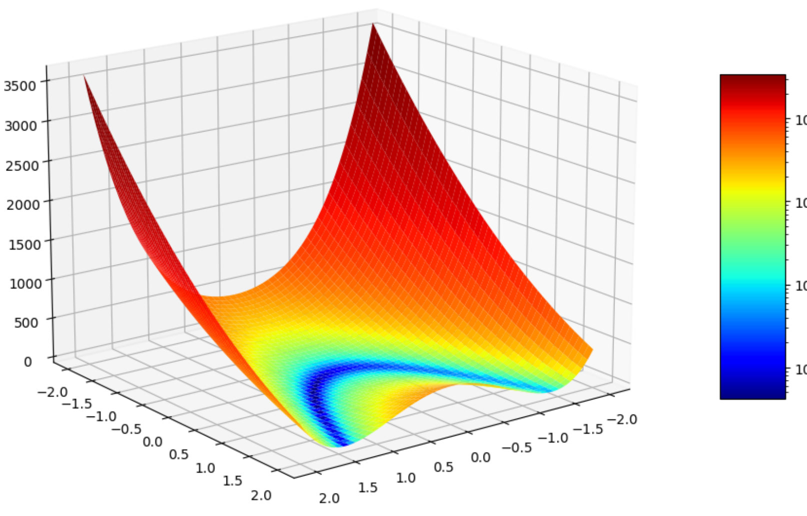
DETERMINISTIC OPTIMISATION

Minimum searching with Conjugate Gradient and Levenberg-Marquardt method



Oriol Fernández Serracanta 1426251

January 10, 2021

1 Introduction

The Rosenbrock function is a function introduced by Howard Rosenbrock, its known for being a non-convex function on the world of mathematical optimisation

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

This function has quite peculiar shape, with a minimum on the point $(x, y) = (a, a^2)$. In our case with $b = 100$ and $a = 1$ the minimum will be in $(1, 1)$.
At first sight this function seems inofensive but with a closer look we can see that there are points with zero gradient, these points can lead to a "fake minimum" and confuse methods that work with gradients.

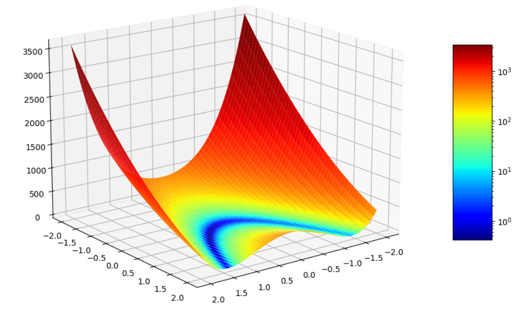
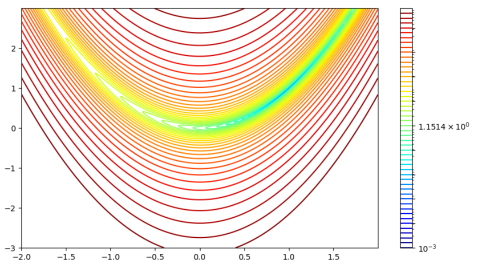
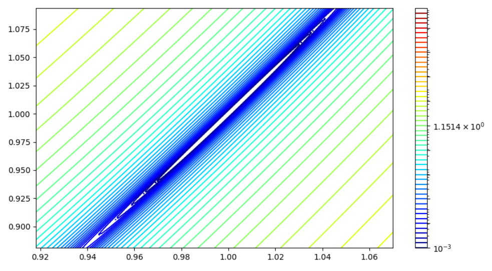


Figure 1: 3D plot of the Rosenbrock function with logarithmic colormap

Also the global minimum has a very narrow point of zero gradient surrounded by steep points around. The algorithm will be tricked easily skipping the real minimum and going to zero gradient points somewhere else in the minima valley. The problem will be to find this minimum from the point $(-1.5, -1)$ with both Conjugate Gradient Method and Levenberg-Marquardt method. The seed $(-1.5, -1)$ seems to be a quite tricky starting point, that is because it is situated on the other side of the slope, so the algorithms will need to first reach the valley and after follow the valley in the direction of the minimum. This coerces the algorithm to go through the valley to reach the minimum, by bibliography the difficult zone of the function.



(a)



(b)

Figure 2: (a) general contour line plot of the function, (b) zoom in on the narrow minimum point.

