Optimisation Stochastique Numérique: du Recuit Simulé à la Méthode CMA-ES

Anne Auger
RandOpt, Inria & CMAP

Journée de Rentrée du CMAP

Mercredi 4 Octobre
Rand Members

Permanent members:

Anne Auger
CR Inria, Team Leader

Dimo Brockhoff
CR Inria

Nikolaus Hansen,
DR Inria

PhD students:

Cheikh Touré, start Oct 2017 - collab Storengy
Konstantinos Varelas, start Dec 2017, CIFRE-DGA, Thales
Pierrick Pochelou, start Dec 2017 [collab Eric Moulines], CIFRE TOTAL

Postdoc and Engineers:

Asma Atamna, postdoc PGMO (collab EDF)
Umut Battu, ADT engineer
Dejan Tusar, engineer, Inria-hub funding
Numerical Black-Box Optimization

Application Domains

Optimize $f : \mathbb{R}^n \mapsto \mathbb{R}^k$

$\mathbf{x} \in \mathbb{R}^n \rightarrow f(\mathbf{x}) \in \mathbb{R}^k$

derivatives are not available or not useful
Numerical Black-Box Optimization
Application Domains
RandOpt Research Overview
Black-box optimization for difficult problems

**Theory**
- stability of Markov chains
- stochastic approximation
- convergence bounds

Benchmarking: **COCO platform**

**Algorithm Design**
- constrained optimization
- multi-objective
- large-scale
- expensive

**Applications**
- Thales, Storengy
- EDF, TOTAL
What Makes a Problem Difficult to Solve?  

Typical “real-world” difficulties

- non-linear
- non-convex
- non-quadratic
- ill-conditioned
- dependencies between the variables
- non-separable
- rugged
- discontinuous
- noisy
- multi-modal
What Makes a Problem Difficult to Solve?

Typical “real-world” difficulties

- non-linear
- non-convex
- non-quadratic
- ill-conditioned
- non-separable dependencies between the variables
- rugged
- discontinuous
- noisy
- multi-modal
Simulated Annealing

Input: temperature decreasing schedule \( \{T_t : t > 0\} \)

Initialize

WHILE NOT HAPPY

# Sample candidate solutions
\[
X_{t+1}^1 = X_t + \Delta(X_t)
\]

IF \( f(X_{t+1}^1) \leq f(X_t) \) # accept if better
\[
X_{t+1} = X_{t+1}^1
\]

ELIF \( f(X_{t+1}^1) > f(X_t) \) AND \( U(0, 1) < \exp \left( \frac{-(f(X_{t+1}^1) - f(X_t))}{T_{t+1}} \right) \) # proba of accept if worse
\[
X_{t+1} = X_{t+1}^1
\]
ELSE
\[
X_{t+1} = X_t
\]

\( t = t + 1 \)

Objective: Minimize \( f : \mathbb{R}^n \to \mathbb{R} \)
Simulated Annealing

**Input:** temperature decreasing schedule \( \{T_t : t > 0\} \)

**Initialize**

WHILE NOT HAPPY

# Sample candidate solutions

\[
X_{t+1}^1 = X_t + \Delta(X_t)
\]

IF \( f(X_{t+1}^1) \leq f(X_t) \)  # accept if better

\[
X_{t+1} = X_{t+1}^1
\]

ELIF \( f(X_{t+1}^1) > f(X_t) \) AND \( U(0, 1) < \exp \left( \frac{-(f(X_{t+1}^1) - f(X_t))}{T_{t+1}} \right) \)  # proba of accept if worse

\[
X_{t+1} = X_{t+1}^1
\]

ELSE

\[
X_{t+1} = X_t
\]

\( t = t + 1 \)

**What is/are the bottleneck(s) in this algorithm?**
“A major difficulty in simulated annealing is to find a good cooling scheme for the temperature” [typical statement]
“[The generation of a new solution] is the most problematical. The literature describes several different schemes for choosing a new solution, none of which, in our view, inspire complete confidence.”

