8 - Optimization

March 13, 2021

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as optimize
import imageio
import os
import cvxpy as cp
import scipy as sp
import sympy as sym
```

1 Washboard Potential

"Washboard potentials" are 1-D potential energy functions with many local minima. They arise in the theory of supercondicting Josephson junctions, the motion of defects in crystals, and many other applications. Consider the following washbarod potential

$$V(r) = A_1 \cos(r) + A_2 \cos(2r) - Fr$$

with $A_1 = 5$, $A_2 = 1$, and F initially set to 1.5.

1.1 Part A

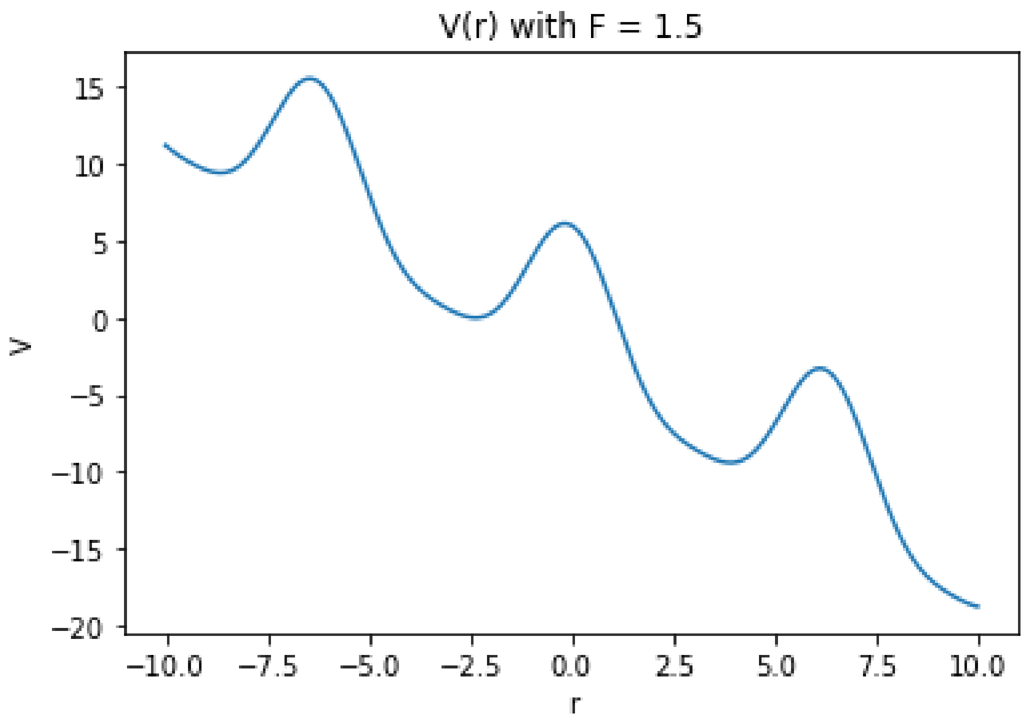
Plot $V(r)$ over $(-10, 10)$. Numerically find the local maximum of V near zero and the local minimum of V to the left (negative side) of zero. What is the potential energy barrier for moving from one well to the next in this potential?

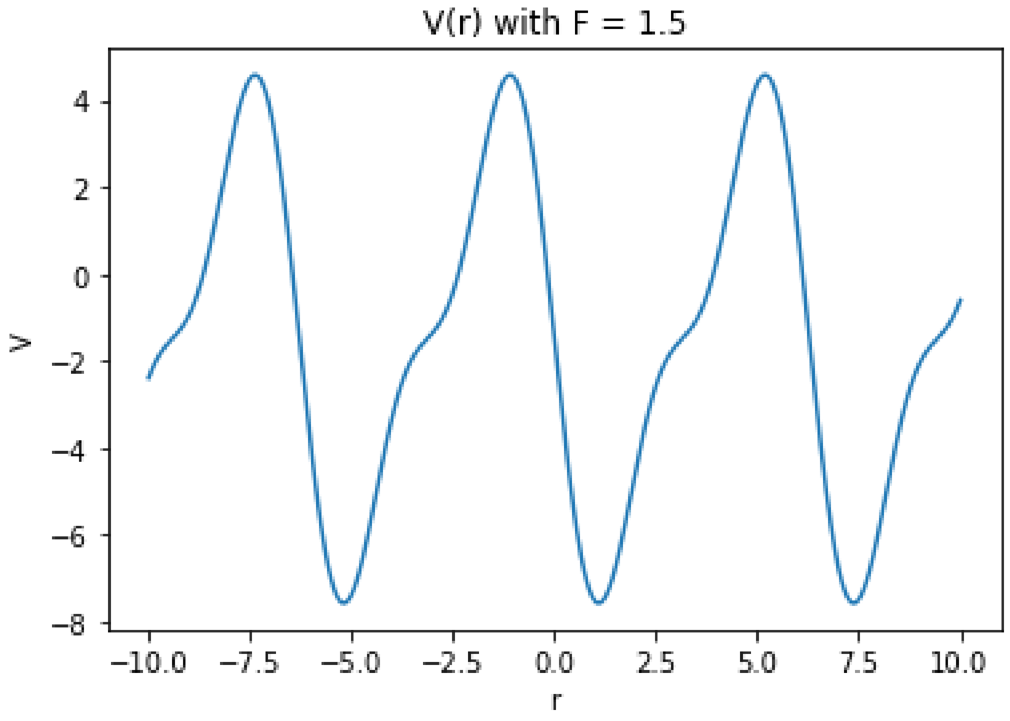
---

[2]:
```python
def V(r,F):
    A_1 = 5
    A_2 = 1
    return A_1 * np.cos(r) + A_2 * np.cos(2*r) - F * r

def diff(func,r,F):
    # Use the complex-step method
    h = 1e-30
    return np.imag(func(r + 1j*h,F))/h
```

```python
def find_extremum(func,F,rLeft,rRight):
    tol = 1e-6
    for i in range(1,500):
        rMid = (rLeft + rRight)/2
        leftPoint = diff(func,rLeft,F)
        rightPoint = diff(func,rRight,F)
        midPoint = diff(func,rMid,F)

        if np.sign(midPoint) == np.sign(leftPoint):
            rLeft = rMid
        else:
            rRight = rMid


