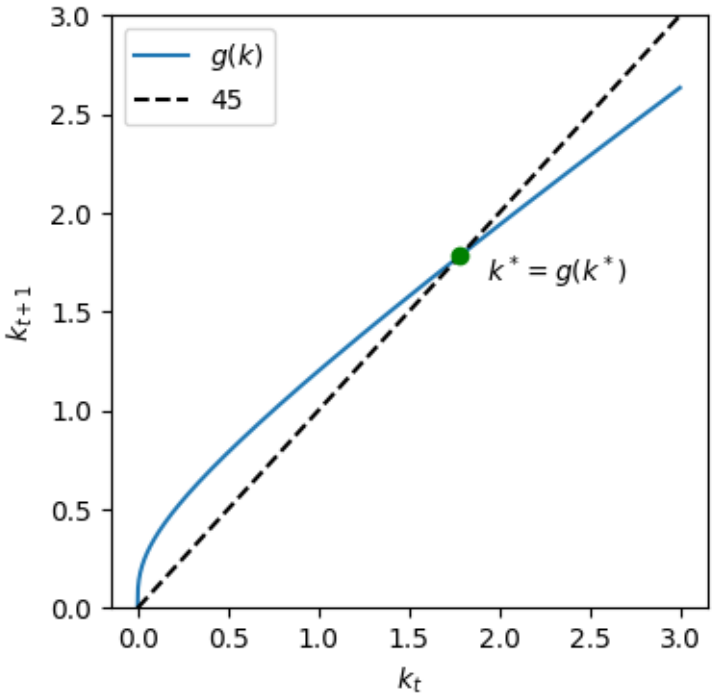
[4]:
```
def g(k, params):
    A, s, a, d = params
    return A * s * k**a + (1 - d) * k
```

```
def exact_fixed_point(params):
    A, s, a, d = params
    return ((s * A) / d) ** (1 / (1 - a))
```

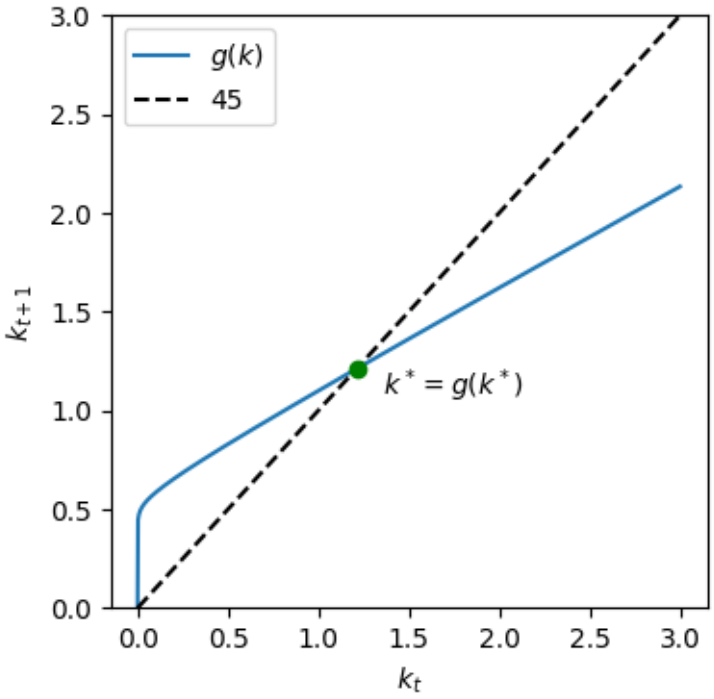
Добавим функцию, строящую графики для модели Солоу.

[5]:
```
def solow_plot(params, k_min=0.0, k_max=3.0, figsize=(4, 4)):
    k_grid = np.linspace(k_min, k_max, 1200)

    fig, (ax) = plt.subplots(figsize=figsize)
    ax.plot(k_grid, g(k_grid, params), label=r"$g(k)$")
    ax.plot(k_grid, k_grid, "k--", label="45")

    k_star = exact_fixed_point(params)
    ax.plot((k_star,), (k_star,), "go")
    ax.annotate(
        r"$k^* = g(k^*)$",
        xy=(k_star, k_star),
        xycoords="data",
        xytext=(10, -10),
        textcoords="offset points",
    )

    ax.legend(loc="upper left")

    ax.set_ylim(0, 3)
    ax.set_xlabel("$k_t$")
    ax.set_ylabel("$k_{t+1}$")
    plt.show()
```