“[The generation of a new solution] is the most problematical. The literature describes several different schemes for choosing a new solution, none of which, in our view, inspire complete confidence.”

“The problem is one of efficiency: A generator of random changes is inefficient if, when local downhill moves exist, it nevertheless almost always proposes an uphill move. A good generator, we think, should not become inefficient in narrow valleys; nor should it become more and more inefficient as convergence to a minimum is approached. Except possibly for [7], all of the schemes that we have seen are inefficient in one or both of these situations.”

What Makes a Problem Difficult to Solve?
Typical “real-world” difficulties

- non-linear
- non-convex
- non-quadratic
- ill-conditioned
- dependencies between the variables
- non-separable
- rugged
- discontinuous
- noisy
- multi-modal
Simulated Annealing
Fixed Step-size (typical setting)

**Input:** temperature decreasing schedule \( \{T_t : t > 0\} \)

**Initialize** \( X_0, \sigma \)

**WHILE NOT HAPPY**

# Sample candidate solution
\[
X_{t+1}^1 = X_t + \sigma U_{t+1} \quad \text{with} \quad U_{t+1} \sim \mathcal{N}(0, I_d)
\]

**IF** \( f(X_{t+1}^1) \leq f(X_t) \)  # accept if better
\[
X_{t+1} = X_{t+1}^1
\]

**ELIF** \( f(X_{t+1}^1) > f(X_t) \) AND \( U(0, 1) < \exp \left( \frac{-(f(X_{t+1}^1) - f(X_t))}{T_{t+1}} \right) \)  # proba of accept if worse
\[
X_{t+1} = X_{t+1}^1
\]

**ELSE**
\[
X_{t+1} = X_t
\]
\[
t = t + 1
\]
Simulated Annealing
Convergence Problem

“more and more inefficient as convergence to a minimum is approached” [Numerical Recipes]

\[ f(x) = \|x\|^2 = \sum_{i=1}^{n} x_i^2 \]

\( n = 10 \)

\( \sigma = 0.1 \)

3 trials
Simulated Annealing  
Adaptive Step-size

**Input:** temperature decreasing schedule \( \{T_t : t > 0\} \)

**Initialize** \( X_0, \sigma_0 \)

**WHILE NOT HAPPY**

# Sample candidate solutions 

\[ X_{t+1}^{1} = X_t + \sigma_t U_{t+1} \text{ with } U_{t+1} \sim \mathcal{N}(0, I_d) \]

**IF** \( f(X_{t+1}^{1}) \leq f(X_t) \)  
# accept if better, incr. step-size 

\[ X_{t+1} = X_{t+1}^{1}, \sigma_{t+1} = 1.5\sigma_t \]

**ELIF** \( f(X_{t+1}^{1}) > f(X_t) \) AND \( U(0, 1) < \exp \left( \frac{-(f(X_{t+1}^{1}) - f(X_t))}{T_{t+1}} \right) \)  
# proba of accept if worse 

\[ X_{t+1} = X_{t+1}^{1}, \sigma_{t+1} = \sigma_t \]

**ELSE** 

\[ X_{t+1} = X_t, \sigma_{t+1} = (1.5)^{-1/4}\sigma_t \]  
# decr. step-size

\( t = t + 1 \)
Improving the Convergence
Adaptive Step-size

\[ f(x) = \|x\|^2 = \sum_{i=1}^{n} x_i^2 \]

\( n = 10 \) 

1 trial

\( f(X_t) = \|X_t\|^2 \)
(1+1)-ES with one-fifth success rule

[Rechenberg, 73], [Schumer, Steiglitz, 68], [Devroye, 72]

Initialize $X_0$, $\sigma_0$

WHILE NOT HAPPY

# Sample candidate solutions

$X_{t+1}^1 = X_t + \sigma_t U_{t+1}$ with $U_{t+1} \sim \mathcal{N}(0, I_d)$

IF $f(X_{t+1}^1) \leq f(X_t)$ # accept if better, incr. step-size

$X_{t+1} = X_{t+1}^1, \sigma_{t+1} = 1.5\sigma_t$

ELSE

$X_{t+1} = X_t, \sigma_{t+1} = (1.5)^{-1/4}\sigma_t$ # decre. step-size

$t = t + 1$
Evolution Strategies (ES)

