Advanced Optimization Open Source Project

CentraleSupelec / ESSEC DSBA M2

Covariance Matrix Adaptation Evolution Strategy

Distance to optimal distribution

Team

1. Gaël DENIZ - B00792533
2. Jedrzej ALCHIMOWICZ - B00780728
3. Johanna OTT - B00782280
4. Meduri Venkata Shivaditya - B00780791
5. Raghuwansh RAJ - B00793388

Index

Description of the CMA-ES algorithm

For our project we worked on the python implementation of the CMA-ES optimization algorithm. For the github page please see: https://github.com/CMA-ES/pycma/issues.

To introduce the algorithm on a high-level, we cite here the description used on the github publication of it: "CMA-ES is an evolutionary algorithm for difficult non-linear non-convex black-box optimisation problems in continuous domain. The CMA-ES is considered as state-of-the-art in evolutionary computation and has been adopted as one of the standard tools for continuous optimisation in many (probably hundreds of) research labs and industrial environments around the world. The CMA-ES is typically applied to unconstrained or bounded constraint optimization problems, and search space dimensions between three and a hundred. The method should be applied, if derivative based methods, e.g. quasi-Newton BFGS or conjugate gradient, (supposedly) fail due to a rugged search landscape (e.g. discontinuities, sharp bends or ridges, noise, local optima, outliers). If second order derivative based methods are successful, they are usually faster than the CMA-ES: on purely convex-quadratic functions, $f(x)=x^THx$, BFGS (Matlabs function fminunc) is typically faster by a factor of about ten (in terms of number of objective function evaluations needed to reach a target function value, assuming that gradients are not available). On the most simple quadratic function $f(x)=||x||^2=x^Tx$ BFGS is faster by a factor of about 30." (cma-es.github.io, last used 01.12.2021)

Motivation behind choosing this project

CMA ES is a state of the art algorithm in derivative free optimization. It can be considered to be one of the top algorithms in 2019. We are honoured to contribute to its analysis and performance testing. The algorithm's implementation in python is a big github project. We hope our code will be accepted and we hope it will help further improve the algorithm.

Our work was centered very specifically on the covariance matrix adaptation. With our implementation, the user can see how quickly the covariance matrix approaches the optimal matrix. This will help him to better understand on which functions the algorithm works better. It will also be of great help to tune the learning rate.

Description of the issue solved

We worked on solving issue #70 (https://github.com/CMA-ES/pycma/issues/70). The issue description was provided by the owner of the repository, Nikolaus Hansen, and divided into a few steps: a basic description, an example and a coding hint.

Basic Description

The basic description states: 'It would be nice to (optimally) show some distance measure to the optimal distribution, if the optimal distribution is known'. Then, we were given a hint as to what distance metric to use:

$\sqrt{\sum_i \log^2(\lambda_i)}$ is a most useful distance to the identity covariance matrix.

Example

As for the example, Nikolaus Hansen has provided us with an example for a specific case of convex quadratic function.

On a convex quadratic function with Hessian H, we compute $\Lambda = \sqrt{H}C\sqrt{H}$, where $\sqrt{H}$ is the symmetric matrix which satisfies $H = \sqrt{H} \times \sqrt{H}$ and C is the covariance matrix of CMA-ES. Let λ_i be the eigenvalues of Λ. Then, our distance measure is

$$D(C,H) = \sqrt{\sum_i (\ln \lambda_i - \overline{\ln \lambda})^2} = \min_\alpha \sqrt{\sum_i (\alpha + \ln \lambda_i)^2}$$

where $\overline{\ln \lambda} = \frac{1}{n}\sum_i \ln \lambda_i$ is the mean of the log eigenvalues.

