Optimization
Deterministic Optimization

Arturo del Cerro Vergara (NIU:1593930)

January 10, 2021

Master's degree in Modelling for Science and Engineering

Abstract

Rosenbrock's function is in some way a test for the performance of algorithms. In this project Levenberg-Marquardt and Conjugate Gradient Descent methods have been implemented, tested and compared showing the possible successes or fails of each one.

Contents

1 Rosenbrock's function

The rosenbrock's function is a function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ such that

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

This Rosenbrock function was presented by Howard H. Rosenbrock in order to give a test for the performance of optimization algorithms. In this case we are studying the 2-dimensional case but in general it is a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. In figure 1 is shown the shape of this function.

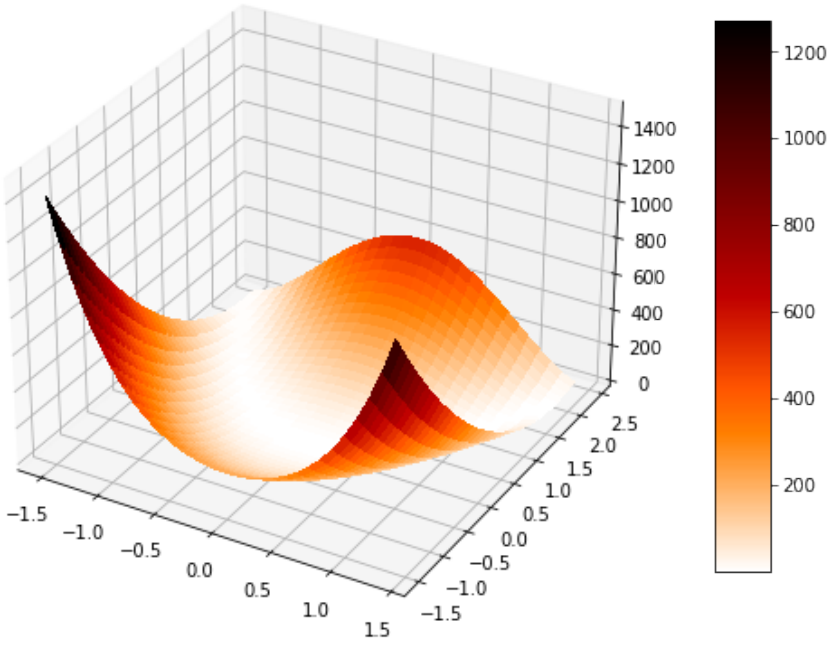

Figure 1

1.1 Convexity

Majority of linear optimization began over convex sets because of their consistent properties that allows the guaranteed achievement of none, some or infinite solutions. Then some algorithms appeared in order to perform optimization algorithms over non-convex sets or functions. First we will need few definitions.
Definition. Let S be a vector space or an affine space over some ordered field. Then $C \subseteq S$ is a convex subset if $\forall x, y \in C$ and $\forall t \in [0, 1]$, $tx + (1 - t)y \in C$
Definition. Let X be a convex subset of a real vector space and let $f : X \longrightarrow \mathbb{R}$ then f is called convex if $\forall x_1, x_2 \in X$ and $\forall\ t \in [0, 1]$

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Now we know the tools we are working with we see the properties about the Rosenbrock's function. Is it convex? We can demonstrate the non-convexity of

this function with a counter-example.
Counter-example. Let $x_1 = (-1, 1)$ and $x_2 = (0, 0)$ then exists $t = 0.5$ that don't satisfy the convexity function condition. Lets see:

$$f(0.5x_1 + 0.5x_2) = 8.5 \not< 2.5 = 0.5f(x_1) + 0.5f(x_2)$$

Besides, this functions seems to be convex over that regions at each side of the 'hill'.

1.2 Global minimum

As our objetive is to find $\min_x f(x)$ some algorithms express the cost function f as a sum of squares, i.e, if $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ then

$$f(x) = \sum_{i=1}^{n} \left(f_i(x)^2\right) \tag{1.2.1}$$

In our case $n = 2$ and Rosenbrock's function can be expressed as

$$f(x) = \left[10(x_2 - x_1^2)\right]^2 + \left[1 - x_1\right]^2$$

So we know it is defined over $\mathbb{R}^+$. Now letting $x = (1, 1)$ then $f(x) = 0$. So $(1, 1)$ is a mininum. Also, this minimum is a global minimum. We can ask ourselves now if this (-1.5,-1) starting point is good or bad in to our algorithms. It seems is a bad one as (-1.5,-1) and (1,1) do not verify the convex conditions over the Rosenbrock's function. Lets see it then below.