Invariance to increasing transformation of $f$

Comparison-based (or ranked-based) algorithm

same ranking (and update of the state of algorithm) when optimizing

$f : \mathbb{R}^n \rightarrow \mathbb{R}$ or $g : \text{Im}(f) \rightarrow \mathbb{R}$ where $g$ strictly increasing
Evolution Strategies (ES)
Invariance to increasing transformation of $f$

- Comparison-based (or ranked-based) algorithm
  
  Same ranking (and update of the state of algorithm) when optimizing
  
  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ or $g : \text{Im}(f) \rightarrow \mathbb{R}$ where $g$ strictly increasing

- Invariance to strictly increasing transformations of $f$

Those three functions are optimized in the same way
Improving Convergence Even Further

(1+1)-ES with one-fifth success rule

\[ f(x) = \|x\|^2 = \sum_{i=1}^{n} x_i^2 \]

\[ n = 10 \]

Linear convergence
Testing Overall Performance

Benchmarking

- Want to have a better idea of performance of the 3 algorithms on representative set of problems typical of difficult black-box problems
  - non-convex, multi-modal (weak structure, strong structure), ill-conditioned

- For different dimensions

- That’s where benchmarking comes into play
automatizing the benchmarking:

the COCO platform

Comparing Continuous Optimizers Platform

https://github.com/numbbo/coco
Numerical Black-Box Optimization Benchmarking Framework

[http://coco.gforge.inria.fr/](http://coco.gforge.inria.fr/)

- 7,902 commits
- 12 branches
- 25 releases
- 13 contributors

Branch: `master`  |  New pull request

**brogkho** committed on GitHub: Merge pull request #1075 from numbbo/development

- `code-experiments`: Merge pull request #1071 from ttusar/debug
- `code-postprocessing`: further clean up of postprocessing output,
- `code-preprocessing/archive-update`: Added empty last lines.
- `docs`: updated reference to biobjective perf-assessment paper on arXiv in ge...
- `howtos`: Update documentation-howto.md
- `.clang-format`: raising an error in bbo2009_logger.c when best_value is NULL. Plus s...
- `.hgignore`: raising an error in bbo2009_logger.c when best_value is NULL. Plus s...
- `AUTHORS`: small correction in AUTHORS
- `LICENSE`: Added acknowledgements to external collaborators...
- `README.md`: Update README.md
- `do.py`: Merge branch `development` of https://github.com/numbbo/coco into pp...
- `doxygen.ini`: moved all files into code-experiments/ folder besides the do.py scrip...
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimization algorithms.
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANST C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
- Python

Contributions to link further languages (including a better example in C++) are more than welcome.

For more information,
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
- Python

Contributions to link further languages (including a better example in C++) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description
- see the bbo-biobj COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbed
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
- Python

Contributions to link further languages (including a better example in C++) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description
- see the bbo-biobj COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbed.
- consult the BBOB workshops series,
- consider to register here for news,
- see the previous COCO home page here and
- see the links below to learn more about the ideas behind COCO.
Getting Started

1. Check out the Requirements above.

2. Download the COCO framework code from github either by clicking the Download ZIP button or (preferred) by typing `git clone https://github.com/numbbo/coco.git` in the terminal. Clone in your workspace or via `git clone` but note that `git pull` keeps the code up-to-date with the latest release.

CAVEAT: this code is still under heavy development. The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

3. In a system shell, `cd into the coco or coco-<version>` folder (framework root), where the file `do.py` can be found. Type, i.e. `execute`, one of the following commands once:

```bash
python do.py run-c
python do.py run-java
python do.py run-matlab
python do.py run-octave
python do.py run-python
```

depending on which language shall be used to run the experiments. `run-*` will build the respective code and run the example experiment once. The build result and the example experiment code can be found under `code-experiments/build/<language>` (Octave). `python do.py` lists all available commands.

