



Line Search (Part II)

WQD7011 Numerical Optimization



Recall previous slides...

- We had learned:
 - Introduction to line search
 - Direction searching using gradient descent method and newton method



Exercises (Discussion)

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.

Step Length (Introduction)

- Trade-off in calculating step length, α_k :
 - Choose α_k to give substantial reduction of f
 - Do not want to spend too much time making the choice.
- Ideal step length?
 - Put it into an optimization algorithm, find the global minimizer.
$$\phi(\alpha) = f(x_k + \alpha p_k), \alpha > 0$$
- BUT, too expensive:
 - Too many evaluations of objective function f and gradient ∇f .

Step Length (Introduction)

- More practical way?
 - Perform **inexact line search** to identify a step length that achieve adequate reductions in f at minimal cost.
- What is Exact Line Search?
 - Search through ALL possible α to find global minimizer:

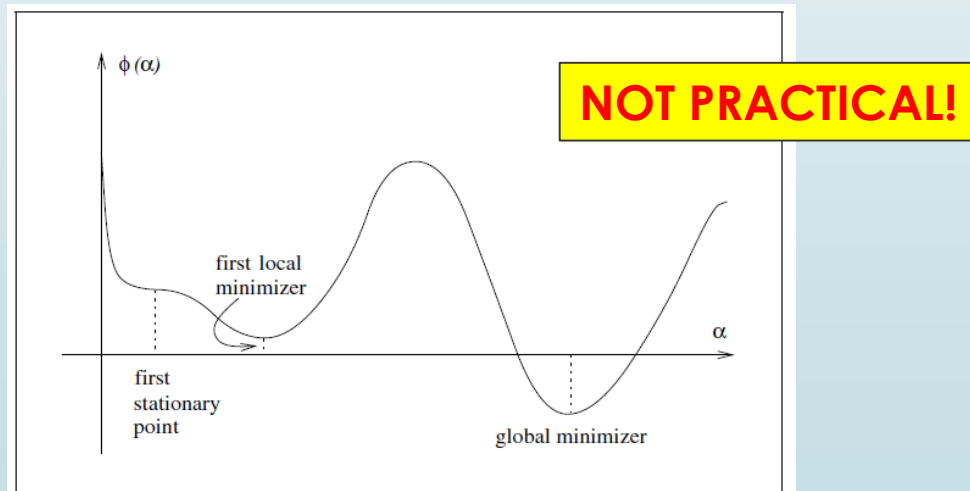


Figure 3.1 The ideal step length is the global minimizer.



Line Search in Finding Step Length

- **Bracketing Phase:**
 - Finds an interval containing desirable step lengths
- **Bisection or Interpolation Phase:**
 - Computes a good step length within this interval

Termination Conditions for Line Search Algorithm

- The simplest method: simple condition of decrement

$$f(x_{k+1}) = f(x_k + \alpha p_k) < f(x_k)$$

- and take multiple iterations to find the minima.
- **Backtracking** line search algorithm to find the step length.

Backtracking Line Search

1. Obtain an α_{init} , then let $\alpha_j = \alpha_{init}$ and initialize $j = 0$.
2. Until $f(x_k + \alpha_j p_k) < f(x_k)$, repeat
 - a) Set $\alpha_{j+1} = \rho \alpha_j$, where ρ is in $(0, 1)$.
 - b) Increase j by 1.
3. Set $\alpha = \alpha_j$.

Problem: May converge before reaching minima.
Thus, we need a **SUFFICIENT DECREASE** condition.

