Homework 7

1 General optimization

$$p^* = \min_{x \in \mathbb{R}^n} f_0(x) \; ; \; f_i(x) \leq 0, \quad i = 1, \ldots, m$$

(a)
$$\begin{cases} g^* = \min_{x,t} t, & \leftarrow \text{linear objective} \\ f_0(x) \leq t \\ f_i(x) \leq 0, \quad i = 1, \ldots, m \end{cases}$$

finding the x, t subject to constraint $f_0(x) \leq t$ essentially minimizes $f_0(x)$

(b) $\min_x f_0(x) + \underbrace{\sum_{i=1}^{m} \max(0, f_i(x))^2}$

adds a penalty every time the constraint is not satisfied, implicitely constraining the problem, forcing it to satisfy the original constrained problem

OR define $g(x) := \begin{cases} f_0(x) & x \in \mathcal{X} \text{ feasible set} \\ \infty & \text{otherwise} \end{cases}$

$\Rightarrow \min_x g(x)$

(c)

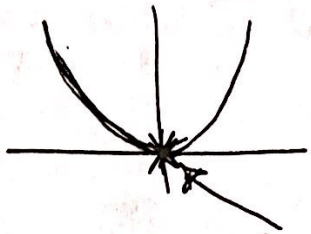
$$\begin{cases} \max\limits_{t} \; t \\ t \le f_0(x) \quad \forall x \in \mathcal{X} \text{ (feasible set)} \end{cases}$$

brings t as close to every value of $f_0(x)$, including $\min\limits_{x} f_0(x)$, and the feasible set may be infinite.

(d) consider

$$\min_{x} \begin{cases} x^2 & |x| \le 1 \\ x & \text{otherwise} \end{cases} \quad \text{where } x < 0$$

subject to $|x| \le 1$



optimum with constraint $|x| \le 1$

← if we remove constraint

x becomes the new minimum

∴ We cannot remove constraint and obtain the same solution

(e) Show

$$\min_x \min_y f_1(x,y) = \min_y \min_x f_0(x,y)$$

$$\min_y F_0(x,y) \le F_0(x,y)$$

$$\Rightarrow \min_x \min_y f_0 \le \min_x F_0(x,y)$$

$$\Rightarrow \min_x \min_y F_0 \le \min_y \min_x f_1(x,y)$$

similarly

$$\min_y \min_x F_0 \le \min_x \min_y F_1(x,y)$$

$$\Rightarrow \min_x \min_y F_1(x,y) = \min_y \min_x F_0(x,y)$$

∎

(f)

$$\min_{x \in X} \max_{y} F_0(x,y) = p^*$$

then $p^* \geq d^*$ where $d^* = \max_{y} \min_{x \in X} F_0(x,y)$

$$\min_{x \in X} F_0(x,y) \leq F_0(x,y) \leq \max_{y} F_0(x,y)$$

$\Rightarrow$ $\min_{x \in X} F_0(x,y) \leq \min_{x \in X} \max_{y} F_0(x,y)$ applying $\min_{x \in X}$ to both sides

$\Rightarrow$ $\max_{y} \min_{x \in X} F_0(x,y) \leq \min_{x \in X} \max_{y} F_0(x,y)$ applying $\max_{y}$ to both sides

$\Rightarrow$ $d^* \leq p^*$

□

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix} \qquad H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

2. Gradient Descent VS Newton-Raphson

(a) $f(x) = x_1^2 + x_2^2 + 8x_1 + 2x_2 + 17$

i) $\nabla f = \begin{pmatrix} 2x_1 + 8 \\ 2x_2 + 2 \end{pmatrix}$

ii) $H = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ $\qquad H^{-1} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$

iii) $\nabla f \overset{\text{set}}{=} \vec{0}$ ⟩ minimum at

$$\Rightarrow \boxed{\begin{pmatrix} x_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} -4 \\ -1 \end{pmatrix}}$$

IV. Jupyter

V. It took gradient descent much longer than N-R

Newton Raphson took just a single step

(b) $f(x) = \cosh(\varepsilon x_1^2 + x_2^2)$

i. $\nabla f(x) = \begin{pmatrix} 2\varepsilon x_1 \sinh(\varepsilon x_1^2 + x_2^2) \\ 2x_2 \sinh(\varepsilon x_1^2 + x_2^2) \end{pmatrix}$

ii.

$$Hf(x) = \begin{pmatrix} 2\varepsilon \sinh(\varepsilon x_1^2 + x_2^2) + 4\varepsilon^2 x_1^2 \cosh(\varepsilon x_1^2 + x_2^2) & 4\varepsilon x_2 x_1 \cosh(\varepsilon x_1^2 + x_2^2) \\ 4\varepsilon x_2 x_1 \cosh(\varepsilon x_1^2 + x_2^2) & 2\sinh(\varepsilon x_1^2 + x_2^2) + 4x_2^2 \cosh(\varepsilon x_1^2 + x_2^2) \end{pmatrix}_{2\times 2}$$

iii Jupyter

iv. They look about the same

(c) Generally Newton-Raphson should do better since it is a second order method

3. Gradient Descent × The Laplacian

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

← adjacency matrix

$$D = \begin{pmatrix} 2 & & & & & \\ & 3 & & & & 0 \\ & & 2 & & & \\ & & & 3 & & \\ 0 & & & & 3 & \\ & & & & & 1 \end{pmatrix}$$

