The rCMA Tutorial:
Examples for using CMA-ES in R

Wolfgang Konen,
Cologne University of Applied Sciences

May, 2015

1 Overview


rCMA realizes an R-binding to CMA-ES using package rJava [Urbanek 2013, 2014], the R-to-Java interface. The main features of rCMA are:

1. Ability to start the Java CMA-ES optimization with fitness functions defined in R.
2. Constraint handling: Arbitrary constraints can be incorporated, see function parameter isFeasible in cmaOptimDP.
3. Extensibility: Full access to all methods of the Java class CMAEvolutionStrategy through package rJava [Urbanek 2013, 2014]. New methods can be added easily. See the documentation of cmaEvalMeanX for further details, explanation of JNI types Oracle 2014 and a full example.
4. Test and Debug: The access of Java methods from R allows for easy debugging and test of programs using CMAEvolutionStrategy through R scripts without the necessity to change the underlying JAR file.

Note that package rCMA differs from package cmaes. cmaes realizes CMA completely in R, but has no methods for constraint handling and has only fewer parameters of CMA accessible than there are in Hansen’s Java class CMAEvolutionStrategy.

2 Installing rCMA

Once you have R (http://www.r-project.org/), > 2.14, up and running, simply install rCMA with
install.packages("rCMA");

Then, library rCMA is loaded with

library(rCMA);

If on starting rCMA there is an error related to rJava, see Appendix A.

3 Lessons

3.1 Lesson 1: Optimizing the 2D sphere problem

```r
# demoCMA1.R
fitFunc <- function(x) { sum(x*x); }
cma <- cmaNew(); cmaInit(cma,seed=42,dimension=2,initialX=1.5, initialStandardDeviations=0.2); res1 = cmaOptimDP(cma,fitFunc,iterPrint=30); plot(res1$fitnessVec,type="l",log="y",col="blue", xlab="Iteration",ylab="Fitness"); str(res1);
```

First we define with `fitFunc` the function to be minimized. It is the sphere function for arbitrary dimensions with its global minimum at the origin.

Next we construct in line 2 with `cmaNew()` a new CMA object which is an Java object of class `CMAEvolutionStrategy`. Various parameters of object `cma` (see rCMA Getters and Setters) could be set at this point, but we do not do it in this demo.

In line 3 the object `cma` is initialized with `cmaInit`. Several parameters are set, especially the dimension is set to $n = 2$. As a side effect, `cmaInit` sets the population size according to the usual CMA rule (see `https://www.lri.fr/~hansen/cmatutorial.pdf` Table 1):

$$\lambda = 4 + \lfloor 3\ln(n) \rfloor$$

which amounts to $\lambda = 6$ in our case (can be verified with `cmaGetPopulationSize(cma)`).

As a further side effect of `cmaInit`, the object `cma` is transformed to an augmented state such that no further modifications on its parameters are allowed. It is now ready for doing optimization.

The CMA optimization starts in line 4 with `cmaOptimDP` from the initial point `initialX=(1.5,1.5)`. Every `iterPrint=30` iterations a printout shows the optimization progress, until CMA terminates through one of its stop conditions. The printout from `cmaOptimDP` looks as follows:
## 0030 1.437185e-03 | 2.3652e-02, 2.9627e-02
## 0060 1.859678e-07 | 3.8003e-04, 2.0383e-04
## 0090 3.777315e-13 | -6.0544e-07, -1.0570e-07
## Terminated due to TolFun: function value changes below stopTolFun=1.0E-12 (iter=108,eval=648)
## cfe,ffe, %infeasible: 648 648 0.000000

The line

## 0030 1.437185e-03 | 2.3652e-02, 2.9627e-02

tells us that after 30 generations the best fitness value is 1.43e-03 with the corresponding best point in input space at (2.36e-02, 2.96e-02).

The termination message tells us that CMA stopped because the change in fitness value dropped below 1e-12 in iteration 108, at which time the fitness function was evaluated 648 times. The last line tells this again: 648 cfe (constraint function evaluations) and 648 ffe (fitness function evaluations) have been done, meaning that every individual was feasible (% infeasible=0.0), which is clear because the whole search space is feasible in this demo.

![Figure 1: The result of the plot-command from demoCMA1.R](image)

The call to `cmaOptimDP` returns an object `res1`, which is a list with several diagnostic informations about the CMA run. See `help(cmaOptimDP)` for further details. With the help of `res1$fitnessVec` we plot Fig. 1 with the command in line 5 showing the development of the ever-best fitness.
Finally, the last line depicts with `str(res1)` an overview of `res1`.

If we like to do the optimization of the sphere function in 100 dimensions, we only have to change `dimension=2` to `dimension=100` in the call to `cmaInit`.

This ends the first lesson and hopefully shows that it is fairly easy to set up and start a CMA optimization with the help of `rCMA`.

### 3.2 Lesson 2: Constrained optimization with rCMA

In this lesson we want to do a simple form of constrained optimization. `rCMA` offers the possibility to hand over a Boolean function `isFeasible(x)` to `cmaOptimDP`.

As an example we consider the problem TR2, which is the sphere problem with an additional tangent inequality constraint

\[ \sum_{i=1}^{n} x_i \geq n. \]  

Points below the tangent line passing through (1, 1) are infeasible. Fig. 2 depicts the situation and shows that the constrained optimum is at point (1, 1).