Structure of implementation suggested

Lastly, we were given a hint for how to implement our solution within the code. There are two basic ways to call the algorithm. First one is through the optimize method of the CMAEvolutionStrategy class. The second one is more step-by-step and utilizes ask/tell functionality. The second method was suggested to us by the repository owner:

```
import cma

es = cma.CMAEvolutionStrategy(4 * [1], 1)
es.sm.C  # is the covariance matrix

fun = cma.ff.sphere
# distance = Distance(fun)  # a callable class instance to compute the distance
while not es.stop():
    X = es.ask()
    es.tell(X, [fun(x) for x in X])
    # es.more_to_write.append(distance(es.sm.C))  # is a list of values plotted additionally in the upper left plot
    es.logger.add()
es.plot()
```

Our understanding of the problem and implementation plan

A significant part of our project was grasping the algorithm and understanding what exactly the distance measure is. At first, we focused on the case of a convex quadratic function as per the example. In this case we can easily calculate the hessian matrix, which is exactly the inverse of the covariance matrix at the optimum, and use it to calculate the distance measure at each iteration. However, this is a very specific case and in fact, it is not a very difficult case. CMA-ES algorithm, like many other stochastic derivative-free algorithms, is most used for solving difficult problems (non-convex, ill-conditioned, multi-modal, rugged, noisy). Therefore, we needed to extend our thinking. For functions where we could not find the hessian matrix, we decided to measure the distance to the identity covariance matrix, which basically represents the distance to the starting point. It is important to notice that in this case the distance will increase as we get closer to the optimum.

Therefore, in the end we came to the conclusion that we need to calculate the distance between the covariance matrix at a given optimization step and either a given matrix (which would represent distance to the optimum) or an identity matrix (which would represent distance to the starting point).

:

Structure of the Implementation of distance metric feature into the CMA-ES library

Our implementation works in the following way: when the user decides to use the step-by-step solution, they can decide to calculate the distance measure at each step of the optimization. To follow the structure of the already implemented code of CMA-ES we created a class Distance which has a function to calculate the distance between an inputted covariance matrix and the matrix to which we calculate the distance to (we will call it the 'optimal matrix' for simplicity, even though depending on the implementation it may not always mean truly the optimal covariance matrix). The optimal matrix is set at the initialization of the class object and can be one of the three things: an arbitrary matrix inputted by the user, an identity matrix or a matrix calculated by us. The way it is set goes as follows:

1. If the user set type = 'identity' we set the optimal matrix to identity matrix
2. If the user set type = 'known', the user needs to pass one more argument which includes a matrix, and we set the optimal matrix to that imputed by the user
3. If the user set type = 'optimal' we run the optimization algorithm to obtain the optimal covariance matrix, and set the optimal matrix to the output. We are aware that this implementation requires to run the optimization twice (first to get optimal covariance matrix, second as usual but we can now calculate the distance to the covariance matrix). We believe that if the calculation of the distance is important for the user, it is acceptable to wait longer for the implementation.

After the optimal matrix is set we can easily calculate the distance to it using the provided formula. Therefore, after each step of the optimization (or after each ask/tell cycle) we call the Distance object, input the current covariance matrix and calculate the distance from it to the optimal matrix. The values at each step are stored in an array, and are at the end used for visualisation purposes.

```
class distance:
    def __init__(self, es, func, type, Hessian = None):
        self.type = type
        if type == 'optimal':
            #es_optimal = cma.CMAEvolutionStrategy(**es.inputargs)
            #es_optimal.optimize(func)
            xopt, es_optimal = cma.fmin2(func, es.inputargs['x0'], es.inputargs['sigma0'], {'verb_disp' : 0})
            self.optimal_C = es_optimal.C
        elif type == 'identity':
            self.optimal_C = np.identity(len(es.inputargs['x0']))
        elif type == 'known':
            self.hessian =  Hessian
        return
    def distance(self, C):
        if self.type == 'known':
            Hes = self.hessian
        else:
            Hes = np.linalg.inv(self.optimal_C)
        square_root_H = sqrtm(Hes)
        M = square_root_H @ C @ square_root_H

        eigenvalues_M, eigenvectors_M = np.linalg.eig(M)

        mean_log_eigenvalues_M = np.mean(np.log(eigenvalues_M))

        distance = 0
        for eigenvalue in eigenvalues_M:
            distance += (math.log(eigenvalue) - mean_log_eigenvalues_M)**2

        return math.sqrt(distance)
```

