Unconstrained Optimization Steepest descent

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

I. Problem overview

Main goal of this project is to implement and compare methods of three different gradient descents using exact derivatives and finite differences method in terms of number of iterations and computing time. We need to implement unconstrained optimization to find the global minimum of the function:

$$\min_{x \epsilon R^n} \sum_{i=1}^{n} \left(\frac{1}{4} x_i^4 + \frac{1}{2} x_i^2 + x_i \right) \tag{1}$$

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

$$h = 10^{-k} \left\| \hat{x} \right\|, k = 2, 4, 6, 8, 10, 12, 14 \tag{2}$$

We should test our own implementation of steepest descent and ones offered by Fletcher-Reeves (FR) and Polak-Ribiere(PR) non-linear conjugate gradient with line-search. Problem is with $n = 10^4$ and $n = 10^5$.

II. Proposed approach

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

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

forward differences

$$f'(x_i) = \frac{f(x_i + h) - f(x_i)}{h}, \tag{4}$$

.

After that we can define function(s) to implement one of the three methods and record the results to get comparison at the end.Before selecting either one of the methods, we need to set some hyperparameters, like maximum iterations, tolerance, α, ρ and etc by default or user preference. Then with selected algorithm function performs either one of three.

In order to compute α in each iteration we are using Armijo condition for steepest descent.

Armijo condition:

$$f(x_i + \alpha p_i) \leq f(x_i) + c_1 \alpha \nabla f(x_i)^T p_i,$$

Algorithm 1: Steepest descent

given $x0$;
compute $f_0 = f(x_0), \nabla f_0 = \nabla f(x_0)$;
set $p_0 = -\nabla f_0$;
while *i < max and* $||f(x)|| \geq tolerance$ **do**
  compute α_i
  compute $x_{i+1} = x_i + \alpha_i p_i$
  compute p_i
  if *condition* **then**
    instructions1;
    instructions2;
  else
    instructions3;
  end
end

Algorithm 2: FR-CG

given $x0$;
compute $f_0 = f(x_0), \nabla f_0 = \nabla f(x_0)$;
set $p_0 = -\nabla f_0$;
while *i < max and* $||f(x)|| \geq tolerance$ **do**
  compute α_i
  compute $x_{i+1} = x_i + \alpha_i p_i$
  compute $\beta_{i+1} = \frac{\nabla f(x_{i+1})^T \nabla f(x_{i+1})}{\nabla f(x_i)^T \nabla f(x_i)}$
  compute $p_{i+1} = -\nabla f(x_{i+1}) + \beta_{i+1} p_i$
end

Algorithm 3: PR-CG

given $x0$;
compute $f_0 = f(x_0), \nabla f_0 = \nabla f(x_0)$;
set $p_0 = -\nabla f_0$;
while *i < max and* $||f(x)|| \geq tolerance$ **do**
  compute α_i
  compute $x_{i+1} = x_i + \alpha_i p_i$
  compute $\beta_{i+1} = \frac{\nabla f(x_{i+1})^T (\nabla f(x_{i+1}) - \nabla f(x_i))}{\nabla f(x_i)^T \nabla f(x_i)}$
  compute $p_{i+1} = -\nabla f(x_{i+1}) + \beta_{i+1} p_i$
end

where α is a stepsize to be updated, c_1 is the constant, while p_i is the slope of the gradient and can be found with three different ways depending on method we use.

- Steepest descent: $p_i = -\nabla \mathrm{f}(\mathrm{x}_i)$
- Fletcher Reeves: $p_{i+1} = -\nabla f(x_{i+1}) + \beta_{i+1} p_i$
- Polak Ribiere: $p_{i+1} = -\nabla f(x_{i+1}) + \beta_{i+1} p_i$

For the FR-CG and PR-CG, algorithm is looking for the correct p_i and strong Wolfe conditions along with Armijo should also be satisfied:

$$\left|\nabla f(x_i + \alpha p_i)^T p_i\right| \leq c_2 \left|\nabla f(x_i)^T p_i\right| \tag{5}$$

where c_2 is Wolfe constant, $0 < c_1 < c_2 < 0.5$

If the conditions are not satisfied α decreases by some predefined amount, ρ.

