Constrained Optimization
Projected gradient method

Hajali Bayramov
Politecnico di Torino
Student id: s288874
hajali.bayramov@studenti.polito.it

I. Problem overview

Main goal of this project is to implement projected gradient method with inputs fed having dimensions $n = 10^4$ and $n = 10^6$ by using exact derivative and finite differences to find local minimum within given interval of function below and compare the results in terms of number of iterations and computing time:

$$\min_{x \epsilon R^n} \sum_{i=1}^{n} \left(\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 - x_i \right) \tag{1}$$
$$\text{s.t. } 1 \leq x_i \leq 2 \ \forall i$$

As the problem specifies we need to use exact derivatives or finite difference methods (e.g. forward) using following values for increment h:

$$h = 10^{-k} \|\hat{x}\|, k = 2, 4, 6, 8, 10, 12 \tag{2}$$

II. Proposed approach

Projected gradient method uses projection of x onto X, where x is our input value, X is the set that values are constrained to. General function is:

$$\Pi_X(x) := argmin_{\hat{x} \epsilon X} \, ||\hat{x} - x|\,|^2 \tag{3}$$

There are couple of cases on how to compute projection depending on the shape of X. In our case, there are 2 boundaries, upper (U) and lower (L), thus shape becomes box shape. Projections are computed as follows:

$$[\Pi_X(x)]_i = \begin{cases} x_i & \text{if } L_i \leq x_i \leq U_i \\ L_i & \text{if } x_i < L_i \\ U_i & \text{if } x_i > U_i \end{cases}$$

$X = \{x \epsilon \mathbb{R}^n : L_i \leq x_i \leq U_i\}$, componentwise $L, U \epsilon \mathbb{R}^n$

Regarding the implementation, first we need to define useful functions at the beginning. Starting with the function 1 itself. Using lambda functions increases the speed. Then we need to define the gradient function with options of exact derivative

$$f'(x_i) = x_i^3 + x_i - 1, \tag{4}$$

forward differences

$$f'(x_i) = \frac{f(x_i + h) - f(x_i)}{h}. \tag{5}$$

After that we can define function to implement projected gradient method and record the results for $n = 10^4$ and $n = 10^6$ to get comparison at the end. Before going on further, we need to set some hyperparameters, like maximum iterations, tolerance, α, ρ, γ and etc by default or user preference.

Algorithm 1: Projected gradient

```
given $x0$;
**while** $i$ < *max and* $||f(x)|| \geq tolerance$ **do**
    *compute* $\hat{x_i} = \Pi_X(x_i - \gamma_i \nabla f(x_i)), \gamma_i > 0$
    *compute* $\alpha$
    *compute* $x_{i+1} = x_i + \alpha_i(\hat{x_i} - x_i)$
**end**
```

α and γ are chosen by limited minimization rule or armijo rule along the feasible direction. In both ways γ is constant (0.1) $\forall i$ and α is found by decreasing it until it satisfies the Armijo condition:

$$f(x_i + \alpha(\hat{x_i} - x_i)) \leq f(x_i) + c_1 \alpha \nabla f(x_i)^T(\hat{x_i} - x_i),$$

where α is a stepsize to be updated, c_1 is the constant. If the condition is not satisfied α decreases by some predefined amount, ρ.

III. Results

First of all, we compare outcome for $n = 10^4$ and $n = 10^6$ using exact derivatives. Results should converge to $x_i \approx 1$, which is lower boundary. Algorithm managed to get convergence for both inputs with the same number of iterations (48), while for obvious reasons, bigger input takes as many time as its difference with the smaller one. $n = 10^6$ takes roughly 100 times more time than $n = 10^4$, 46 seconds and 0.58 seconds respectively.