        if np.abs(rLeft - rRight) < tol:
            break

    return rMid
```

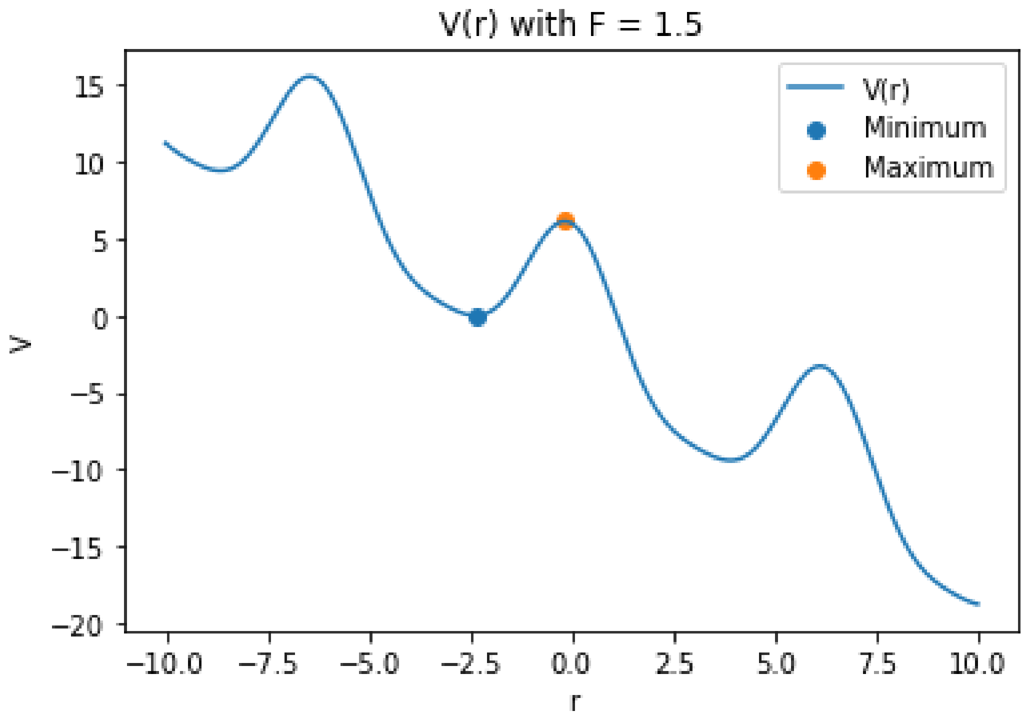
[3]:
```python
r = np.linspace(-10,10,1000)

plt.figure()
plt.plot(r,V(r,1.5))
plt.title("V(r) with F = 1.5")
plt.xlabel("r")
plt.ylabel("V")

plt.figure()
plt.plot(r,diff(V,r,1.5))
plt.title("V(r) with F = 1.5")
plt.xlabel("r")
plt.ylabel("V");
```


V(r) with F = 1.5
V
r
15
10
5
0
−5
−10
−15
−20
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


V(r) with F = 1.5
V
r
4
2
0
−2
−4
−6
−8
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

[4]:
```
# Searching for the minimum and maximum
r_min = find_extremum(V,1.5,-5,-1)
r_max = find_extremum(V,1.5,-1,1)

plt.figure()
plt.plot(r,V(r,1.5),label = "V(r)")
#plt.plot(r,diff(V,r,1.5), label = "diff(V(r))")
plt.scatter(r_min,V(r_min,1.5), label = "Minimum")
plt.scatter(r_max,V(r_max,1.5), label = "Maximum")
plt.title("V(r) with F = 1.5")
plt.xlabel("r")
plt.ylabel("V")
plt.legend()

print("The potential energy barrier is ",V(r_max,1.5) - V(r_min,1.5)," Volts.")
```

```
The potential energy barrier is  6.127109544004874  Volts.
```



1.2 Part B

Usually finding the minimum is only a first step. Often, one wants to explore how the minimum moves and disappears. Increase the external tilting field F, roughly estimate the field F_c at which the barrier disappears. Also, estimate the location r_c at this field where the potential minimum and maximum merge. (This is an example of a saddle-node bifurication.) Give the criterion on the first derivative and the second derivative of $V(r)$ at F_c and r_c. Using these two equations, numerically use a root-finding routine to locate the saddle-node bifurication F_c and r_c.

---

[5]:
```python
filenames = []

# Guesses taken from part A
r_min = -2.366
r_max = -0.169

r = np.linspace(-10,10,1000)

color_min = "blue"
color_max = "orange"

for F in np.linspace(1.5,8,150):

    if abs(V(r_min,F) - V(r_max,F)) < 0.1:
        print("Saturation at r = ",r_min," and F = ",F)
        color_min = "red"
        color_max = "red"

    r_min = find_extremum(V,F,r_min - 0.1,r_min + 0.1)
    r_max = find_extremum(V,F,r_max - 0.1,r_max + 0.1)

    fig = plt.figure()
    plt.plot(r,V(r,F),label = "V(r)")
    plt.scatter(r_min,V(r_min,F), label = "Minimum",color = color_min)
    plt.scatter(r_max,V(r_max,F), label = "Maximum",color = color_max)
    plt.title("V(r)")
    plt.xlabel("r")
    plt.ylabel("V")
    #plt.ylim(-20,20)
    plt.legend()
    plt.draw()

    # create file name and append it to a list
    filename = f'{F}.png'
    filenames.append(filename)

    # save frame
    fig.savefig(filename)
    plt.close()

# # build gif form the individual images
# with imageio.get_writer('example.gif', mode='I') as writer:
#     for filename in filenames:
```

```
#         image = imageio.imread(filename)
#         writer.append_data(image)

# Remove auxillary files
for filename in filenames:
    os.remove(filename)
```

```
Saturation at r =  -1.3507259521484294  and F =  5.775167785234899
Saturation at r =  -1.3329410705566325  and F =  5.818791946308725
Saturation at r =  -1.313941589355461  and F =  5.8624161073825505
Saturation at r =  -1.2933887634277266  and F =  5.906040268456376
Saturation at r =  -1.2707561645507737  and F =  5.949664429530201
Saturation at r =  -1.245141998291008  and F =  5.993288590604027
Saturation at r =  -1.2147197875976485  and F =  6.0369127516778525
Saturation at r =  -1.1744129333496016  and F =  6.080536912751678
Saturation at r =  -1.0744136962890547  and F =  6.124161073825503
Saturation at r =  -0.9744144592285078  and F =  6.167785234899329
```

The estimated saturation is at $r_c \approx -1.35$ and $F_c \approx 5.78$. In case the above plot looks weird, just know that it was a fun gif of the saturation process.

To solve for this in a more robust manner, we use the derivatives. At a saddle point, the first and second derivatives with respect to r must be zero:

$$V'(r) = -A_1 \sin(r) - 2A_2 \sin(2r) - F = 0$$

$$V''(r) = -A_1 \cos(r) - 4A_2 \cos(2r) = 0$$

We can solve this system of equations by using `scipy.optimize.fsolve`:

[6]:
```
A_1 = 5
A_2 = 1

def equations(vars):
    r,F = vars
    return [-A_1*np.sin(r) - 2*A_2*np.sin(2*r) - F, -A_1*np.cos(r) - 4*A_2*np.
 ↪cos(2*r)]