$D_{ii} = d_i$ degree of vertex

← eg degree of vertex 6 is 1

(a) Laplacian

$$L = D - A$$

$$= \begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

(b) $L = D - A$

$L^T = (D - A)^T$

$= D^T - A^T$

$= D - A = L$ since $A^T = A$

from HW 2 adjacency matrix is symmetric

(c) $x^T L x = x^T D x - x^T A x$

$= \sum_{i=1}^{n} x_i^2 d_i - x^T \sum_{j=1}^{n} a_{ij} x_j$

$= \sum x_i^2 d_i - \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j$

$= \sum x_i^2 d_i - \sum_{(i,j) \in E} x_i x_j = \sum_{i=1}^{n} d_i - \sum_{i,j \in E} x_i x_j$

$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} d_1 & d_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} d_1 x_1 \\ d_2 x_2 \end{pmatrix} = d_1 x_1^2 + d_2 x_2^2$

$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$

$\begin{pmatrix} -x_2 \\ -x_1 - x_3 \\ -x_2 \end{pmatrix} =$

working backwards

$\frac{1}{2} \sum_{(i,j) \in E} (x_i - x_j)^2 = \frac{1}{2} \sum x_i^2 + x_j^2 - 2 x_i x_j$

$= \frac{1}{2} \sum_{i \in V} x_i^2 + \frac{1}{2} \sum_{j \in V} x_j^2 - \sum_{i,j \in E} x_i x_j$

$= \frac{1}{2} 2 \sum_{i=1}^{n} d_i - \sum x_i x_j$

since A is symmetric

$\sum_{i \in V} x_i^2 = \sum_{j \in V} x_j^2$

and since $x_i^2 = 1$

$\sum x_i = d_i$

$x_i^2 = 1$

equal ☑

(d) $L = D - A$ $\quad \mathbb{1} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$

$L\mathbb{1} = D\mathbb{1} - A\mathbb{1}$ ← justification

$= 0$

$A\mathbb{1}$ sums rows and by definition of adjacency matrix, this is exactly the degree of vertexts.

(e)

$$x^T(L + \lambda \mathbb{1}\mathbb{1}^T)x - \lambda \mathbb{1}^T x$$

$$= x^T L x + \lambda x^T \mathbb{1}\mathbb{1}^T x - \lambda \mathbb{1}^T x$$

from part (c) ↓

$$= \frac{1}{2}\sum (x_i - x_j)^2 + \lambda x^T \mathbb{1}\mathbb{1}^T x - \lambda \mathbb{1}^T x$$

$$2\lambda \left(\sum x_i - 1\right)^2 = 2\lambda \left(\sum_{i \in V} x_i - 1\right)\left(\sum x_i - 1\right)$$

$$= 2\lambda \left(\sum x_i \sum x_i - 2\sum x_i + 1\right)$$

$$= 2\lambda x^T \mathbb{1}\mathbb{1} x - 4\lambda \mathbb{1}^T x + 2\lambda$$

scratch

$\mathbb{1}^T x = \sum x_i$

$x^T \mathbb{1} = \sum x_i$

$$= \frac{1}{2}\sum (x_i - x_j)^2 + \lambda\left(\sum x_i \sum x_i - 2\sum x_i\right)$$

$x^2 - 2x + 1 - 1$

$(x-1)^2$

completing the square

$$= \frac{1}{2}\sum (x_i - x_j)^2 + \lambda\left(\sum x_i - 1\right)^2$$

$\therefore x^T(L + \lambda \mathbb{1}\mathbb{1}^T)x - \lambda \mathbb{1}^T x$

$= \frac{1}{2}\sum (x_i - x_j)^2 + \lambda\left(\sum x_i - 1\right)^2$

(f) Objective function

$$f(x) = \frac{1}{2}x^T L x + \frac{1}{2}\lambda x^T \mathbb{1}\mathbb{1}^T x - \lambda \mathbb{1}^T x$$

$$\nabla f(x) = \frac{1}{2}\left(Lx + L^T x\right) + \frac{1}{2}\left(\lambda\left(\mathbb{1}\mathbb{1}^T x + (\mathbb{1}\mathbb{1}^T)^T x\right)\right) - \lambda(\mathbb{1}^T)^T$$

$$= Lx + \lambda \mathbb{1}\mathbb{1}^T x - \lambda \mathbb{1}$$

L symmetric $\quad (\mathbb{1}\mathbb{1}^T)^T = \mathbb{1}\mathbb{1}^T$

set equal to zero

$$(L + \lambda \mathbb{1}\mathbb{1}^T)x - \lambda \mathbb{1} = 0$$

$$\boxed{x = \lambda(L + \lambda \mathbb{1}\mathbb{1}^T)^{-1}\mathbb{1}}$$

is optimum

(g) $$\boxed{x_{k+1} = x_k - \eta\left(Lx_k + \lambda \mathbb{1}\mathbb{1}^T x_k - \lambda \mathbb{1}\right)}$$

(h) $$x_{k+1} - x^* = x_k - \eta\left(Lx_k + \lambda \mathbb{1}\mathbb{1}^T x_k - \lambda \mathbb{1}\right) - \lambda(L + \lambda \mathbb{1}\mathbb{1}^T)^{-1}\mathbb{1}$$

= next page

$$\left(I-\eta\left(L+\lambda 1 1^T\right)\right)x_k+\eta\lambda 1-\lambda\left(L+\lambda 1 1^T\right)^{-1}1$$

writing backwards

② $\left(I-\eta\left(L+\lambda 1 1^T\right)\right)x_k-\left(I-\eta\left(L+\lambda 1 1^T\right)\right)\lambda\left(L+\lambda 1 1^T\right)^{-1}1$

x^*

③ $=\left(I-\eta\left(L+\lambda 1 1^T\right)\right)x_k-\lambda\left(L+\lambda 1 1^T\right)^{-1}1+\eta\lambda 1$ the same