Backtracking Line Search

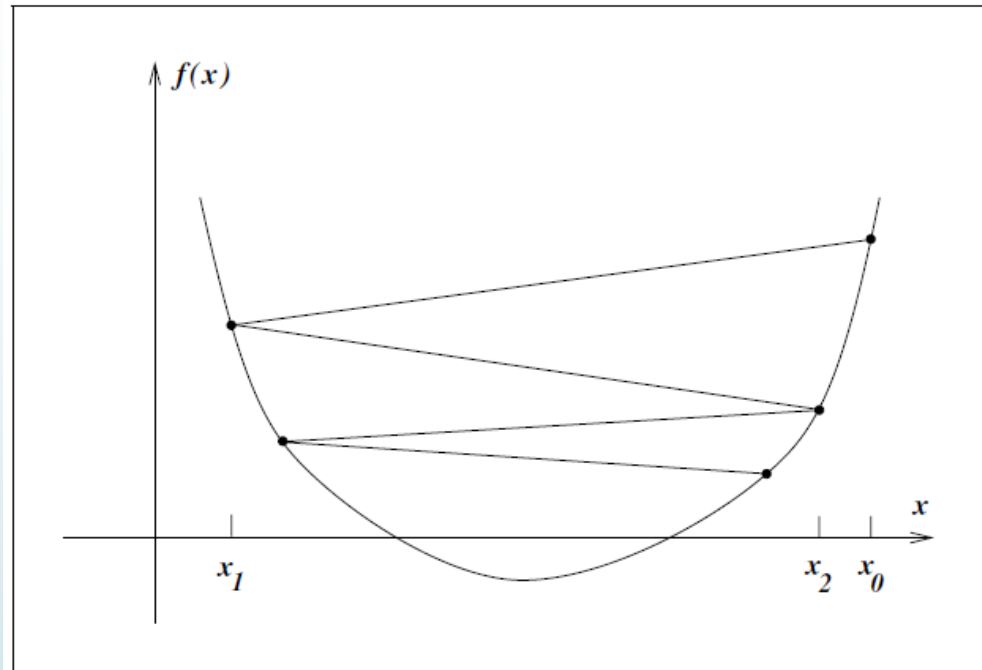


Figure 3.2 Insufficient reduction in f .

Problem: May converge before reaching minima.
Thus, we need a **SUFFICIENT DECREASE** condition.

Armijo Condition

- Ensure α gives sufficient decrease in f (when finding step length), Armijo condition is required:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k$$

- Here, c_1 is a positive constant in $(0, 1)$.
- Practically, c_1 is a small number, such as 10^{-4} .
- f_k is a short hand of $f(x_k)$.
- BUT,

Why “+”???

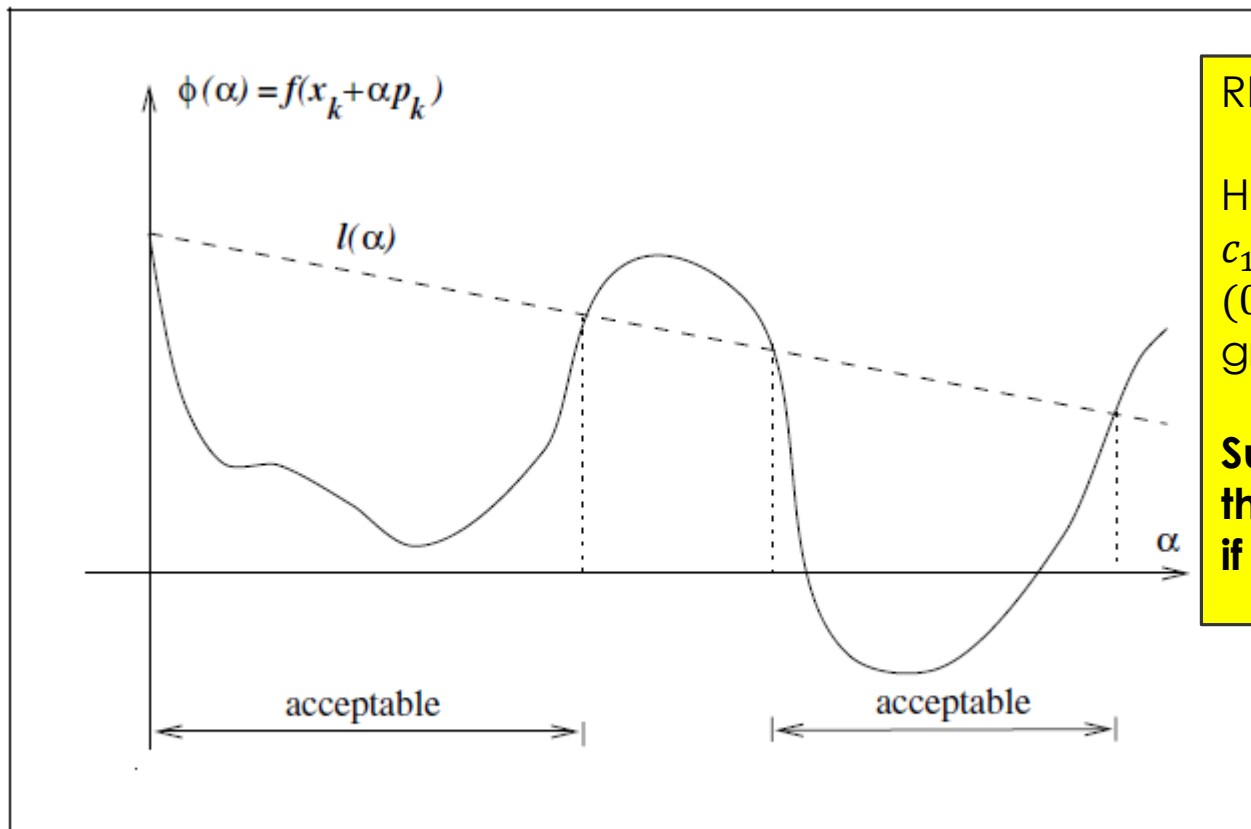
- Note: RHS of the equation is a linear function with α as the slope.