2 Conjugate Gradient Method

For the Conjugate Gradient Method we used the Fletcher-Reeves β and we used the golden search algorithm for finding the minimum on the gradient direction for every orthogonal step. The code is on section 3. Here we have a small pseudocode of the algorithm:

```
initialize(points,seed,epsilon,maxiter,step=-grad(X))
while( |grad(Xnew)|>epsilon || iter<maxiter ):
    alpha=Golden_Search(X,step,delta)
    Xnew=X+alpha*step
    points.append(X)
    Beta=(grad(Xnew)*grad(Xnew))/(grad(X)*grad(X))
    step=-grad(Xnew)+Beta*step
    iter++
    X=Xnew
```

This algorithm with the Rosenbrock function showed a quite caracteristic behaviour: in the beginning the steps are quite robust and go quite fast to the minimum, but although it reaches the minimum zone fast it takes a while to get closer and closer, when it is close the minimum is just making "blind steps" around the minimum and little by little improving the accuracy. This blindness and this prowling behaviour of the algorithm around the minimum can be changed with the inner α-searching algorithm, that sub-algorithm has an accuracy too which if we set it higher we can improve the outer number of iterations.
In summary we have two main tunable parameters, ϵ which gives you the final algorithm tolerance and tells you how close you want the algorithm to be to the minimum before you stop, and δ which tells you how much accuracy do you want to have on the Golden Search, and so how accurate you want the alpha.

2.1 Golden Search

The algorithm used in the sub-algorithm for finding the α lenght is the called "Golden Search" or "Golden Section" algorithm.
This algorithm is used to find an extreme, either maximums or minimums of a function inside an specific interval. It works for unimodal functions with extreme inside the interval.
The idea of the algorithm is similar to the Bolzano theorem for searching zeros, it consists on narrowing the intervals in where the minimum or maximum is. The name of the algorithm comes from the fact that it uses the golden ratio in order to get the intervals, the three intervals relative widths are $2-\phi, 2\phi-3, 2-\phi$. These ratios are kept for all the steps and are maximally efficient. excepting if extremes are on boundary points.
As a curiosity Golden search is the limit of the Fibonacci search for many function evaluations. The pseudocode adapting the golden search for a determined direction in a 2d function is the following:

```
Golden_Search(interval_i,interval_f,delta,x,dir)
    x_in=x[0]
    y_in=x[1]
    a=interval_i
    d=interval_f
    phi=((-1+sqrt(5))/2)
    while(abs(a-d)>delta):
        step=phi(d-a)
        b=a+step
        c=d-step
        fc=f(x+c*dir[0],y+c*dir[1])
        fb=f(x+b*dir[0],y+b*dir[1])
        if(fc<fb):
            a=a
            d=c
        else:
            d=d
            a=b
        min=(d-a)/2
    return min
```

being the Golden search a good choice for a sub-algorithm because it has a very good complexity $(\mathcal{O}(\log(\frac{1}{\delta}))$ for example in order to reach a precision of 10^{-6} there are needed less than 14 steps. And it is tunable with the value of δ, so if our program is taking a lot of time computing the alpha value we can tune it and reduce its precision making the golden search faster.

2.2 Optimisation of ϵ and δ

We could say that putting a very low δ is the key so we would obtain the result in less iterations but we should take in account the minimization time of the Golden search, so in order to take all in account we measured the total elapsed time and plotted this times in terms of ϵ and δ: Here we can see how the performance varies substantially