Understanding the distance metric

Distance to Hessian

The covariance matrix signifies the gaussian variance of the parameter distribution vector. The hessian at the optimal point tells us about the shape of the graph at the optimal point. if the variance of the parameter distribution as the optimal is low, it can be thought of as small changes in the parameter reflect huge changes in the objective function value, so the graph is steep signalling high value of second derivate of objective function w.r.t that parameter, Therefore we can see that the covariance matrix of the parameter distribution is proportional to the inverse of the Hessian. This is a key step for us in understanding how we can measure the distance to optimal using the Covariance matrix and the Hessian. Lambdas in the formula refer to the eigen values of $\Lambda = \sqrt{H}C\sqrt{H}$,

$$D(C,H) = \sqrt{\sum_i (\ln\lambda_i - \overline{\ln\lambda})^2} = \min_\alpha \sqrt{\sum_i (\alpha + \ln\lambda_i)^2}$$

where $\overline{\ln\lambda} = \frac{1}{n}\sum_i \ln\lambda_i$ is the mean of the log eigenvalues.

Distance to Identity

While initializing the parameter distribution for the CMA-ES algorithm, we initialize the covariance matric to Identity matrix. As the iterations flow, the covariance matrix is adapted at each step and inches it's way towards the actual parameter distibution. We can visualize how the iterations are effecting the covariance matrix by plotting the distance vector which is populated throughout the iterations. Distance to identty can can computer very simply using the covariance matrix at an iteration point using the formula below

$$D(C, I) = \sqrt{\sum_i \log^2(\lambda_i)}$$

Using these 2 formulas, we added the distance metric feature to the CMA library. Please go through the .ipynb file with this report to see how to use the features created can be used.

Implementation of the distance metric feature into the CMA-ES library

A new file called distance.py is added which includes the feature to calculate the distance metric. It is a class which can be initialized using CMAEvolutionaryStrategy class object created, the objective function and type of metric we would like to calculated. The various use cases of the distance class is mentioned below.

Hessian at the optimal is known

The user can choose from ‘known’ in cases where he/she is aware of the hessian at the optimal, or can use the automatic hessian calculation function implemented which can easily calculate the hessian matrix of any quadratic function.

Hessian at the optimal is unknown

Approximating Hessian

In this case, hessian at the optimal can be approximated using the inverse of the covariance matrix of the parameter distribution. Choosing type as optimal will enable the feature to automatically calculate the approximate hessian and use it to calculate the distance. This method is general and is not limited cases only where hessian is computable but it can approximate hessian for any function.

Distance to Identity

In this case, Distance to identity metric discussed above is used. To enable using the ditstance to identity matrix, choose type as identity. This method is general as well, and can be used to perform distance analysis of any function while optimizing it.

Error Handling

The distance metric is prone to raising math errors. Situations where the distance metric cannot be computed include cases where any of eigen value is 0 or eigen value is negative. Therefore before computing the distance metric, the feature implementation is concious of this problem and raises the appropriate error allowing the user to understand why an error occured

Plotting

Suitable line plot methods are written to the distance class which allows the user to plot all the distances over the iterations. This feature is made possible by creating a logger list for each distance object which stores the values of the distances with each iteration

Implementation of the automatic hessian calculation for quadratic polynomial function

Hessian calculation

This subprocedure of the overall solution has as input a function and its dimension. It outputs a boolean value that indicates whether a Hessian meeting certain conditions could be found and if it could be found than also the Hessian. The Hessian of a function is a matrix containing the 2nd order derivatives of this function with respect to all its variables. Mathematically, it is written as:

$$(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}$$

For this procedure the following functions were used. They are explained below:

1. partial
2. gradient
3. hessian
4. prep_dimensions
5. conditiones_fullfilled_and_if_yes_hessian
6. conditiones_fullfilled_and_if_yes_hessian_with_error_check

Code parts that have been used for this are coming from the following sources: blog post Hessian Matrix and Optimization Problems in Python 3.8 from Lous Brulé Naudet, link: https://towardsdatascience.com/hessian-matrix-and-optimization-problems-in-python-3-8-f7cd2a615371) and the python package `sympy` and numpy have been used extensively.

