POLITECNICO DI TORINO

Master Degree course in Ingegneria Matematica 2022/2023

Numerical optimization for large scale problems and Stochastic Optimization

Unconstrained Optimization

Racca Riccardo s315163 Filippo Scaramozzino s312856

Gennaio 2023

Contents

Chapter 1

Introduction

1.1 Theoretical background

In this work we try to implement specific methods in order to find the minimum of complex functions, in particular we will focus on two main solving methods :

- Steepest descent with backtracking
- Newton method

Tested on the functions:

- Chained Rosenbrock function
- Chained Powell singular function
- Generalized Broyden tridiagonal function

taken from the document available at: https://www.researchgate.net/publication/325314497_Test_Problems_for_Unconstrained_Optimization

1.1.1 Methods - Steepest Descent

The steepest descent method is an iterative method to find the minimum of a given function.
Let the function $f : R^n \to R$ be given, starting from an initial point x_0 (a column vector), such that $x_0 \in R^n$, the steepest descent computes a sequence of vectors $\{x_k\}_{k \in N}$ in the following way :

$$x_{k+1} = x_k + \alpha p_k, \qquad \forall k \geq 0$$

where p_k is a vector that represent the most descent direction that the method follows in order to find the minimum.

$$p_k = -\nabla f(x_k)$$

$\alpha \in R^+$ represent the step length of each iteration.

To better optimize this method we implemented also a backtracking strategy, the backtracking strategy for an iterative method consists of looking for a value α_k satisfying the Armijo condition at each iteration of the method.

The backtracking strategy is an iterative process that looks for the value of α_k, remembering that the choice of the starting α_0 is problem-dependent.

Given an arbitrary factor $\rho \in (0,1)$ and an arbitrary starting value α_0 we decrease iteratively α_0, multiplying it by ρ, such that $\alpha_k = \rho^{t_k}\alpha_0$, for a $t_k \in N$, until the Armijo condition is satisfied.

The Armijo condition provides an α_k in order to have a sufficient decrease in f,looking for a value of α_k satisfying condition at each step k.

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

1.1.2 Methods - Newton Method

We implemented two different newton methods to compare the results, the Newton method with finite differences and the Newton method matrix free.

The general idea behind Newton method is the following :

Let the function $f : R^n \to R$ be given, starting from an initial point x_0 (a column vector), such that $x_0 \in R^n$, the steepest descent computes a sequence of vectors $\{x_k\}_{k \in N}$ in the following way :

$$x_{k+1} = x_k + \alpha p_k, \qquad \forall k \geq 0$$

where p_k is the descent direction and is the solution of the linear system given by :

$$H_f(x_k)p = -\nabla f$$

where H_f is the positive definite hessian matrix of f

The Newton method originates from the idea that we can approximate $f(x+p)$ with a Taylor expansion:

$$f(x+p)' \approx f(x) + p^T \nabla f(x) + \frac{1}{2} p^T H_f(x) p = m_{f,x}(p)$$

Assuming that $H_f(x)$ is positive semidefinite, for each $p \in R^n$, we have that $p^T H_f(x) p \geq 0$ and, therefore, $m_{f,x}(p)$ is convex.
Then, we can compute the p^* that minimizes $m_{f,x}(p)$, p^* such that

$$\nabla f(x) + H_f(x)p^* = 0$$

p^* is solution of the linear system and is a descent direction for $f(x)$.

The Newton method has advantages with respect to the steepest descent but also some disadvantages.

The Newton method has quadratic rate of convergence if a good initial point (x_0) is chosen.
On the other side, the computation of the hessian is expensive.
Another disadvantage is that p_k has a descent direction only if the hessian of $f(x_k)$ is positive definite, in case it is not, we should work with the modified Newton method to perform a correction on the matrix.

1.1.3 Newton method - finite differences - matrix free

We define $f : R^n \to R$ and assume $f \in C^2$, we can then define the gradient and the hessian of the function :

$$G = \nabla f = \begin{bmatrix} f_{x_1} \\ . \\ . \\ . \\ f_{x_n} \end{bmatrix} \in R^n$$

$$H_f = \nabla^2 f = \begin{bmatrix} f_{x_1 x_1} & . & . & . & f x_1 x_n \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ f_{x_n x_1} & . & . & . & .f_{x_n x_n} \end{bmatrix} \in R^{n \times n}$$

The hessian is a symmetric matrix.

Since the computation of the gradient and the hessian for complex functions can be expensive, we can approximate them using the finite differences approach.