[6]:
```
solow_plot(params)
```



[7]:
```
solow_plot(SolowParameters(alpha=0.05, delta=0.5))
```



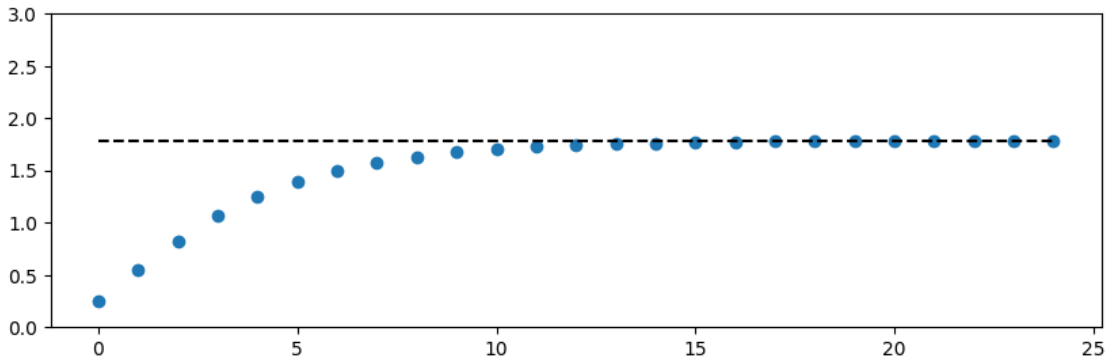
1.1.2. Последовательные аппроксимации

Найдем решение задачи при помощи последовательных аппроксимаций. Напишем функцию, которая будет принимать на вход начальное значение и другую функцию, после чего будет проводить заданное число последовательных итераций.

[8]:
```
def iterates(k_0, f, params, n=25):
    k = k_0
    res = []

    for t in range(n):
        res.append(k)
        k = f(k, params)

    return res
```

Скормим ей функцию g (вот зачем мы ее писали).

[9]:
```
k_series = iterates(0.25, g, params)
k_star = exact_fixed_point(params)

fig, (ax) = plt.subplots()
ax.plot(k_series, "o")
ax.plot([k_star] * len(k_series), "k--")
ax.set_ylim(0, 3)
plt.show()
```



Посмотрим, к чему мы придем на длинной дистанции.

[10]:
```
k_approx = iterates(0.25, g, params, n=10_000)[-1]
k_approx
```

[10]:
```
1.7846741842265788
```

Это близко к истинному значению k^*.

[11]:
```
np.isclose(k_approx, k_star)
```

[11]:
```
np.True_
```

1.2. Метод Ньютона

Задачу поиска неподвижной очки, описанную выше, также можно решить при помощи метода Ньютона. В этом методе мы начинаем с начального предположения x_0, которое уточняется при помощи поиска неподвижной точки касательной в x_0.

Начнем с разложения по Тейлору функции g

$$\hat{g}(x) \approx g(x_0) + g'(x_0)(x - x_0)$$

Пусть у нас $x = x_{t+1}$ и $x_0 = x_t$, тогда (с точностью до слагаемых более высоких порядков)

$$x_{t+1} = g(x_t) + g'(x_t)(x_{t+1} - x_t) =: q(x_t)$$

Раскроем скобки и перегруппируем слагаемые:

$$\begin{aligned} x_{t+1} &= g(x_t) + g'(x_t)(x_{t+1} - x_t) \\ x_{t+1} &= g(x_t) + g'(x_t)x_{t+1} - g'(x_t)x_t \\ x_{t+1}[1 - g'(x_t)] &= g(x_t) + g'(x_t)x_t \\ x_{t+1} &= \frac{g(x_t) + g'(x_t)x_t}{1 - g'(x_t)} \end{aligned}$$

Решим описанную выше задачу методом Ньютона. Определим функцию, возвращающую производную g

$$g'(k) = \alpha s A k^{\alpha - 1} + (1 - \delta)$$

[12]:
```
def Dg(k, params):
    A, s, alpha, delta = params
    return alpha * A * s * k ** (alpha - 1) + (1 - delta)
```

И функцию q.

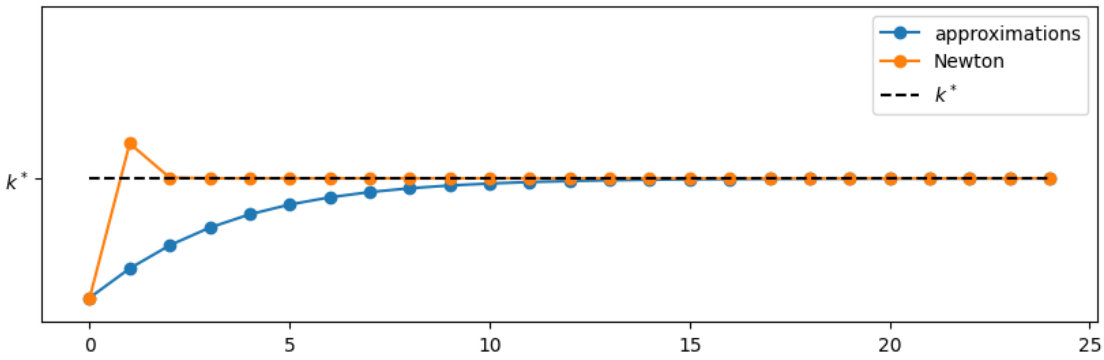
[13]:
```
def q(k, params):
    return (g(k, params) - Dg(k, params) * k) / (1 - Dg(k,␣
  ↪params))
```

Напишем функцию для сравнения траекторий, полученных последовательными аппроксимациями и методом Ньютона.

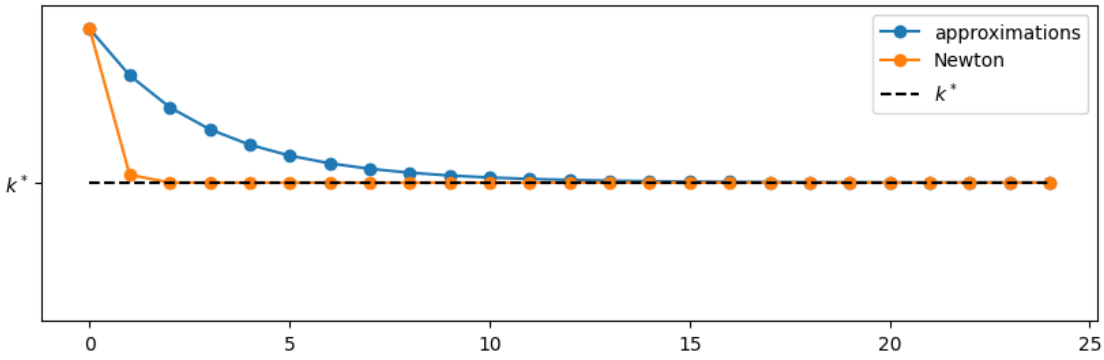
[14]:
```
def plot_paths(params, k_0=0.8, k_lim=(0.6, 3.2)):
    k_series = iterates(k_0, g, params)
    n_series = iterates(k_0, q, params)
    k_star = exact_fixed_point(params)

    k_lim = (
        min(k_lim[0], min(k_series) - 0.2, min(n_series) -␣
  ↪0.2),
        max(k_lim[1], max(k_series) + 0.2, max(n_series) +␣
  ↪0.2),
    )

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

```
    ax.plot(k_series, "-o", label="approximations")
    ax.plot(n_series, "-o", label="Newton")
    ax.plot([k_star] * len(k_series), "k--", label="$k^*$")
    ax.set_ylim(*k_lim)
    ax.set_yticks((k_star,))
    ax.set_yticklabels(("$k^*$",))
    ax.legend()
    plt.show()
```

[15]:
```
plot_paths(params, 0.8)
```



[16]:
```
plot_paths(params, 3.1)
```



Как можно увидеть на графиках, метод Ньютона сходится гораздо быстрее.

1.2.1. Решение уравнений

Метод Ньютона чаще ассоциируется с поиском корней уравнения $f(x) = 0$.

Представим себе, что у нас есть **гладкая** функция $f(x)$, и мы хотим найти такое x, что $f(x) = 0$. Пусть у нас есть предположение x_0, которое мы хотим уточнить, получив x_1.

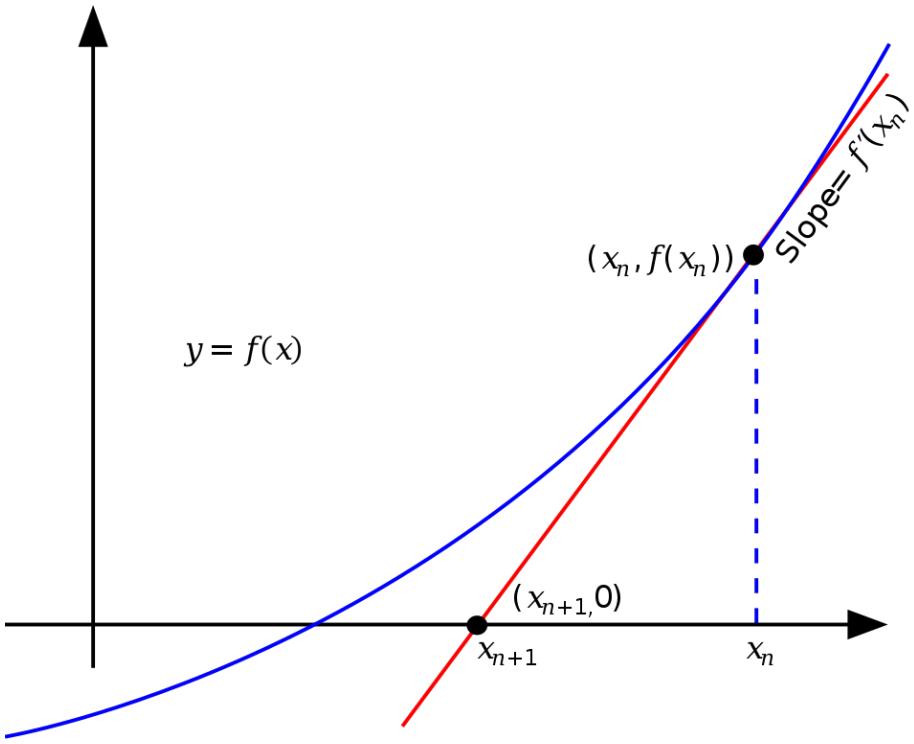
Опять таки, рассмотрим разложение по Тейлору функции $f(x)$ в окрестности x_0:

$$\hat{f}(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

Пусть также $\hat{f}(x_1) = 0$ (то есть мы «попали»), тогда

$$0 = \hat{f}(x_1) = f(x_0) + f'(x_0)(x_1 - x_0)$$
$$-f(x_0) + f'(x_0)x_0 = f'(x_0)x_1$$
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Последнее равенство дает нам рекурсивное соотношение, позволяющее пошагово приблизиться к искомому корню.



Original: Olegalexandrov Vector: Pbroks13, Public domain, via Wikimedia Commons

[17]:
```python
def newton(f, df, x_0, tol=1e-7, iter=100_000):
    x = x_0

    err = tol + 1
    n = 0

    while err > tol:
        n += 1
        if n > iter:
            raise Exception(f"Не достигнута сходимость за {iter} шагов")
        y = x - f(x) / df(x)
        err = np.abs(y - x)
        x = y
        print(f"шаг {n}, ошибка = {err}")
```

```
    return x
```

Пример 1 Если представить, что $f(x) = g(x) - x$, то понятно, что задача поиска неподвижной точки в модели Солоу сводится к поиску корня какого-то уравнения.

[18]:
```
f = lambda x: g(x, params) - x
df = lambda x: Dg(x, params) - 1
```

[19]:
```
np.isclose(newton(f, df, 3.2), k_star)
```

```
шаг 1, ошибка = 1.3409570835209281
шаг 2, ошибка = 0.07393024029293849
шаг 3, ошибка = 0.0004384758047146775
шаг 4, ошибка = 1.6154841020465938e-08
```

[19]:
```
np.True_
```

В пакете `SciPy` есть встроенный метод `newton`...

[20]:
```
sco.newton(f, 3.2)
```

[20]:
```
np.float64(1.7846741842265785)
```

Пример 2 Рассмотрим уравнение $x^2 - 1 = 0$ и попробуем найти его корень (в данном случае один из) при начальном предположении $x_0 = 10$. Очевидно, что $\frac{\mathrm{d}}{\mathrm{d}x}(x^2 - 1) = 2x$.

[21]:
```
f = lambda x: x**2 - 1
df = lambda x: 2 * x
```

[22]:
```
newton(f, df, 10)
```