① $=\left(I-\eta\left(L+\lambda 1 1^T\right)\right)\left(x_k-x^*\right)$

(i) $\|x_k-x^*\|_2=\|\left(I-\eta\left(L+\lambda 1 1^T\right)\right)(x_{k-1}-x^*)\|_2$ from part (b)

$\leq \|I-\eta\left(L+\lambda 1 1^T\right)\|_2\,\|x_{k-1}-x^*\|_2$

let $\rho=\|I-\eta\left(L+\lambda 1 1^T\right)\|_2$

$\Rightarrow \|x_k-x^*\|\leq \rho\|x_{k-1}-x^*\|$

$\leq \rho^2\|x_{k-2}-x^*\|$

$\cdots$

$\leq \rho^k\|x_0-x^*\|$

Since zero is an eigenvalue, then max eigenvalue of

$-\eta L$ is <u>zero</u>

the eigenvalues of $\lambda \mathbb{1}\mathbb{1}^T$ are n (multiplicity 1) and 0 (multiplicity $n-1$)

so max eigenvalue of $-\eta L - \eta \lambda \mathbb{1}\mathbb{1}^T$ is

$$= P\left(-\eta D_L - P^{-1}\eta\lambda\mathbb{1}\mathbb{1}^T P\right)P^{-1}$$

$$= P\left(-\eta D_L - P^T Q \begin{pmatrix} 0 & & \\ & \ddots & \\ & & 0_{\eta\lambda n} \end{pmatrix} Q^T P\right) P^T$$

then $I + \left(-\eta L - \eta\lambda\mathbb{1}\mathbb{1}^T\right)$ max eigenvalue is $1 + \eta\lambda n$

max singular value then is

$$\rho = \sqrt{1+\eta\lambda n} \quad ?$$

$$\therefore \left\| x_k - x^* \right\| \leq \sqrt{1+\eta\lambda n}^{\,k} \left\| x_0 - x^* \right\|$$

↑ something like this

(j) we want

$$(1+\eta\lambda_n)^{k/2}\|x_0-x^*\| \le \varepsilon$$

$$(1+\eta\lambda_n)^{k} \le \left(\frac{\varepsilon}{\|x_0-x^*\|}\right)^2$$

$$k \le \frac{\log\left(\frac{\varepsilon}{\|x_0-x^*\|}\right)^2}{\log(1+\eta\lambda_n)}$$

(k) j is wrong

i is wrong

(l) ...

4. Fourier representation of Images

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi}{N} kn}$$

(a)

$$X = \frac{1}{\sqrt{3}} \underbrace{\begin{pmatrix} 1 & e^{\frac{-j2\pi}{N}} & e^{\frac{-j4\pi}{N}} \\ 1 & e^{\frac{-j4\pi}{N}} & e^{\frac{-j8\pi}{N}} \\ 1 & e^{\frac{-j6\pi}{N}} & e^{\frac{-j12\pi}{N}} \end{pmatrix}}_{F_x} x$$

(n=0, n=1, n=2; k=1, k=2, k=3)

(b) $\phi = e^{\frac{-j2\pi}{N}}$

$$F = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & \phi & \phi^2 & \phi^3 & \cdots & \phi^{N-1} \\ 1 & \phi^2 & \phi^4 & \phi^6 & \cdots & \phi^{2(N-1)} \\ 1 & \phi^3 & \phi^6 & \phi^9 & \cdots & \phi^{3(N-1)} \\ & & & & & \\ 1 & \phi^N & \phi^{2N} & \phi^{3N} & & \phi^{N(N-1)} \end{pmatrix}$$

(j=1, j=2; i=1, i=2)

$$F_{ij} = \phi^{(i-1)j}$$

for $i, j \in \{1, \ldots, N\}$

(c) Kernel Regularized least squares

gd_and_nr

November 1, 2019

1 Performance Testing of Gradient Descent and Newton-Raphson Method

1.1 Places where you have to write code are marked with #TODO

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pdb
```

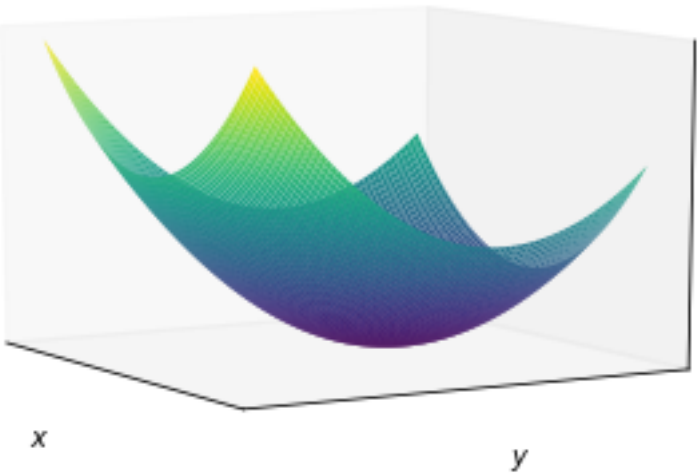
1.2 Tesing with a paraboloid

1.2.1 Define the paraboloid function

[4]:
```
def f(x):
    (x1,x2) = x
    return x1*x1 + x2*x2 - 8*x1 + 2*x2 + 17
```

1.2.2 Visualize the function in 3D

[5]:
```
x = np.arange(0, 8, 0.1)
y = np.arange(-4, 4, 0.1)
xx, yy = np.meshgrid(x, y, sparse=True)
z = f([xx,yy])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
ax.view_init(elev=10., azim=150)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
plt.show()
```



1.2.3 Define first derivative and hessian of function

[6]:
```
def delta_f(x):
    (x1,x2) = x
    return np.array([2*x1+8, 2*x2+2]) #TODO define first derivative of␣
 ↪paraboloid
def hessian_f():
    return np.array([[2, 0],[0, 2]]) #TODO define hessian of paraboloid
```

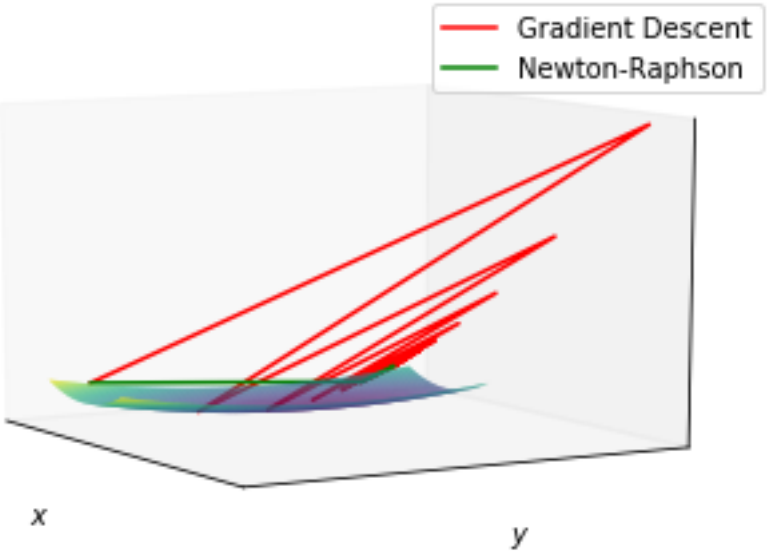
1.2.4 Take an initial guess of optimal solution and perform iterations to update it

[7]:
```
xin = np.array([8,3]) #initial guess of optimal solution
num_steps = 100
step_size = 0.9