Using forward finite differences we have (for the gradient) :

$$f_{x_i}(x) = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}, \forall i = 1, ..., n$$

Instead, using centered finite differences we have (for the gradient) :

$$f_{x_i}(x) = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i) - f(x-he_i)}{2h}, \forall i = 1, ..., n$$

Using finite differences we have (for the hessian) :

$$(H_f(x))_{ii} \approx \frac{f(x+he_i) - 2f(x) + f(x-he_i)}{h^2}, \forall i = 1, ..., n$$

$$(H_f(x))_{ij} \approx \frac{f(x+he_i+he_j) - f(x+he_i) - f(x+he_j) + f(x)}{h^2}, \forall i,j = 1, ..., n, \forall i \neq j$$

Where $he_i = he_j = p$ and $h = \sqrt{\epsilon_m}$.

These expression comes from both the definitions of partial derivatives and Taylor expansion.

The matrix free approach instead, we approximate the product $H_f(x_k)\dot{p}$ of the Newton method as :

$$H_f(x_k)\dot{p} \approx \frac{\nabla f(x + hp - \nabla f(x))}{h}$$

This formula allow us to not compute directly the hessian, and to solve the linear system $H_f(x_k)\dot{p} = \nabla f(x_k)$ we use the conjugate gradient method, by mean of the command *pcg* on matlab, that is an iterative solver that approximates the solution of the linear system.

The matrix free works with the exact gradient of the f because the approximation of both the gradient and the hessian might produce inexact result if used in a optimization method, knowing this issue we tried to perform a matrix free method with the approximation of the gradient to see whether in our case the approximation was too inexact or could produce acceptable results.
The results produced by the method were good.

1.2 Test Rosenbrock

We start by applying our optimization methods to the Rosenbrock function to see how they perform on a simple problem, we start from two different points. The Rosenbrock function is defined as :

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

and the two starting points are :

$$x_1 = (1.2,1.2), \quad x_2 = (-1.2,1)$$

We will analyze this function by mean of two methods :

- Steepest descent with backtracking
- Newton method (in this case exact Newton method)

In the following we analyze the result.

We note that with this simple function it is possible to compute directly the gradient and the hessian of the function.

For this reason we decided to use the exact Newton method for this particular case, to obtain a better result, even if we know that it will soon become very difficult to obtain good results in short time with this approach.

We can visualize the result obtained with the steepest descent.

	x_1	x_2
fk	0.0032	0.1089
xk	(1.0563,1.1156)'	(1.3300,1.7687)'

We can see that the starting points x_1 and x_2 produce two different results, by looking at the table we can see that x_1 seems to be a better starting point to find the minimum of the function, and the minimum seems to be around the point (1,1) and the value of the minimum is zero.

The results with the steepest descent are not very precise, we can try to obtain better results with the Newton method.

Since we know how to compute the exact gradient of the function we can proceed with the exact Newton method to find out if with this method we obtain a better result.

	x_1	x_2
fk	0	0
xk	(1,1)'	(1,1)'

The Newton method by using the exact gradient and hessian produce a very precise result.

1.3 Analysis and results

Now we test the three functions mentioned above :

- Chained Rosenbrock function
- Chained Powell singular function
- Generalized Broyden tridiagonal function

By applying the methods described above.

In our analysis we tested 5 different starting points.

The choice of the points, in some cases, brought us to discover interesting results, such as being able to make the hypothesis of having discovered the minimum of the function in analysis.

Chapter 2

Analysis

In this chapter we present our analysis and our results by means of tables and graphs summarizing the results.

2.1 Tables and graphs

The tables are organized in the following way :

- method : solving method applied to the function
- time : time needed to reach the value in the column "min"
- min : the supposed minimum found by the method, if 0 is written we had a result less than 1e-5, and we assume it to be close enough to zero
- n.iterations : number of iterations necessary to reach the minimum indicated in the table(in some cases in which we didn't see any improvement in the solutions from some values of k on we reduced the number of max iterations without compromising the solution and obtaining improvements in time)
- alpha0 : parameter for backtracking
- rho : coefficient for backtracking
- nabla : method used to compute the gradient of f
- nabla^2 : : method used to compute the hessian of f
- l.s. : method used to solve the linear system
- pcg : number of max iterations in case the pcg solver is used
- bt : the mean of the backtracking iterations at each step

2.2 BROYDEN

In this section we analyze the results obtained with Generalized Broyden Tridiagonal function.