# A root-finding function
rc,Fc = optimize.fsolve(equations, [-1.35, 5.78])

print("rc = ", rc,", Fc = ",Fc)
```

```
rc =  -1.092145208249667 , Fc =  6.073367741558359
```

We can see here that my estimate was close to the actual solution.

2 Sloppy Minimization

Optimization problems that arise in data fitting are often extremely ill-conditioned (a phenomenon sometimes called “sloppiness”). A classically ill-conditioned fitting problem involves a sum of exponentials to data. Consider the function

$$y(t) = \frac{1}{N}\sum_{n=0}^{N-1} \lambda_n e^{-\lambda_n t}$$

Now suppose you don’t know the parameters λ_n. Can you reconstruct them by fitting to experimental data $y_0(t)$?

Consider the case of $N = 2$. Suppose the actual parameters are $\lambda_0 = 2$ and $\lambda_1 = 3$ (so the experiment has data $y_0(t) = e^{-2t} + \frac{3}{2}e^{-3t}$) and we try to minimize the least squared error in the fits C:

$$C(\lambda) = \int_0^\infty (y(t) - y_0(t))^2 dt$$

You can check that this integral evaluates to

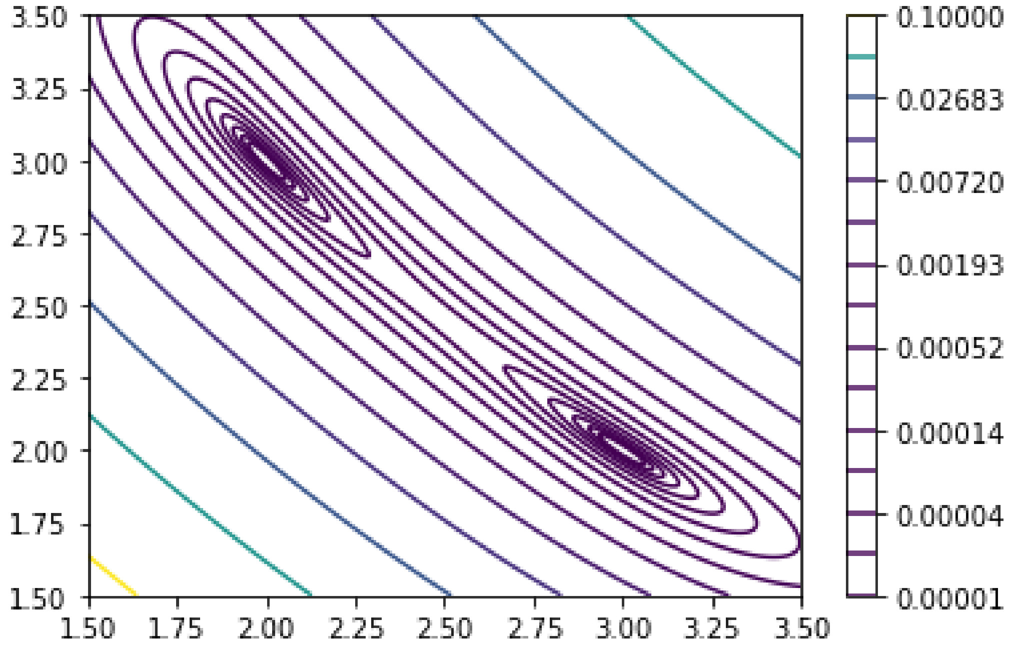
$$C(\lambda) = -\frac{51}{40} + \frac{\lambda_0}{8} - \frac{\lambda_0}{2+\lambda_0} - \frac{3\lambda_0}{2(3+\lambda_0)} + \frac{\lambda_1}{8} + \frac{2}{2+\lambda_1} + \frac{9}{2(3+\lambda_1)} + \frac{\lambda_0\lambda_1}{2(\lambda_0+\lambda_1)}$$

2.1 Part A

Draw a contour plot of C in the square $1.5 < \lambda_n < 3.5$ with enough contours (perhaps non-equally spaced) so that the two minima can be distinguished. (You’ll also need a fairly fine grid of points.) You should see that inferring the two rate constants separately is challenging. This is because the two exponentials have similar shapes, so increasing one decay rate and decreasing the other can almost perfectly compensate for one another.

---

[7]:
```python
def C(lam_0,lam_1):
    return -51/40 + lam_0/8 - lam_0/(2 + lam_0) - 3*lam_0/(2*(3 + lam_0)) +␣
 ↪lam_1/8 + 2/(2 + lam_1) + 9/(2*(3+lam_1)) + lam_0*lam_1/(2*(lam_0 + lam_1))

lam_0 = np.linspace(1.5,3.5,500)
lam_1 = np.linspace(1.5,3.5,500)

LAM_0,LAM_1 = np.meshgrid(lam_0,lam_1)

fig,ax = plt.subplots()
contour = ax.contour(LAM_0,LAM_1,C(LAM_0,LAM_1),levels = np.
 ↪logspace(-5,-1,15),cmap = 'viridis')
cbar = fig.colorbar(contour)
```



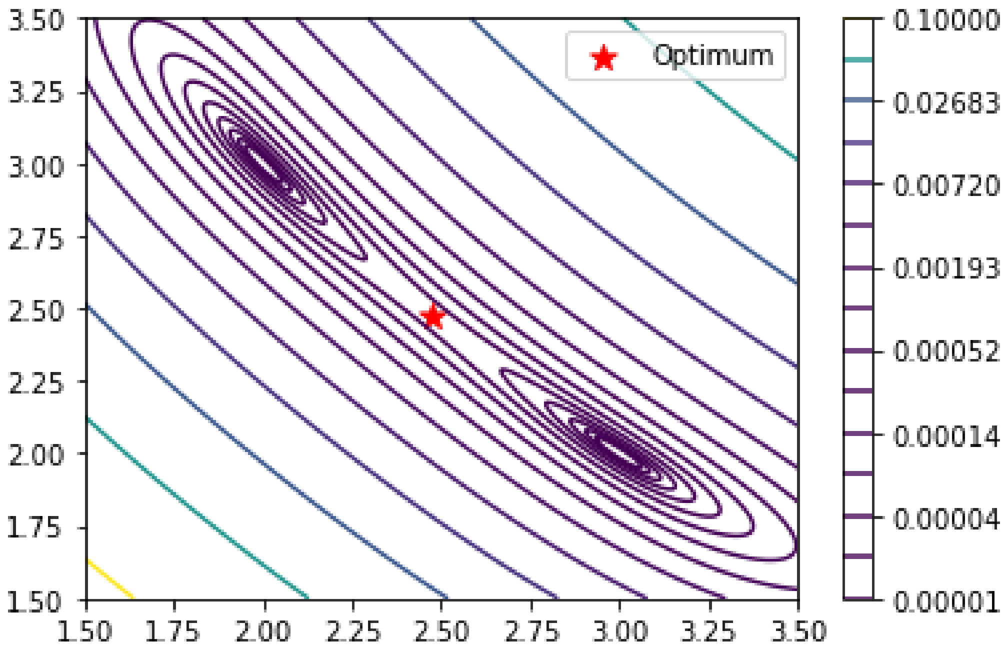
2.2 Part B

Assume the two parameters are equal ($\lambda = \lambda_0 = \lambda_1$), minimize the cost to find the optimal choice for λ. Where is this point on the contour plot? Plot $y_0(t)$ and $y(t)$ with thtis single-exponent best fit on the same graph, over $0 < t < 2$. Do you agree that it would be difficult to distinguish these two fits?

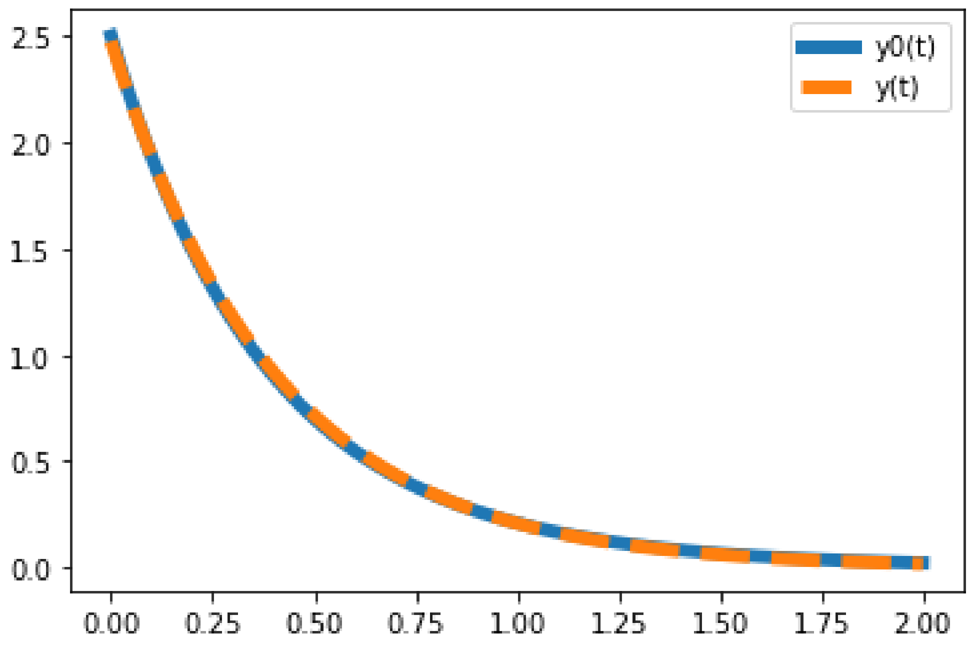
Let's minimize C with the case that $\lambda_0 = \lambda_1 = \lambda$.

[8]:
```
def objective(lam):
    return C(lam,lam)

sol = optimize.minimize(objective,1)
lam = sol.x[0]
print("Optimal Lambda = ",lam)

fig,ax = plt.subplots()
contour = ax.contour(LAM_0,LAM_1,C(LAM_0,LAM_1),levels = np.
 ↪logspace(-5,-1,15),cmap = 'viridis')
cbar = fig.colorbar(contour)
plt.scatter(lam,lam, s = 100, label = "Optimum", marker = "*",color = "red");
plt.legend();
```

```
Optimal Lambda =  2.474938102032043
```



[9]:
```
def y0(t):
    return np.exp(-2*t) + 3/2 * np.exp(-3*t)
def y(t,lam):
    N = len(lam)
    sum = 0;
    for n in range(0,N):
        sum = sum + lam[n] * np.exp(-lam[n] * t)

    return 1/N * sum

t = np.linspace(0,2,1000)
plt.plot(t,y0(t),label = "y0(t)",linewidth = 5)
plt.plot(t,y(t,[lam]),label = "y(t)",linestyle = "--",linewidth = 5)
plt.legend();
```



I can see that these two fits would indeed be hard to distinguish.

2.3 Part C

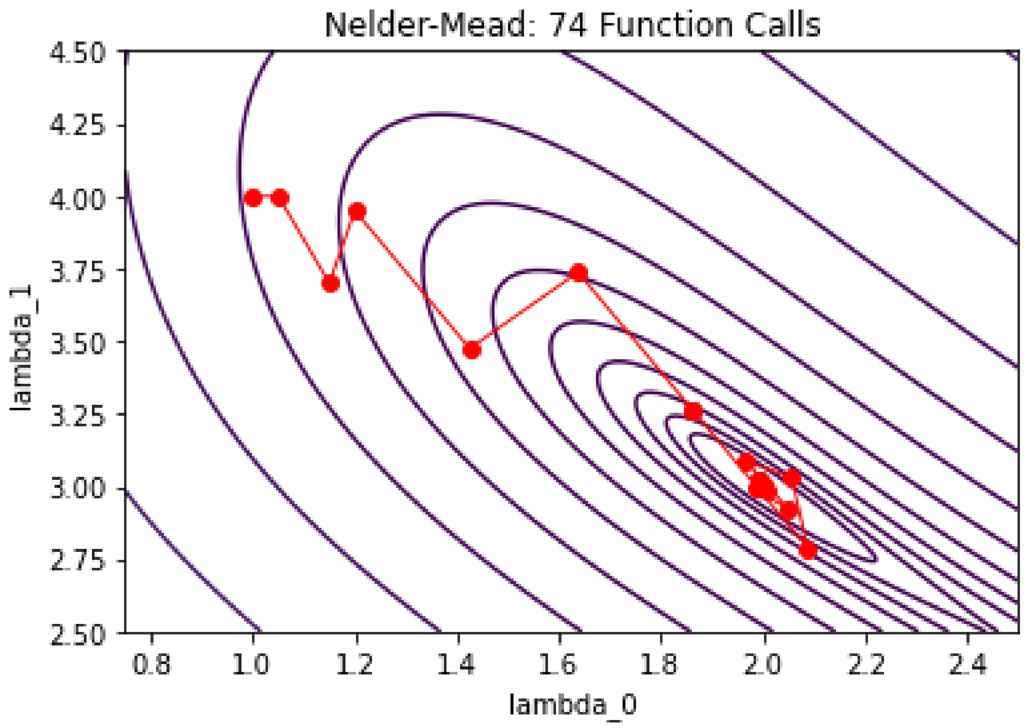
This problem becomes much more severe in high dimensions. The banana-shaped ellipses on your contour plot can become needle-like, with aspect ratios of more than a thousand to one (about the same as a human hair). Following these thin paths down to the true minima can be a challenge for multi-dimensional minimization programs. Find a method for storing and plotting the locations visited by your minimization routine (ie. values (λ_0, λ_1) at which it evaluates C while searching for the minimum). Starting from $\lambda_0 = 1, \lambda_1 = 4$, minimize the cost using as many methods as is convenient within your programming environment, eg. Nelder-Mead, Powell, Newton, Quasi-Newton, conjugate gradient, Levenberge-Margquardt (aka nonlinear least squares). Try to use at least one that does not demand derivatives of the function. Plot the evaluation points on top of the contour plot of the cost for each method. Compare the number of function evaluations needed for each method.

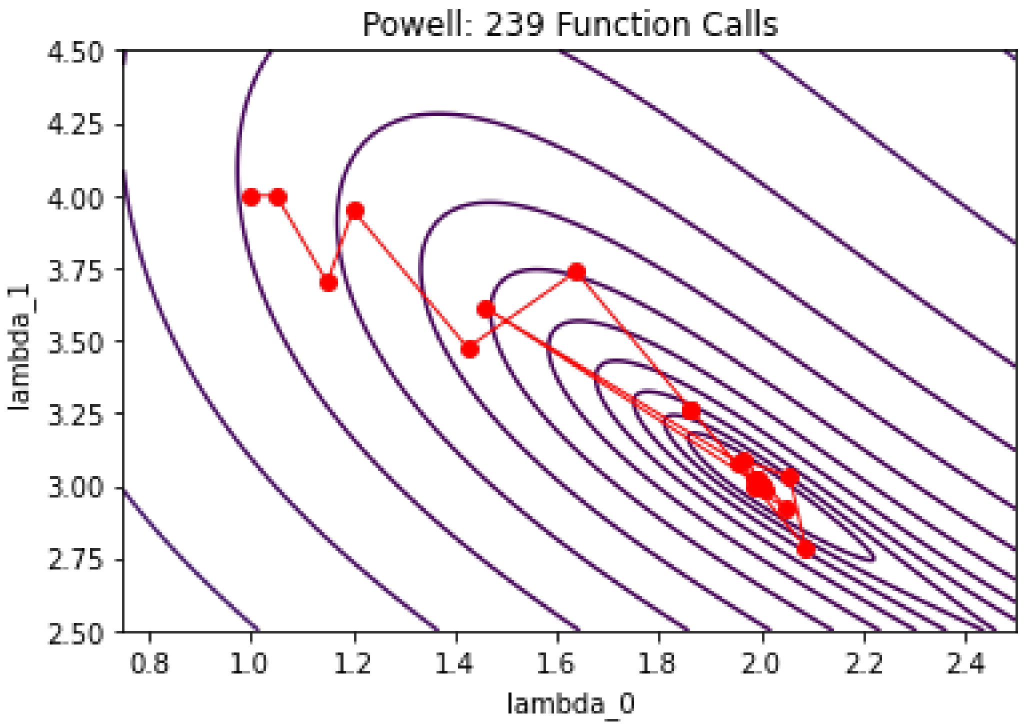
---

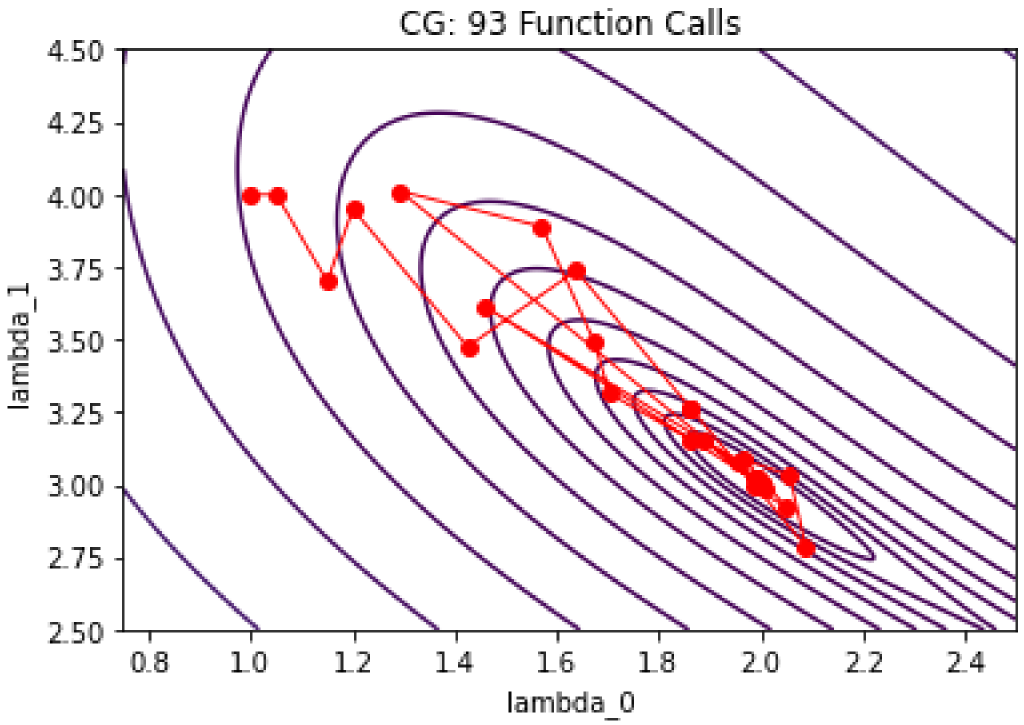
[10]:
```python
def objective(lam):
    return C(lam[0],lam[1])