2 Algorithms

Now we are going to describe the algorithms that have been used and the variants for them comparing how they perform to the solution in a better or worse way.
Note: This work have been developed in Python software. For project's request, functions as matrix sum, matrix multiplication, matrix transpose, matrix inverse and norm have been self written in order to not use any external packages or libraries. The only external libraries used are plots and time libraries.

2.1 Non-linear Conjugate Gradient Method

To this specific problem we need to use the non-linear conjugate gradient method that is a generalization of the conjugate gradient method but using the gradient alone when finding the minimum. Having $f(x)$ to minimize, then the solution (if exists) is in the form of

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

where α_k is the stepsize obtained with line search and d_k is the search direction defined by

$$d_k = \begin{cases} -g_k & \text{if } k = 1 \\ -g_k + \beta_k d_{k-1} & \text{if } k \geq 2 \end{cases}$$

where $g_k = \nabla f(x_k)$ and β_k is a parameter we will explain below.
This means that, as (x_k) is the direction of maximum increase, we take the opposite, the steepest descent direction.
Although there exists lots of line search techniques as the strong and weak Wolfe line search here we have implemented the backtranking one that appears in [3]. This method basicaly consists in actualizing α until first of the strong Wolfe conditions is satisfied:

$$f(x_k + \alpha_k d_k) - f(x_k) \leq \rho \alpha_k g_k^T d_k$$

with $\rho \in (0, 1)$. Below is shown the code used for the line search.

```
def backtracking(x,alpha,iter,tau,rho):
    f = rosenbrock(x)
    gT = transpose(gradient(x))
    d = [[-gT[0][0]],[-gT[0][1]]]  #d is steepest descent direction
    t = -rho*multiply(gT,d)[0][0]
    while rosenbrock(x)-rosenbrock( matrix_sum( x , [[alpha*i[0]] for i in d] )
    ↪  )  <  alpha*t:
        alpha = tau*alpha          #actualize alpha by a tau step
    return alpha
```

As this line search tries to get the minimum argument for α as $arg\min_{\alpha} f(x_k + \alpha_k d_k)$ one could ask to do a brute force search for this minimum. This is way computational efficient but later we will show the effects of this in the convergence of the algorithm. Now we find different conjugate gradient methods [5] for computing the β_k value that performs way different deppending in the parameters. Here we present the four are used.

- Fletcher-Reeves Method. May fail for non-exact line search.

$$\beta_k^{FR} = \frac{||g_k||^2}{||g_{k-1}||^2}$$

  A variation for this method that works better is taking $\beta_k = 0$ if $k \equiv 0 \mod(n)$. (n=2 in our case)

- Polak-Ribiere Method. In this case β is computed as as

$$\beta_k^{PR} = \frac{g_k^T (g_k - g_{k-1})}{||g_{k-1}||^2}$$

  This method works better theorically than Fletcher-Reeves one and as we will see in our case occurs the same. Nevertheless this method only ensure convergence for strict convex functions (not our case). To fix it, a variation for this method is the $\beta_k^{PR+} = \max\{0, \beta_k^{PR}\}$

- Hestenes-Stiefel Method. The β_k is calculated as

$$\beta_k = \frac{g_k^T(g_k - g_{k-1})}{d_{k-1}^T(g_k - g_{k-1})}$$

  As before, there exists a variation for β that ensures in more cases the convergence of the method calculated as $\beta_k^{HS+} = \max\{0, \beta_k^{HS}\}$

- Dai-Yuan Method. In this last case, a value for β_k proposed by Dai-Yuan is

$$\beta_k^{DY} = \frac{||g_k||^2}{d_{k-1}(g_k - g_{k-1})}$$

As we have described it theoretically in Appendix can be found the code for the implementation of this algorithm. Now we are going to see how this different methods performs within the non-linear conjugate gradient descent.