4. On the computer where experiment data shall be post-processed, run:

```bash
python do.py install-postprocessing
```
3. In a system shell, cd into the coco or coco-<version> folder (framework root), where the file do.py can be found. Type, i.e. `execute`, one of the following commands once:

```
python do.py run-c
```

or

```
python do.py run-matlab
python do.py run-octave
python do.py run-python
```

depending on which language shall be used to run the experiments. `run-*` will build the respective code and run the example experiment once. The build result and the example experiment code can be found under code-experiments/build/<language> (where `<language>` = matlab for Octave). `python do.py` lists all available commands.

4. On the computer where experiment data shall be post-processed, run:

```
python do.py install-postprocessing
```

to (user-locally) install the post-processing. From here on, do.py has done its job and is only needed again for updating the builds to a new release.

5. Copy the folder code-experiments/build/YOUR-FAVORITE-LANGUAGE and its content to another location. In Python it is sufficient to copy the file example_experiment.py. Run the example experiment (it already is compiled, in case). As the details vary, see the respective read-me's and/or example experiment files:

- read me and example experiment
- Java read me and example experiment
- Matlab/Octave read me and example experiment
https://github.com/numbbo/coco

to (user-locally) install the post-processing. From here on, do.py has done its job and is only needed again for updating the builds to a new release.

5. Copy the folder code-experiments/build/YOUR-FAVORITE-LANGUAGE and its content to another location. In Python it is sufficient to copy the file example_experiment.py. Run the example experiment (it already is compiled, in case). As the details vary, see the respective read-me's and/or example experiment files:

- C read me and example experiment
- Java read me and example experiment
- Matlab/Octave read me and example experiment
- Python read me and example experiment

If the example experiment runs, connect your favorite algorithm to Coco: replace the call to the random search optimizer in the example experiment file by a call to your algorithm (see above). Update the output result_folder, the algorithm_name and algorithm_info of the observer options in the example experiment file.

Another entry point for your own experiments can be the code-experiments/examples folder.

6. Now you can run your favorite algorithm on the bbo-biobj (for multi-objective algorithms) or on the bbo suite (for single-objective algorithms). Output is automatically generated in the specified data result_folder.

7. Postprocess the data from the results folder by typing

```python
python -m bbo_proc [-o OUTPUT_FOLDERNAME] YOURDATA_FOLDER [MORE_DATA FOLDERS]
```

The name bbo_proc will become cocopp in future. Any subfolder in the folder arguments will be searched for logged data. That is, experiments from different batches can be in different folders collected under a single "root" YOURDATA_FOLDER folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.
if solver._name_ in ("random_search",):
    solver(fun, fun.lower_bounds, fun.upper_bounds, remaining_evals)
elif solver._name_ == 'fmin' and solver._globals_['__name__'] in ['cma', 'cma.evolution_strategy', 'cma.es']:
    if x0[0] == center[0]:
        sigma0 = 0.02
        restarts_ = 0
    else:
        x0 = "%f + %f * np.random.rand(%d)" % (center[0], 0.8 * range_[0], fun.dimension)
        sigma0 = 0.2
        restarts_ = 6 * (observer_options.as_string.find('IPOP') >= 0)
    solver(fun, x0, sigma0 * range_[0], restarts=restarts_,
           options=dict(scaling=range_/range_[0], maxevals=remaining_evals,
                         termination_callback=lambda es: fun.final_target_hit,
                         verb_log=0, verb_disp=0, verbosee=9))
elif solver._name_ == 'fmin_slsqp':
    solver(fun, x0, iter=1 + remaining_evals / fun.dimension,
           jprint=1)

elif solver._name_ == 'simulated_annealing_adaptive':
    solver(fun, x0, 3, remaining_evals, temp_fast)
    # CALL MY SOLVER, interfaces vary

else:
    raise ValueError("no entry for solver %s" % str(solver._name_))

if fun.evaluations >= max_evals or fun.final_target_hit:
    break
# quit if fun.evaluations did not increase
if fun.evaluations <= max_evals - remaining_evals:
    if max_evals - fun.evaluations > fun.dimension + 1:
        print("WARNING: %d evaluations > fun.dimension" % remaining_evals)
    if fun.evaluations < max_evals - remaining_evals:
        raise RuntimeError("function evaluations decreased")
    break
return restarts + 1