On the other hand, we need to compare the results of the same methods using finite differences to compute the gradient function. While using increments, for every value of k result converges to a solution with 48 number of iterations with both input parameters. Figure 1 shows that iterations are equal to 48 (47 starting from 0). Time as well does not differ so much. We can see from figure 2 time elapsed decreases slightly till $k = 10$ then increases a little bit. But considering the insignificance of that time we can consider all values converge to a solution. However using finite differences to compute gradient is costly,

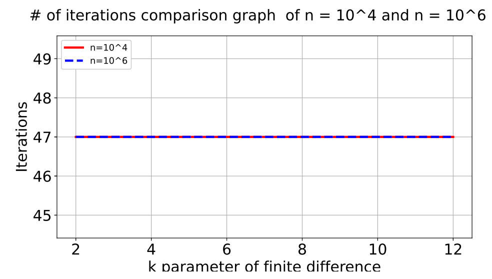


Fig. 1. Iteration comparison graph of $n = 10^4$ and $n = 10^6$

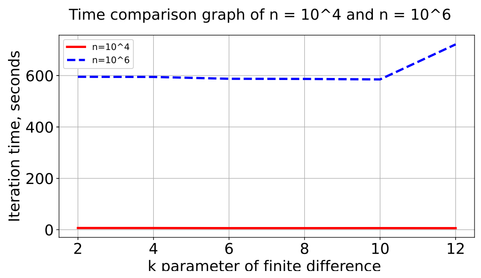


Fig. 2. Time comparison graph of $n = 10^4$ and $n = 10^6$

we can exploit it if exact derivative of the function is not possible to find.

We set maximum iteration to 300, c_1 to 10^{-5}. By tuning these parameters one can reach better performance but keeping these parameters as is helps me deal with the time and power constraints of working PC. At the end, results are quite good.

IV. Conclusion

In general, we successfully overcome the problem by applying projected gradient method to find local minimum within given interval. Looking at the results, using exact derivatives in the application gives us faster convergence. However, not always we can get exact function of derivative. In that case we need to use finite differences method.

Assignment

[1]:
```python
import numpy as np
import time

import matplotlib.pyplot as plt
```

$\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 - x_i$

[2]:
```python
# def f(x):
#     '''
#     1/4 * x^4 + 1/2 * x^2 - x
#     '''
#     f = np.zeros(len(x))
#     for i in range(len(x)):
#         f[i] = np.power(x[i], 4)/4 + np.power(x[i], 2)/2 - x[i]
#     return f

f = lambda x: (1/4)*np.power(x, 4) + (1/2)*np.power(x,2) - x #function


def delta_f(x, method = 'e', k = 10):
    '''
    method: default='e'
        e : exact derivative,
        f : forward differences,
        b : backward differences,
        c : centered differences
    k: default=10, value for increment h; is ignored when method='e'

    Exact gradient function:
        x^3 + x - 1

    Forward differences:
        (f(x+h) - f(x)) / h

    Backward differences:
        (f(x) - f(x-h)) / h
    Centered differences:
        (f(x+h) - f(x-h)) / 2h

    where h = 10^-k norm(x)
    '''

    if method == 'e':
        return np.power(x, 3) + x - 1
    else:
        ga = np.zeros(len(x))
        h = 10**(-k) * np.linalg.norm(x)

        if method =='f':
            for i in range(len(x)):
                ga[i] = (f(x[i]+h) - f(x[i])) / h
            return ga
        elif method =='b':
            for i in range(len(x)):
                ga[i] = (f(x[i]) - f(x[i]-h)) / h
            return ga
        elif method =='c':
            for i in range(len(x)):
                ga[i] = (f(x[i]+h) - f(x[i]-h)) / (2*h)
            return ga

def project(y, L, U):
    p = np.zeros(len(y))
    for i in range(len(y)):
        if y[i]<L[i]:
            p[i] = L[i]
        elif y[i]>U[i]:
            p[i] = U[i]
        else:
            p[i] = y[i]
    return p
```

[1]:
```
def descent(x0, L, U, kmax = 300, tollgrad = 10**(-4), alpha = 0.3, gamma = 0.1,␣
 ↪grad_method = 'e', grad_method_k = 10):
    '''
    x0: initial points for x
    L: lower bound
    U: upper bound
    kmax: max iteration
    tollgrad: tolerance for gradient
    alpha: default=0.31, initial alpha
    gamma: default=0.1
    grad_method: default='e'
```

```
        e : exact derivative,
        f : forward differences,
        b : backward differences,
        c : centered differences
    grad_method_k: default=10, value for increment h; is ignored when method='e'
    '''

    x = np.zeros((len(x0), kmax+1)) # kmax+1 because we need to also consider␣
↪x[k+1] case later on
    x[:, 0] = x0

    delta_fk_norm = np.linalg.norm(delta_f(x0, method = grad_method, k =␣
↪grad_method_k))

    k = 0

    # constants for line search algorithm
    rho = 0.35
    jmax = 30
    c1_arm = 10**(-5)

    gamma_k = gamma
    alpha_k = 1

    p_k = -delta_f(x[:, 0], method = grad_method, k = grad_method_k)

    s = time.time()
    while(k < kmax and delta_fk_norm >= tollgrad):

        x_k_hat = project(x[:, k] - gamma_k * (-p_k), L, U)

        # Find alpha using line search(backtracking method) with respect to the␣
↪Armijo condition
        j = 0
        while (j < jmax and
               (f(x[:, k] + alpha_k * (x_k_hat-x[:, k])) - f(x[:, k])).sum() >=␣
↪c1_arm * alpha_k * p_k.T @ (x_k_hat-x[:, k])) and alpha_k>0.25:
            alpha_k = alpha_k * rho
            j+=1

        # Compute the new value for xk
        x[:, k+1] = x[:, k] + alpha_k * (x_k_hat - x[:, k])

        if np.allclose(x[:, k+1], L, rtol=1e-03) or np.allclose(x[:, k+1], U,␣
↪rtol=1e-03):
            break
        p_k = -delta_f(x[:, k+1], method = grad_method, k = grad_method_k)

        # Compute the gradient of f in xk
        delta_fk_norm = np.linalg.norm(-p_k)

        # Increase the step by one
        k += 1

    t = time.time() - s
    # Compute f(xk)
    fk = f(x[:, k-1]).sum()

    # "Cut" xseq to the correct size and return
    return x[:, :k], k-1, fk, t
```

Naming guide "variable"+ _A _B

`A - length of vector as a power of 10 (10^4, 10^6)`

`B - gradient method (e, f, c, b)`

Example: k_4_e - iterations with n=10^4, using exact derivative

1 1. n = 10^4

1.1 A) Exact gradient

[33]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)
L = np.ones(n)     #lower bound (1)
U = np.ones(n) * 2 #upper bound (2)

x_seq_4_e, k_4_e, fk_4_e, t_4_e = descent(x0, L, U, grad_method='e')

print(f"Exact derivative for 10^4 --- Iteration: {k_4_e+1}, time:{t_4_e} s")
```

```
Exact derivative for 10^4 --- Iteration: 48, time:0.5868959426879883 s
```

1.2 B) Forward differences gradient

[6]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)
L = np.ones(n)     #lower bound (1)
U = np.ones(n) * 2 #upper bound (2)
```

```
x_seq_k_4_f = []
k_k_4_f = []
fk_k_4_f = []
t_k_4_f = []

for i in range(2, 13, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0, L, U, grad_method='f', grad_method_k=i)

    print(f"Iteration: {k_i+1}, time:{t_i} s \n")

    x_seq_k_4_f.append(x_seq_i)
    k_k_4_f.append(k_i)
    fk_k_4_f.append(fk_i)
    t_k_4_f.append(t_i)
```

```
k: 2
Iteration: 48, time:6.691483974456787 s

k: 4
Iteration: 48, time:6.573555946350098 s

k: 6
Iteration: 48, time:6.187804937362671 s

k: 8
Iteration: 48, time:6.171245574951172 s

k: 10
Iteration: 48, time:6.299541473388672 s

k: 12
Iteration: 48, time:6.072954893112183 s
```

2 2. n = 10ˆ6

2.1 A) Exact gradient

[8]:
```
n = 10**6
np.random.seed(288874)
x0 = np.random.random(n)
L = np.ones(n)      #lower bound (1)
U = np.ones(n) * 2 #upper bound (2)

x_seq_6_e, k_6_e, fk_6_e, t_6_e = descent(x0, L, U, grad_method='e')
```

```
print(f"Exact derivative for 10^6 --- Iteration: {k_6_e+1}, time:{t_6_e} s")
```

```
Exact derivative for 10^6 --- Iteration: 48, time:46.902482748031616 s
```

2.2 B) Forward differences gradient

[4]:
```
n = 10**6
np.random.seed(288874)
x0 = np.random.random(n)
L = np.ones(n)      #lower bound (1)
U = np.ones(n) * 2 #upper bound (2)

x_seq_k_6_f = []
k_k_6_f = []
fk_k_6_f = []
t_k_6_f = []

for i in range(2, 9, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0, L, U, grad_method='f', grad_method_k=i)

    print(f"Iteration: {k_i+1}, time:{t_i} s \n")

    x_seq_k_6_f.append(x_seq_i)
    k_k_6_f.append(k_i)
    fk_k_6_f.append(fk_i)
    t_k_6_f.append(t_i)
```

```
k: 2
Iteration: 48, time:595.2214076519012 s

k: 4
Iteration: 48, time:594.6810591220856 s

k: 6
Iteration: 48, time:587.5145070552826 s

k: 8
Iteration: 48, time:586.9224534034729 s
```

[5]:
```
for i in range(10, 13, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0, L, U, grad_method='f', grad_method_k=i)

    print(f"Iteration: {k_i+1}, time:{t_i} s \n")
    x_seq_k_6_f.append(x_seq_i)
    k_k_6_f.append(k_i)
    fk_k_6_f.append(fk_i)
    t_k_6_f.append(t_i)
```

```
k: 10
Iteration: 48, time:585.0535550117493 s

k: 12
Iteration: 48, time:721.4772608280182 s
```

[ ]:

```

```

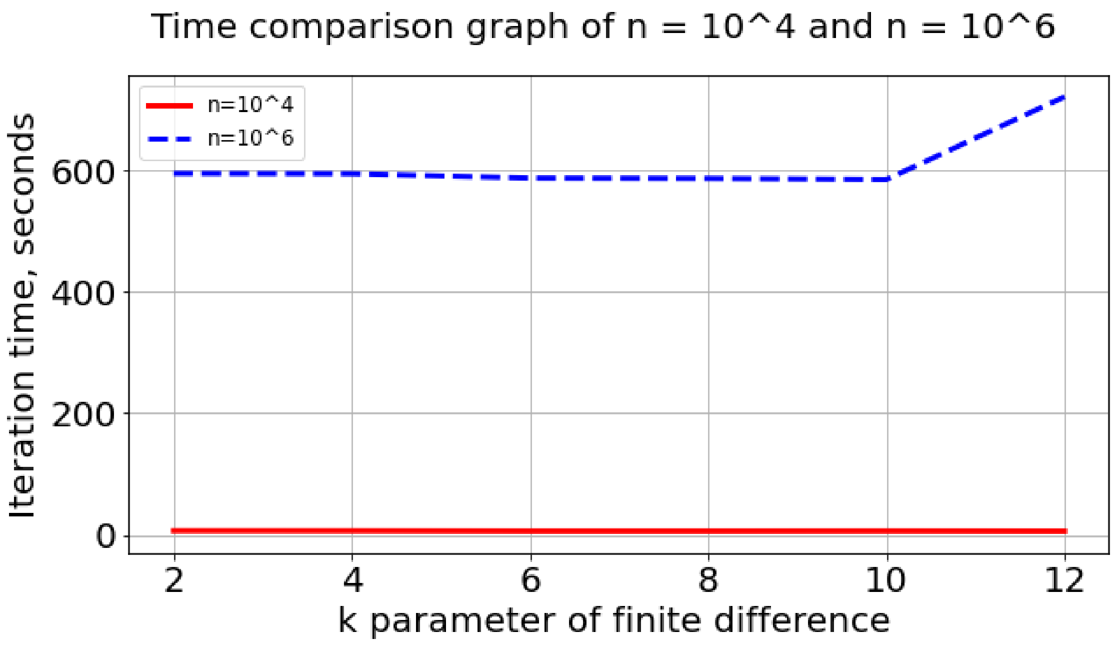
[13]:

```
def draw_time_comparison(first, second, title= ' '):
    fig, ax = plt.subplots(figsize = (10,5))
    plt.grid()

    fig.suptitle('Time comparison graph' + title, fontsize=20)
    ax = plt.plot(range(2, 13, 2), first, 'r-',linewidth=3)
    ax = plt.plot(range(2, 13, 2), second, 'b--',linewidth=3)
    plt.legend(['n=10^4', 'n=10^6'], loc ="upper left", prop={'size': 12})
    plt.xlabel('k parameter of finite difference ', fontsize=20)
    plt.ylabel('Iteration time, seconds', fontsize=20)
    plt.tick_params(axis='x', labelsize=20)
    plt.tick_params(axis='y', labelsize=20)

    fig.savefig('Time_'+title+'.eps', format='eps')

def draw_iteration_comparison(first, second, title= ' '):
    fig, ax = plt.subplots(figsize = (10,5))
    plt.grid()

    fig.suptitle('# of iterations comparison graph ' + title, fontsize=20)
    ax = plt.plot(range(2, 13, 2), first, 'r-',linewidth=3)
    ax = plt.plot(range(2, 13, 2), second, 'b--',linewidth=3)

    plt.legend(['n=10^4', 'n=10^6'], loc ="upper left", prop={'size': 12})
    plt.xlabel('k parameter of finite difference', fontsize=20)
    plt.ylabel('Iterations', fontsize=20)
    plt.tick_params(axis='x', labelsize=20)
    plt.tick_params(axis='y', labelsize=20)

    fig.savefig('Iter_'+title+'.eps', format='eps')
```

[14]:

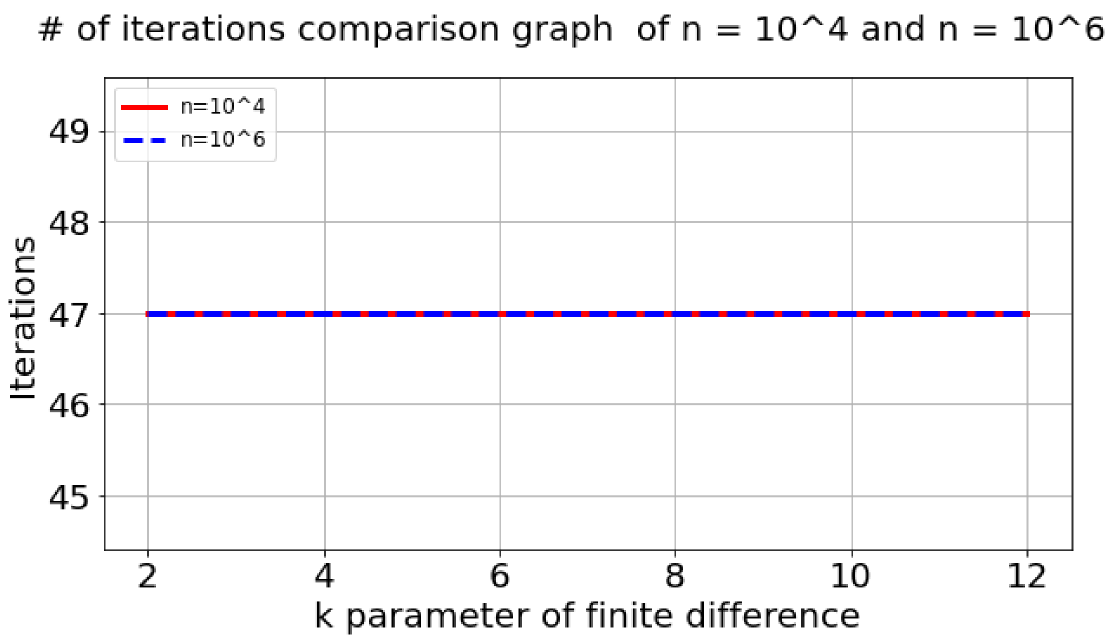
```
draw_time_comparison(t_k_4_f, t_k_6_f, ' of n = 10^4 and n = 10^6')
```

```
The PostScript backend does not support transparency; partially transparent
artists will be rendered opaque.
The PostScript backend does not support transparency; partially transparent
artists will be rendered opaque.
```



[12]:
```
draw_iteration_comparison(k_k_4_f, k_k_6_f, ' of n = 10^4 and n = 10^6')
```

```
The PostScript backend does not support transparency; partially transparent
artists will be rendered opaque.
The PostScript backend does not support transparency; partially transparent
artists will be rendered opaque.
```



# of iterations comparison graph of n = 10^4 and n = 10^6
n=10^4
n=10^6
Iterations
45
46
47
48
49
2
4
6
8
10
12
k parameter of finite difference

```
[ ]:
```

```
[ ]:
```