Table 1: Results for different methods

Method	τ	ρ	Error	Iterations	Time (s)
Fletcher-Reeves	0.2	0.2	10^{-4}	270	0.001
	0.001	0.02	10^{-4}	2442	0.001
	0.4	0.2	10^{-4}	5158	0.001
Polak-Ribiere	0.01	0.1	10^{-4}	116	0.015
	0.2	0.1	10^{-4}	141	0.029
	0.2	0.2	10^{-4}	67	0.009
Hestenes-Stiefel	0.1	0.01	10^{-4}	640	0.09
	0.1	0.1	10^{-4}	676	0.01
	0.2	0.1	10^{-4}	298	0.06
Dai-Yuan	0.01	0.01	10^{-4}	70095	9.88
	0.1	0.01	10^{-4}	3332	0.75
	0.2	0.01	10^{-4}	381	0.07

In the table above we can see some of the results with different parameters. As we anticipated before there are some of this methods that do not ensure the convergence. To avoid this, in the majority of this tests the have been used the β_k^+ form because if not maybe the algorithm gets stucked or converges to an other point. So a conclussion it has been lighted up is the importance of the line search and its parameters. The theory ensures that with this $\rho \in (0, 1)$ it can converge but we have to take care with the election of it. From the results, the best one is the Polak-Ribiere one with parameters τ and ρ equal to 0.2 both as it takes only 67 iterations, the least of all the tests made.
In figure 2 we can see how the best performances of each method reach to the

solution. In green the Fletcher-Reeves, in red the Poliak-Ribiere, in yellow the Hestenes-Stiefel and in blue the Dai-Yuan. It is obvius the bad performance of all of them as they get a lot of iterations around (0,0) and then gets a lot of effort to get the solution (1,1) when they are near.

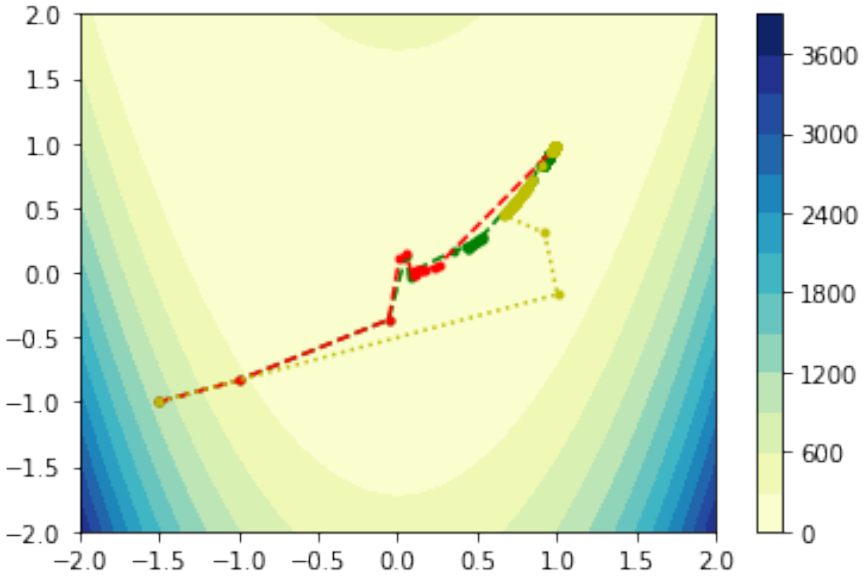

Figure 2

As we remark the importance of the line search algorithm that could be going to the bad our conjugate gradient descent algorithm we want to show how this algorithm would perform by a exact line search. For this we search for the α_k that is minimum in the line search by 'brute force'. Obviously this is not efficient but here are the results:

Table 2: Results for different methods by exact line search

Method	Error	Iterations	Time (s)
Fletcher-Reeves	10^{-4}	28	18.16
Polak-Ribiere	10^{-4}	14	9.19
Hestenes-Stiefel	10^{-4}	13	8.4
Dai-Yuan	10^{-4}	12	7.39

In figure 3 are shown the paths of the algorithms made by exact line search (brute mode) and we can compare now how well they perform. It seems that all algorithms do the same path but the Dai-Yuan one that takes a shortest path that goes directly to the solution.