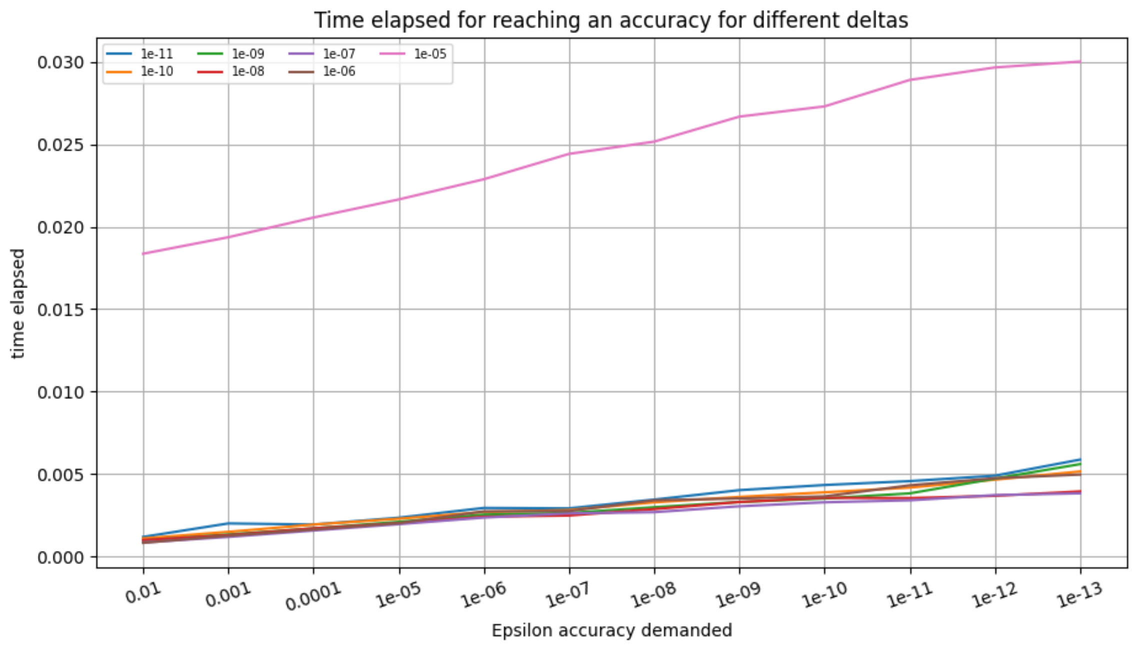


Figure 3: Iterations needed to achieve a certain accuracy, every plot is a different δ

by changing the δ value. By far we see how putting a small delta might improve the Golden search time but has serious repercussions on the global algorithm up to a point that putting a $\delta < 10^{-5}$ was totally messing up the algorithm and it was not finishing for the accuracies plotted. So we can extract that the lowest δ the worse. for not very precise accuracies (ϵ) it doesn't matter to choose between $\delta = [10^{-6} - 10^{-12}$ but for better accuracies it results that very small deltas start to affect the performance, what we found is that the optimal delta values are between $10^{-7} - 10^{-8}$ which have a very low increase on the elapsed time when increasing the accuracy. The value we will use to compare it with the LM method are that for an accuracy of 10^{-9} it took 40 steps with a $\delta = 10^{-8}$ and 41 steps with a $\delta = 10^{-7}$

3 Levenberg-Marquardt Method

The Levenberg-Marquardt is an hybrid method between the steepest descent or gradient method and the Newton method, for large lambdas we have the Gradient descent method and for small ones the algorithm tends to Gauss-Newton method.
This method is powerful because it modifies the lambda depending on the situation of $f(X)$ and $f(X_{new})$. In most cases when we are approaching the minimum ($f(X) > f(X_{new})$ the algorithm tends to a Gauss-Newton method. While when it is searching for the correct direction ($f(X) < f(X_{new})$ the algorithm tends to a gradient descent trying to get the correct direction.
Here we have a small pseudocode of the algorithm

```
Initialize(seed,maxiter,Lambda,epsilon,points)
while( |grad(Xnew)|>epsilon || i<maxiter ):
    step=inverse(Hessian(X)+Lambda*I)*-Grad(X)
    Xnew=X+step
    if(f(Xnew)<f(X)):
        Lambda=Lambda/2
    elif(f(Xnew)>f(X)):
        Lambda=2*Lambda
    points.append(X)
    X=Xnew
    iter++
```

The fact that λ is upgraded every iteration makes the algorithm very adaptable to different situations. And also λ has some "memory", if we come from a very lost situation with a large λ and suddenly we find a correct descent direction, lambda will decrease but still be large.
This facts make Levenberg-Marquardt an algorithm with a better performance than the Conjugate-Gradient method.
We could also study what happens when we vary the value of initial λ, we will do a plot showing how many steps are needed to reach certain tolerances depending on the initial λ value Here we can observe how for smaller λ the