![Figure 2: Sketch of the TR2 problem. The green area is the infeasible region. The feasible region is the white area above and including the diagonal. The blue point at (1,1) is the optimum (minimum).](image)

Now we look at the code for solving this optimization problem. The first 5 lines are identical to Lesson 3.1:

```r
# demoCMA2.R
fitFunc <- function(x) { sum(x*x); }
n = 2;
cma <- cmaNew();
```
```r
cmaInit(cma, seed=42, dimension=n, initialX=1.5, initialStandardDeviations=0.2);
res1 = cmaOptimDP(cma, fitFunc, iterPrint=30);

isFeasible <- function(x) { (sum(x) - length(x)) >= 0; }
cma <- cmaNew();
cmaInit(cma, seed=42, dimension=n, initialX=1.5, initialStandardDeviations=0.2);
res2 = cmaOptimDP(cma, fitFunc, isFeasible, iterPrint=30);

fTarget = c(0, n);
plot(res1$fitnessVec-fTarget[1], type="l", log="y",
     xlim=c(1, max(res1$nIter, res2$nIter)),
     xlab="Iteration", ylab="Distance to target fitness");
lines(res2$fitnessVec-fTarget[2], col="red");
legend("topright", legend=c("TR2", "sphere"), lwd=rep(1, 2), col=c("red", "black"))
str(res2);
bestSolution=rCMA::cmaEvalMeanX(cma, fitFunc, isFeasible);
str(bestSolution);
```

In line 6 we define function `isFeasible` according to Eq. (2). Then we call in line 9 `cmaOptimDP` with `isFeasible` as the third argument. The termination message from `cmaOptimDP`:

```
## Terminated due to TolFun: function value changes below stopTolFun=1.0E-12 (iter=210, eval=1260)
## cfe, ffe, %infeasible: 1889 1260 0.499206
```

tells us that `cfe` exceeds `ffe` by roughly 50%, meaning that every second feasible check returned `FALSE`. This is in agreement with the expected placement of the CMA-ellipsoid in all but the first few iterations: Its mean is centered near the minimum `c(1,1)`, so it is at the border of feasibility. Then half of the individuals drawn at random from the distribution will be infeasible.

In line 11 we plot the TR2 result together with the unconstrained `sphere` result from `res1`. We see that it takes about twice as many iterations to solve TR2, but finally we reach a similar accuracy.

Now we look at the last two lines where `bestSolution` is calculated with the help of `cmaEvalMeanX`. It is stated in the CMA-tutorial [Hansen 2011] that the population mean from the last generation may be an even better solution than the best-so-far solution. With the help of `cmaEvalMeanX` we calculate this mean, compare its fitness value with the best-so-far solution and update `bestSolution`, if the mean is better and feasible. If the mean is better, then `bestEvalNum = lastEvalNum`. From the printout `str(bestSolution)` we see that it is not the case here:

```r
bestSolution=rCMA::cmaEvalMeanX(cma, fitFunc, isFeasible);
str(bestSolution);
```

```
## List of 5
```

# List of 5
Note that `bestX` and `meanX` are very close to the true optimum \(c(1,1)\). The difference in the order of \(1e^{-8}\) is only seen when subtracting the true optimum from `bestX` or `meanX`.

Again, as in Lesson 3.1, if we like to do the optimization TR2 in 100 dimensions, we only have to change `n=2` to `n=100` in line 2.

We close this lesson with a warning remark: The constraint handling approach is a very simple one: DP = death penalty. That is, if we get an infeasible individual, it is immediately discarded and a new one is drawn from the current CMA distribution. This approach will run into trouble (infinite while-loop) if the current distribution does not allow to reach any feasible solutions. But for the simple constrained problem TR2 it works well.

### 4 Further informations

Further informations on package `rJava` are found in Urbanek [2013, 2014].
Further informations on JNI (Java Native Interface) and JNI types are found in Oracle [2014].

A Fixing problems with the rJava installation

rCMA uses package rJava [Urbanek 2013, 2014] for Java-R-communication. On some operating systems, especially Windows 7, it may happen that the command require(rJava) issues an error of the form

```
Error: .onLoad failed in loadNamespace() for 'rJava', details: ...
```

This means that rJava was not installed properly on your computer. Try then the following:

1. Define the environment variable JAVA_HOME: Explorer - RightMouse on "Computer" - Properties - Environment Variables, and add there

   ```
   JAVA_HOME = C:\Program Files\Java\jdk1.7.0_11\jre7
   ```

   and restart R. (The path is the correct one on my computer, on others it might be slightly different. It is the path to the Java Runtime Environment within your JDK.)

2. Package rJava needs to find the Java DLL jvm.dll. To enable this, expand the environment variable Path: Explorer - RightMouse on "Computer" - Properties - Environment Variables - Path - Edit, and add at the end of the Path string

   ```
   ;C:\Program Files\Java\jdk1.7.0_11\jre\bin\server
   ```

   and restart R. (The path is the correct one on my computer, on others it might be slightly different. It is the subdirectory in the current Java installation containing jvm.dll.)

Note that the above remarks are for 64-bit-Java and 64-bit-R. If you use 32-bit-Java, the locations might be slightly different as well.

On some Linux/UNIX systems there might be also problems with the installation of rJava because R cannot locate the Java installation. In that case, fix it permanently by issuing the command

```
sudo R CMD javareconf -e
```

at the UNIX prompt (needs superuser rights). If you do not have superuser rights, you may invoke

```
R CMD javareconf -e
```

in each session where you need rJava.
References


Simon Urbanek. rJava: Low-level R to Java interface, 2013. URL http://cran.r-project.org/web/packages/rJava