III. Results

First of all, we compare outcome from SD, FR-CG, PR-CG for $n = 10^4$ and $n = 10^5$ using exact derivatives. Results should converge to $f \approx -3953$ and $x_i \approx -0.68$. As can be seen from Table 1, results are pretty close and there is no big difference between methods. We get best from FR-CG but other methods use insignificantly more time and iterations.

TABLE I
Comparison table for methods using exact derivatives

n	**Method**	**# iterations**	**Iteration time, in s**
10^4	SD	15	0.07
	FR-CG	14	0.05
	PR-CG	17	0.06
10^5	SD	16	0.89
	FR-CG	15	0.83
	PR-CG	18	0.98

On the other hand, we need to compare the results of the same methods using finite differences to compute the gradient function. While using increment (2), $k = 2$ does not converge to a solution and reaches maximum iteration. Thus, we start comparison with $k = 4$ and increment it by 2 until it reaches 14.

We set maximum iteration to 300, c_1 to 10^{-5}, c_2 to 0.1. By tuning this parameter one can reach better performance but keeping maximum iteration relatively low helps me deal with the time and power constraints of working PC. At the end, results are quite good. As can be seen from figures 1 and 2, for every method used $k = 4$ gives poor performance and this is because we set tolerance of our accuracy pretty low. Also, we can see that FR and PR conjugate gradient methods actually performs better than SD until $k = 10$ and slightly poor after that.

Looking at the time comparison of iterations, namely figures 3 and 4, it can be seen that outcome from methods behave in a similar way to the number of iterations we looked at before. Steepest descent method took longer time with finite differences till $k = 10$ for both numbers of inputs. Meanwhile,

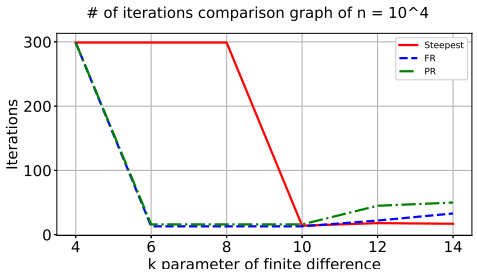


Fig. 1. Iteration comparison graph of $n = 10^4$

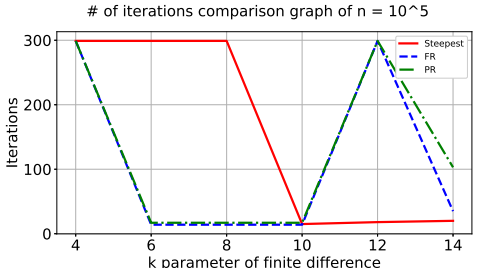


Fig. 2. Iteration comparison graph of $n = 10^5$

for $k = 12$ algorithm acted strangely for FR and PR due to computational precision.

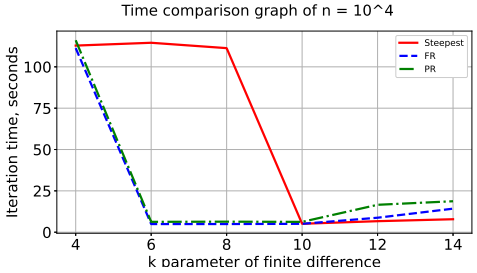


Fig. 3. Time comparison graph of $n = 10^4$

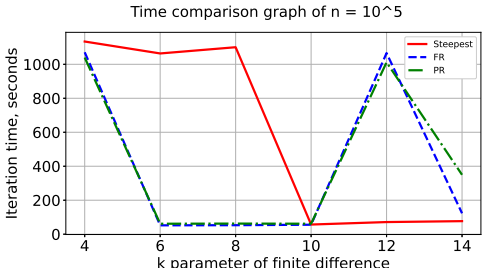


Fig. 4. Time comparison graph of $n = 10^5$

IV. Conclusion

In general, we successfully overcome the problem by applying three different gradient methods to find global minimum. 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. Comparing the results, using 10 as k gives us the best results for each methods. As is using the

exact derivatives FR reaches the optimal solution with smaller number of iterations in less time. After that comes SD. PR might be slower but it is proven to be robust of all, meaning have less failures and performs better in practice, however weaker convergence than FR is experienced. Increasing the dimensions of input (n) results in slower convergence and more iterations. But given time and computational power it reaches the global minimum.

