Line Search (Part I)

WQD7011 Numerical Optimization

Recall previous slides...

- We had learned:
  - Differences between **constrained and unconstrained optimization**.
  - Differences between **global and local solutions**.
  - Three **types of local minimizer**.
  - **Convex**
  - **Recognizing local minimizer** – Taylor's Approximation + optimality conditions for sufficient and necessary

Introduction to Line Search Methods

Introduction to Line Search Methods

How to find your way back to the lodge?

Introduction to Line Search Methods

What is the current available information?
Lodge located at the base. ← minimization function

- Pick a downward direction
- Until meeting upward direction, change downward direction again
- Repeat process until reaching base.

Linear Search

Generic approach for Line Search Algorithm

- Initial **guess** at the minimizer x_0
- Iteratively produces $x_1, x_2, x_3, \ldots$ and until it is converged (hopefully) to minimizer x_k.
- Two steps from x_k to x_{k+1}:
  - Select **search direction p_k** to proceed with certain point
  - Specify **step size α_k** to traverse along this direction.
- **Next point** is determined by:

$$x_{k+1} = x_k + \alpha_k p_k$$

  where positive scalar α_k is the step length / step size.

- Success of a line search method depends on **effective choices of both the direction p_k and α_k.**

Search Direction I – Gradient Descent / Steepest Descent

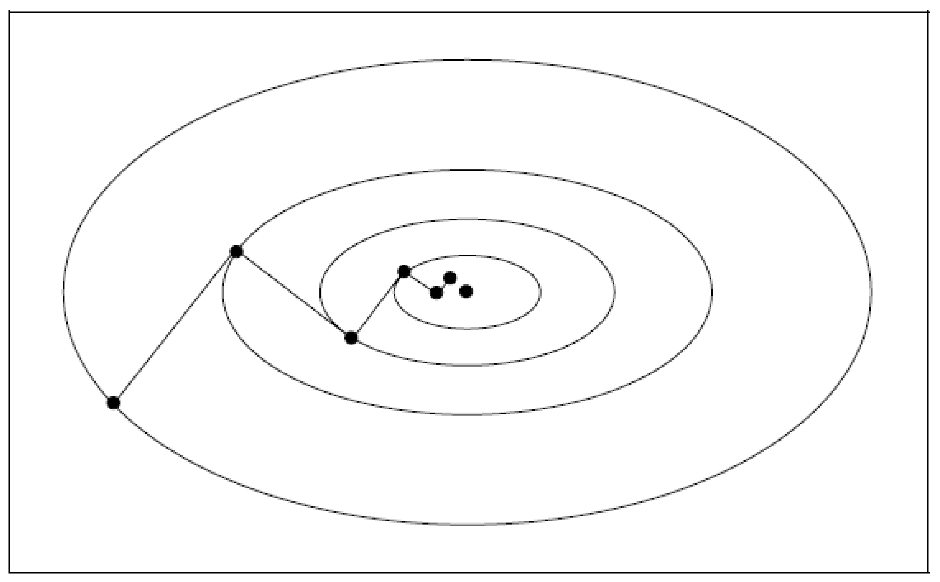


Fig. 3.7 Steepest descent steps.

- **Gradient** of function at given point gives the fastest increasing direction.
- In minimization, we use the **negative of the gradient points** (i.e., fastest decreasing direction):

$$p_k = -\nabla f_k$$

- to get **nearer to minimizer soonest possible**.

Search Direction II – Newton's Method

- Utilize both **gradient and Hessian matrix** (provide curvature information of function at a point) to select search direction:

$$p_k = -\nabla^2 f_k^{-1} \nabla f_k$$

- where $\nabla^2 f_k^{-1}$ is the inverse of Hessian matrix of f at point x.

- Comparison with Gradient Descent method?

Rate of Convergence

- Optimization algorithm with good convergence properties:
  - p_k does not tend to become orthogonal to the gradient ∇f_k (steepest descent steps are taken regularly)
  - Simply compute $\cos\theta_k$ at every iteration, turn p_k towards steepest descent direction if $\cos\theta_k$ is smaller than some preselected constant $\delta > 0$.
- Easy?

Rate of Convergence

- Undesirable. WHY?
- Angle test said YES. THEN WHY?
- **Reason:**
  - May impede a fast rate of convergence, because for problems with an ill-conditioned Hessian → maybe necessary to produce search directions that are almost orthogonal to the gradient + inappropriate choice of parameter δ may cause such steps to be rejected.

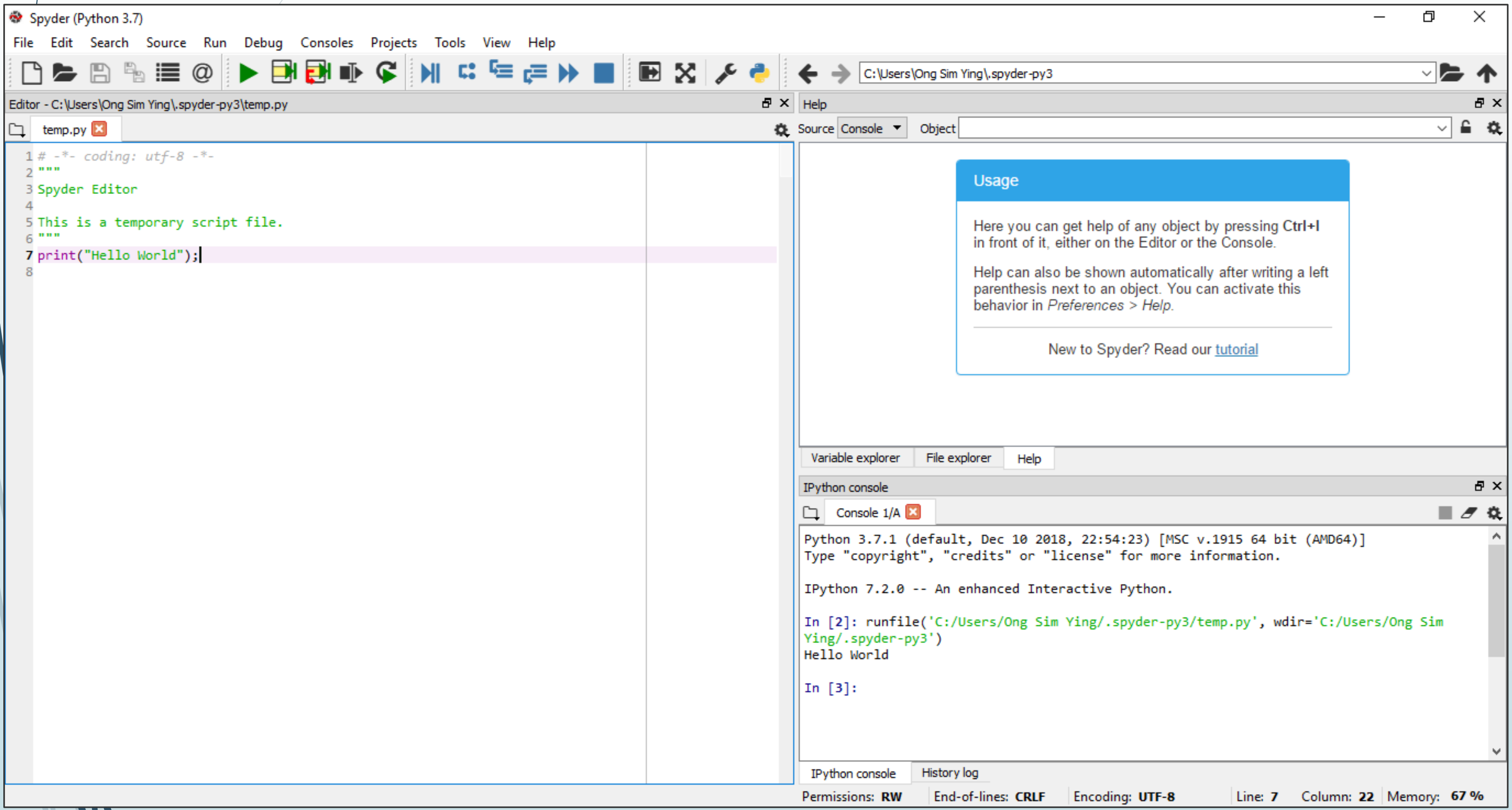
Rate of Convergence

- **Algorithmic strategies that achieve rapid convergence can sometimes conflict with the requirements of global convergence.**
- Challenge?
- Design algorithms that incorporate both properties:
  - Good global convergence guarantees
  - Rapid rate of convergence

Practical Lab

- Launch your Anaconda Navigator.
- Choose Spyder ← GUI platform to Python.

Practical Lab



Gradient Descent Method

- Create a new file and named it as GradientDescent.py.
- Import these two libraries:

```
import numpy as np
import matplotlib.pyplot as plt
```

- numpy library documentation: https://docs.scipy.org/doc/numpy/user/whatisnumpy.html
- matplotlib library documentation: https://matplotlib.org/users/index.html

Gradient Descent Method

- Create a function to store our objective function.
- But, WHY need to create function? What is a function?

```
def func(x):
    return 100*np.square(np.square(x[0])-x[1])+np.square(x[0]-1)
```

Gradient Descent Method

- Create another function to store our first order and second order derivative functions.
- Utilized to determine our steepest descent.

```
def dfunc(x):
    df1 = 400*x[0]*(np.square(x[0])-x[1])+2*(x[0]-1)
    df2 = -200*(np.square(x[0])-x[1])
    return np.array([df1, df2])
```

Gradient Descent Method

- A new function to calculate run our Gradient Descent method.
- Let's define few important variables:

```
def grad(x, max_int):
    miter = 1
    step = .0001/miter
    vals = []
    objectfs = []
```

Gradient Descent Method

- Continue with grad() function, integrating the function with the functions created in prior:

```
    while miter <= max_int:
        vals.append(x)
        objectfs.append(func(x))
        temp = x-step*dfunc(x)
        if np.abs(func(temp)-func(x))>0.01:
            x = temp
        else:
            break
        print(x, func(x), miter)
        miter += 1
    return vals, objectfs, miter
```

Gradient Descent Method

- Let's initiate a starting point.
- And call the Gradient Descent Method!

```
start = [5, 5]
val, objectf, iters = grad(start, 50)
```

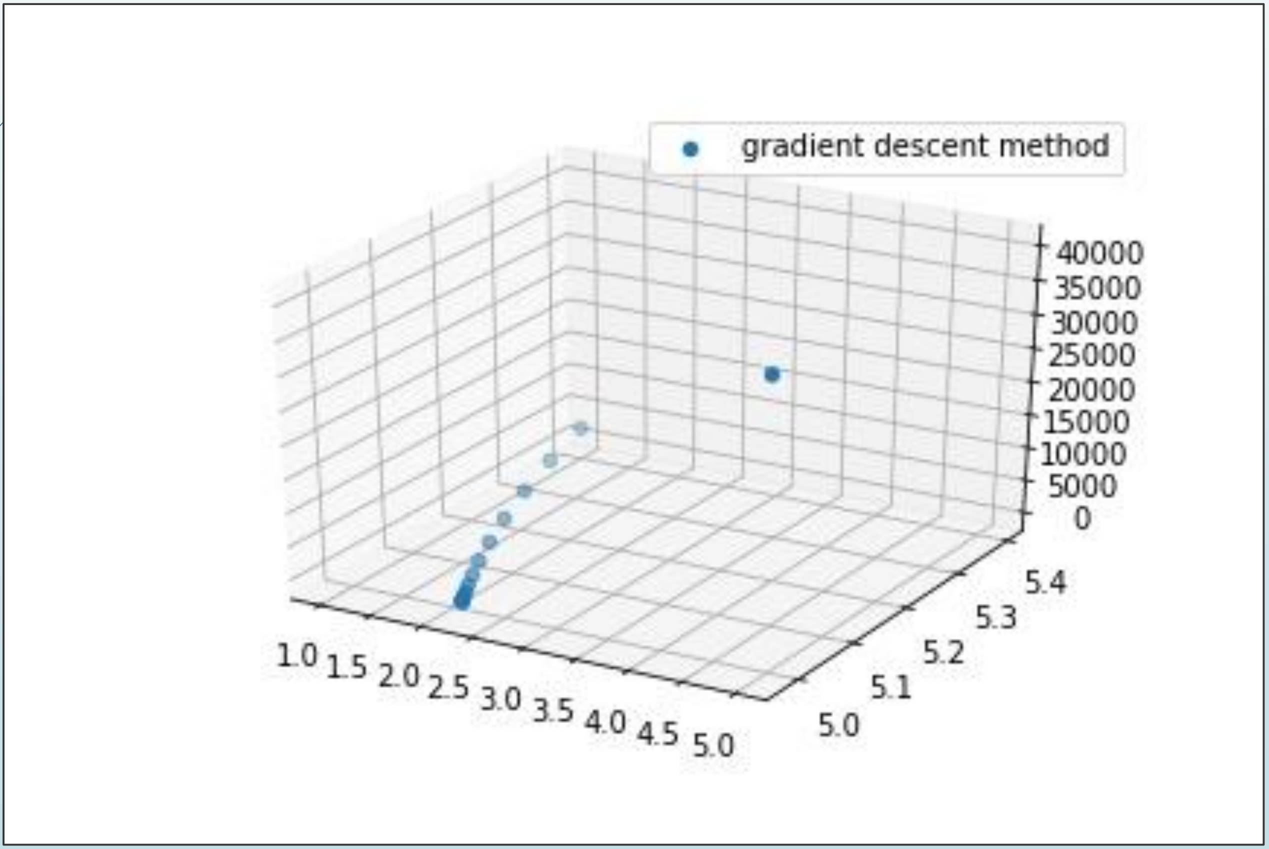
Gradient Descent Method

- Visualize your result in 3D plot to understand the method better:

```
x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
z = np.array(objectf)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x, y, z, label='gradient descent method')
ax.legend()
plt.savefig('GradientDescent.jpg')
```

Gradient Descent Method

- Result:



Newton Method

- Let's create a new file called Newton.py.
- Copy all our codes from GradientDescent.py into Newton.py.
- Add another library here:

```
from numpy.linalg import inv
```

Newton Method

- Create another function to calculate Hessian matrix:

```
def invhess(x):
    df11 = 1200*np.square(x[0])-400*x[1]+2
    df12 = -400*x[0]
    df21 = -400*x[0]
    df22 = 200
    hess = np.array([[df11, df12], [df21, df22]])
    return inv(hess)
```

Newton Method

- Change your temp assignment:

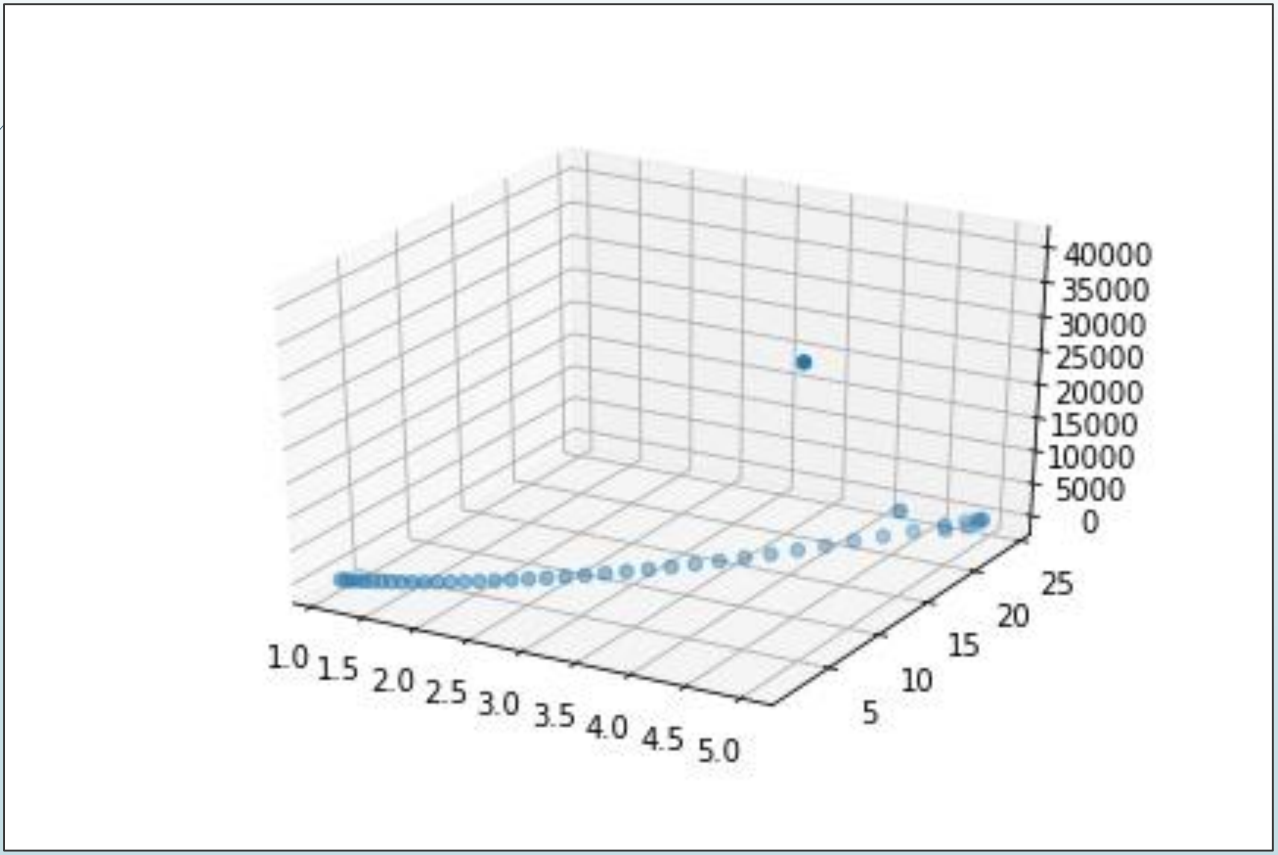
```
temp = x-step*(invhess(x).dot(dfunc(x)))
```

- Then, change your graph information:

```
ax.scatter(x, y, z, label='newton method')
plt.savefig('newton.jpg')
```

Newton Method

- Result:



Exercises

1. What is the role of second derivative function in Gradient Descent method?
2. Explain invHess() in Newton.py, in line manner.
3. Calculate the time utilized to run both Gradient Descent method and Newton method when start is set to [5, 5].
4. What are the initial observations from the both results obtained when start is set to [5, 5]?
5. If start is set to [15, 15], rerun both methods. What are the observations now?
6. Compare and contrast between Gradient Descent method and Newton method.

Complete the answers and submit to spectrum before next class.