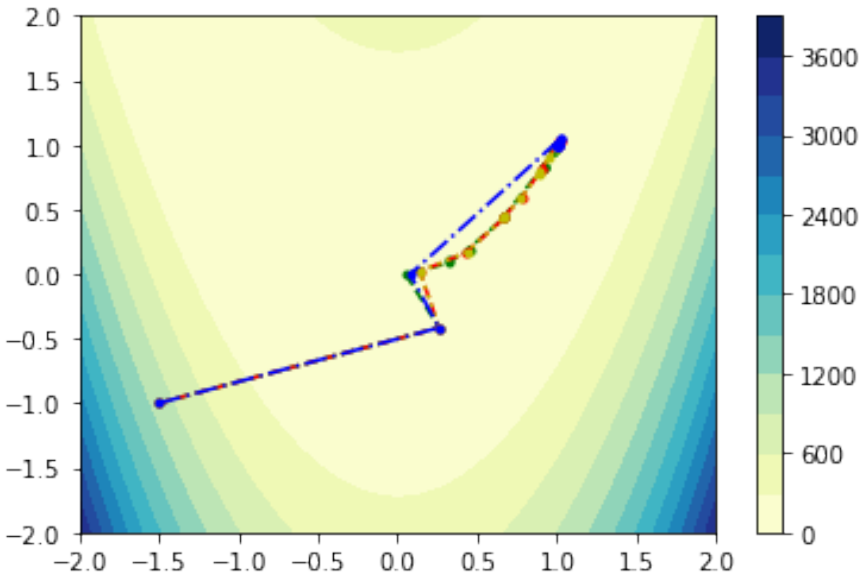

Figure 3

2.2 Levenberg-Marquardt

Levenberg-Marquardt algorithm is an algorithm for finding the solution of certain non-linear problems in least squares. It combines the advantages of the gradient descent method and the Gauss-Newton method. Its steps are a linear combination of both methods based on some rules that adapt by the path to the solution. In the case of the Rosenbrock's function, the gradient descent steps dominates until the canyon is reached and then the Gauss-Newton steps begins to dominate.
Now the notation used in 1.2.1 appears as considering this problem as a least squares problem. Let $F \in \mathbb{R}^2$ be

$$F = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} 10(x_2 - x_1^2) \\ (1 - x_1) \end{pmatrix}$$

then we can calculate the jacobian used for this algorithm. Hre is shown the pseudocode inspired in [2] and [6]:

```
Levenberg-Marqueardt Pseudocode
    x = x_0
    λ = max diag (J^T J)
    WHILE not converged:
        Find δ such that (J^T J + λ diag(J^T J) )δ = J^T f
        x' = x + δ
        if f(x') ≤ f(x) then
            x = x'
            λ = λ/a
        else
            λ = λ · b
    return x
```

Again the entire code for this algorithm can be found in Appendix. Note that this algorithm is adaptative as well as we said before. Depending on this parameters a and b the algorithm performs differently. The case $a = b$ is called the Direct method as it increase the λ value by a fixed factor for uphill steps and decrease the λ with this same fixed step for downhill steps. The case $0 \leq a \neq b$ is called the Delayed gratification as it increase λ by a small factor when uphill steps and decrease λ value by a large factor when downhill steps occurs. Usually it is taken $a = 2, b = 8$. Also it is interesant to comment that Levenberg algorithm (not Levenberg-Marquardt) is the same but with the identity matrix instead of $J^T J$ when multiplying by λ to find δ.
Then, here is presented the results of the different valued Levenberg-Marquardt algorithm.

Table 3: Results for different methods

Levenberg-Marquardt	a	b	Error	Iterations	Time (s)
Direct Method	2	2	0	50	0.004
	4	4	0	30	0.001
	8	8	0	43	0.003
Delayed Gratification	2	8	0	46	0.001
	4	6	0	29	0.001
	4	8	0	33	0.001
	6	8	0	32	0.001
Dealing with precission	4	6	10^{-4}	26	0.001
	4	6	10^{-3}	25	0.001
	4	6	10^{-2}	24	0.001
	4	6	10^{-1}	23	0.001
Levenberg	6	6	0	26	0.002

As we can see the best performance is get with the delayed gratification with parameters $a = 4, b = 6$ with a 29 iterations. Also, with Levenberg algorithm we get an improvement of this Levenberg-Marquardt algorithm reaching 26 iterations in the same time.
In the figures below are presented the 'paths' the algorithm follows. The left one represents, in red, the Levenberg-Marquardt algorithm with $a = b = 2$ and the green one the Levenberg-Marquardt with $a = 4, b = 6$. In the right image is shown the Levenberg-Marquardt algorithm in green and the Levenberg algorithm in blue. Both performs very well reaching in few iterations to the solution.