Appendix

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

import matplotlib.pyplot as plt
```

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

[15]:
```
# 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 (f(x+h) - f(x-h)) / (2*h)
```

[22]:
```
def descent(x0, kmax, tollgrad, descent_method = 'steepest', alpha = 0.31,␣
↪grad_method = 'e', grad_method_k = 10):
    '''
    x0: initial points for x
    kmax: max iteration
    tollgrad: tolerance for gradient
    descent_method: default='steepest'
        steepest: steepest descent
        fr: Fletcher Reeves
        pr: Polak Ribier
    alpha: default=0.31, initial alpha
    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
```

```
   p_k = -delta_f(x0, method = grad_method, k = grad_method_k)

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

   alpha_k = alpha
   k = 0

   # constants for line search algorithm
   rho = 0.45
   jmax = 30
   c1_arm = 10**(-5)
   c2_wol = 0.1

   s = time.time()
   if descent_method == 'steepest':
       while(k < kmax and delta_fk_norm >= tollgrad):

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

           # Compute the new value for xk
           x[:, k+1] = x[:, k] + alpha_k * p_k

           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(delta_f(x[:, k+1], method =␣
↪grad_method, k = grad_method_k))

           # Increase the step by one
           k += 1

   elif descent_method == 'fr':
       gradf_k = delta_f(x0, method = grad_method, k = grad_method_k)
       while(k < kmax and delta_fk_norm >= tollgrad):

           # Find alpha using line search(backtracking method)
           j = 0
           while (j < jmax and
```

```
                (f(x[:, k] + alpha_k * p_k).sum() - f(x[:, k]).sum() - c1_arm *␣
↪alpha_k * delta_f(x[:, k], method = grad_method, k = grad_method_k).T @ p_k >0.
↪001 and
                (np.abs(delta_f(x[:, k] + alpha_k * p_k, method = grad_method, k␣
↪= grad_method_k).T @ p_k) - c2_wol * np.abs(delta_f(x[:, k], method =␣
↪grad_method, k = grad_method_k).T @ p_k >0.001))):

                alpha_k = alpha_k * rho
                j+=1

            # Compute the new value for xk
            x[:, k+1] = x[:, k] + alpha_k * p_k

            # Compute the new value for gradfk
            gradf_k1 = delta_f(x[:, k+1], method = grad_method, k =␣
↪grad_method_k)

            # Compute the new value for betak
            beta_k1 = (gradf_k1.T @ gradf_k1) / (gradf_k.T @ gradf_k)

            # Compute the new value for pk
            p_k = -gradf_k1 + beta_k1 * p_k

            # Compute the gradient of f in xk
            delta_fk_norm = np.linalg.norm(delta_f(x[:, k+1], method =␣
↪grad_method, k = grad_method_k))

            # Increase the step by one
            k += 1

    elif descent_method == 'pr':
        gradf_k = delta_f(x0, method = grad_method, k = grad_method_k)
        while(k < kmax and delta_fk_norm >= tollgrad):

            # Find alpha using line search(backtracking method)
            j = 0
            while (j < jmax and
                (f(x[:, k] + alpha_k * p_k).sum() - f(x[:, k]).sum() - c1_arm *␣
↪alpha_k * delta_f(x[:, k], method = grad_method, k = grad_method_k).T @ p_k >0.
↪001 and
                (np.abs(delta_f(x[:, k] + alpha_k * p_k, method = grad_method, k␣
↪= grad_method_k).T @ p_k) - c2_wol * np.abs(delta_f(x[:, k], method =␣
↪grad_method, k = grad_method_k).T @ p_k >0.001)))):

                alpha_k = alpha_k * rho
                j+=1
```

```
            # Compute the new value for xk
            x[:, k+1] = x[:, k] + alpha_k * p_k

            # Compute the new value for gradfk
            gradf_k1 = delta_f(x[:, k+1], method = grad_method, k =␣
↪grad_method_k)

            # Compute the new value for betak
            beta_k1 = (gradf_k1.T @ (gradf_k1 - gradf_k)) / (gradf_k.T @ gradf_k)

            # Compute the new value for pk
            p_k = -gradf_k1 + beta_k1 * p_k

            # Compute the gradient of f in xk
            delta_fk_norm = np.linalg.norm(delta_f(x[:, k+1], method =␣
↪grad_method, k = grad_method_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 _C _D (D optional)