$x0$ = point given by the paper, $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	7.6	0	32	5	0.1891	fw		\		5
FD	1121	0.012	6		0.5	c	fw	pcg	5	0
N.MF*	18.8	0	10		0.5	fw	MF	pcg	2	0

With the first point we analyze we can see that for what concern the time and the minimum found, the best method is the steepest descent, that reaches the minimum in 7.6 seconds with a initial point of dimension $n = 1e3$.
In this case we modified the parameter ρ used in the backtracking to optimize the computational time.
We can note that after 32 iterations of steepest descent the result was stationary and no change at all happened, so we decreased the maximum iterations of the method from 1000 to 32.
The Newton method with finite differences was less accurate and much longer to execute with respect to the steepest descent, we tried to improve precision by using the centered finite differences, but the time necessary to have a feasible solution was much longer, to improve the time of execution of the method we tried to reduce the number of iterations, even if this can bring to a less accurate solution, in this case we found the best trade-off for the execution between solution and time of execution(16 min approximatly).

With the Newton method with matrix free we obtained a good solution both in terms of value of the minimum found and computational time.

$x0$ =-ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	12.7	0	53	5	0.1891	fw		\		2.47
N.FD	1085	0.0012	6		0.5	c	MF	pcg	5	0
N.MF*	31	0	17		0.5	c	MF	pcg	2	0

We tried another point, a vector of values $= -1$ and dimension $1e3$, also in this case we obtained results similar to what we observed above, but with a little increase in the computational time and a small decrease in the number of iterations of the backtracking. This is a worse starting point compared to the original one.

$x0$ =zeros , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	8.7	0	36	6	0.25	fw		\		2.7

The starting point made of all zeros seems good for the method steepest descent, but with the other two methods the execution time is greater than 15 minutes, so we conclude that the starting point made of all zeros is not a good starting point.

$x0 = x0$+ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	10.17	0	43	5	0.3	fw		\		3.2

With the starting point made by adding 1 to our original point we obtained good results only with the steepest descent, as with the other two methods the execution time is greater than 15 minutes, we conclude that the best point by far is the original one.

$x0$=ones , $n = 1e3$ An important result we obtained with the starting point equal to a vector of all 1 was that the function was exactly 0 at the first iteration and the methods didn't do any iterations, we can say that by choosing this point we found a local minimum in which the methods cannot move in a decreasing direction.

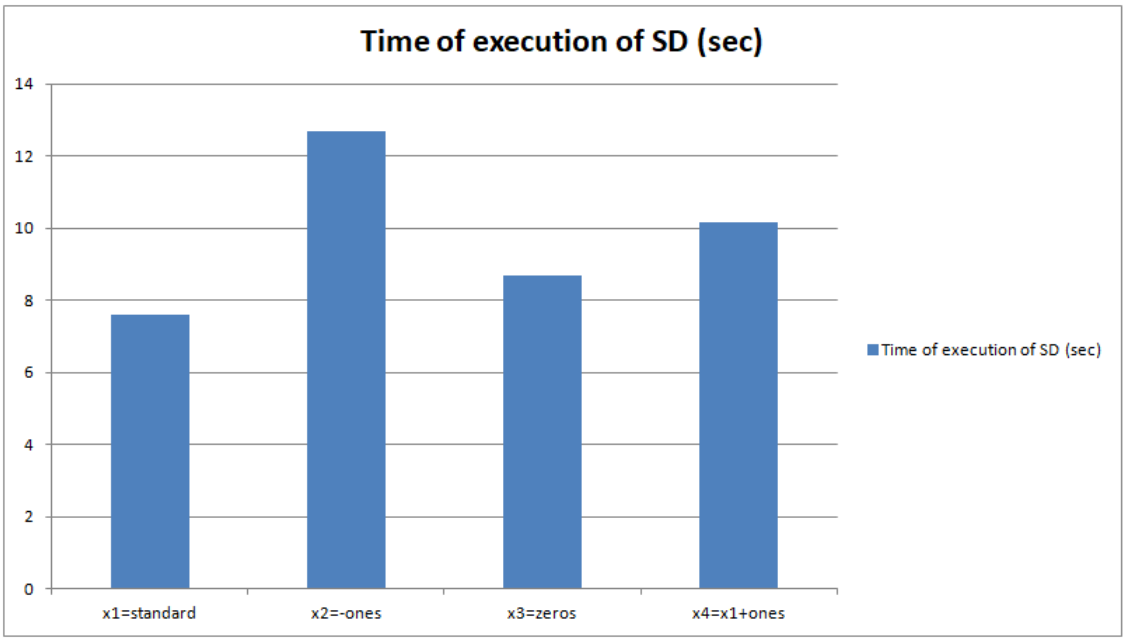


Figure 2.1. Time of SD for the different points

Now we try to run the methods that gave us the best results in terms of time of execution and value of minimum, in this case the steepest descent, for a higher dimension using $n = 1e4$.