Description of each step: what happens + theory behind it + which modules are used

Function: partial

This function will be given a function consisting out of one or several variables and one of these variables (in the code referred to as “element”). It derives the partial derivative with respect to that given variable, i.e. the other variables are treated as constants.
Exemplary Use:

```
print("Apply partial to the function",function,"with respect to the symbol",symbols_list[0])
print("The result is:")
partial(symbols_list[0],function)
```

executed in 7ms, finished 06:16:06 2021-12-02

```
Apply partial to the function x_0**2 + x_1 with respect to the symbol x_0
The result is:
```

$2x_0$

Function: gradient

This function will be given an array of partials derived from the previous function and outputs the gradient in the form of a matrix.

Exemplary Use:

```
print("After applying the partial function to every element of the function:",function)
print("Apply gradient to the derived partials",partials)
print("The result is:")
gradient(partials)

```

executed in 7ms, finished 06:16:26 2021-12-02

```
After applying the partial function to every element of the function: x_0**2 + x_1
Apply gradient to the derived partials [2*x_0, 1]
The result is:
```

```
matrix([[2*x_0, 1]], dtype=object)
```

Function: hessian

This function makes use of the previously described function partial. It will be given a function and outputs the hessian of that function (https://en.wikipedia.org/wiki/Hessian_matrix).

Exemplary Use:

```
user_input_function_string = "10*x_0**2 + x_1**2+x_2**2+x_3**2+x_4**2"
function = sympify(user_input_function_string)
symbols_list = prep_dimensions(5)


hessian(symbols_list, function)
```

```
matrix([[20, 0, 0, 0, 0],
        [0, 2, 0, 0, 0],
        [0, 0, 2, 0, 0],
        [0, 0, 0, 2, 0],
        [0, 0, 0, 0, 2]], dtype=object)
```

Function: prep_dimensions

This function takes an integer, dimensions, as input. It creates that amount of sympy variables and returns a list containing these variables.

Exemplary Use:

```
Entrée [4]:  prep_dimensions(5)

Out[4]:  [x_0, x_1, x_2, x_3, x_4]
```

Function: Conditiones_fullfilled_and_if_yes_hessian
The goal of this function is to check whether our function is convex quadratic with positive, semi-definite hessian. If this is the case it will return (True, hessian), otherwise it will return False. We basically wrote this function that works well for convex quadratic functions if you input them as polynomials. As many things can lead to errors in this function, we use the function below to return False might an error appear.

More technically, we take a function and its dimensions. We calculate the partials and the hessian and return this. Note that this function can be simplified.

Exemplary Use:
See the function below for examples.

Function: Conditiones_fullfilled_and_if_yes_hessian_with_error_check
This function adds to the Conditiones_fullfilled_and_if_yes_hessian an error handling functionality. It catches any error made in the previous steps. If the Hessian calculation could not be used for example because it does exist only with an imaginary part or because the requirement of a semi-positive definite matrix is not met, it returns false . If there were no such circumstances, it passes on the Hessian.
Exemplary Use:

- input: not positive semi-definite function:

```
1  # user input   (variables are given as x_i for i element of [1,dimensions])
2  user_input_function_string= "x_0**1-(3/2)*x_0*x_1+x_1**2"
3  dimensions = 2
4
5  #application of main function
6  output = conditiones_fullfilled_and_if_yes_hessian_with_error_check(
7      dimensions,user_input_function_string)
8
9  #output
10 print("for the given user input",sympify(user_input_function_string),
11       "we receive the following output:",output)
```

executed in 10ms, finished 16:12:04 2021-12-01

```
Returned False on whether hessian semi-positive definite
for the given user input -3*x_0*x_1/2 + x_0 + x_1**2 we receive the following output: False
```

(Note: since the function is not positive semi-definite, we get the output “False” which indicates that we can not use any output of this sub-part of code and have to use one of the alternatives described in the “Implementation and Integration of Distance metric functionality with CMA” chapter)

- input: positive semi-definite function:

```
]:
# user input   (variables are given as x_i for i element of [1,dimensions])
user_input_function_string= "x_0**2+x_1**1"
dimensions = 2

#application of main function
output = conditiones_fullfilled_and_if_yes_hessian_with_error_check(
    dimensions,user_input_function_string)

#output
print("for the given user input",sympify(user_input_function_string),
      "we receive the following output:",output)
```

executed in 8ms, finished 16:12:33 2021-12-01

```
for the given user input x_0**2 + x_1 we receive the following output: (True, Matrix([
[2, 0],
[0, 0]]))
```