`A - length of vector as a power of 10 (10^4, 10^5)`

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

`C - descent method (st, pr, fr)`

`D - k (2, 4, 6, ..., 12)`

Example: k_4_e_st - iterations with n=10^4, using exact derivative, steepest descent method

1 1. n = 10^4

1.1 A) Exact gradient

1.1.1 Steepest descent, Fletcher Reeves, Polak Ribier

[23]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)

x_seq_4_e_st, k_4_e_st, fk_4_e_st, t_4_e_st = descent(x0,  kmax, tol,␣
 ↪grad_method='e')
x_seq_4_e_fr, k_4_e_fr, fk_4_e_fr, t_4_e_fr = descent(x0,  kmax, tol,␣
 ↪descent_method='fr', grad_method='e')
x_seq_4_e_pr, k_4_e_pr, fk_4_e_pr, t_4_e_pr = descent(x0,  kmax, tol,␣
 ↪descent_method='pr', grad_method='e')

print(f"Steepest, exact for 10^4 --- Iteration: {k_4_e_st+1}, time:{t_4_e_st} s")
print(f"Fletcher Reeves, exact for 10^4 --- Iteration: {k_4_e_fr+1}, time:
 ↪{t_4_e_fr} s")
print(f"Polak Ribier, exact for 10^4 --- Iteration: {k_4_e_pr+1}, time:
 ↪{t_4_e_pr} s")
```

```
Steepest, exact for 10^4 --- Iteration: 15, time:0.07494759559631348 s
Fletcher Reeves, exact for 10^4 --- Iteration: 14, time:0.05388331413269043 s
Polak Ribier, exact for 10^4 --- Iteration: 17, time:0.06692218780517578 s
```

1.2 B) Forward differences gradient

1.2.1 Steepest descent

[26]:
```
x_seq_k_4_f_st_2, k_k_4_f_st_2, fk_k_4_f_st_2, t_k_4_f_st_2 = descent(x0,  kmax,␣
 ↪tol, grad_method='f', grad_method_k=2)

k_k_4_f_st_2, x_seq_k_4_f_st_2[:, k_k_4_f_st_2] # it does not converge when k=2␣
 ↪even with max iteration
```

[26]:
```
(299,
 array([-0.77690584, -0.80669571, -0.79001164, ..., -0.76868856,
        -0.76348641, -0.74433103]))
```

[27]:
```
x_seq_k_4_f_st_4, k_k_4_f_st_4, fk_k_4_f_st_4, t_k_4_f_st_4 = descent(x0,  kmax,␣
 ↪tol, grad_method='f', grad_method_k=4)
k_k_4_f_st_4, x_seq_k_4_f_st_4[:, k_k_4_f_st_4] # it does converge when k=4 with␣
 ↪max iteration
```

[27]:
```
(299,
 array([-0.68417812, -0.68475658, -0.68446366, ..., -0.68396454,
```

```
       -0.68381084, -0.68173708]))
```

[28]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_4_f_st = []
k_k_4_f_st = []
fk_k_4_f_st = []
t_k_4_f_st = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, grad_method='f',␣
 ↪grad_method_k=i)

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

    x_seq_k_4_f_st.append(x_seq_i)
    k_k_4_f_st.append(k_i)
    fk_k_4_f_st.append(fk_i)
    t_k_4_f_st.append(t_i)
```

```
k: 4
Iteration: 300, time:112.8567726612091 s

k: 6
Iteration: 300, time:114.55893111228943 s

k: 8
Iteration: 300, time:111.30999660491943 s

k: 10
Iteration: 15, time:5.087850093841553 s

k: 12
Iteration: 19, time:6.664440631866455 s

k: 14
Iteration: 18, time:7.843650579452515 s
```

1.2.2 Fletcher Reeves

[29]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_4_f_fr = []
k_k_4_f_fr = []
fk_k_4_f_fr = []
t_k_4_f_fr = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, descent_method='fr',␣
 ↪grad_method='f', grad_method_k=i)

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

    x_seq_k_4_f_fr.append(x_seq_i)
    k_k_4_f_fr.append(k_i)
    fk_k_4_f_fr.append(fk_i)
    t_k_4_f_fr.append(t_i)
```