$x0 =$ point given by the paper , $n = 1e4$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	712.6	0.0186	32	5	0.1891	fw		\		5

We can observe that the result was not as precise as for $n = 1e3$ this is because we tried to improve the computational time by reducing the number of iterations required to obtain a better solution.

$x0 =$zeros , $n = 1e4$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	772.7	0	36	5	6	fw		\		3

For the point composed of all zeros we can obtain a good result for what concern the value of the minimum found with a time similar to the one above.

2.3 POWELL

In this section we analyze the results obtained with Chained Powell Singular function.

$x0$ = point given by the paper , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	230	0	1000	5	0.5	fw		\		8.3
N.MF*	30	0	10		0.5	c	MF	pcg	5	0

By looking at the results, we can observe an opposite behaviour with respect to the Broyden function.
In this scenario the Newton method matrix free is performing better than the steepest descent, also with a significantly smaller number of iterations required to find the value of the minimum.
We used the centered approximation for the gradient for the matrix free method.

$x0$ =ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	233	0	1000	5	0.5	fw		\		8.3
N.MF*	29	0	10		0.5	c	MF		5	0

$x0$ =-ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	230	0	1000	5	0.5	fw		\		6.8
N.MF*	32	0	10		0.5	fw	MF		5	0

$x0$ =2*ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	235	0	1000	5	0.5	fw		\		8.2
N.MF*	30	0	10		0.5	fw	MF		5	0

Even changing the starting point doesn't seem to improve the results obtained before, the steepest descent require all the iterations to arrive to the solution.
We can conclude that our results both with the steepest descent and Newton method don't allow us to run the methods with a dimension of $x0$ equal to $1e4$ in a time < 15 minutes.
$x0$ =zeros , n =1e3
Interesting results are obtained by the starting point made of all zeros, in which the function is exactly zero without any iteration of the methods, we can assume to have found a local minimum.

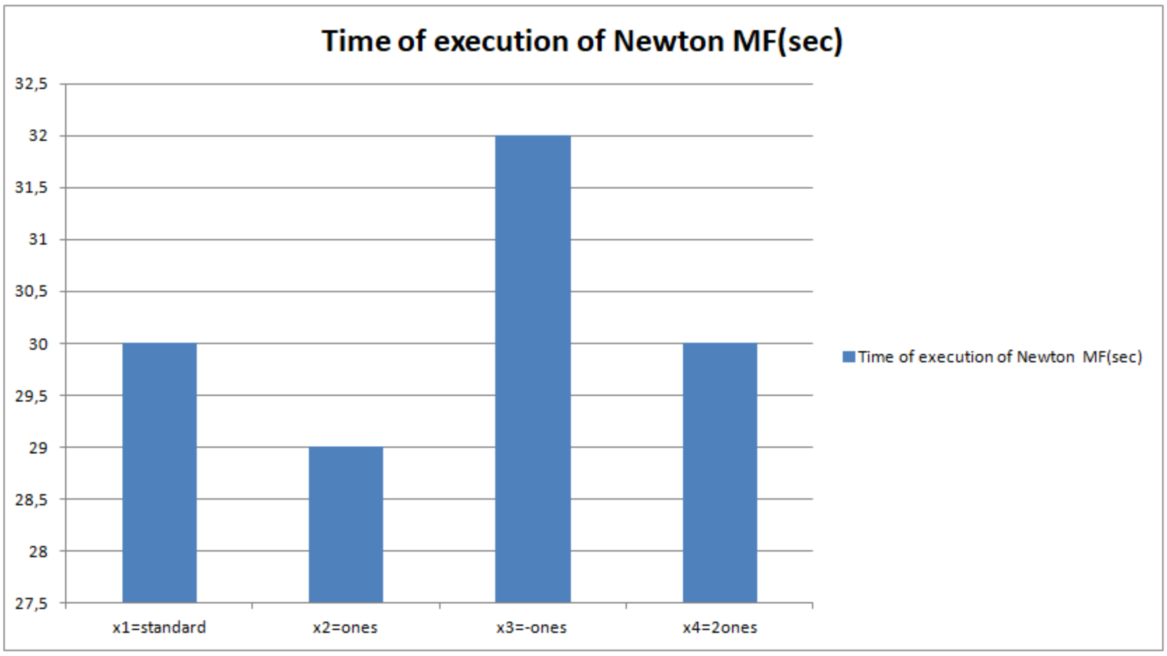


Figure 2.2. Time of Newton matrix free for the different points

2.4 ROSENBROCK

In this section we analyze the results obtained with the Chained Rosenbrock function.

$x0$ = point given by the paper , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	5.8	976	1000	5.8	0.5	fw		\		10
N.MF*	23	969	40		0.5	fw	MF		50	0

In the Rosenbrock function with $x0$ = given by the paper we obtained with both method a minimum value that is quite high, the steepest descent used all the 1000 iterations to arrive to the value, even modifying the parameters for the backtracking the solution doesn't seem to improve.
For the matrix free approach even setting maximum iterations equal to 50 for the pcg solver the solution doesn't seem to improve and after 40 iterations it stays fixed at 969.
We assumed to be in the position of not knowing where the exact minimum is, so we can investigate whether our value found is the real minimum or not, one possible strategy to search for another minimum is to change the starting point $x0$, and see if we get another solution (also to avoid the problem of oversolving in a point far from the solution we will change initial point).

$x0$ =2*ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	5.8	0.0044	1000	8.09	0.5	fw		\		11.5
N.FD	95	0	17		0.5	fw	fw	pcg	20	0.3
N.MF*	3.4	0	10		0.5	fw	MF		20	0

Changing point to $x0$ =2*ones we can see that this time we obtained a better value for the minimum, probably the minimum as the values found by the methods seem to be around zero.
The fastest method is the Newton method matrix free.
To find if the minimum is zero we can test other points and analyze their solutions.

$x0$ =-ones , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	5.6	0	1000	0.9	0.00999	fw		\		2

$x0$ =zeros , $n = 1e3$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	5.9	0.01	1000	5	0.5	fw		\		11.5

The other points confirm that the value of the minimum is zero.
With the best results we had previously we can increase the dimension of the starting point to $n = 1e4$, we try to find the best trade-off between computational time and value

found for the minimum.

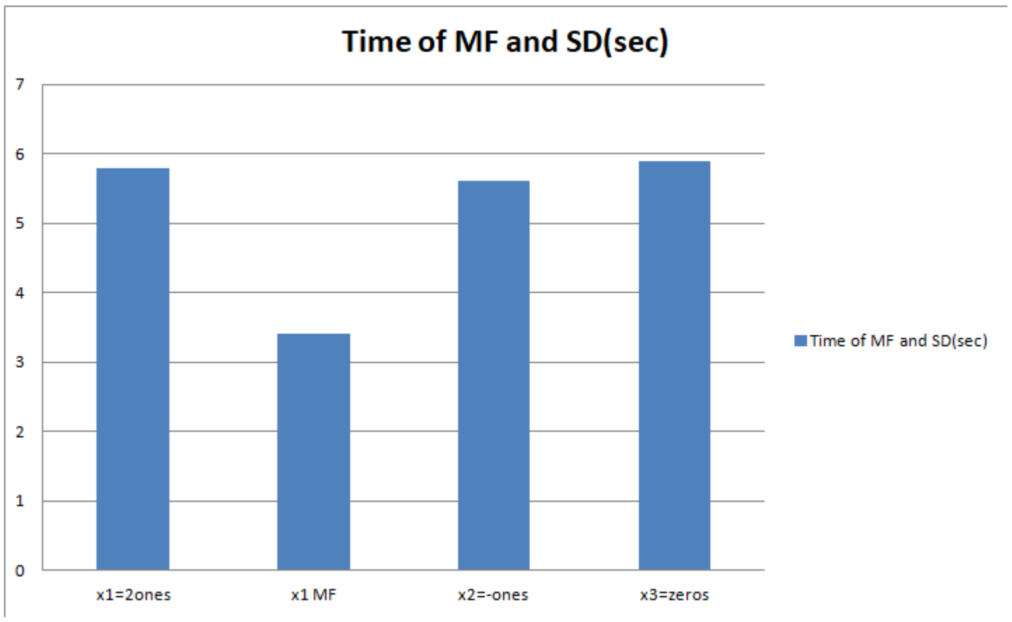


Figure 2.3. Time of Newton matrix free and SD

$x0$ =2*ones , $n = 1e4$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	642.8	0.0046	32	8.09	0.00999	fw		\		2

$x0$ =zeros , $n = 1e4$

Results										
Method	time	min	n.iterations	α_0	ρ	∇	∇^2	l.s.	pcg	bt
SD	708.6	0.0157	1000	5	0.5	fw		\		11.5
N.MF*	497	0	15		0.5	fw	MF		20	0

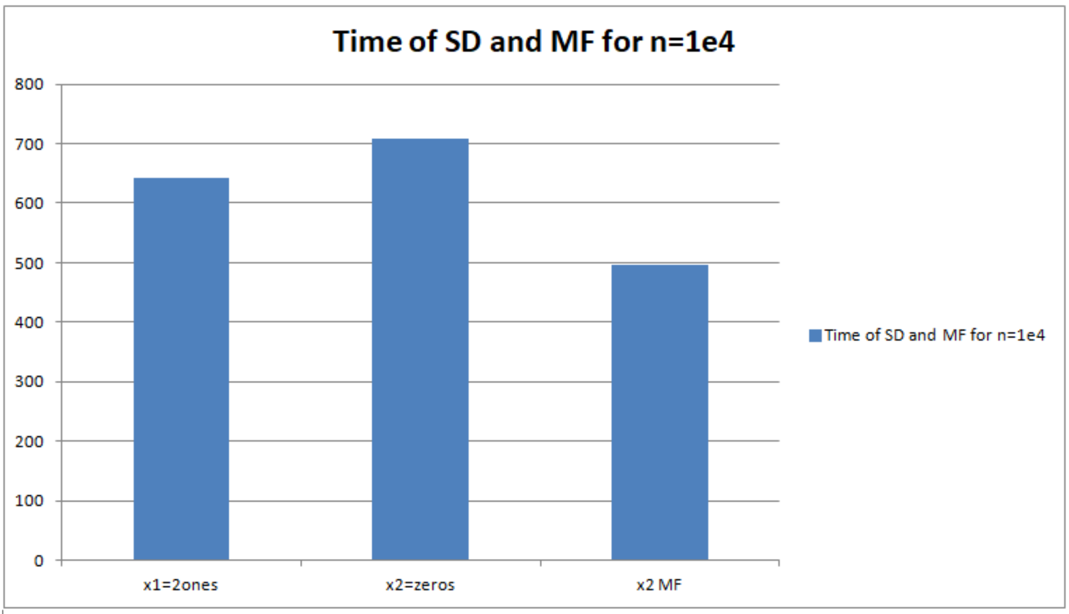


Figure 2.4. Time of Newton matrix free and SD for $n = 1e4$

2.5 Conclusions and possible improvements

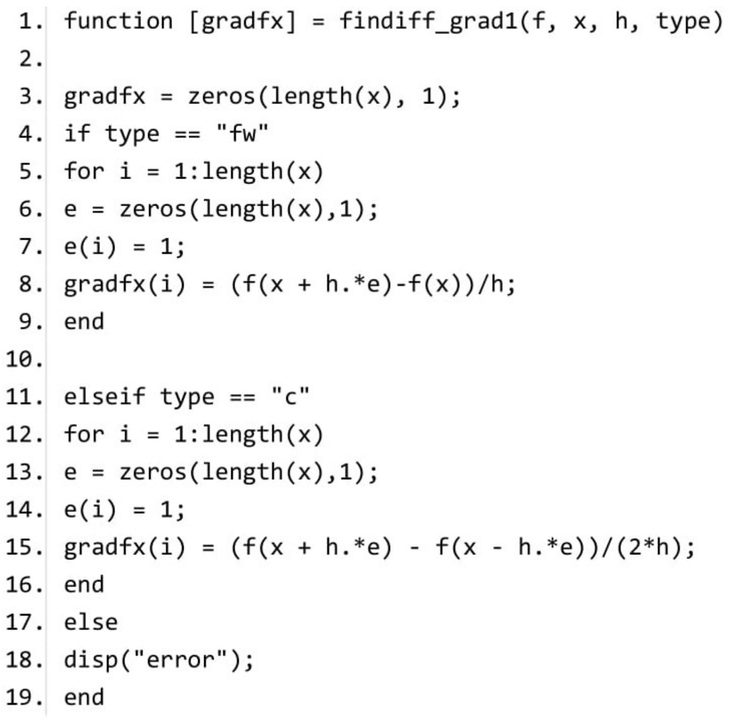
Analyzing our results we can see that the best method is not a specific one for every function but it depends on the situation and the choice of the starting point plays an important role.
The steepest descent and the Newton method matrix free show good result in terms of time, while the finite differences approach is usually slower but in some cases more precise.
Once we analyzed our results we can think about possible improvements of our methods in term of solutions and computational time.
Our procedure to change and try different starting points helped us in finding the exact minimum because in some cases we were also able to find the exact solution in one step, and so, we were able to make the hypothesis of finding the exact minimum.
One first big improvement could be the explicit computation of the gradient for our functions, because by having the explicit form we can avoid nested loops that make our methods slower.
Another improvement we can think about is to analyze the minimum number of backtracking iterations and set the ρ to be set at the first step to avoid those backtracking iterations.

2.6 APPENDIX

Here we listed the code used to implement our functions.

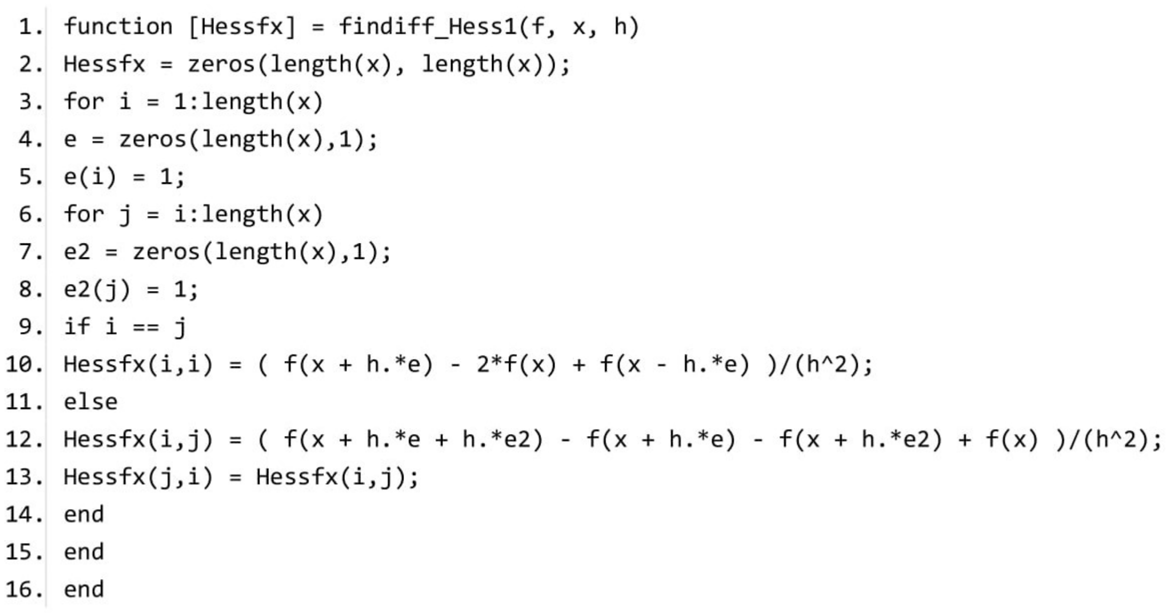
```matlab
function [xk, fk, gradfk_norm, k, xseq, btseq] = SD2(x0, f, alpha0, kmax, tolgrad, c1, rho, btmax, h, FDgrad)

xk = x0;
k = 0;
xseq = zeros(length(xk),kmax);
btseq = zeros(1,kmax);


if strcmp(FDgrad,'ex')

gradf = @(x) gradf_ad_B(xk); % B , R , P where used to computed the exact gradients for each function in AD
gradfk = gradf(xk);

elseif strcmp(FDgrad,'fw') || strcmp(FDgrad,'c')

gradf = @(x) findiff_grad1(f, xk, h, FDgrad);
gradfk = gradf(xk);

else
disp('error')
end

for i = 1:kmax
p = -gradfk;
x2 = xk;
alpha = alpha0;
xk = x2 + alpha*p;

fk = f(xk);
fk2 = f(x2);
b = 0;
while (fk > fk2 + c1*alpha*(gradfk)'*p) && b<btmax
alpha = rho * alpha;
xk = x2 + alpha*p;
b = b+1;
fk = f(xk);
end

if strcmp(FDgrad,'ex')

gradf = @(x) gradf_ad_B(xk); % B , R , P where used to computed the exact gradients for each function in AD
gradfk = gradf(xk);

else

gradf = @(x) findiff_grad1(f, xk, h, FDgrad);
gradfk = gradf(xk);

end

gradfk_norm = norm(gradfk);

if gradfk_norm <= tolgrad
break
end

k = k+1;
xseq(:,i) = xk;
btseq(i) = b;
end

fk = f(xk);

xseq = xseq(:,1:k);
btseq = btseq(1:k);

end
```

Figure 2.5. Function Steepest Descent used for our analysis



```
function [gradfx] = findiff_grad1(f, x, h, type)

gradfx = zeros(length(x), 1);
if type == "fw"
for i = 1:length(x)
e = zeros(length(x),1);
e(i) = 1;
gradfx(i) = (f(x + h.*e)-f(x))/h;
end

elseif type == "c"
for i = 1:length(x)
e = zeros(length(x),1);
e(i) = 1;
gradfx(i) = (f(x + h.*e) - f(x - h.*e))/(2*h);
end
else
disp("error");
end
```

Figure 2.6. Function used for finite differences



```
function [Hessfx] = findiff_Hess1(f, x, h)
Hessfx = zeros(length(x), length(x));
for i = 1:length(x)
e = zeros(length(x),1);
e(i) = 1;
for j = i:length(x)
e2 = zeros(length(x),1);
e2(j) = 1;
if i == j
Hessfx(i,i) = ( f(x + h.*e) - 2*f(x) + f(x - h.*e) )/(h^2);
else
Hessfx(i,j) = ( f(x + h.*e + h.*e2) - f(x + h.*e) - f(x + h.*e2) + f(x) )/(h^2);
Hessfx(j,i) = Hessfx(i,j);
end
end
end
```

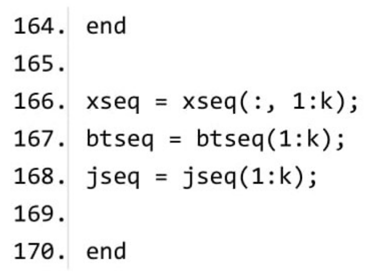
Figure 2.7. Function used for hessian

```
function [xk, fk, gradfk_norm, k, xseq, btseq, jseq] = newton_star_7(x0, f, kmax, FDgrad, FDHess, tolgrad, verbose, c1, rho, btmax, jmax, h, fterms,pcg_maxit)


flagG = 0;
flagMF = 0;

if isa(fterms, 'function_handle')
flag_etak = 1; %flag per forcing terms
else
flag_etak = 0;
end

switch FDgrad

case 'fw'
gradf = @(x) findiff_grad1(f,x,h,'fw');

case 'c'
gradf = @(x) findiff_grad1(f,x,h,'c');

case 'ex'
flagG = 1;
gradf = @(x) gradf_ad_B(x);

otherwise
disp('error')
end

switch FDHess

case 'fw'
Hessf = @(x) findiff_Hess1(f, x, sqrt(h));

case 'Jfw'
if flagG == 0
disp('error');
return
end
Hessf = @(x) findiff_J1(@gradf_ad_B, x, h, 'fw');

case 'Jc'
if flagG == 0
disp('error');
return
end
Hessf = @(x) findiff_J1(@gradf_ad_B, x, h, 'c');

case 'MF'
flagMF = 1;

case 'ex' %voglio hess ex
if flagG == 0
disp('error');
return
end
Hessf = @(x) Hessf_ad_B(x);

otherwise
disp('error');

end

xk = x0;
k = 0;
xseq = zeros(length(xk), kmax);
btseq = zeros(1, kmax);
jseq = zeros(1, kmax);


for i = 1:kmax

gradfk = gradf(xk);

if flagMF == 0 %se non voglio MF calcolo Hess
Hessfk = Hessf(xk);
end

if verbose && flagMF == 0 %Choleski per direzioni discendenti
A = Hessfk;
beta = norm(A,'fro');
a = min(diag(A));
```

Figure 2.8. Function used for Newton method part one

```
if a > 0
tao = 0;
else
tao = beta/2;
end
for j = 1:jmax

try
R = chol(A + tao*eye(length(xk)));
flag=1;
catch
tao = max(2*tao, beta/2);
flag = 0;
end

if flag==1
break
end

jseq(k) = j;

end
Hessfk = R'*R;
end

if pcg_maxit == 0
if flagMF == 1
disp('error')
return
end

p = Hessfk\-gradfk;

elseif flagMF == 1

Hessfk_p = @(p) (gradf(xk + h*p) - gradf(xk))/ h;

if flag_etak == 0
[p,xflag] = pcg(Hessfk_p, -gradfk, tolgrad, pcg_maxit);

else
eta_k = fterms(gradfk, k);
[p,xflag] = pcg(Hessfk_p, -gradfk, eta_k, pcg_maxit);
end

else


if flag_etak == 0
[p,xflag] = pcg(Hessfk, -gradfk, tolgrad, pcg_maxit);

else
eta_k = fterms(gradfk, k);
[p,xflag] = pcg(Hessfk, -gradfk, eta_k, pcg_maxit);
end

end

x2 = xk;
fk2 = f(x2);
alpha = 1;
xk = x2 + alpha * p;

b = 0;
fk = f(xk);
while (fk > fk2 + c1 * alpha * gradfk'*p) && b < btmax
alpha = rho * alpha;
xk = x2 + alpha * p;
b = b + 1;
fk = f(xk);
end

btseq(i) = b;
gradfk_norm = norm(gradfk);

if gradfk_norm < tolgrad
break
end

k = k + 1;
xseq(: , k) = xk;
```

Figure 2.9. Function used for Newton method part two



```
end

xseq = xseq(:, 1:k);
btseq = btseq(1:k);
jseq = jseq(1:k);

end
```

Figure 2.10. Function used for Newton method part three