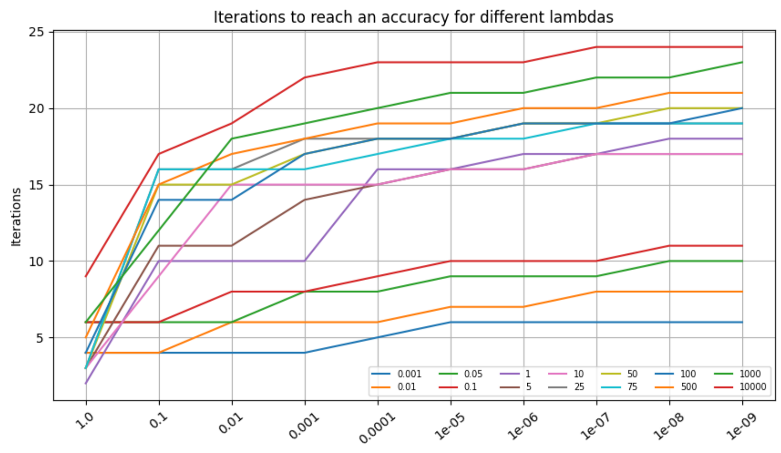


Figure 4: Iterations needed to reach different accuracies for a sort of λ

number of steps needed to reach the accuracy is really low, in fact we can see a gap separating the regime $\lambda < 0$ and the regime $\lambda > 0$. As a qualitative analysis we can say that for $\lambda < 1$ the algorithm has a exploitatory behaviour being more similar to Gauss-Newton method and gets to the point faster, for $\lambda > 1$ we have an algorithm closer to the gradient descent making more steps and in conclusion taking a bit more time to reach the desired point.
Anyway in general the number of iterations in order to reach the point is really small being in the worst cases of λ lower than 25 iterations for an accuracy of 10^{-9} which is a quite good accuracy.

4 Conclusions

We have been able to find a minimum of a non-convex function like the Rosenbrock function with both Conjugate Gradient Method and Levenberg-Marquardt Method.
We observed how the number of iterations was changing in terms of the accuracy we asked for in the final result.

For the Conjugate Gradient Method we found that it took about 40 steps to reach the minimum with an accuracy of 10^{-9} in a total elapsed time of $0.00380s$ which shows a very fast convergence to the minimum knowing that we are treating with a function which was used as a performance test problem for mathematical optimisation.
Golden Search algorithm was used to minimize the value of alpha and so we could get a fast response for a part of this algorithm. We could also see how the performance of the CGM is very sensible to a parameter of Golden Search algorithm and we had to find for the optimal δ(which is the accuracy of the Golden Search) that gets better performance on the global CGM, in our case we found small deltas are needed if we want to ensure the convergence, and in terms of performance, all deltas work pretty good but for high demanding accuracies we purposed a $\delta = [10^{-7} - 10^{-8}$ which has the smaller slope time elapsed-accuracy. Conjugate Gradient Method is a quite good method which after tuning and checking a bit the parameters, can lead to very good results, nonetheless a disadvantage is the heaviness of the algorithm having a minimisation algorithm inside every step.

On the other hand for the Levenberg-Marquardt method we saw how it took less than 30 steps to reach accuracies of 109 which are really good results, in the best cases ($\lambda = 0.0001$) it took 7 iterations and $0.0002526s$ which shows an improvement of more than 10x compared to Conjugate Gradient Method.
This shows Levenberg-Marquardt performed better in this problem due to the duality and adaptability of the model (working as Gradient descent or Gauss-Newton method depending on the situation) despite of being a more "simple" algorithm.
A small performance analysis was also done in the LM method by checking which values of initial λ give better results. After checking different lambdas for different accuracies we got that the smaller the lambda, the faster the algorithm was getting to the minimum.

As an hypothesis for this we got that lambda value shows more one face of the dual algorithm behaviour, having for small lambdas a more ”Gauss-Newton” behaviour and for large lambdas a more ”Gradient Descent” behaviour, from here we thought that the seed point was good enough so that the algorithm doesn’t have to explore that much, that gives advantage to the Gauss-Newton method which was going more directly to the point. For a worse seed maybe large lambdas would have given a better result making that the algorithm doesn’t get lost that easily.
For other functions we recommend to do a lambda performance study before selecting a final lambda, so we can see which of the two behaviours of the algorithm fits better for the start according to the seed.

Moreover we can suggest the Conjugate Gradient method will have different performance results depending on the 1-D minimization algorithm for the α searching, we used Golden Search which seems to be quite good but other algorithms could be implemented like Backtracking, we suspect this changes can change the performance of the algorithm but anyway they seem to stay far from the Levenberg-Marquardt method.

In conclusion we found the minimum of this function with both algorithms and used Python to do a deeper analysis with different kinds of plots, the program could be easily implemented in C by following the pseudocode purposed on each algorithm section but Python was used to optimize time on writing the code and making the results analysis. C or a compiled program is not necessary in this cases and an interpreted language as Python performed well running ensembles of the algorithm (The lambda analysis with 10 accuracies and 14 lambdas which is a total of 140 algorithm runs took less than 10 seconds), the only case where C would start to make a difference is in a demanding accuracy in the case of Conjugate Gradient Method which in the case of $1e15$ in Python was taking a bit more than 60 seconds.