```
k: 4
Iteration: 300, time:111.32002425193787 s

k: 6
Iteration: 14, time:4.922631740570068 s

k: 8
Iteration: 14, time:4.933692455291748 s

k: 10
Iteration: 14, time:5.06583833694458 s

k: 12
Iteration: 23, time:8.75761103630066 s

k: 14
Iteration: 34, time:14.232372760772705 s
```

1.2.3 Polak Ribier

[30]:
```
n = 10**4
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_4_f_pr = []
k_k_4_f_pr = []
fk_k_4_f_pr = []
t_k_4_f_pr = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, descent_method='pr',␣
 ↪grad_method='f', grad_method_k=i)

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

    x_seq_k_4_f_pr.append(x_seq_i)
    k_k_4_f_pr.append(k_i)
    fk_k_4_f_pr.append(fk_i)
    t_k_4_f_pr.append(t_i)
```

```
k: 4
Iteration: 300, time:116.07892155647278 s

k: 6
Iteration: 17, time:6.269172191619873 s

k: 8
Iteration: 17, time:6.366508960723877 s

k: 10
Iteration: 17, time:6.245284557342529 s

k: 12
Iteration: 46, time:16.573433876037598 s

k: 14
Iteration: 51, time:18.721948623657227 s
```

2 2. n = 10^5

2.1 A) Exact gradient

2.1.1 Steepest descent, Fletcher Reeves, Polak Ribier

[31]:
```
n = 10**5
np.random.seed(288874)
x0 = np.random.random(n)

x_seq_5_e_st, k_5_e_st, fk_5_e_st, t_5_e_st = descent(x0,  kmax, tol,␣
 ↪grad_method='e')
x_seq_5_e_fr, k_5_e_fr, fk_5_e_fr, t_5_e_fr = descent(x0,  kmax, tol,␣
 ↪descent_method='fr', grad_method='e')
x_seq_5_e_pr, k_5_e_pr, fk_5_e_pr, t_5_e_pr = descent(x0,  kmax, tol,␣
 ↪descent_method='pr', grad_method='e')

print(f"Steepest, exact for 10^4 --- Iteration: {k_5_e_st+1}, time:{t_5_e_st} s")
print(f"Fletcher Reeves, exact for 10^4 --- Iteration: {k_5_e_fr+1}, time:
 ↪{t_5_e_fr} s")
print(f"Polak Ribier, exact for 10^4 --- Iteration: {k_5_e_pr+1}, time:
 ↪{t_5_e_pr} s")
```

```
Steepest, exact for 10^4 --- Iteration: 16, time:0.893566370010376 s
Fletcher Reeves, exact for 10^4 --- Iteration: 15, time:0.8367772102355957 s
Polak Ribier, exact for 10^4 --- Iteration: 18, time:0.9805235862731934 s
```

2.2 B) Forward difference gradient

2.2.1 Steepest descent

[32]:
```
n = 10**5
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_5_f_st = []
k_k_5_f_st = []
fk_k_5_f_st = []
t_k_5_f_st = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, grad_method='f',␣
 ↪grad_method_k=i)

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

```
    x_seq_k_5_f_st.append(x_seq_i)
    k_k_5_f_st.append(k_i)
    fk_k_5_f_st.append(fk_i)
    t_k_5_f_st.append(t_i)
```

```
k: 4
Iteration: 300, time:1134.0428175926208 s

k: 6
Iteration: 300, time:1063.8334455490112 s

k: 8
Iteration: 300, time:1100.5457110404968 s

k: 10
Iteration: 16, time:56.885350465774536 s

k: 12
Iteration: 19, time:71.42638349533081 s

k: 14
Iteration: 21, time:76.83855128288269 s
```

2.2.2 Fletcher Reeves

[33]:
```
n = 10**5
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_5_f_fr = []
k_k_5_f_fr = []
fk_k_5_f_fr = []
t_k_5_f_fr = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, descent_method='fr',␣
 ↪grad_method='f', grad_method_k=i)

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

    x_seq_k_5_f_fr.append(x_seq_i)
    k_k_5_f_fr.append(k_i)
    fk_k_5_f_fr.append(fk_i)
    t_k_5_f_fr.append(t_i)
```

```
k: 4
Iteration: 300, time:1070.8488726615906 s

k: 6
Iteration: 15, time:52.20074796676636 s

k: 8
Iteration: 15, time:53.00279378890991 s

k: 10
Iteration: 15, time:55.75055527687073 s

k: 12
Iteration: 300, time:1065.8683197498322 s

k: 14
Iteration: 36, time:123.00361013412476 s
```