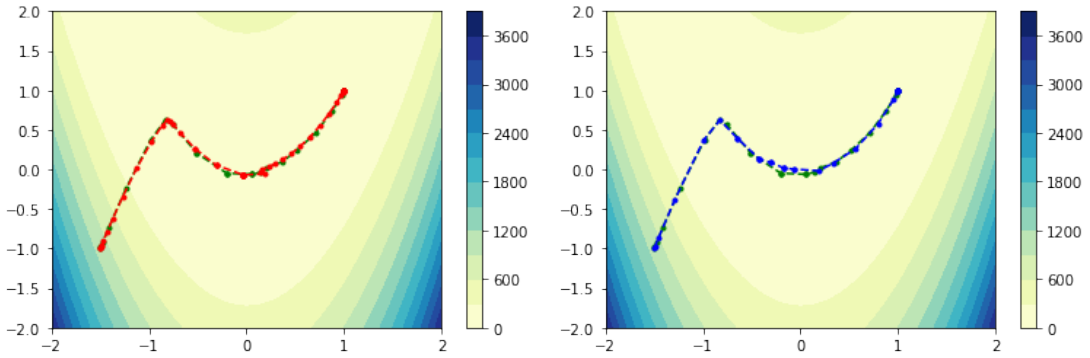

Figure 4

2.3 Discussion

Taking in consider the issue that the line search smears a bit the conjugate gradient method and the Levenberg-Marquardt do not have any issue that makes its performance worse, we can compare both algorithm. We know that Levenberg-Marquardt algorithm is a better or improved algorithm because it mix Gauss-Newton method with gradient descent method and in fact this has been shown in our results. Also, both methods perform in a time perspective. As we know that is not very fair to compare this two methods as one performs way better than the other, we are going to compare the Levenberg-Marquardt algorithm with the two of best of the conjugate gradient descent using exact line search.

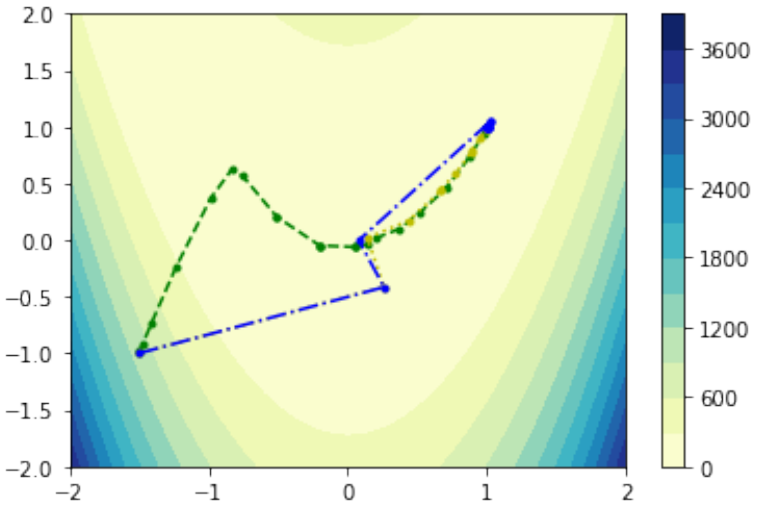

Figure 5

As we see all methods goes through (0,0) and meet there and it makes sense because we are going from both negative coordenate points to both positive ones.In fact, Levenberg-Marquardt seems to be smoothier than the conjugate gradient that takes longers steps and then correct them.

3 Conclusions

In this project we were asked to program two different methods for the Rosenbrock's function. This function is used as a test for algorithms and we have shown why. Some of the methods used have been shown as problematic or to converge slowly, due principally to line searching. Others as Levenberg-Marquardt have been shown to be full well functioning. This tests and graphics have been matched with theoretical shots in order to undestand and give a better explanation of what we are working with.

4 Appendix

As commented before, no external library has been used to write this code unless the time and plot ones.