Backtracking Line Search with Armijo Condition

1. Obtain an α_{init} , then let $\alpha_j = \alpha_{init}$ and initialize $j = 0$.
2. Until $f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k$, repeat
 - a) Set $\alpha_{j+1} = \rho \alpha_j$, where ρ is in $(0, 1)$.
 - b) Increase j by 1.
3. Set $\alpha = \alpha_j$.

The contraction factor ρ is often allowed to vary at each iteration.

Armijo Condition



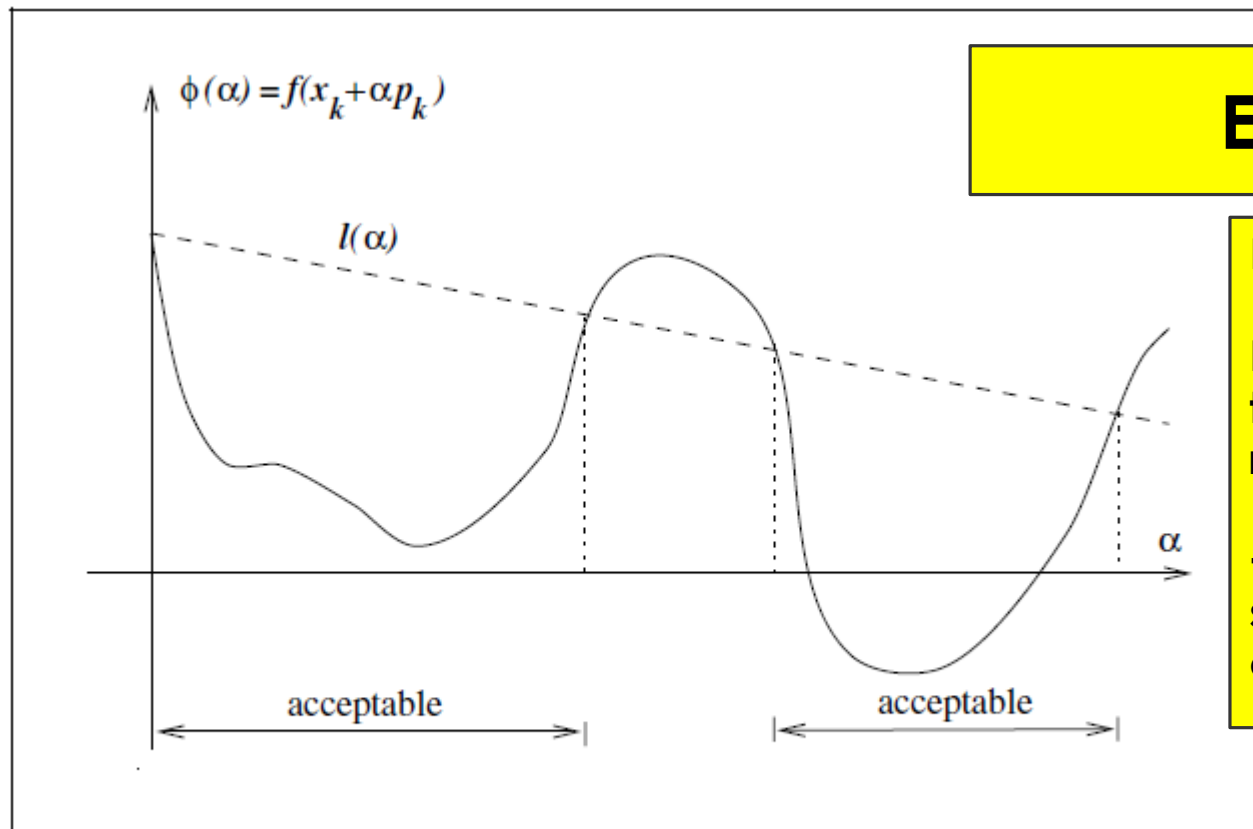
RHS is denoted as $l(\alpha)$.