2.2.3 Polak Ribier

[34]:
```
n = 10**5
np.random.seed(288874)
x0 = np.random.random(n)


x_seq_k_5_f_pr = []
k_k_5_f_pr = []
fk_k_5_f_pr = []
t_k_5_f_pr = []

for i in range(4, 15, 2):
    print(f"k: {i}")
    x_seq_i, k_i, fk_i, t_i = descent(x0,  kmax, tol, descent_method='pr',␣
 ↪grad_method='f', grad_method_k=i)

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

    x_seq_k_5_f_pr.append(x_seq_i)
    k_k_5_f_pr.append(k_i)
    fk_k_5_f_pr.append(fk_i)
    t_k_5_f_pr.append(t_i)
```

```
k: 4
Iteration: 300, time:1039.8310105800629 s

k: 6
Iteration: 18, time:61.54909825325012 s
```

```
k: 8
Iteration: 18, time:62.463219165802 s

k: 10
Iteration: 18, time:61.65953755378723 s

k: 12
Iteration: 300, time:1011.7506356239319 s

k: 14
Iteration: 104, time:349.62147092819214 s
```

[ ]:

[ ]:

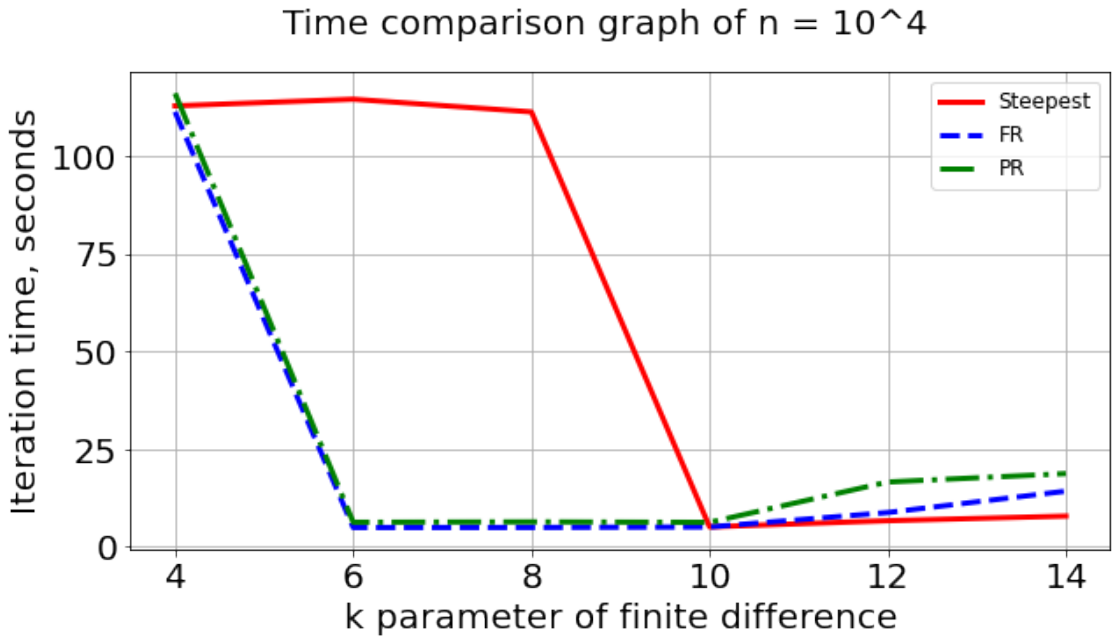
[ ]:

[35]:
```python
def draw_time_comparison(st, fr, pr, title= ' '):
    fig, ax = plt.subplots(figsize = (10,5))
    plt.grid()

    fig.suptitle('Time comparison graph' + title, fontsize=20)
    ax = plt.plot(range(4, 15, 2), st, 'r-',linewidth=3)
    ax = plt.plot(range(4, 15, 2), fr, 'b--',linewidth=3)
    ax = plt.plot(range(4, 15, 2), pr, 'g-.',linewidth=3)
    plt.legend(['Steepest', 'FR', 'PR'], loc ="upper right", 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(st, fr, pr, title= ' '):
    fig, ax = plt.subplots(figsize = (10,5))
    plt.grid()

    fig.suptitle('# of iterations comparison graph' + title, fontsize=20)
    ax = plt.plot(range(4, 15, 2), st, 'r-',linewidth=3)
    ax = plt.plot(range(4, 15, 2), fr, 'b--',linewidth=3)
    ax = plt.plot(range(4, 15, 2), pr, 'g-.',linewidth=3)
    plt.legend(['Steepest', 'FR', 'PR'], loc ="upper right", 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')
```

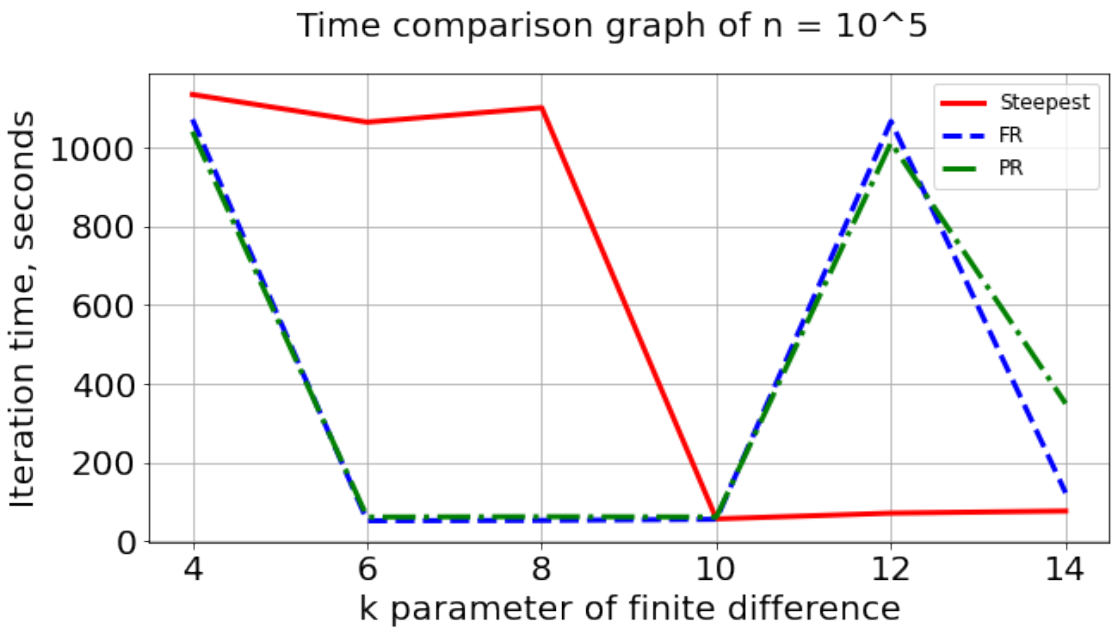
[36]:
```
draw_time_comparison(t_k_4_f_st, t_k_4_f_fr, t_k_4_f_pr, ' of n = 10^4')
```

```
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.
```