4.1 Conjugate Gradient Descent Algorithm Code

Below is written the code for conjugate gradient descent algorithm. It is a python function that uses line search or brute force to get the α_k and it returns the 'path' the algorithm follows to get the solution of our problem. Number of max iters, methods, tau and rho can be inputed.

```
def ncg(x0,max_iter,epsilon,method,tau,rho,beta_option = True):
    start = time.time()
    x = x0
    track_x = [x]
    f = rosenbrock(x)
    g = gradient(x)
    d = [[-g[0][0]],[-g[1][0]]] #steepest descent direction
    iter = 0

    while p_norm(g) > epsilon and iter < max_iter:
        #alpha = brute_force(x,d) #for exact line search
        alpha = backtracking(x,iter, tau, rho) #for backtracking
        x = matrix_sum(x, [[alpha*i[0]] for i in d]  )
        g1 = gradient(x)
        if method == "Fletcher-Reeves":
            if iter % 2 == 0:
                beta = 0
            else:
                beta = multiply(transpose(g1),g1)[0][0]  / multiply(
                ↪  transpose(g),g)[0][0]
        elif method == "Polak-Ribiere":
            beta = multiply(transpose(g1), substract(g1,g))[0][0] /
            ↪  multiply(transpose(g),g)[0][0]
        elif method == "Hestenes-Stiefel":
            beta = multiply( transpose(g1), substract(g1,g) )[0][0] / multiply(
            ↪  transpose(d), substract(g1,g))[0][0]
        elif method == "Dai{Yuan":
            beta = multiply( transpose(g1),g1 )[0][0] / multiply( transpose(d),
            ↪  substract(g1,g))[0][0]
```

```
        g = g1
        if beta_option == True and beta < 0: #In some methods this ensure
        ↪  convergence in more cases
            beta = 0
        d = matrix_sum( [[-g[0][0]],[-g[1][0]]] ,  [[beta*i[0]] for i in d]  )
        iter = iter +1
        track_x.append(x)
    end = time.time()
    print("Time elapsed:" , end-start)
    print("Number of iterations:", iter)
    print("Algorithm has converged to the point",(x[0][0],x[1][0]))
    return track_x
```

4.2 Levenberg-Marquardt Algorithm Code

Below is the python function for the Levenberg-Marquardt algorithm. It returns the values of each iteration as the conjugate gradiente descent does. It can be inputed the max iterations and the error approach for the solution (epsilon). As we are solving a least squares problem we make use of the residuals (r).

```
def LM(x0,a,b, epsilon, max_iter=1000):
    start = time.time()
    track_x = []
    iter = 0;
    x = x0;
    J = jacobian(x)
    JT = transpose(J)
    mu = max(diagonal( multiply( JT,J  )))

    while iter <= max_iter and sqrt((x[0][0]-1)**2+(x[1][0]-1)**2) > epsilon:
        J = jacobian(x)
        JT = transpose(J)
        JT_J =  multiply( JT,J  )

        r = residuals(x)
        #I = [[mu,0],[0,mu]] for Levenberg algorithm
        I = [[mu*JT_J[0][0],0],[0,mu*JT_J[1][1]]] #for Levenberg-Marquardt
        ↪  algorithm
        g = matrix_sum(JT_J,I)
        grad_C = multiply(JT,r)
        C = r[0][0]**2 + r[1][0]**2
        inv_g = inverse_matrix(g)
        xnew = matrix_sum(x,multiply(inv_g,grad_C))
        rnew = residuals(xnew)
        Cnew = rnew[0][0]**2 + rnew[1][0]**2
        if Cnew < C:
            x = xnew
            r = rnew
            mu = mu/a
        else:
            mu = mu*b
        track_x.append(x)
        iter = iter +1
```

```
print("Number of iters = ", iter)
print("Algorithm converged to",(x[0][0],x[1][0]))
end = time.time()
print("Time elapsed:" , end-start)
return (track_x)
```

References

[1] Alsedà, L. (2021). Optimisation. Master's degree in Modlling for Science and Engineering. http://mat.uab.cat/~alseda/MasterOpt/index.html. [Online; accesed 07-January-2021].

[2] Brunet, F. (2010). Contributions to Parametric Image Registration and 3D Surface Reconstruction. pp. 27-44. https://www.brnt.eu/phd/node10.html [Online; accesed 09-January-2021].

[3] Quarteroni, A. Sacco R. and Saleri F. (2000) Numerical Mathematics. pp. 281-327.

[4] Wikipedia. Convex function. https://en.wikipedia.org/wiki/Convex_function

[5] Yu-Hong Dai. Nonlinear Conjugate Gradient Methods. http://www.apmath.spbu.ru/cnsa/pdf/obzor/Nonlinear%20Conjugate%20Gradient%20Methods.pdf. [Online; accesed 09-January-2021]

[6] Zinn-Bjorkman, L. Numerical Optimization using the Levenberg-Marquardt Algorithm. https://mads.lanl.gov/presentations/Leif_LM_presentation_m.pdf [Online; accesed 09-January-2021]