def callback(lam_i):
    lam_convergence.append([lam_i[0],lam_i[1]])

def plotConvergence(lam_convergence,solver):
    fig,ax = plt.subplots()
    contour = ax.contour(LAM_0,LAM_1,C(LAM_0,LAM_1),levels = np.
 ↪logspace(-4,0,15),cmap = 'viridis')
    lam_convergence = np.array(lam_convergence)
    plt.plot(lam_convergence[:,0],lam_convergence[:,1],marker = 'o',linewidth =␣
 ↪1, color = "red");
    title_string = str(solver + ": " + str(results.nfev) + " Function Calls")
    plt.title(title_string)
    plt.xlabel("lambda_0")
    plt.ylabel("lambda_1")

# Defining the grid
lam_0 = np.linspace(0.75,2.5,500)
lam_1 = np.linspace(2.5,4.5,500)
LAM_0,LAM_1 = np.meshgrid(lam_0,lam_1)

startingPoint = [1,4]
lam_convergence = [startingPoint];

# Trying different solvers
solver = 'Nelder-Mead'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'Powell'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'CG'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'BFGS'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'L-BFGS-B'
print("Using the ", solver, " method")
```

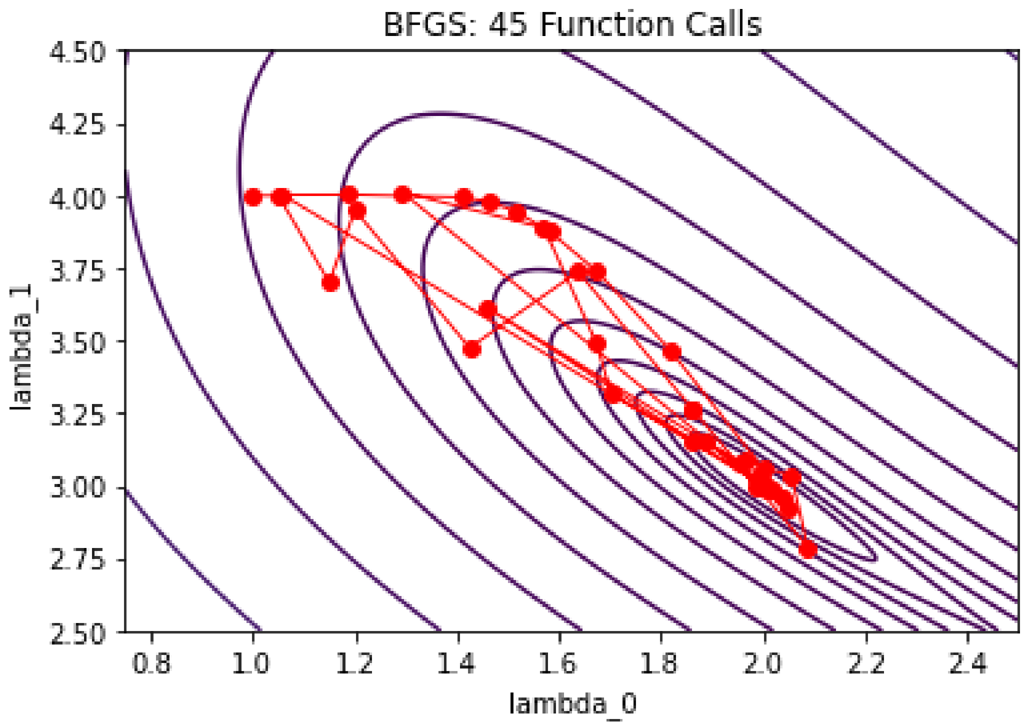
```
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'TNC'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)

solver = 'SLSQP'
print("Using the ", solver, " method")
results = optimize.minimize(objective,startingPoint,method = solver,callback =␣
 ↪callback,options = {'disp': False})
plotConvergence(lam_convergence,solver)
```

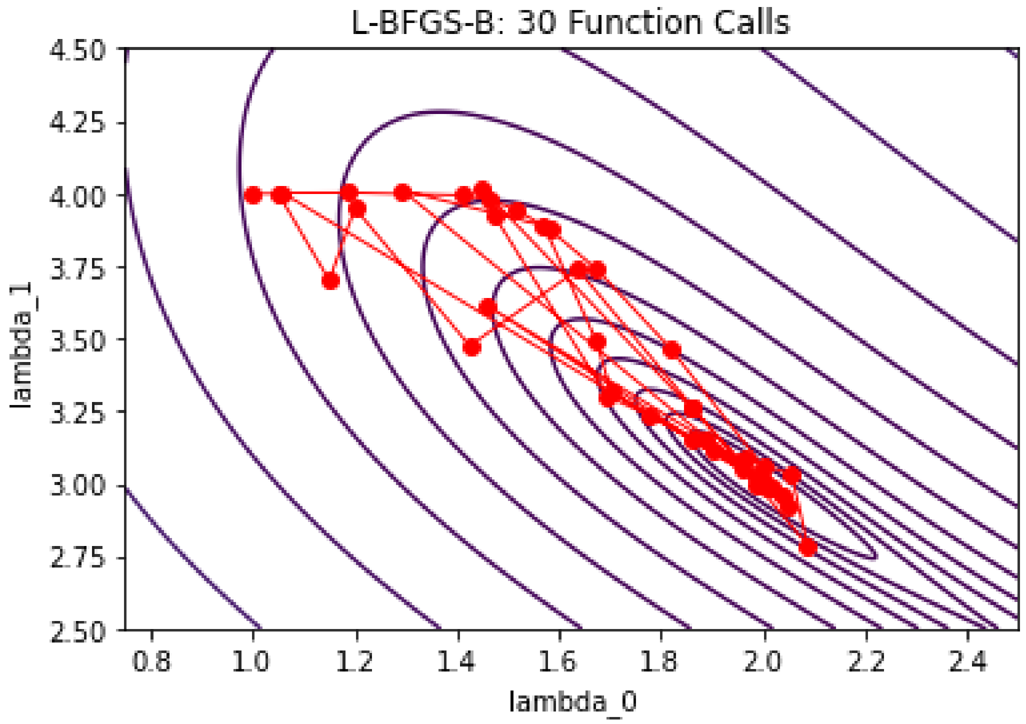
```
Using the  Nelder-Mead  method
Using the  Powell  method
Using the  CG  method
Using the  BFGS  method
Using the  L-BFGS-B  method
Using the  TNC  method
Using the  SLSQP  method
```

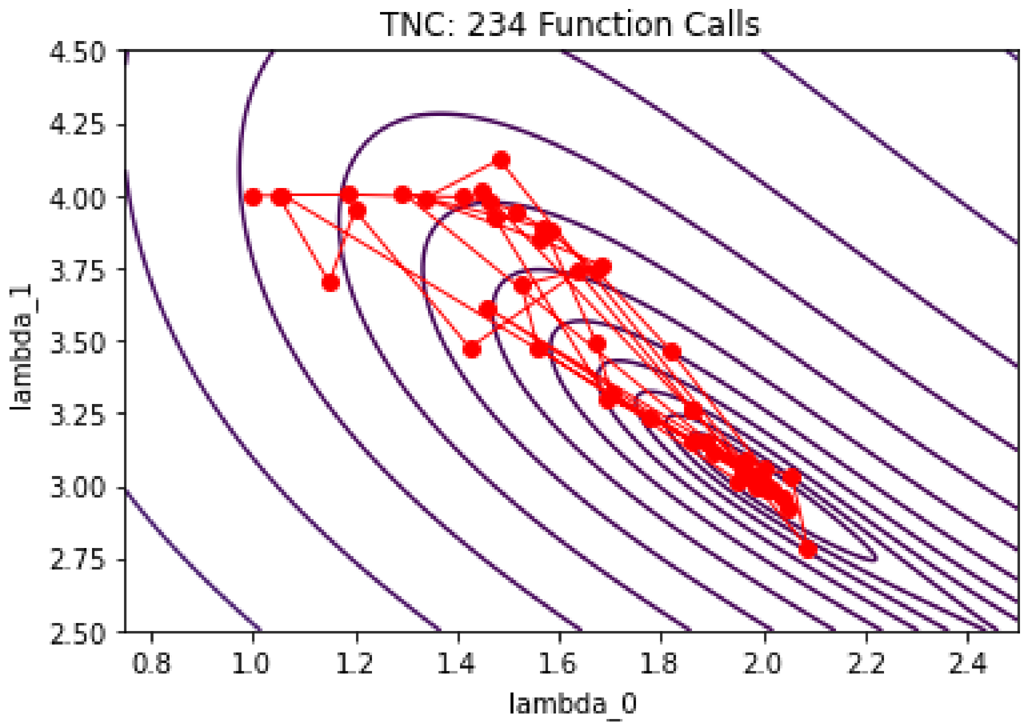


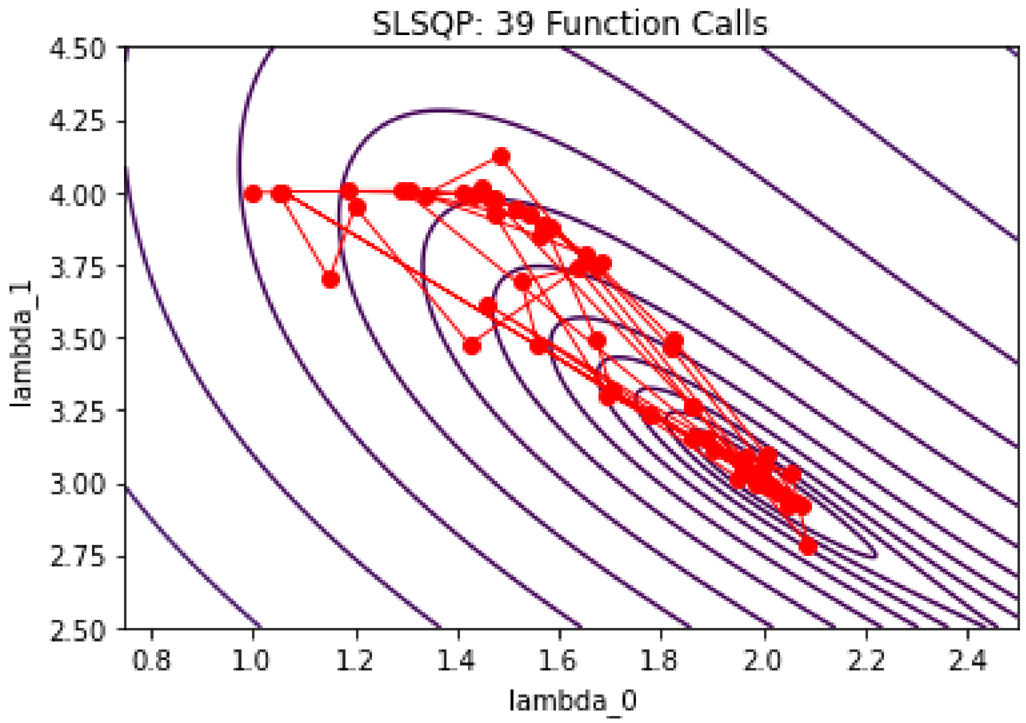

Powell: 239 Function Calls
lambda_1
lambda_0
4.50
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4


CG: 93 Function Calls
lambda_1
lambda_0
4.50
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4


BFGS: 45 Function Calls
lambda_0
lambda_1


L-BFGS-B: 30 Function Calls
lambda_0
lambda_1


TNC: 234 Function Calls
lambda_1
lambda_0
4.50
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4


SLSQP: 39 Function Calls
lambda_1
lambda_0
4.50
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4

Although it didn't have the fewest number of function calls, I think that I trust the Nelder-Mead method the most after looking at these convergence plots because it seems to travel almost directly to the correct spot and the others seems to wander quite a bi and look like they even shoot out of the minimum sometimes.

3 Convex Functions

Consider a vector in $x \in R^n$. The L-p norms are a family of vector norms parameterized by p:

$$||x||_p = \left(\sum_{k=0}^{n-1} |x_k|^p\right)^{\frac{1}{p}}$$

3.1 Part A

Show that $||x||_p$ is a convex function for any $p \geq 1$. Hint, you may find the Minkowski inequality helpful:

$$||x+y||_p \leq ||x||_p + ||y||_p$$

For a function f to be convex, it must satisfy

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

Which is shown in section 1.3 of Boyd. For this to be true, x and y need to be real numbers and $\alpha + \beta = 1$, where $\alpha \geq 0$ and $\beta \geq 0$. Let's just try plugging this in directly:

$$\left(\sum_{k=0}^{n-1} |\alpha x + \beta y|^p\right)^{\frac{1}{p}} \leq \alpha \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}} + \beta \left(\sum_{k=0}^{n-1} |y|^p\right)^{\frac{1}{p}}$$

Let's now apply the Minkowski inequality to the left-hand side of the equation

$$\left(\sum_{k=0}^{n-1} |\alpha x + \beta y|^p\right)^{\frac{1}{p}} \leq \left(\sum_{k=0}^{n-1} |\alpha x|^p\right)^{\frac{1}{p}} + \left(\sum_{k=0}^{n-1} |\beta y|^p\right)^{\frac{1}{p}} \leq \alpha \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}} + \beta \left(\sum_{k=0}^{n-1} |y|^p\right)^{\frac{1}{p}}$$

$$\left(\sum_{k=0}^{n-1} |\alpha x|^p\right)^{\frac{1}{p}} + \left(\sum_{k=0}^{n-1} |\beta y|^p\right)^{\frac{1}{p}} \leq \alpha \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}} + \beta \left(\sum_{k=0}^{n-1} |y|^p\right)^{\frac{1}{p}}$$

The inequality will hold if both similar terms obey the inequality too

$$\left(\sum_{k=0}^{n-1} |\alpha x|^p\right)^{\frac{1}{p}} \leq \alpha \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}} \text{ and } \left(\sum_{k=0}^{n-1} |\beta x|^p\right)^{\frac{1}{p}} \leq \beta \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}}$$

Investigating just the α case gives

$$\left(\sum_{k=0}^{n-1} |\alpha x|^p\right)^{\frac{1}{p}} \leq \alpha \left(\sum_{k=0}^{n-1} |x|^p\right)^{\frac{1}{p}}$$

Taking both sides to the p^{th} power,

$$\sum_{k=0}^{n-1} |\alpha x|^p \leq \alpha \sum_{k=0}^{n-1} |x|^p$$

Bringing the α into the sum on the left-hand side:

$$\sum_{k=0}^{n-1} |\alpha x|^p \leq \sum_{k=0}^{n-1} \alpha |x|^p$$

We can also look at this as individual terms:

$$|\alpha x|^p \leq \alpha |x|^p$$

$$|\alpha|^p |x|^p \leq \alpha |x|^p$$

Dividing out the x terms gives

$$|\alpha|^p \leq \alpha$$

We know from the conditions for α and β that $0 \leq \alpha \leq 1$. Therefore

$$\alpha^p \leq \alpha$$

This inequality will always hold for $0 \leq \alpha \leq 1$ if $p \geq 1$. It will be violated if $p < 1$, as shown below in code. Therefore, the L-p norm is convex for $p \geq 1$.

[11]:
```
def test_inequality(alpha,p):
    LHS = alpha**p
    RHS = alpha
    return LHS - RHS # Should be less than or equal to zero if alpha >= 1 and p␣
 ↪>= 1