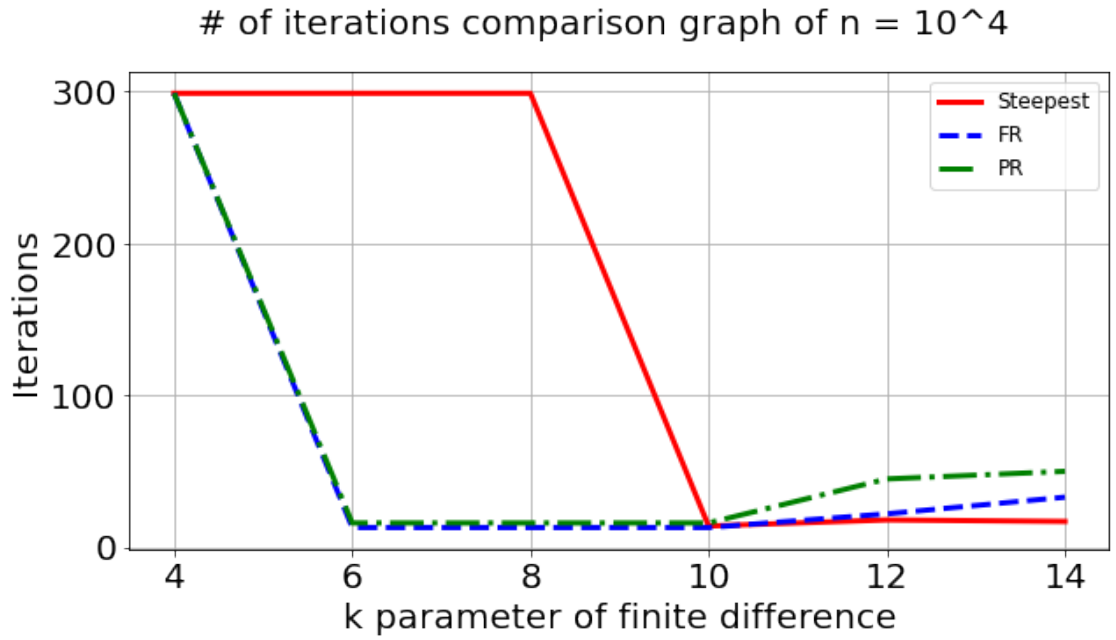
[37]:
```
draw_time_comparison(t_k_5_f_st, t_k_5_f_fr, t_k_5_f_pr, ' of n = 10^5')
```

```
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.
```



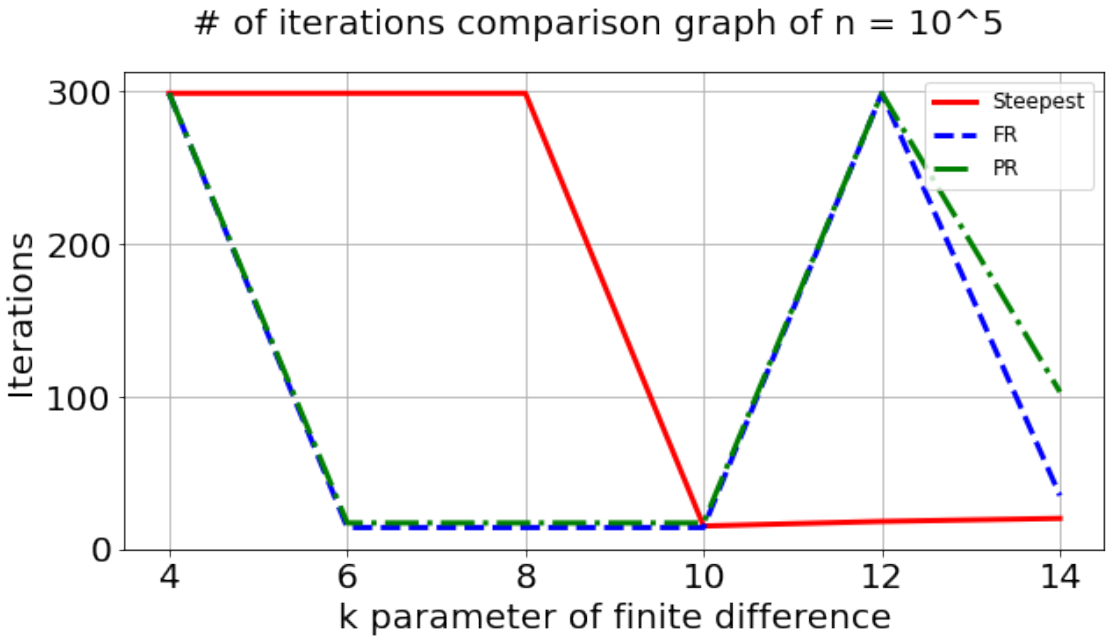
[38]:
```
draw_iteration_comparison(k_k_4_f_st, k_k_4_f_fr, k_k_4_f_pr, ' of n = 10^4')
```

```
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.
```



[39]:
```
draw_iteration_comparison(k_k_5_f_st, k_k_5_f_fr, k_k_5_f_pr, ' of n = 10^5')
```

```
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.
```



[ ]:

[ ]: