

Problem Set 6

☰ Contents

[Problem 1](#)

[Solution](#)

[Problem 2](#)

[Solution](#)

[Problem 3](#)

[Solution](#)

[Problem 4](#)

[Solution](#)

[Problem 5](#)

[Solution](#)

[Problem 6](#)

[Solution](#)

(a)

(b)

(c)

(d)

Given the data:

x_i	0	1	2	3
-------	---	---	---	---

y_i	1	3	3	4
-------	---	---	---	---

(a) Find the best linear fit by solving the 2×2 normal system by hand $A^T A u = A^T b$, with $A = [x^T \ 1]$ and $b = y^T$. Plot the data and the fit.

(b) Calculate the Moore-Penrose pseudo-inverse A^+ of A directly from its definition.

(c) Write down the SVD of A^+ .

(d) What is the error e of the approximation and its 2-norm?

Solution

(a)

Firstly, let's construct the normal equation:

$$x = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 5 \\ 1 \\ 3 \\ 3 \\ 4 \\ 6 \end{bmatrix},$$

$$y = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 5 & 1 \end{bmatrix}, \quad A^T A = \begin{bmatrix} 14 & 6 \\ 6 & 4 \end{bmatrix}$$

Now we can write that

$$\begin{bmatrix} 39 & 11 \\ 11 & 5 \end{bmatrix} u = \begin{bmatrix} 0 & 1 & 2 & 3 & 5 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} y = \begin{bmatrix} 0 & 1 & 2 & 3 & 5 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 3 \\ 3 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 51 \\ 17 \end{bmatrix}.$$

Then we solve the system of linear equations for u (a and d):

$$\begin{cases} 39a + 11d = 51 \\ 11a + 5d = 17 \end{cases}$$

We see that $\det(A^T A) = 74 \neq 0$, thus, we have a single solution:

$$u = \begin{bmatrix} 34/37 \\ 51/37 \end{bmatrix},$$

and the best linear fit would be

$$\hat{y} = au = [x \ 1] \begin{bmatrix} 34/37 \\ 51/37 \end{bmatrix} = \frac{1}{37}(34x + 51)$$

```

import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'svg'

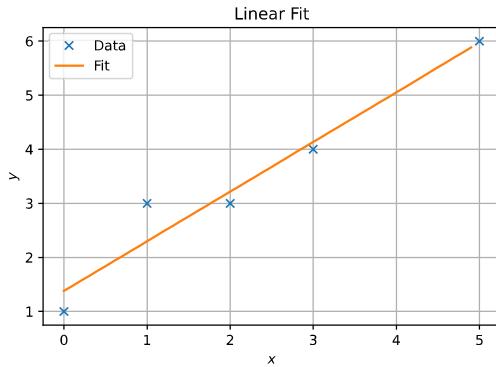
a = 34
d = 51

x1 = [0, 1, 2, 3, 5]
y1 = [1, 3, 3, 4, 6]

x = np.arange(0, 5, 0.1)
y = 1/37*(a*x + d)

plt.plot(x1, y1, lw=0, marker='x')
plt.plot(x, y)
plt.legend(['Data', 'Fit'])
plt.title('Linear Fit')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.grid()

```



(b)

The definition of Moore-Penrose pseudo-inverse is as follows:

$$A^+ = (A^T A)^{-1} A^T$$

Let's perform some calculations:

```

import sympy as sp
import numpy as np

x = sp.Matrix([0, 1, 2, 3, 5])
y = sp.Matrix([1, 3, 3, 4, 6])

ones = sp.ones(5, 1)
A = x.row_join(ones)

Aplus = (A.T * A).inv() * A.T
Aplus_disp = sp.MathMul(sp.gcd(tuple(Aplus)), Aplus / sp.gcd(tuple(Aplus)), evaluate=False)
Aplus_disp

```

$$\left(\frac{1}{74}\right) \begin{bmatrix} -11 & -6 & -1 & 4 & 14 \\ 39 & 28 & 17 & 6 & -16 \end{bmatrix}$$

This is our Moore-Penrose pseudo-inverse A^+ .