Has negative slope for $c_1 \nabla f_k^T p_k$, but because $c_1 \in (0, 1)$, so it lies above the graph for ϕ .

Sufficient condition stated that α is ONLY ACCEPTABLE if $\phi(\alpha) \leq l(\alpha)$.

Figure 3.3 Sufficient decrease condition.

Armijo Condition



Enough?

NO.

Not enough to ensure that the algorithm make reasonable progress.

++ It is satisfied by ALL sufficiently small values of α .

Figure 3.3 Sufficient decrease condition.

Curvature Condition

- If we DON'T WANT small value of α , curvature condition can be applied:

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k$$

- where c_2 is a constant in $(c_1, 1)$, typically close to 1 (e.g., 0.9).
- This condition means the gradient at x_{k+1} (if the step with α is taken) must be greater than or equal to c_2 times the initial gradient.
- Note: the gradient near the minima should be “less negative” than the gradient at the further point.

Curvature Condition

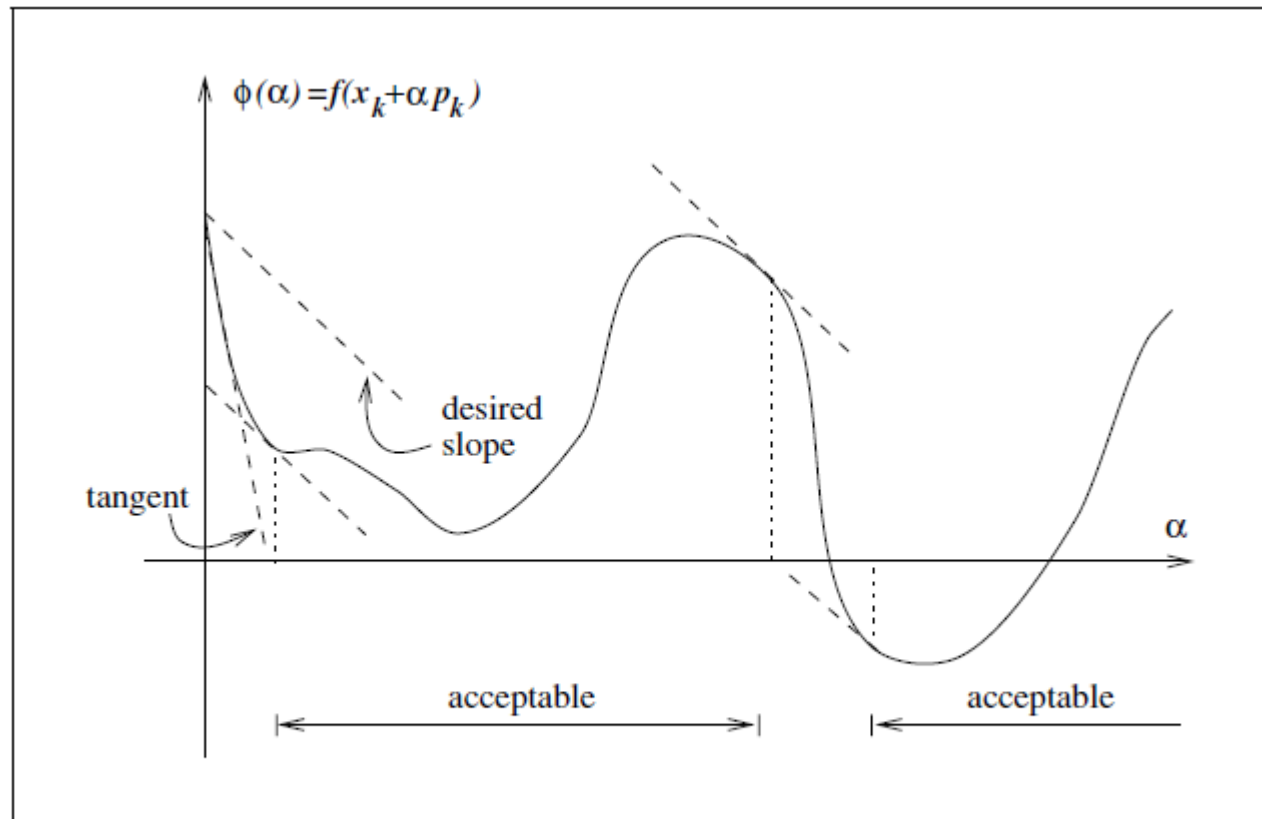


Figure 3.4 The curvature condition.

Wolfe Condition

- Armijo condition and curvature conditions are known collectively as the **Wolfe Conditions**:

$$\begin{aligned} f(x_k + \alpha p_k) &\leq f(x_k) + c_1 \alpha \nabla f_k^T p_k \\ \nabla f(x_k + \alpha_k p_k)^T p_k &\geq c_2 \nabla f_k^T p_k \end{aligned}$$

- with $0 < c_1 < c_2 < 1$

Wolfe Condition

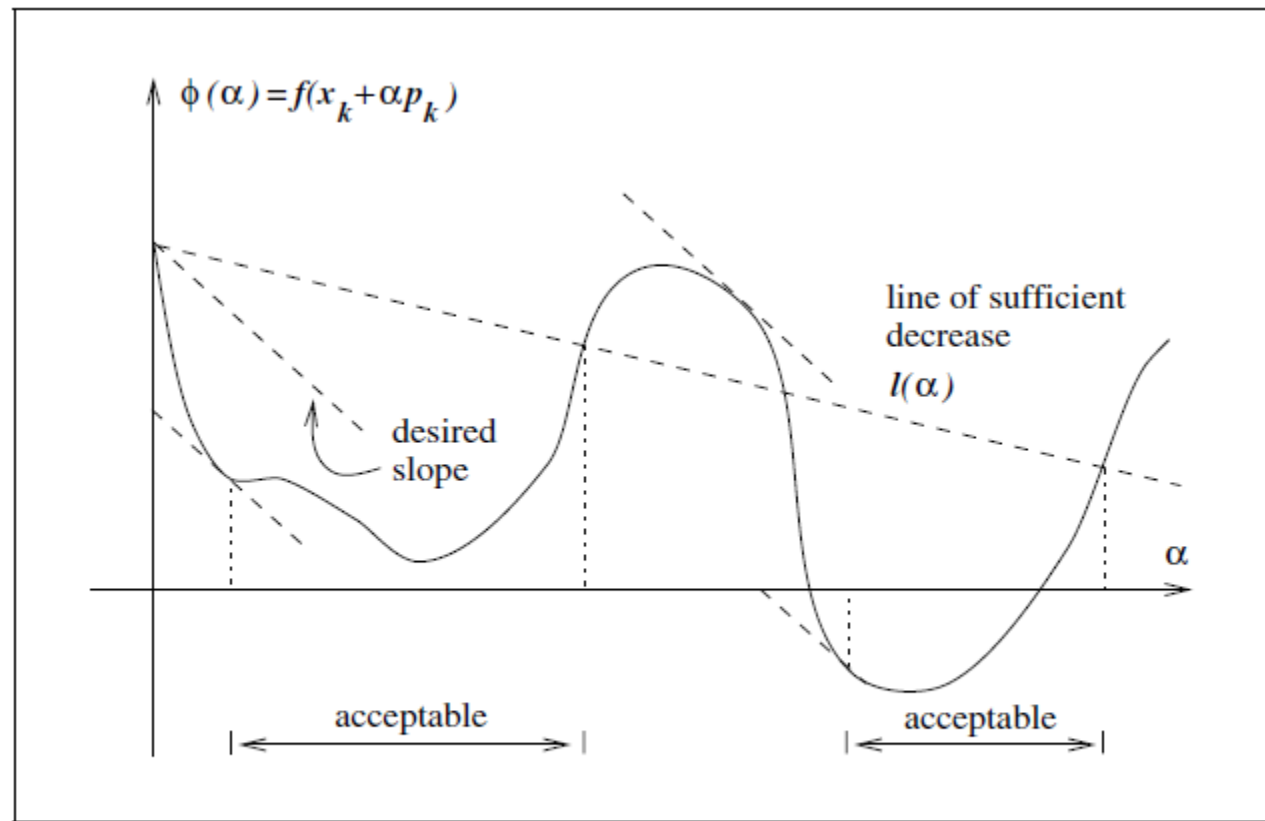


Figure 3.5 Step lengths satisfying the Wolfe conditions.

Goldstein Conditions

- ▶ Like the Wolfe conditions, the Goldstein Conditions ensure that the step length α achieves sufficient decrease BUT IS NOT TOO SHORT.
- ▶ With $0 < c < 0.5$:

$$f(x_k) + (1 - c)\alpha_k \nabla f_k^T p_k \leq f(x_k + \alpha_k p_k) \leq f(x_k) + c\alpha_k \nabla f_k^T p_k$$

Control
Step Length

Ensure sufficient
Decrease condition

Goldstein Conditions

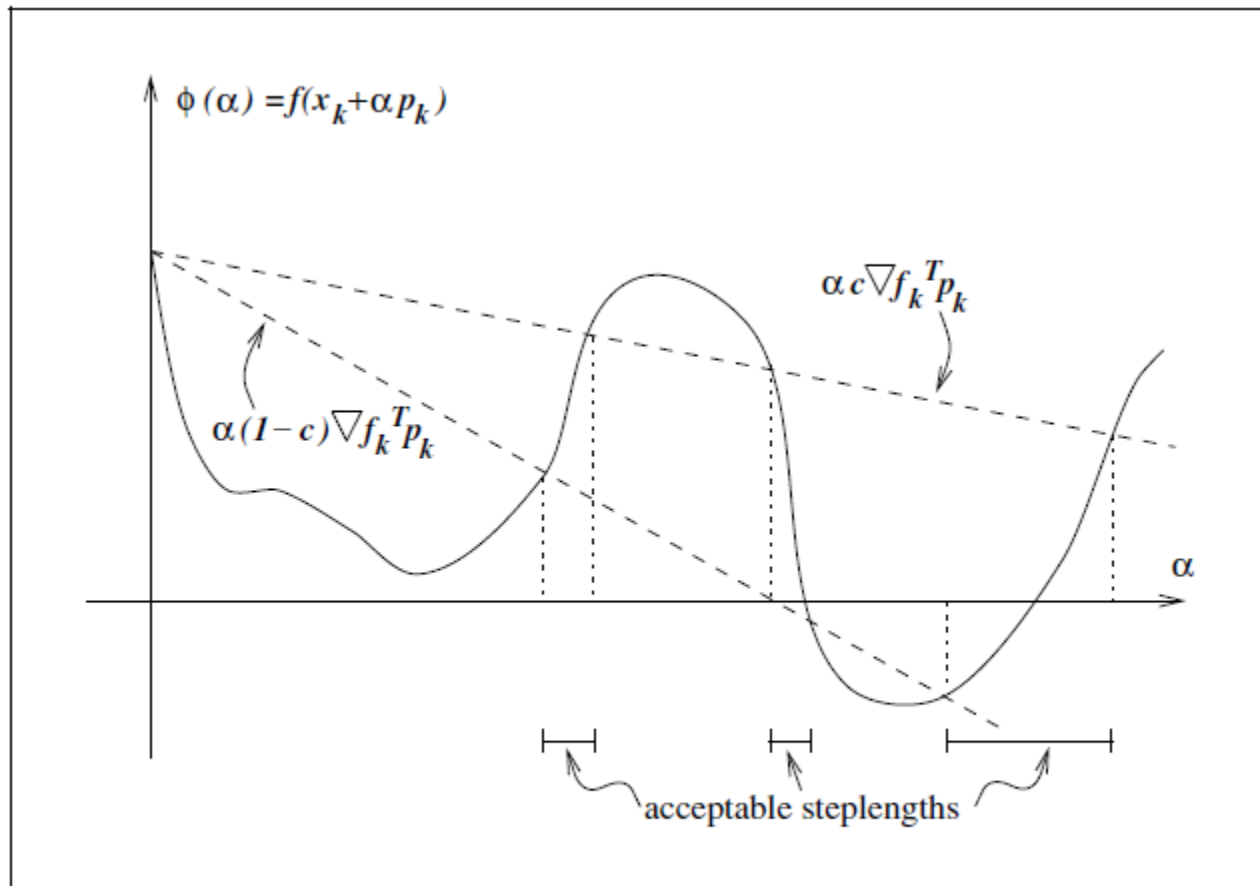



Figure 3.6 The Goldstein conditions.