```
шаг 1, ошибка = 4.95
шаг 2, ошибка = 2.425990099009901
шаг 3, ошибка = 1.1214568889914178
шаг 4, ошибка = 0.4185095446959888
шаг 5, ошибка = 0.08078561620663183
шаг 6, ошибка = 0.0032525615317913203
шаг 7, ошибка = 5.289550279563571e-06
шаг 8, ошибка = 1.3989698288696673e-11
```

[22]:
```
1.0
```

1.2.2. Автоматическое дифференцирование

Разумеется, считать производные руками не всегда хочется и не всегда возможно (за разумное время). Воспользуемся пакетом `autograd`.

ВНИМАНИЕ! `autograd` — птица гордая и не хочет работать в целыми числами, поэтому x_0 надо передавать как число `float`!

ВНИМАНИЕ Есть еще более продвинутый пакет, `jax`, который еще более гордый и крайне жесткий в плане типов параметров.

[23]:
```
def newton2(f, x_0: float, tol=1e-7, iter=100_000):
    x = x_0

    err = tol + 1
    n = 0

    df = autograd.grad(f, 0)

    while err > tol:
        n += 1
        if n > iter:
            raise Exception(f"Не достигнута сходимость за
↪{iter} шагов")
        y = x - f(x) / df(x)
        err = np.abs(y - x)
        x = y
        print(f"шаг {n}, ошибка = {err}")

    return x
```

[24]:
```
f = lambda x: x**2 - 1
```

[25]:
```
newton2(f, 10.0)
```

```
шаг 1, ошибка = 4.95
шаг 2, ошибка = 2.425990099009901
шаг 3, ошибка = 1.1214568889914178
шаг 4, ошибка = 0.4185095446959888
шаг 5, ошибка = 0.08078561620663183
шаг 6, ошибка = 0.0032525615317913203
шаг 7, ошибка = 5.289550279563571e-06
шаг 8, ошибка = 1.3989698288696673e-11
```

[25]:
```
np.float64(1.0)
```

Разумеется, мы не открыли Америку, и кто-то уже за нас все придумал. Воспользуемся методом `root` из `scipy.optimize`.

[26]:
```
sco.root(f, 10.0, method="hybr")
```

[26]:
```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 0.000e+00]
       x: [ 1.000e+00]
  method: hybr
    nfev: 15
    fjac: [[-1.000e+00]]
```

```
     r: [-2.000e+00]
   qtf: [-1.128e-12]
```

1.3. Многомерный случай

Рассмотрим следующую задачу. Мы имеем два продукта — 0 и 1 — спрос и предложение на которые определяются выражениями

$$q_i^s(p) = b_i\sqrt{p_i}$$
$$q_i^d(p) = \exp(-(a_{i0}p_0 + a_{i1}p_1)) + c_i$$

В равновесии избыточный спрос

$$e_i(p) = q_i^d(p) - q_i^s(p)$$

$$e(p) = \exp(Ap) + c - b\sqrt{p}$$

должен быть равен 0 для всех товаров.

[27]:
```
def e(p, A, b, c):
    return np.exp(-A @ p) + c - b * np.sqrt(p)
```

$$A = \begin{bmatrix} 0.5 & 0.4 \\ 0.8 & 0.2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

[28]:
```
A = np.array([[0.5, 0.4], [0.8, 0.2]])
b = np.ones(2)
c = np.ones(2)
```

Посмотрим на избыточный спрос при $p = (1, 0.5)$.

[29]:
```
e((1.0, 0.5), A, b, c)
```

[29]:
```
array([0.4965853 , 0.69946288])
```

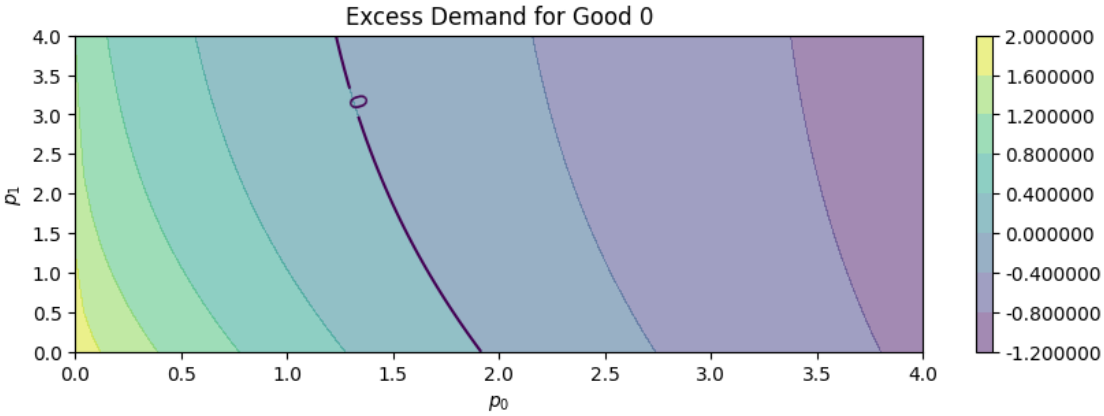
Построим графики.

[30]:
```
def plot_excess_demand(ax, good=0, grid_size=100,␣
  ↪grid_max=4, surface=True):
    # Create a 100x100 grid
    p_grid = np.linspace(0, grid_max, grid_size)
    z = np.empty((100, 100))

    for i, p_1 in enumerate(p_grid):
        for j, p_2 in enumerate(p_grid):
            z[i, j] = e((p_1, p_2), A, b, c)[good]

    if surface:
        cs1 = ax.contourf(p_grid, p_grid, z.T, alpha=0.5)
        plt.colorbar(cs1, ax=ax, format="%.6f")

    ctr1 = ax.contour(p_grid, p_grid, z.T, levels=[0.0])
```

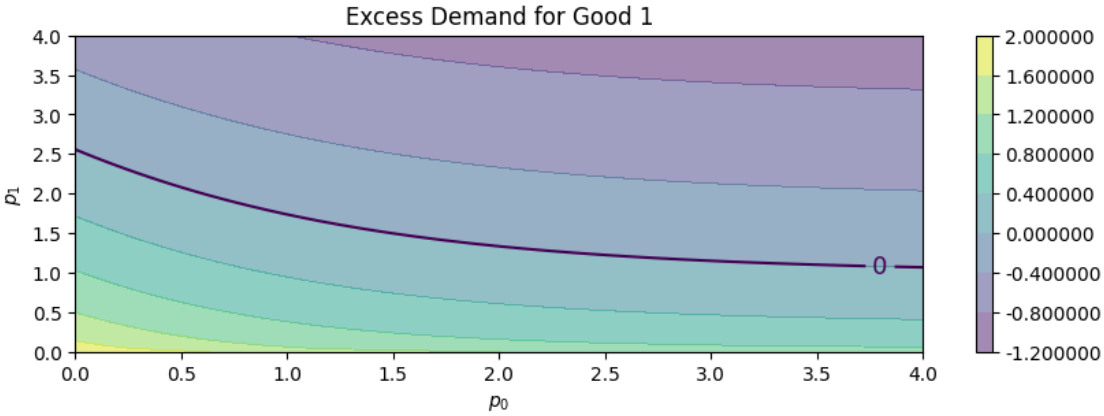
```
    ax.set_xlabel("$p_0$")
    ax.set_ylabel("$p_1$")
    ax.set_title(f"Excess Demand for Good {good}")
    plt.clabel(ctr1, inline=1, fontsize=13)
```

Обратите внимание, что данная функция принимает на вход **ось** (Axis). Другими словами, создавать рисунок нужно до вызова функции. Концечно, проще было бы создавать рисунок внутри функции, но, как мы увидим ниже, это позволит нам несколькими вызовами функции разместить несколько линий на одном графике.

[31]:
```
fig, ax = plt.subplots()
plot_excess_demand(ax, good=0)
plt.show()
```

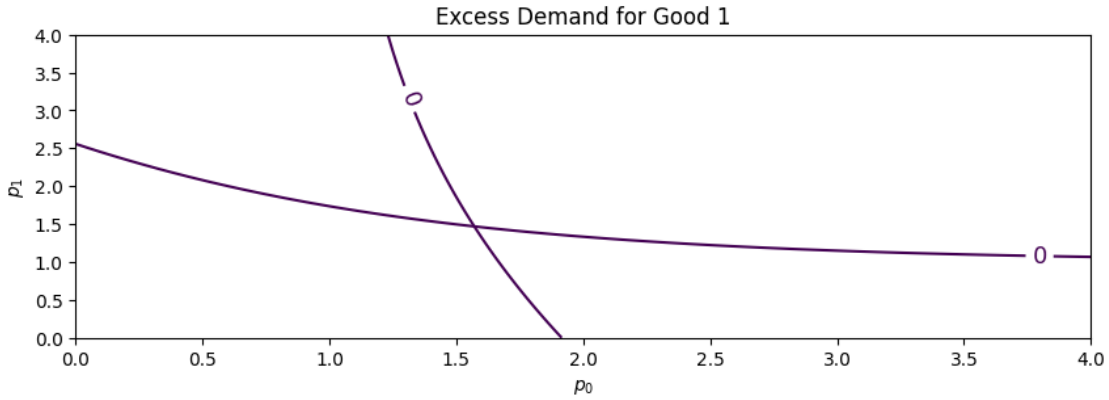


[32]:
```
fig, ax = plt.subplots()
plot_excess_demand(ax, good=1)
plt.show()
```



[33]:
```
fig, ax = plt.subplots()
for good in (0, 1):
    plot_excess_demand(ax, good=good, surface=False)
```

```
plt.show()
```



Найдем корни при помощи **root**.

[34]:
```
p_init = np.ones(2)
```

[35]:
```
%%time
solution = sco.root(lambda p: e(p, A, b, c), p_init,␣
↪method="hybr")
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

[36]:
```
solution.x, np.max(np.abs(e(solution.x, A, b, c)))
```

[36]:
```
(array([1.57080182, 1.46928838]), np.float64(2.
↪0383694732117874e-13))
```

Для ускорения сходимости в **root** при помощи аргумента **jac** можно передать функцию для расчета **якобиана**

$$J(p) = \begin{bmatrix} \frac{\partial e_0}{\partial p_0}(p) & \frac{\partial e_0}{\partial p_1}(p) \\ \frac{\partial e_1}{\partial p_0}(p) & \frac{\partial e_1}{\partial p_1}(p) \end{bmatrix}$$

[37]:
```
def jacobian_e(p, A, b, c):
    p_0, p_1 = p
    a_00, a_01 = A[0, :]
    a_10, a_11 = A[1, :]
    j_00 = -a_00 * np.exp(-a_00 * p_0 - a_01 * p_1) - (b[0]␣
↪/ 2) * p_0 ** (-1 / 2)
    j_01 = -a_01 * np.exp(-a_00 * p_0 - a_01 * p_1)
    j_10 = -a_10 * np.exp(-a_10 * p_0 - a_11 * p_1)
    j_11 = -a_11 * np.exp(-a_10 * p_0 - a_11 * p_1) - (b[1]␣
↪/ 2) * p_1 ** (-1 / 2)
    return np.array([[j_00, j_01], [j_10, j_11]])
```

[38]:
```
%%time
solution = sco.root(
    lambda p: e(p, A, b, c),
    p_init,
    jac=lambda p: jacobian_e(p, A, b, c),
    method="hybr",
)
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

[39]:
```
solution.x, np.max(np.abs(e(solution.x, A, b, c)))
```

[39]:
```
(array([1.57080182, 1.46928838]), np.float64(2.
  ↪0383694732117874e-13))
```

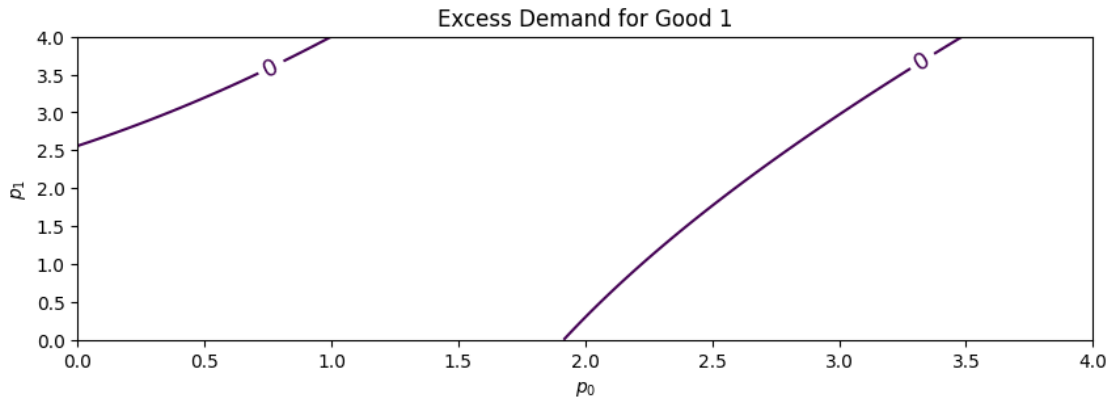
Пример с отсутствием решения у задачи Пытливые умы спросили меня: что будет при отсутствии точки пересечения? Посмотрим.

Для примера умножим элементы на побочной диагонали матрицы A на -1...

[40]:
```
A = np.array([[0.5, -0.4], [-0.8, 0.2]])
b = np.ones(2)
c = np.ones(2)
```

...и посмотрим на графики и попытки решения.

[41]:
```
fig, ax = plt.subplots()
for good in (0, 1):
    plot_excess_demand(ax, good=good, surface=False)
plt.show()
```



[42]:
```
%%time
solution = sco.root(lambda p: e(p, A, b, c), p_init,␣
  ↪method="hybr")
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

```
C:\Users\zyama\AppData\Local\Temp\ipykernel_5256\1012735744.py:2:
RuntimeWarning: invalid value encountered in sqrt
  return np.exp(-A @ p) + c - b * np.sqrt(p)
```