(Note: no issue regarding the semi-definiteness of the function detected and thus the Hessian gets returned)

- Input: with imaginary parts. Gives false

```
Entrée [11]:
# user input should be:

# user please give you variables as x_i for i element of [1,dimensions]
user_input_function_string= "im(x_0**2 - (3/2)*x_0*x_1 + x_1**2)"
dimensions = 2

# then we call the function
output = conditiones_fullfilled_and_if_yes_hessian_with_error_check(dimensions,user_input_function_string)

# and here you can see the output
print("Output:",output)
```

```
Returned False on whether hessian semi-positive definite
Output: False
```

- Input: with imaginary numbers. Gives false

```
# user input should be:

# user please give you variables as x_i for i element of [1,dimensions]
user_input_function_string= "x_0**2*i- (3/2)*x_0*x_1 + x_1**2"
dimensions = 2

# then we call the function
output = conditiones_fullfilled_and_if_yes_hessian_with_error_check(dimensions,user_input_function_string)

# and here you can see the output
print("Output:",output)
```

```
Returned False on whether hessian semi-positive definite
Output: False
```

- Input: complicated mathematical functions like sin(1/x)

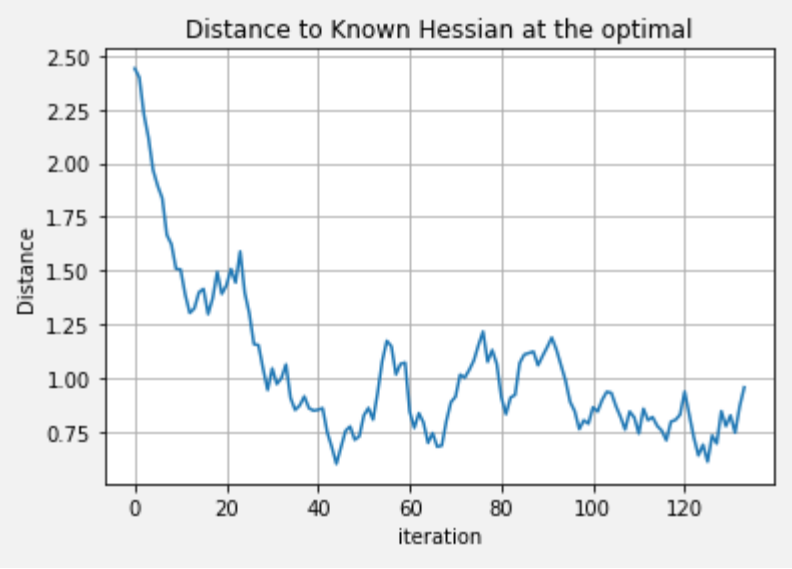
```
# user input should be:

# user please give you variables as x_i for i element of [1,dimensions]
user_input_function_string= "x_0**2*sin(1/x)"
dimensions = 2

# then we call the function
output = conditiones_fullfilled_and_if_yes_hessian_with_error_check(dimensions,user_input_function_string)

# and here you can see the output
print("Output:",output)
```

```
Returned False on whether hessian semi-positive definite
Output: False
```

Visualization

- Distance to Hessian Plot
- Distance to Identity Plot

Consider a convex quadratic function $10{x_0}^2 + {x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2$. We know that for all functions, the covariance matrix at the optimality is known to be the Inverse of the Hessian. The CMA ES algorithm at each iteration moves one step closer towards the optimal distribution. We tried to look how is the distance between covariance matrix C and optimal Covariance matrix H^{-1} is evolving with time.
From the plot shown below,It is clear that the moving average has a descending trend and the distance is decreasing with the number of iterations.
There is one important thing that needs to be looked at in the below plot. The plot has a gradual decreasing trend but at some point The distance started increasing ,now it is not a straightforward to explain such abnormality but one potential reason could be that the step size was bigger than the distance to optiman and hence we overshoot the optimal distribution Then slowly algorithms realised that ,the steps size was brought down and eventually the curve smoothes towards a distance.

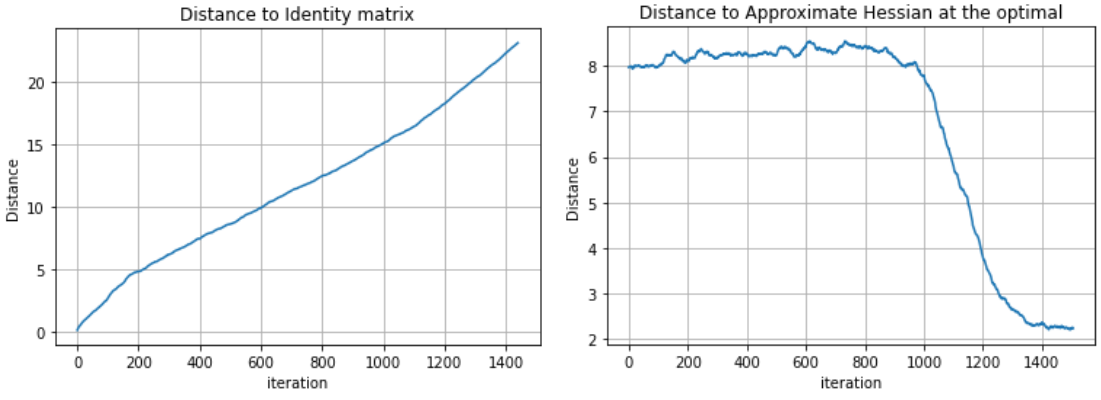


Test cases of using the new features added

Rosenbrock Function

Let us now consider an example objective function (https://en.wikipedia.org/wiki/Rosenbrock_function). Our code does not allow to calculcate the hessian of the Rosenbrock function if we import it like cma.ff.rosen . As this is the case, we calculate the distance from the covariance matrix to either the identity, or the optimal covariance matrix i.e. the final covariance matrix after running the code a first time beforehand. Both distances are plotted here below. We expect the distance to the identity to increase and approximate a constant. And the distance to the optimal to decrease to zero over time.

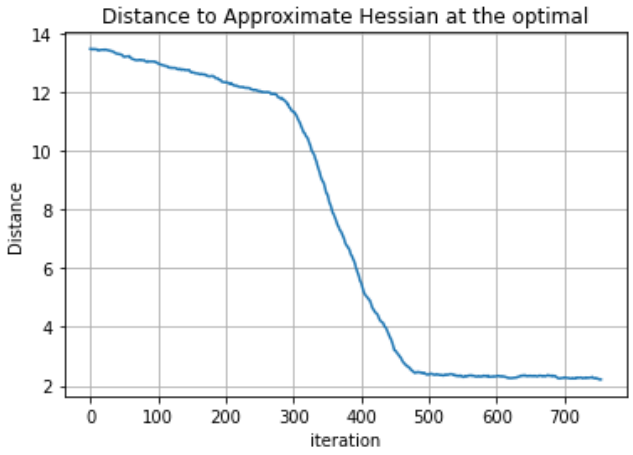
What we observe matches our expectations quite well. The first plot rises and the second decreases. However we see that after 1400 iterations, the distance to the optimal covariance matrix remains constant around 2.5 instead of further decaying to 0. We now do not know which of the two covariances is far from the optimal one. probably both are different. Note as well that in the first place the covariance matrix does not converge to the optimal. This is because far from the solution, the hessian is different, and because if the stepsize is to small, the covariance matrix we want (usually a large ellips in the) is very different from what we want close to the solution.



Cigar Function

Distance to Hessian

The cigar function that is implemented inside the CMA functional Package fits the definition of a convex quadratic function. Hence we tried to plot the distance of the Covariance matrix and the optimal Distribution matrix (here Inverse of the Hessian).The distance is decreasing with the iteration as per our expectations and finally smoothes towards a fixed value. The reason for a straight shape in the beginning can be explained by the adaptability of the step size.



Distance to Identity

In case the functions passed to the CMA ES algorithm tend not to be a convex quadratic function and the optimal distribution is not known to us. We try to see the distance to the Identity matrix. The Intuition for that is if after a continuous number of iteration The Distance plot seems flattened ,that could be an alias for the possibility the algorithm has reached towards an optimal and hence the step size covariance matrix is not changing much.