alpha = np.linspace(0,1,5)
```

```
p = 0.25
print("p =",p,":", test_inequality(alpha,p))

p = 0.5
print("p =",p,":", test_inequality(alpha,p))

p = 1
print("p =",p,":", test_inequality(alpha,p))

p = 2
print("p =",p,":", test_inequality(alpha,p))

p = 20
print("p =",p,":", test_inequality(alpha,p))
```

```
p = 0.25 : [0.         0.45710678 0.34089642 0.18060486 0.        ]
p = 0.5 : [0.         0.25       0.20710678 0.1160254  0.        ]
p = 1 : [0. 0. 0. 0. 0.]
p = 2 : [ 0.     -0.1875 -0.25   -0.1875  0.    ]
p = 20 : [ 0.         -0.25       -0.49999905 -0.74682879  0.        ]
```

3.2 Part B

The L-p norm defines a valid vector norm for any $p \geq 1$. for $p < 1$, it no longer defines a norm, but sometimes it is useful to think about the function for $p < 1$. Show that the "L-0" norm counts the number of non-zero components of the vector. Show that the "L-0" norm is not convex.

Again starting with the definition of the L-p norm:

$$||x||_p = \left(\sum_{k=0}^{n-1} |x_k|^p \right)^{\frac{1}{p}}$$

Let's set $p = 0$:

$$||x||_0 = \left(\sum_{k=0}^{n-1} |x_k|^0 \right)^{\frac{1}{0}}$$

This does not make much mathematical sense because we have an undefined exponent. Let's rewrite the original expression as

$$||x||_p = \left(\sum_{k=0}^{n-1} |x_k|^p \right)^{\frac{1}{p}} = \exp\left[\frac{1}{p} \ln \left(\sum_{k=0}^{n-1} |x_k|^p \right) \right]$$

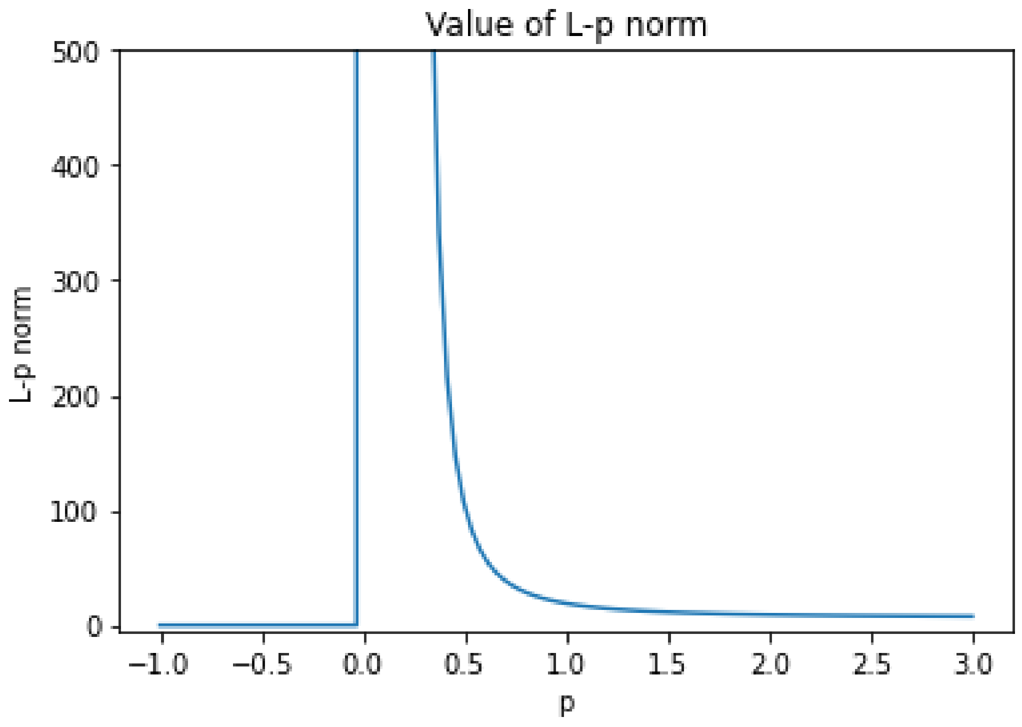
We can take the limit as $p \to 0$:

$$\lim_{p\to 0} \exp\left[\frac{1}{p}\ln\left(\sum_{k=0}^{n-1} |x_k|^p\right)\right]$$

Unfortunately, in the limit as $p \to 0$, all elements in the x array will become equal to 1. Then we still have the problem of dividing by zero. Let's try plotting the limit to see if it does attempt to converge at the number of non-zero elements:

[12]:
```python
x = np.array([0,1,2,3,4,0,7,0,2])
p = np.linspace(-1,3,100)

def p_norm(x,p):
    return np.sum(np.abs(x)**p)**(1/p)

def test_limit(x,p):
    results = np.zeros(len(p))
    for i in range(len(p)):
        results[i] = p_norm(x,p[i])
    return results

plt.plot(p,test_limit(x,p))
plt.ylim(-5,500)
plt.xlabel("p")
plt.ylabel("L-p norm")
plt.title("Value of L-p norm");
```

```
<ipython-input-12-fcfc8b56996e>:5: RuntimeWarning: divide by zero encountered in
power
  return np.sum(np.abs(x)**p)**(1/p)
```



It seems apparent from the plot that the L-0 norm as defined here does not actually converge to the number of non-zero elements in an array. According to this source they say that most engineers and mathematicians use an alternate definition of the L-0 norm that counts the number of non-zero elements:

$$||x||_0 = \#(i|x_i \neq 0)$$

where (I believe, according to this source), the # symbol counts the number of i's that satisfy the condition that $x_i \neq 0$.

Therefore, I feel pretty good about saying that the definition of the L-p norm does not converge to counting the number of non-zero array elements as $p \to 0$. However, we do have an alternative definition for the L-0 norm that does count the number of non-zero array elements.

Showing that the L-0 norm is not convex

In part A of this problem, we showed that the criteria for the L-p norm to be convex boiled down to the following statements being true

$$\alpha^p \leq \alpha$$

$$\beta^p \leq \beta$$

where

$$0 \leq \alpha \leq 1$$

$$0 \leq \beta \leq 1$$

$$\alpha + \beta = 1$$

According to these conditions, if $p = 0$, then

$$\alpha^p = \alpha^0 = 1 = \alpha \quad (\text{if} \quad \alpha = 1)$$

but, if $\alpha = 1$, then $\beta = 0$ and

$$\beta^p = 0^0 = 1 \nleq \beta$$

The same logic applies when initially applied the β instead of α. Because these considtions for α and β cannot be simulatenously met, the L-0 norm is not convex.

4 Convex Optimization

4.1 Part A

In your environment, find a convex optimization DSL. What are some of the algorithms that are available?

The DSL that I found for python is `cvxpy`. The solvers used are the open-source solvers ECOS, OSQP, and SCS:

- ECOS:
  - solves convex second-order cone programs
- OSQP:
  - quadratic programming
- SCS:
  - Solves large-scale convex cone problems

4.2 Part B

Solve the following optimization problem:

$$\min_x ||J(x - x_0)||_2^2 + \lambda ||x||_1$$

$$x_i \geq 0$$

for the J and x_0 in the attached file (get zip file online).

$||\cdot||_p$ denotes the p-norm and $\lambda = 0, 10^{-8}, 10^{-4}, 10^{-2}, 1$ is a Lagrange multiplier. For each value of λ, report the value of your optimized objective function and the number of components of x that are zeros (ie. the number of constraints that are saturated at your solution vector)

---

[13]:
```
J = np.loadtxt("/Users/markanderson/Desktop/HW_Optimiztion/J.txt")
x0 = np.loadtxt("/Users/markanderson/Desktop/HW_Optimiztion/x0.txt")
```

[14]:
```
x = cp.Variable(len(x0))
constraints = [x >= 0]

for lam in [0,1e-8,1e-4,1e-2,1]:

    # Define objective function
    objective = cp.norm(J @ (x - x0),2)**2 + lam * cp.norm(x,1)

    prob = cp.Problem(cp.Minimize(objective),constraints)

    prob.solve(solver = cp.SCS)

    print("lambda: ",lam,", Objective = ",prob.value,", Number of x < 1e-6:␣
 ↪",len([i for i in x.value if i < 1e-6]) )
```

```
lambda:  0 , Objective =  0.00017574846685325285 , Number of x < 1e-6:  3
lambda:  1e-08 , Objective =  0.000177552017750098 , Number of x < 1e-6:  5
lambda:  0.0001 , Objective =  0.4810604052803991 , Number of x < 1e-6:  245
lambda:  0.01 , Objective =  38.02075110882707 , Number of x < 1e-6:  299
lambda:  1 , Objective =  2695.0242462084825 , Number of x < 1e-6:  345
```