Discussion on Group Assignment 1

- Refer to the assignment instructions in Spectrum.
- 



Practical Exercise

- Using the same objective function and first order derivatives, we will calculate the step length using:
 - Armijo conditions
 - Wolfe conditions

Armijo Conditions

- Import libraries

```
import random
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

Armijo Conditions

- Objective function and first-order derivatives:

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

# first order derivatives of the function
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])
```

Armijo Conditions

- The Armijo function:

```
# the armijo algorithm
def armijo(valf, grad, niters):
    #beta = random.random()
    #sigma = random.uniform(0, .5)
    beta = 0.25
    sigma = 0.25
    (miter, iter_conv) = (0, 0)
    conval = [0,0]
    val = []
    objectf = []
    val.append(valf)
    objectf.append(func(valf))
    while miter <= niters:
        leftf = func(valf+np.power(beta, miter)*grad)
        rightf = func(valf) + sigma*np.power(beta, miter)*dfunc(valf).dot(grad)
        if leftf-rightf <= 0:
            iter_conv = miter
            conval = valf+np.power(beta, iter_conv)*grad
            break
        miter += 1
        val.append(conval)
        objectf.append(func(conval))
    return conval, func(conval), iter_conv, val, objectf
```


Armijo Conditions

- Initialization, Run and Plotting:

```
# initialization
start = np.array([-0.3, 0.1])
direction = np.array([1, -2])
maximum_iterations = 30

converge_value, minimal, no_iter, val, objf = armijo(start, direction, maximum_iterations)
print("The value, minimal and number of iterations are " + str(converge_value) + \
      ", " + str(minimal) + ", " + str(no_iter))
x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
z = np.array(objf)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x, y, z, label='Armijo Rule')
ax.legend()
plt.savefig('armijo.jpg')
```

Wolfe Conditions

- Use back the same objective functions, derivatives and libraries from the Armijo Condition practical.
- Add in new method for Wolfe Conditions:

```
def wolfe(valf, direction, max_iter):  
    (alpha, beta, step, c1, c2) = (0, 1000, 5.0, 0.15, 0.3)  
    i = 0  
    stop_iter = 0  
    stop_val = valf  
    minima = 0  
    val = []  
    objectf = []  
    val.append(valf)  
    objectf.append(func(valf))  
    while i <= max_iter:  
        # first confition  
        leftf = func(valf + step*direction)  
        rightf = func(valf) + step* c1*dfunc(valf).dot(direction)  
        if leftf > rightf:  
            beta = step  
            step = .5*(alpha + beta)  
            val.append(valf+step*direction)  
            objectf.append(leftf)  
        elif dfunc(valf + step*direction).dot(direction) < c2*dfunc(valf).dot(direction):
```

Wolfe Conditions

► Continue on Wolfe Conditions:

```
elif dfunc(valf + step*direction).dot(direction) < c2*dfunc(valf).dot(direction):
    alpha = step
    if beta > 100:
        step = 2*alpha
    else:
        step = .5*(alpha + beta)
    val.append(valf+step*direction)
    objectf.append(leftf)
else:
    val.append(valf+step*direction)
    objectf.append(leftf)
    break
i += 1
stop_val = valf + step*direction
stop_iter = i
minima = func(stop_val)
print(val, objectf)
return stop_val, minima, stop_iter, step, val, objectf
```

Wolfe Conditions

- Initializing, run and plotting for Wolfe:

```
start = np.array([.6, .5])
dirn = np.array([-0.3, -0.4])
converge_value, minimal, no_iter, size, val, objectf = wolfe(start, dirn, 30)
print("The value, minimal and iterations needed are " + str(converge_value) + ", " \
+ str(minimal) + ", " + str(no_iter) + ', ' + str(size))
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='Wolfe Rule')
ax.legend()
plt.savefig('wolfe.jpg')
```



Exercises



1. Why the “+”?
2. Identify ONE technique / method (not in lecture) to perform step length calculation.
3. Identify ONE technique / method (not in lecture) to perform direction searching.
4. Explain the practical steps in Armijo function using words.
5. Explain the practical steps in Wolfe function using words.
6. What is your observation for the results generated using Armijo conditions and wolfe conditions? Discuss.