x_curr_grad = xin #variable to be updated using gradient descent
x_curr_nr = xin #variable to be updated using Newton-Raphson method

####### These are logging variables for visualization ######
gradient_path = [xin]
nr_path = [xin]
fn_val_grad = [f(xin)]
fn_val_nr = [f(xin)]
############################################################

for step in range(num_steps):
```

```
    x_curr_grad = x_curr_grad - step_size*delta_f(x_curr_grad) #TODO Write␣
↪update rule using gradient descent
    x_curr_nr = x_curr_nr - step_size*np.linalg.
↪inv(hessian_f())@delta_f(x_curr_nr) #TODO Write update rule using␣
↪Newton-Raphson method
    #### Updating logs ###################
    gradient_path.append(x_curr_grad)
    nr_path.append(x_curr_nr)
    fn_val_grad.append(f(x_curr_grad))
    fn_val_nr.append(f(x_curr_nr))
    ####################################
```

1.2.5 Visualize the path followed by Gradient Descent and Newton-Raphson

[8]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
ax.plot([i[0] for i in gradient_path], [i[1] for i in␣
↪gradient_path],fn_val_grad,color='r')
ax.plot([i[0] for i in nr_path], [i[1] for i in nr_path],fn_val_nr,color='g')
ax.view_init(elev=10., azim=150)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
plt.legend(['Gradient Descent','Newton-Raphson'])
plt.show()
```



1.3 Testing with a halfpipe

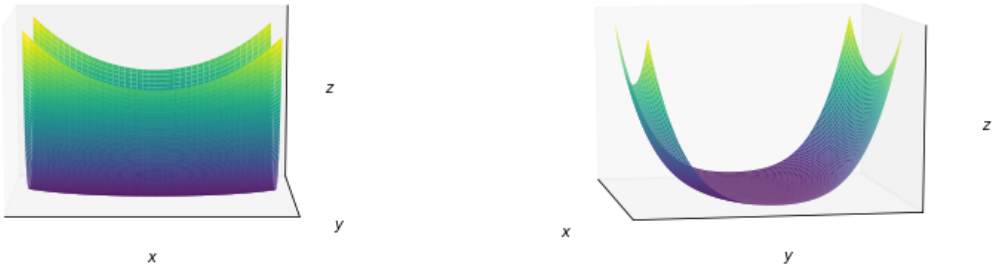
1.3.1 Define the Halfpipe function

[9]:
```
eps = 0.05
def halfpipe(x):
    (x1,x2) = x
    return np.cosh(eps*x1*x1 + x2*x2)
```

1.3.2 Visualize the halfpipe function in 3D

[10]:
```
x = np.arange(-2, 2, 0.1)
y = np.arange(-1, 1, 0.01)
xx, yy = np.meshgrid(x, y, sparse=True)
z = halfpipe([xx,yy])
fig = plt.figure(figsize=plt.figaspect(0.3))
ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
ax.view_init(elev=10., azim=270)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
```

```
ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
ax.view_init(elev=10., azim=170)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
plt.show()
```



1.3.3 Define first derivative and hessian of Halfpipe function

[11]:
```
def delta_halfpipe(x):
    (x1,x2) = x
    val = eps*x1*x1 + x2*x2
    return np.array([2*eps*x1*np.sinh(val), 2*x2*np.sinh(val)]) #TODO define␣
 ↪first derivative of halfpipe
def hessian_halfpipe(x):
    (x1,x2) = x
    val = eps*x1*x1 + x2*x2
    return np.array([2*eps*np.sinh(vals) + 4*eps*eps*x1*x1*np.cosh(val),␣
 ↪4*eps*x1*x2*np.cosh(val)],
                    [4*eps*x1*x2*np.cosh(val), 2*sinh(val) + 4*x2*x2*np.
 ↪cosh(val)]) #TODO define hessian of halfpipe
```

1.3.4 Take an initial guess of optimal solution and perform iterations to update it

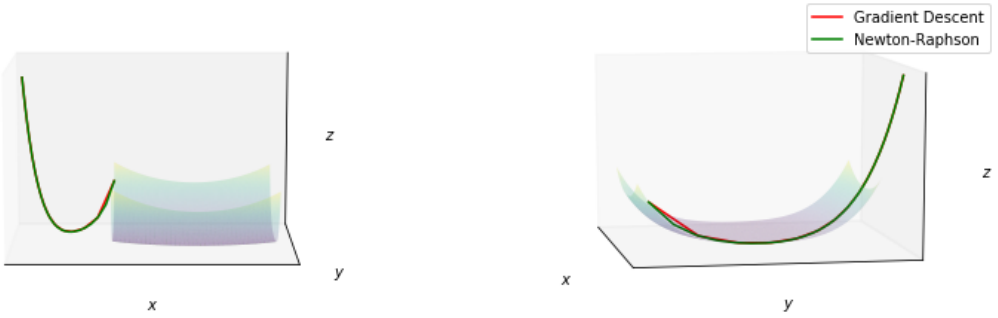
[16]:
```
xin = np.array([-2,0.9])
num_steps = 5000
step_size = 0.1
```

```
x_curr_grad = xin #variable to be updated using gradient descent
x_curr_nr = xin #variable to be updated using Newton-Raphson method

####### These are logging variables for visualization ######
gradient_path = [xin]
nr_path = [xin]
fn_val_grad = [halfpipe(xin)]
fn_val_nr = [halfpipe(xin)]
############################################################

for step in range(num_steps):
    x_curr_grad = x_curr_grad - step_size*delta_f(x_curr_grad) #TODO Write␣
 ↪update rule using gradient descent
    x_curr_nr = x_curr_nr - step_size*np.linalg.
 ↪inv(hessian_f())@delta_f(x_curr_nr) #TODO Write update rule using␣
 ↪Newton-Raphson method

    #### Updating logs ###################
    gradient_path.append(x_curr_grad)
    nr_path.append(x_curr_nr)
    fn_val_grad.append(halfpipe(x_curr_grad))
    fn_val_nr.append(halfpipe(x_curr_nr))
    ######################################
```

1.3.5 Visualize the path followed by Gradient Descent and Newton-Raphson

[17]:
```
fig = plt.figure(figsize=plt.figaspect(0.3))
ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot([i[0] for i in gradient_path], [i[1] for i in␣
 ↪gradient_path],fn_val_grad,color='r')
ax.plot([i[0] for i in nr_path], [i[1] for i in nr_path],fn_val_nr,color='g')
ax.view_init(elev=10., azim=270)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none',alpha=0.2)
ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot([i[0] for i in gradient_path], [i[1] for i in␣
 ↪gradient_path],fn_val_grad,color='r')
ax.plot([i[0] for i in nr_path], [i[1] for i in nr_path],fn_val_nr,color='g')
ax.view_init(elev=10., azim=170)
ax.set_xlabel('$x$')
```

```
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
ax.get_xaxis().set_ticks([])
ax.get_yaxis().set_ticks([])
ax.get_zaxis().set_ticks([])
ax.plot_surface(xx,yy,z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none',alpha=0.2)
plt.legend(['Gradient Descent','Newton-Raphson'])
plt.show()
```



[ ]:

image_transform_fundamentals

November 1, 2019

[6]:
```
import numpy as np
import matplotlib.pyplot as plt
import pywt
```

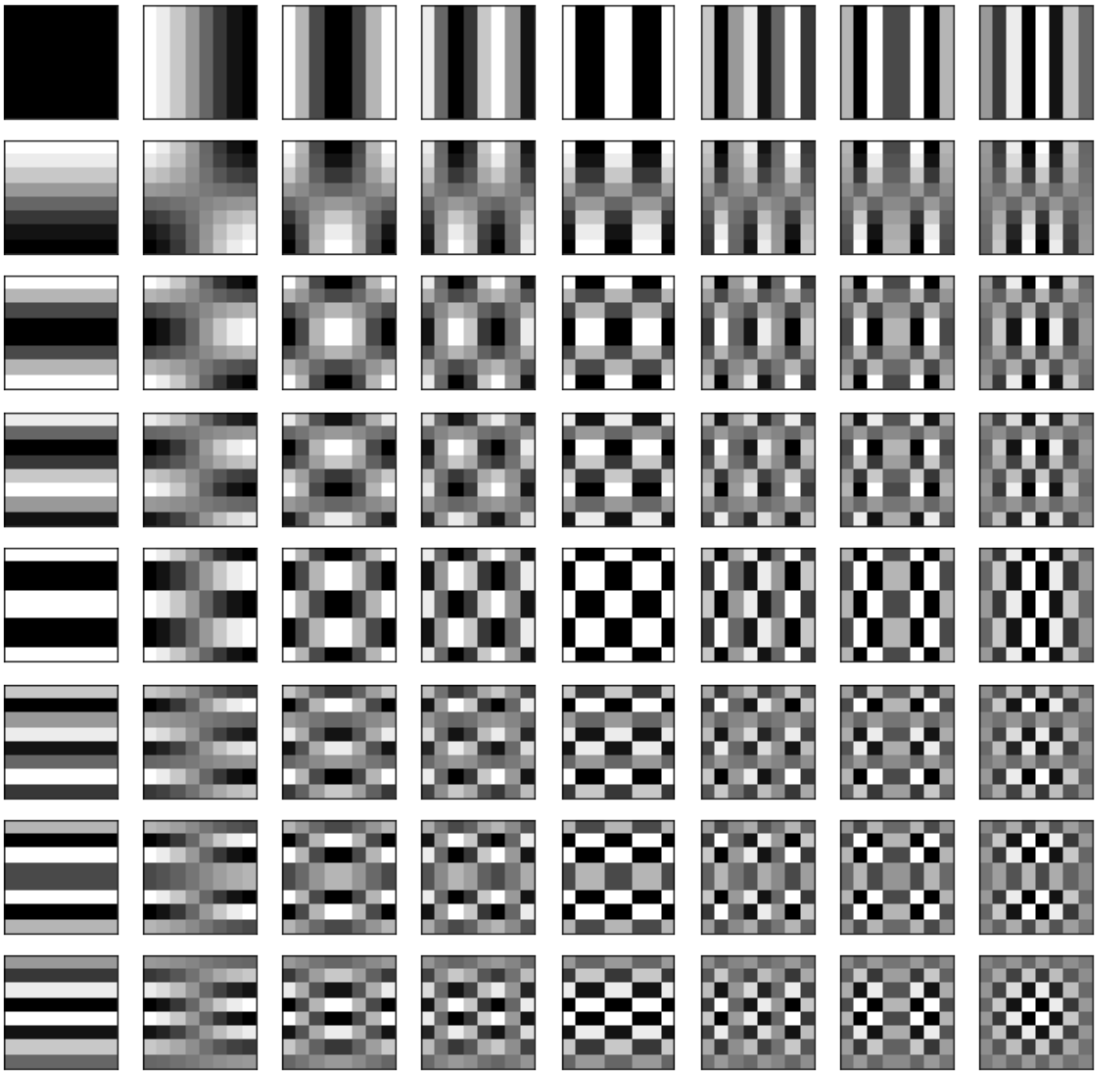
1 Frequency and Basis Function Intuition

We've discussed the Fourier transform as a change of basis from a spacial domain to a frequency domain, but what does this really mean? When we talk about frequency in images, we refer to how quickly the values of the pixels change with respect to their location in the image. If every pixel is the same, the frequency present is the lowest possible, because there is no variation in pixel value with respect to their location. If the pixels values keep changing (from 0 to 1 and back), the frequency present is the highest possible.

To help you visulize the meaning of frequency, we will use the Discrete Cosine Transform (DCT), which is similar to the Fourier transform, except it has no imaginary component. Run the cells below to plot the DCT basis images.

[7]:
```
def dct_basis(u,v):
    basis = np.zeros((8,8))
    for x in range(8):
        for y in range(8):
            basis[x,y] = np.cos(np.pi*(2*x+1)*u/16)*np.cos(np.pi*(2*y+1)*v/16)
    return basis
```

[8]:
```
plt.figure(figsize=(10,10))
for i in range(8):
    for j in range(8):
        plt.subplot(8, 8, 8*i+j+1)
        cur_axes = plt.gca()
        cur_axes.axes.get_xaxis().set_ticks([])
        cur_axes.axes.get_yaxis().set_ticks([])
        plt.imshow(dct_basis(i,j), cmap='gray')
```



The above images are the DCT basis images for 8x8 images. From left to right, we have the frequency increasing along the x axis of the image, and from top to bottom, we have the frequency increasing along the y axis. The top left has a frequency of 0 - the pixels in the image do not vary at all. And, in the bottom right, we have the highest frequency image - each adjacent pixel in the x and y directions is the opposite of that pixel.

You can think of every 8x8 image as the linear combination of these basis functions, and the DCT/DFT coeffecients as the weights of the linear combination.

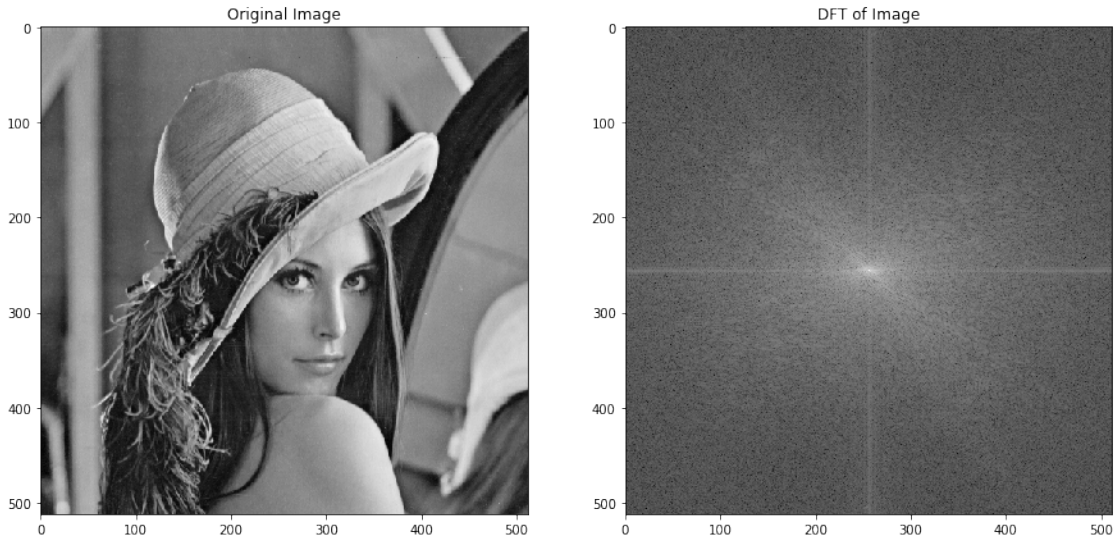
2 DFT and Image Manipulation

Let's now experiment with the DFT, and see how we can use it to manipulate images. Before we get the manipulation, let's read an image and take a look at its DFT.

[9]:
```
def fft2c(im):
    return np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(im)))

def ifft2c(im):
    return np.fft.fftshift(np.fft.ifft2(np.fft.ifftshift(im)))
```

[12]:
```
# The use of np.mean just converts RGB to greyscale by averaging the values in
 ↪those channels
im = np.mean(plt.imread("lenna.png"), axis=2)
# im = np.mean(plt.imread("Lichtenstein_img_processing_test.png"), axis=2)
```

[13]:
```
plt.figure(figsize=(15,10))
plt.subplot(1, 2, 1)
plt.imshow(im, cmap='gray')
plt.title("Original Image")

ft_im = fft2c(im)

plt.subplot(1, 2, 2)
plt.title("DFT of Image")
plt.imshow(np.log10(np.abs(ft_im)), cmap='gray')
plt.show()
```



The above DFT image shows the coeffecients of the DFT basis functions, where the brightness of each pixel indicates the magnitude of the corresponding basis function. The lowest frequency coeffecients are in the center of the image, and the highest frequeny coeffecients are towards the edges.

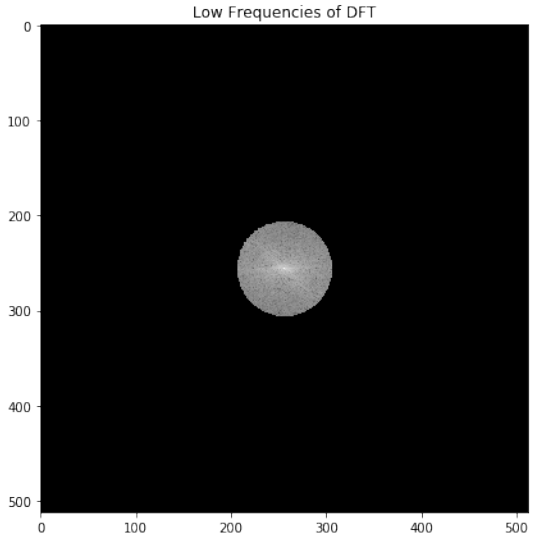
Let's now see how we can manipulate these coeffecients, and how those changes translate to changes in the image. We will do this by using the two functions below, `lpf` and `hpf`, which stand for low pass filter and high pass filter, respectively. The low pass filter "passes" low frequency

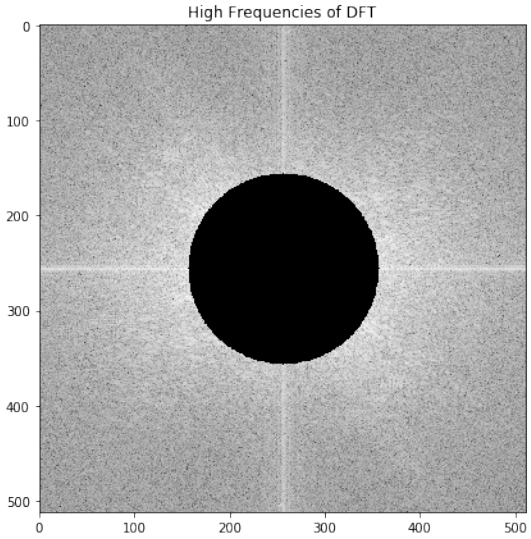
coeffecients (zeros out high frequency coefficients), while the high pass filter does the opposite (zeros out low frequency coeffecients).

Let's take a look at what the coeffecients look like after applying the filters.

[14]:
```
def lpf(im, r):
    filt_im = np.zeros(im.shape, dtype=np.complex64)
    c = np.array(im.shape)//2
    for i in range(im.shape[0]):
        for j in range(im.shape[1]):
            pt = np.array([i,j])
            d = np.linalg.norm(pt-c)
            if d < r:
                filt_im[i,j] = im[i,j]
    return filt_im

def hpf(im, r):
    filt_im = np.zeros(im.shape, dtype=np.complex64)
    c = np.array(im.shape)//2
    for i in range(im.shape[0]):
        for j in range(im.shape[1]):
            pt = np.array([i,j])
            d = np.linalg.norm(pt-c)
            if d > r:
                filt_im[i,j] = im[i,j]
    return filt_im
```

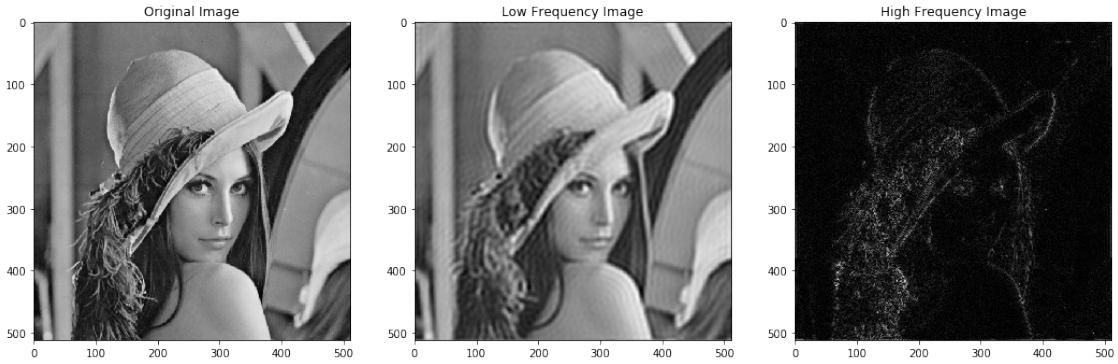
[15]:
```
lpf_im = lpf(ft_im, 50)
hpf_im = hpf(ft_im, 100)

plt.figure(figsize=(15,10))
plt.subplot(1, 2, 1)
plt.title("Low Frequencies of DFT")
plt.imshow(np.log10(np.abs(lpf_im)+0.01), cmap='gray')

plt.subplot(1, 2, 2)
plt.title("High Frequencies of DFT")
plt.imshow(np.log10(np.abs(hpf_im)+0.01), cmap='gray')
plt.show()
```





Now, if we take the inverse DFT of the filtered coeffecients, we can see how the images have changed

[19]:
```
plt.figure(figsize=(18,10))
plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(im, cmap='gray')

plt.subplot(1, 3, 2)
plt.title("Low Frequency Image")
plt.imshow(np.abs(ifft2c(lpf_im)), cmap='gray')

plt.subplot(1, 3, 3)
plt.title("High Frequency Image")
plt.imshow(np.abs(ifft2c(hpf_im)), cmap='gray')
plt.show()
```



In the low frequency image, we see more or less the same image, but with a "ringing" effect. What is important to note is that, even though we got a rid of a lot of coeffecients (we only kept around 3%), the image more or less looks the same. This idea is the key to using the Fourier transform for image compression, which we will take a closer look at next week.

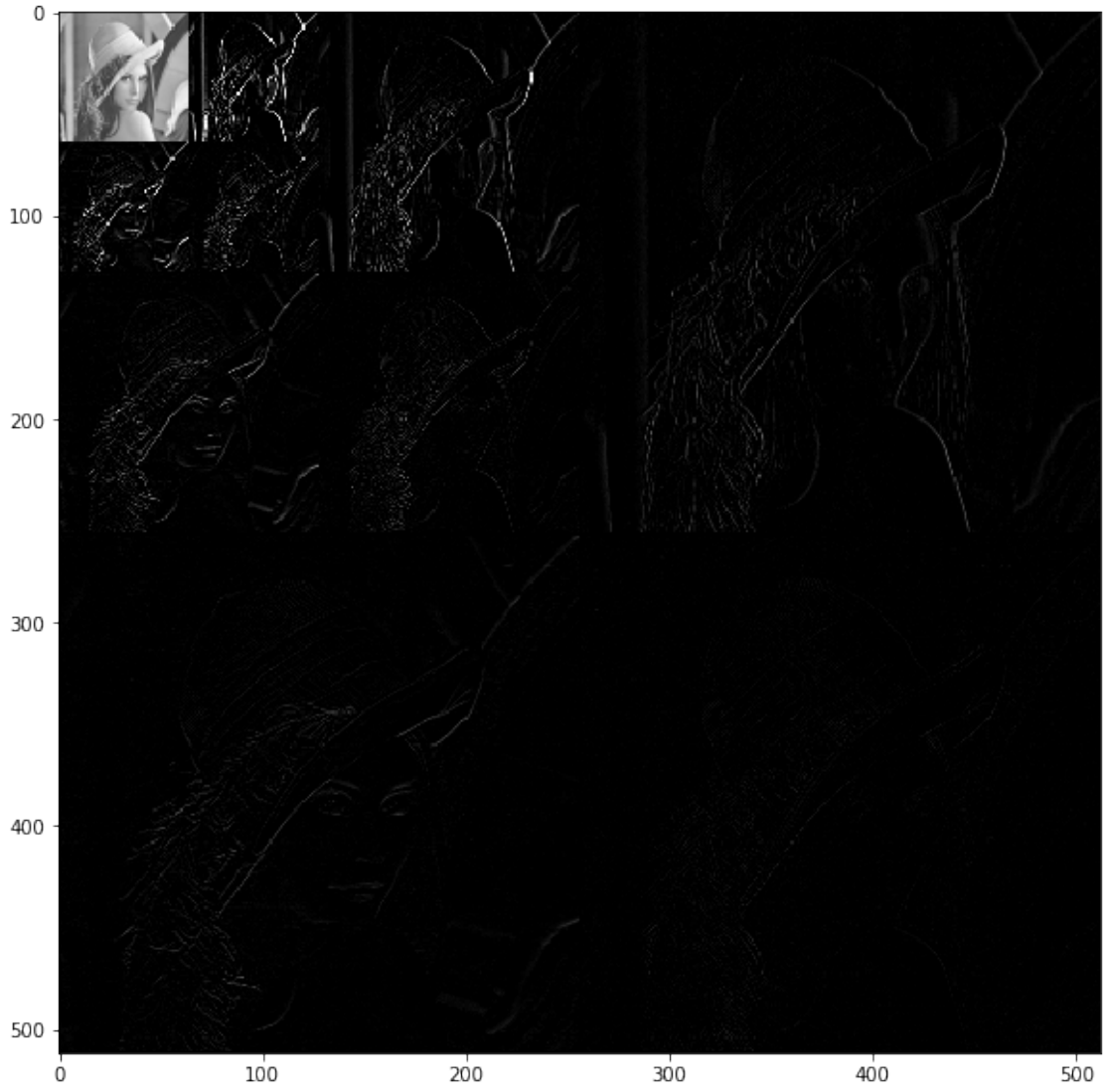
The high frequency image highlights the edges in our image (because edges are where pixels vary the most), and this kind of technique can be used for object or edge detection.

3 Wavelet Transform

Run the cell below to see the wavelet transform of our image.

[17]:
```
wvlt_im = pywt.wavedec2(im, 'haar', level=3)
wvlt_im[0] /= np.abs(wvlt_im[0]).max()
data, slices = pywt.coeffs_to_array(wvlt_im)
```

[18]:
```
plt.figure(figsize=(15,10))
plt.imshow(data, cmap='gray', vmin=0, vmax=1)
plt.show()
```



To reiterate, the way this transform works is well beyond the scope of this class. What is important to note here is that the majority of the information is in the top left of our image, and the rest of the image is quite sparse. We will use this property of the wavelet transform next week for image compression.