[43]:
```
solution
```

[43]:
```
 message: The iteration is not making good progress, as␣
  ↪measured by the
          improvement from the last ten iterations.
 success: False
  status: 5
     fun: [ 9.048e-01  1.822e+00]
       x: [ 1.000e+00  1.000e+00]
  method: hybr
    nfev: 17
    fjac: [[       nan        nan]
           [       nan        nan]]
       r: [       nan        nan        nan]
     qtf: [       nan        nan]
```

1.3.1. Многомерный метод Ньютона

Многомерный вариант метода Ньютона выглядит следующим образом:

$$x_{t+1} = x_t - J_f^{-1}(x_t) f(x_t)$$

где $J_f(x)$ — якобиан f, вычисленный в точке x.

В пакете **autograd** есть метод для автоматического расчета якобиана, называемый, как это ни было бы странно, **jacobian**.

1.4. Альтернативы

1.4.1. Полиномиальные уравнения

Если вы хотите найти корни полинома, то есть более простой способ найти его корни. Покажем на примере $x^2 + 2x + 1 = 0$.

[44]:
```
pp = np.poly1d([1, 0, -4])
pp.roots
```

[44]:
```
array([-2.,  2.])
```

Полиномы в этом представлении можно интегрировать, дифференциировать и даже сокращать.

[45]:
```
pp.deriv(), pp.integ(), np.polydiv(pp, [1, -2])
```

[45]:
```
(poly1d([2, 0]),
 poly1d([ 0.33333333,  0.        , -4.        ,  0.        ]),
```

```
(poly1d([1., 2.]), poly1d([0.])))
```

1.4.2. Символьные вычисления

Пакет `sympy` предназначен для символьных вычислений. То есть это аналог Wolfram Mathematica, Maxima, Maple и иже с ними.

Пакет оперирует «символами», которые привязываются к именам пайтона, но не тождественны им. Для удобства есть подмодуль `abc`, в котором уже определены символы для латинских и греческих букв, за исключением зарезервированных: например, `E` — это число e, а `I` — мнимая единица.

[46]:
```
import sympy
from sympy.abc import x
```

Опишем уравнение при помощи класса `Eq`: первый аргумент конструктора — это LHS, второй — RHS.

[47]:
```
eq = sympy.Eq(x**2 - 2 * x + 1, 0)
```

Решим это уравнение относительно `x`. Получим список корней, в нашем случае корень один.

[48]:
```
sympy.solve(eq, x)
```

[48]:
```
[1]
```

Найдем аналитически k^* из модели Солоу, решив уравнение $g(x) = x$.

[49]:
```
from sympy.abc import a, s, d, A

x = sympy.Symbol("x")
f = sympy.Eq(s * A * x**a + (1 - d) * x, x)
res = sympy.solve(f, x)

res
```

[49]:
```
[(A*s/d)**(-1/(a - 1))]
```

[50]:
```
res[0]
```

[50]:
$$\left(\frac{As}{d}\right)^{-\frac{1}{a-1}}$$