(c)

Now let's find SVD of A^+ :

```
svd = Aplus.singular_value_decomposition()
```

Here we get U :

```
svd[0].evalf()
```

$$\begin{bmatrix} 0.959054262581672 & -0.283222388634664 \\ 0.283222388634664 & 0.959054262581672 \end{bmatrix}$$

Σ :

```
svd[1].evalf()
```

$$\begin{bmatrix} 0.153848963697702 & 0 & 0 \\ 0 & 0.755595851605696 & 0 \end{bmatrix}$$

V :

```
svd[2].evalf()
```

$$\begin{bmatrix} 0.0435734709874309 & 0.724657422271472 \\ 0.191122975415485 & 0.510655760337264 \\ 0.33867247984354 & 0.296654098403056 \\ 0.486221984271594 & 0.0826524364688476 \\ 0.781320993127703 & -0.345350887399569 \end{bmatrix}$$

(d)

To find the error vector, we simply subtract the fit vector from the data vector:

```
a, d = sp.symbols('a, d')
u = sp.Matrix([a, d])
subs = sp.solve(A.T*A*u - A.T*y)
yhat = A*u.subs(subs)
e = y - yhat
e_disp = sp.MatMul(sp.gcd(tuple(e)), e / sp.gcd(tuple(e)))
e_disp
```

$$\left(\frac{1}{37}\right) \begin{bmatrix} -14 \\ 26 \\ -8 \\ -5 \\ 1 \end{bmatrix}$$

And $\|e\|_2$ is:

```
e.norm()
```

$$\frac{\sqrt{962}}{37}$$

Problem 2

Find the best plane in \mathbb{R}^3 , in the least-square sense, through the data given in the table:

x_i	1	1	2	3	5
y_i	5	3	4	10	7
z_i	2	1	2	5	5

What is the error vector and its norm?

Solution

The plane equation looks as follows:

$$ax + by + cz + d = 0$$

In our case, this equation will define the form of the fit:

$$ax + by + d = -cz$$

Or, in matrix form:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ d \end{bmatrix} = -c[\hat{z}],$$

where x, y are the argument vectors and \hat{z} is a fit vector.

The matrix A and the vector u should look like this:

$$A = \begin{bmatrix} 1 & 5 & 1 \\ 1 & 3 & 1 \\ 2 & 4 & 1 \\ 3 & 10 & 1 \\ 5 & 7 & 1 \end{bmatrix}, \quad u = \begin{bmatrix} a \\ b \\ d \end{bmatrix}$$

and the normal equation takes the form of:

$$A^T A u = -c A^T z$$

Thus, we can find the pseudo-inverse $(A^T A)^{-1} A^T$ and multiply from right side both sides of the equation to instantly find u .

Note

We should move the coefficient c to the left and put it into the vector u :

$$A^T A \hat{u} = A^T z;$$

$$u = \begin{bmatrix} a/c \\ b/c \\ 1/c \end{bmatrix}.$$

Then we name the new coefficients $\hat{a}, \hat{b}, \hat{d} = -a/c, -b/c, -d/c$ respectively and call it a day!

```
x = sp.Matrix([1, 1, 2, 3, 5])
y = sp.Matrix([5, 3, 4, 10, 7])
z = sp.Matrix([2, 1, 2, 5, 5])

ones = sp.ones(5, 1)

A = x.row_join(y.row_join(ones))

A.pinv() #(A.T * A)^(-1) * A^T

u = A.pinv()*z
u = (A.T*A).inv()*A.T*z
u
```

$$\begin{bmatrix} \frac{611}{1075} \\ \frac{437}{1075} \\ \frac{1}{1075} \\ -\frac{776}{1075} \end{bmatrix}$$

Although we omitted the coefficient c , the result beautifully transforms to

$$611x + 437y - 776 = 1075z$$

Let's plot the result:

```
%config InlineBackend.figure_format = 'svg'

x = list(x)
y = list(y)
z = list(z)

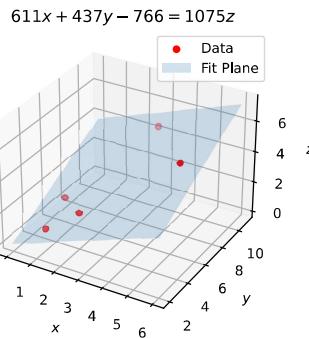
X = np.linspace(0, 6, 10)
Y = np.linspace(2, 11, 10)

X, Y = np.meshgrid(X, Y)

u = np.array(u)

Z = X*u[0] + Y*u[1] + u[2]
Z = np.array(Z, dtype=np.float64)

ax = plt.subplot(projection='3d')
ax.scatter(x, y, z, color='red', label='Data')
s = ax.plot_surface(X, Y, Z, alpha=0.2, label='Fit Plane')
s._facecolors3d = s._facecolor3d
s._edgecolors3d = s._edgecolor3d
ax.legend()
ax.set_title('$611x + 437y - 766 = 1075z$')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
```



⚠ Warning

Nice!

The error vector is found by subtracting the vector \hat{z} from z :

$$e = A u - z =$$

```

x = sp.Matrix(x)
y = sp.Matrix(y)
z = sp.Matrix(z)
u = sp.Matrix(u)
A = x.row_join(y.row_join(sp.ones(5, 1)))

zhat = A*u
e = z-zhat
e_disp = Sp.MatMul(sp.gcd(list(e)), e/sp.gcd(list(e)))
e_disp

```

$$\left(\frac{1}{1075} \right) \begin{bmatrix} 130 \\ -71 \\ -44 \\ -52 \\ 37 \end{bmatrix}$$

And the error norm is

e.norm()

$\frac{\sqrt{1118}}{215}$

or

```
e.norm().evalf()
```

0.155518637184191

Problem 3

Determine the dominant nodes in the function f of space x and time t :

$$f(x,t) = e^{-x^2} \sin(x + 3t) \cos(x - t).$$

considering the interval $x \in [-5, 5], t \in [0, 10]$.

- (a) Plot the singular values in uniform as well as semilog scales.
 - (b) Plot the solution in the $x - t$ plane over the given interval.
 - (c) How much "energy" of the solution is contained in mode 1 and in modes 1 + 2?
 - (d) Split the first two columns of U and V in the SVD of matrix F obtained by calculating $f(x, t)$ over a grid with 100 points in x and 50 points in t . Explain their meaning.

Solution

(a)

Let's perform the discretization and construct the matrix F :

```
X = np.linspace(-5, 5, 100)
T = np.linspace(0, 10, 50)

X, T = np.meshgrid(X, T)

f = lambda x, t: np.exp(-np.power(x, 2)) * np.sin(x + 3*t) * np.cos(x - t)

F = f(X, T)
np_svd = np.linalg.svd(F)

print('Singular Values:')
print(np_svd[1])
```

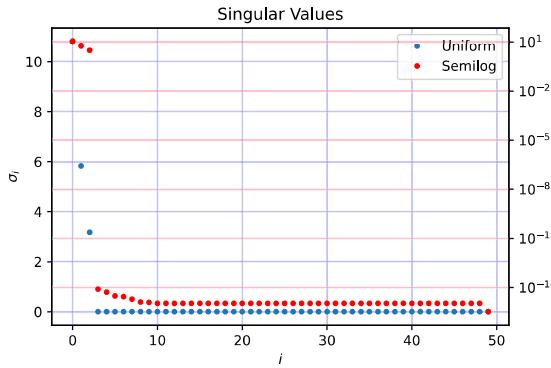
```

fig, ax = plt.subplots(1, 1)
uniform = ax.plot(np_svd[1], label='Uniform', lw=0, marker='.')
ax2 = ax.twinx()

ax2.set_yscale('log')
semilog = ax2.plot(np_svd[1], color='red', label='Semilog', lw=0, marker='.')
lines = uniform + semilog
labels = [l.get_label() for l in lines]
ax.legend(lines, labels)

plt.title('Singular Values')
ax.set_xlabel('$i$')
ax.set_ylabel('$\sigma_i$')
ax.grid(color='blue', alpha = 0.25)
ax2.grid(color='red', alpha = 0.25)

```



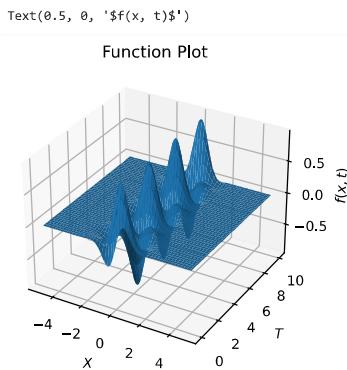
(b)

If by "solution" we understand the function itself, then the plot is down below:

```

ax3 = plt.subplot(projection='3d')
ax3.plot_surface(X, T, F)
ax3.set_title('Function Plot')
ax3.set_xlabel('$x$')
ax3.set_ylabel('$t$')
ax3.set_zlabel('$f(x, t)$')

```



(c)

By the "energy" we understand the term $E_i = \frac{\sigma_i}{\sum_{k=1}^K \sigma_k}$. By the "energy" of k we understand the "energy" of k^{th} expansion: $E_k = \frac{\sum_{i=1}^k \sigma_i}{\sum_{i=1}^K \sigma_i}$.

So,

```

sigmas = np_svd[1]

e1 = np.max(sigmas) / np.sum(sigmas)
e12 = np.sum(sigmas[0:2]) / np.sum(sigmas) #from numpy the sigmas come sorted already

print('E1: {}'.format(e1))
print('E12: {}'.format(e12))

```

```

E1: 0.5455778666776585
E12: 0.8397746647151946

```

(d)

When we SVD the matrix F , obtained as a discretized calculation of $f(x, t)$ over a meshgrid (X, T) , we get the U and V matrices containing the left and right singular vectors. These vectors are the components of the function factorization corresponding to their singular values. The first two columns of U and V will correspond to the first two largest singular values, thus, these "vectors" will carry the

most "energetic" part of the function. In other words, when we reduce the information, such a reduction will preserve the most valuable information.

```

U, S, V = np_svd
U_red = U[:, 0:2]
V_red = V.T[:, 0:2]
S_red = S[0:2]

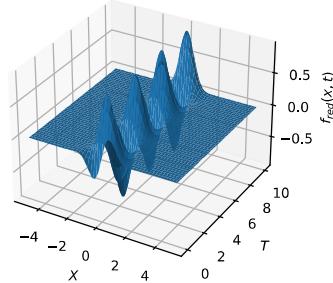
F_red = np.dot(U_red, np.dot(np.diag(S_red), V_red.T))

ax4 = plt.subplot(projection='3d')
ax4.plot_surface(X, T, F_red)
ax4.set_title('Reduced Function Plot')
ax4.set_xlabel('$x$')
ax4.set_ylabel('$t$')
ax4.set_zlabel('$f_{red}(x, t)$')

```

Text(0.5, 0, '\$f_{red}(x, t)\$')

Reduced Function Plot



As we can see, we preserved the most valuable information from the function as it barely differs from the plot of the "full" function.

Problem 4

To find a root of $f(x) = 0$, Newton's method tells to start with some initial guess x_0 and then to iterate following the scheme:
 $x_{n+1} = x_n - f(x_n)/f'(x_n)$.

(a) Use this method to find the root $x = 1$ of $f(x) = x^2 - 1$.

(b) What is the range of initial conditions x_0 that give convergence to $x = 1$?

(c) How fast do the iterations converge? Plot the error $e_n = |x_n - 1|$ as a function of n (maybe, in log scale).

Solution

(a, b, c)

Okay...

```

x = sp.symbols('x')
f = sp.Function('f')

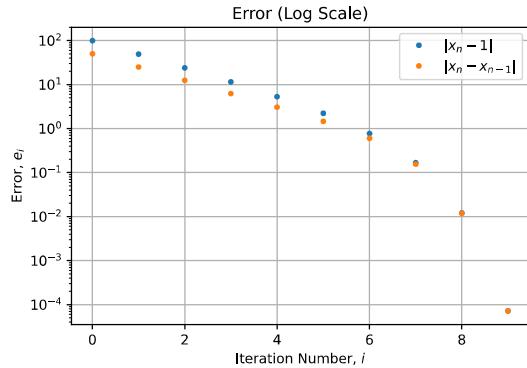
f = f(x)
f = x**2 - 1
f_dif = f.diff(x)

n = 50 #1000 steps
errors = np.zeros(n)
deltax = np.zeros(n)
x_prev = 100 #x_0
for i in range(n):
    errors[i] = np.abs(x_prev - 1) #we know the root is x_r = 1
    xn = x_prev - f.subs({x: x_prev}).evalf() / f_dif.subs({x: x_prev}).evalf()
    deltax[i] = np.abs((xn - x_prev))
    if (x_prev == xn):
        break
    else:
        x_prev = xn
    max_i = i

ax5 = plt.subplot()
ax5.set_yscale('log')
#ax5.set_xscale('log')
ax5.plot(np.arange(max_i), errors[:max_i], marker='.', lw = 0, label='$|x_n - 1|$')
ax5.set_title('Error (Log Scale)')
ax5.set_xlabel('Iteration Number, $i$')
ax5.set_ylabel('Error, $e_i$')
ax5.plot(np.arange(max_i), deltax[:max_i], marker='.', lw = 0, label='$|x_n - x_{n-1}|$')
ax5.grid()
ax5.legend()

```

<matplotlib.legend.Legend at 0x1e17447f908>



The root of the equation we obtain through iteration method is:

`xn`

1.0

The error is so low that python interprets this number as a strict $x = 1$, which is nice.

This function has continuous derivatives f'_x and f''_{xx} . The function definition scope is $x \in \mathbb{R}$, so the method will give the convergence to any of the roots at any arbitrary $x_0 \in \mathbb{R} \setminus 0$. For it to converge to $x = 1$, we have to start from $x > 0$. Oppositely, if we want to converge to $x' = -1$, we have to start from $x_0 < 0$. The point $x_0 = 0$ leads to $x_1 = \infty$, it is a critical point.

For more detailed information on convergence conditions we should refer to Kantorovich theorem.

The method gives quadratic convergence. However, this may vary from function to function, as some features change (e.g. $f''(x_{root}) = 0$ or \exists).

Note

When we don't know the real root, it is better to plot the error $|x_n - x_{n-1}|$ to see if the found value converges to some finite value. This is plotted on the graph as an orange line.

Problem 5

Now apply the same Newton iterations as in the previous problem to the equation $f(x) = x^2 + 1 = 0$. Clearly, this equation has no real roots.

- (a) The question is: What do the iterations do? Do they converge to anything?
- (b) How does the behaviour of the iterations depend on the initial point x_0 ?
- (c) What if you start the iterations in the complex plane? Can you get the convergence to the actual roots $\pm i$ of the equation? What are the domains of attraction of the roots?

Solution

```

x = sp.symbols('x')
f = sp.Function('f')
f = x ** 2 + 1
f_dif = f.diff(x)
x0 = 200
n = 1000

def iter(x_prev, f, n):
    iters_ = np.zeros(n, dtype=np.object)
    for i in range(n):
        max_i = i
        iters_[i] = x_prev
        subs = {x: x_prev}
        xn = x_prev - f.subs(subs).evalf() / f_dif.subs(subs).evalf()
        if (xn.is_infinite):
            print('Divergence at step {} with initial x = {}'.format(i, iters_[0]))
            break
        else:
            x_prev = xn.evalf()
    return x_prev, max_i, iters_

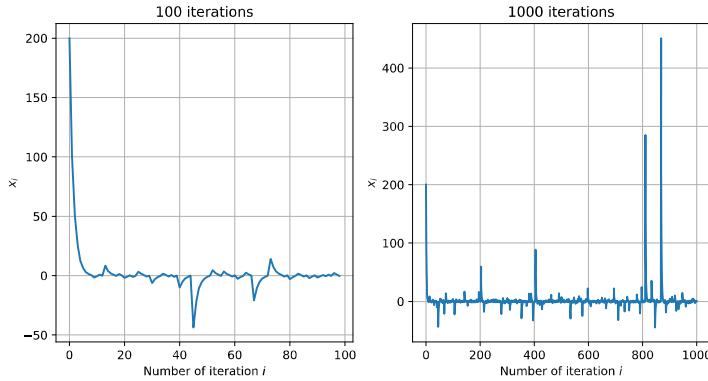
fig, (ax1, ax2) = plt.subplots(1, 2)

x_r, max_i, iters = iter(x0, f, 100)
ax1.plot(np.arange(max_i), iters[:max_i])
ax1.set_title('100 iterations')
ax1.set_xlabel('Number of iteration $i$')
ax1.set_ylabel('$x_i$')
ax1.grid()

x_r, max_i, iters = iter(x0, f, 1000)
ax2.plot(np.arange(max_i), iters[:max_i])
ax2.set_title('1000 iterations')
ax2.set_xlabel('Number of iteration $i$')
ax2.set_ylabel('$x_i$')
ax2.grid()

fig.set_size_inches(10, 5)

```



(a)

It is clear that there would be no convergence: due to the form of the graph, we will approach the lowest point of the function $f(0) = 1$, from which the next iteration will shoot to either of two infinities (plus or minus) because the $f'(x)$ term will be almost zero, thus, the fraction will be almost ∞ . So, as we see, the iterations tend to $x = 0$, then shoot, or diverge, then tend back to zero again.

(b)

If we take the initial point $x_0 = 0$, we will instantly get the infinity as the next approximation. As we go further from the global minimum, we will just need more iterations to get close to 0, where the scenario described in (a) will come into play. Let's demonstrate this on plot:

```

n = 100
iters = np.zeros(n)
x0 = np.array([-1000, -500, -50, -20, -5, -1, -0.1, 0])*(-1)
x_r = []
max_i = []
iters = []
for x1 in x0:
    a, b, it = iter(x1, f, n)
    x_r.append(a)
    max_i.append(b)
    iters.append(it)

fig, (ax1, ax2) = plt.subplots(1, 2)
for i in range(len(x0)):
    ax1.plot(np.arange(max_i[i]), iters[i][:max_i[i]], label='{} = {}'.format(x0[i]))

fig.suptitle('Behaviour of the iterations wrt different $x_0$')
ax1.legend()
ax1.set_xlabel('Number of iteration $i$')
ax1.set_ylabel('$x_i$')
ax1.grid()

n = 10
iters = np.zeros(n)
x0 = np.array([-1000, -500, -50, -20, -5, -1, -0.1, 0])*(-1)
x_r = []
max_i = []
iters = []
for x1 in x0:
    a, b, it = iter(x1, f, n)
    x_r.append(a)
    max_i.append(b)
    iters.append(it)

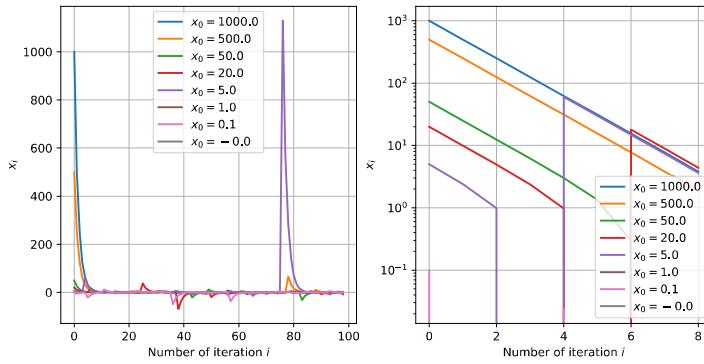
for i in range(len(x0)):
    ax2.plot(np.arange(max_i[i]), iters[i][:max_i[i]], label='{} = {}'.format(x0[i]))

fig.set_size_inches(10, 5)
ax2.set_yscale('log')
ax2.legend(loc=0)
ax2.set_xlabel('Number of iteration $i$')
ax2.set_ylabel('$x_i$')
ax2.grid()

```

Divergence at step 1 with initial $x = 1.0$
Divergence at step 0 with initial $x = -0.0$
Divergence at step 1 with initial $x = 1.0$
Divergence at step 0 with initial $x = -0.0$

Behaviour of the iterations wrt different x_0



(c)

Let us move to the complex plane: $x = a + bi$

```

#x = a + bi
x = sp.symbols('x')
f = x**2 + 1
f_diff = f.diff(x)

a0 = np.arange(-5, 5, 0.05)
b0 = np.arange(-5, 5, 0.05)
a0, b0 = np.meshgrid(a0, b0)
x0 = a0 + b0*1j

def compl_iter(a, b, n):

    def f(x): return x**2 + 1
    def df(x): return 2*x

    x_prev = a + b*1j

    for i in range(n):
        x = x_prev - f(x_prev) / df(x_prev)
        x_prev = x

    #print(np.max(x))

    return x_prev

z = compl_iter(a0, b0, 100)

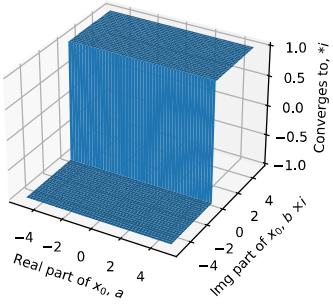
ax = plt.subplot(projection='3d')
ax.plot_surface(a0, b0, np.imag(z))

ax.set_title('Domain of convergence')
ax.set_zlabel('Converges to, $i$')
ax.set_xlabel('Real part of $x_0$, $a$')
ax.set_ylabel('Img part of $x_0$, $b$')

```

Text(0.5, 0, 'Img part of \$x_0\$, \$b\$ \$\times i\$')

Domain of convergence



So the domains of attraction are:

$$\begin{cases} a \in \mathbb{R}, b > 0 \Rightarrow x_{\text{root}} = i, \\ a \in \mathbb{R}, b < 0 \Rightarrow x_{\text{root}} = -i, \\ b = 0 \Rightarrow \text{divergence}, \end{cases}$$

where $a + bi = x_0$.

Problem 6

Consider the function $f = 2x^2 + 2xy + y^2 - x - 2y$.

(a) Find its minimum analytically by representing f as $\frac{1}{2}u^T Au - b^T u$. Plot the function together with its contour levels using, for example, **surf** function in Matlab.

(b) Now find the minimum using the gradient descent. Determine the step τ in the descent method.

(c) Starting with $(x_0, y_0) = (0, 4)$, calculate the first two steps of the gradient explicitly and indicate on a single plot both the positions and the gradient vectors at those positions. Also plot the level curves of f going through these points.

(d) Implement the descent algorithm in Matlab or Python and starting with the same initial condition as in (c) find the minimum within a tolerance of $tol = 10^{-6}$. how many iterations does it take to reach the minimum?

Solution

(a)

$$f = \frac{1}{2}u^T Au - b^T u;$$

$$u = \begin{bmatrix} x \\ y \end{bmatrix}, \quad A = \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Let us differentiate the function:

```

x, y = sp.symbols('x, y')
u = sp.Matrix([x, y])
A = sp.Matrix([[4, 2], [2, 2]])
b = sp.Matrix([1, 2])

s = (u.T*A*u/2 - b.T*u)
ss = s[0].expand()
ss

df = sp.Matrix([ss.diff(x), ss.diff(y)])

```

$$\begin{bmatrix} df/dx \\ df/dy \end{bmatrix} = \begin{bmatrix} 4x + 2y - 1 \\ 2x + 2y - 2 \end{bmatrix}$$

$[-1/2, 3/2]$ is an extremum. Let's define if it is a minimum or maximum:

```

d2fx = df[0].diff(x)
d2fy = df[0].diff(y)
d2fy = df[1].diff(y)

d2f = sp.Matrix([
    [d2fx, d2fy],
    [d2fy, d2fx]
])

d2f

```

[Print to PDF](#)

$$\begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix}$$

This is the matrix of $d^2 f$. If it is positive-definite, then we are at the minimum.

```
d2f.det()
```

12

It obviously is. Moreover, as we can see, the second differential of f is constant and doesn't depend on x and y .

So, our minimum is

$$\begin{bmatrix} x \\ y \end{bmatrix}_{\min} = \begin{bmatrix} -1/2 \\ 3/2 \end{bmatrix}$$

Let's plot-a-lot:

```

X = Y = np.arange(-10, 10, 0.1)
X, Y = np.meshgrid(X, Y)

F = 2*X**2 + 2*X*Y + Y**2 - X - 2*Y

ax1 = plt.subplot(121, projection='3d')
ax2 = plt.subplot(122)

plt.tight_layout()

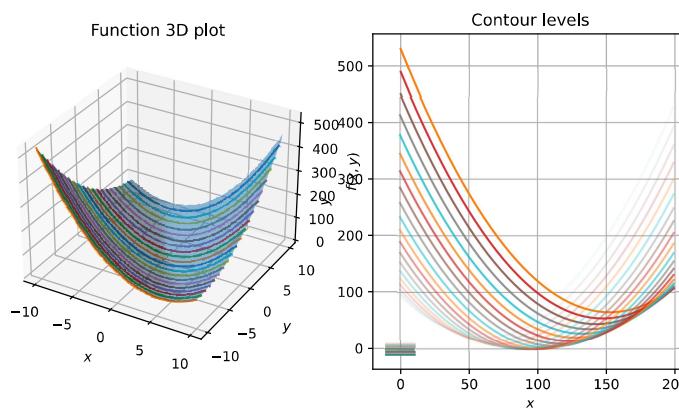
ax1.plot_surface(X, Y, F, alpha = 0.5)

for i in range(0, len(X), 10):
    ax1.plot(X[i], Y[i], F[i], )
    ax2.plot(X[i], Y[i], F[i], alpha=1 - i/len(X))

ax1.set_title('Function 3D plot')
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')
ax1.set_zlabel('$f(x, y)$')

ax2.set_title('Contour levels')
ax2.set_xlabel('$x$')
ax2.set_ylabel('$y$')
ax2.grid()
plt.gcf().set_size_inches(7.5, 4)

```



Yes, I know, that the second plot is not the function level curves, but I wasn't able to do it easily in **Python**. If necessary, I will provide a self-written function for plotting.

(b)

To the code-a-loo!

```
def func(p):
    x = p[0]
    y = p[1]
    #guess what it returns =
    return 2*x**2 + 2*x*y + y**2 - x - 2*y

def grad(p, tau, func):
    #returns the antigradient multiplied by tau
    x = p[0]
    y = p[1]

    dfdx = 4*x + 2*y - 1
    dfdy = 2*x + 2*y - 2

    dir = np.array([dfdx, dfdy])
    norm = np.linalg.norm(dir)
    dir = dir/norm

    return dir*tau

def grad_desc(func, tau, p0, err):
    #main mess
    f_prev = func(p0)
    p_prev = p0
    n=1000000
    grads = np.zeros((n, 2))
    for i in range(n):
        p_new = p_prev - grad(p_prev, tau, func)
        grads[i] = grad(p_new, tau, func)
        f_new = func(p_new)

        if (np.abs(f_new - f_prev) <= err):
            print(i)
            return p_new, f_new, True, grads

        f_prev = f_new
        p_prev = p_new

    return p_new, f_new, False, grads

p, f, converged, grads = grad_desc(func, 0.001, [10, 10], 1e-6)
print(p, f, converged)
```

```
14251
[-0.50000245  1.50000397] -1.249999999916955 True
```

The numerically calculated minimum is:

```
sp.Matrix(p)
```

$$\begin{bmatrix} -0.50000245137345 \\ 1.50000396640556 \end{bmatrix}$$

Our grad descent converged to the analytical solution! Hooray! With $\tau = 0.001$ it took 14251 iterations to find the minimum within 10^{-6} error. However, I started from another $p_0 = [10 \quad 10]$.

(c)

Let's modify the code a bit...

```

def grad_desc(func, tau, p0, err):
    #main mess
    f_prev = func(p0)
    p_prev = p0
    n=5
    grads = np.zeros((n, 2))
    p_news = np.zeros((n, 2))
    for i in range(n):
        p_new = p_prev - grad(p_prev, tau, func)
        grads[i] = grad(p_prev, tau, func)
        p_news[i] = p_new
    f_new = func(p_new)

    f_prev = f_new
    p_prev = p_new

    return grads, p_news

grads, p_news = grad_desc(func, 1, [0, 4], 1e-4)

funcs = np.array([func(i) for i in p_news])

ax1 = plt.subplot(111, projection='3d')
s = ax1.plot_surface(X, Y, F, alpha=0.5, label='Function Surface')
s._facecolors2d = s._facecolor3d
s._edgecolors2d = s._edgecolor3d

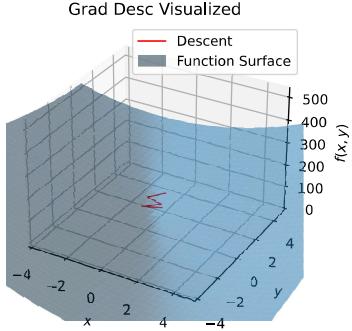
s2 = ax1.plot(xs = p_news[:, 0], ys = p_news[:, 1], zs = funcs, lw=1, color='red'
, label='Descent')

ax1.legend()
ax1.set_title('Grad Desc Visualized')
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')
ax1.set_zlabel('$f(x, y)$')

ax1.set_xlim((-4, 5))
ax1.set_ylim((-4, 5))

```

(-4.0, 5.0)



I suppose this plot provides enough visualization for (c). I was not able to easily plot the function level curves using **Python**, sorry for that.

(d)

I think, this task was accomplished in p. (b) =)

By Arseniy Buchnev
© Copyright 2021.