# set up: CHANGE HERE SOLVER AND FURTHER SETTINGS AS DESIRED

# recover any previous settings

# final setup

# run experiment

# save results
6. Now you can **run** your favorite algorithm on the `bbob-biobj` (for multi-objective algorithms) or on the `bbob` suite (for single-objective algorithms). Output is automatically generated in the specified data folder.

7. **Postprocess** the data from the results folder by typing

   ```bash
   python -m cocopp [-o OUTPUT_FOLDERNAME] YOURDATA_FOLDER [MORE_DATA_FOLDERS]
   ```

   The name `bbob_pproc` will become `cocopp` in future. Any subfolder in the folder argument is taken into account. That is, experiments from different batches can be in different folders collected under the `YOURDATA_FOLDER` folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.

   A folder, `ppdata` by default, will be generated, which contains all output from the post-processing, including a `ppdata.html` file, useful as main entry point to explore the result with a browser. Data might be overwritten, it is therefore useful to change the output folder name with the `-o OUTPUT_FOLDERNAME` option.

   For the single-objective `bbob` suite, a summary pdf can be produced via LaTeX. The corresponding templates in ACM format can be found in the `code_postprocessing/latex-templates` folder. LaTeX templates for the multi-objective `bbob-biobj` suite will follow in a later release. A basic html output is also available in the result folder of the postprocessing (file `templateBBOBarticle.html`).

8. Once your algorithm runs well, **increase the budget** in your experiment script, if necessary implement randomized independent restarts, and follow the above steps successively until you are happy.

If you detect bugs or other issues, please let us know by opening an issue in our issue tracker at https://github.com/numbbo/coco/issues.
result folder

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Size</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>cocopp_commands.tex</td>
<td>Yesterday, 8:52 AM</td>
<td>17 KB</td>
<td>text</td>
</tr>
<tr>
<td>index.html</td>
<td>Yesterday, 8:51 AM</td>
<td>599 bytes</td>
<td>HTML text</td>
</tr>
<tr>
<td>SA_ad_CMA_p_SA_on_ONEFI</td>
<td>Yesterday, 8:47 AM</td>
<td>--</td>
<td>Folder</td>
</tr>
<tr>
<td>SA_ad_SA_on_ONEFI</td>
<td>Yesterday, 8:52 AM</td>
<td>--</td>
<td>Folder</td>
</tr>
</tbody>
</table>
```
automatically generated results

COCO Post-Processing Results

Comparison data

SA_ad_CMA_p_SA_on_ONEFI

SA_ad_SA_on_ONEFI

Single algorithm data
automatically generated results
Overall Performance on 24 Functions
Performance on Multi-modal Functions
Performance on Ill-conditioned Problems

- bbob f10-f14, 20-D
- 51 targets: 100..1e-08
- 15 instances

Fraction of function-target pairs vs. log10(# f-evals / dimension)

- Best 2009
- ONEFIFTH
- SA adapt
- SA on bbo

Logarithmic scale for the x-axis and linear scale for the y-axis.
What is Missing to the Algorithms?

Solving of ill-conditioned problems

“A good generator, we think, should not become inefficient in narrow valleys” [Numerical Recipes]

cond. numbers up to $10^{10}$ frequently encountered

Better performance for “global” optimization
from simulated annealing to CMA-ES
CMA-ES = Covariance Matrix Adaptation - Evolution Strategy

*state-of-the-art stochastic black-box algorithm*  
(widely) used in industry or academic domains

[Hansen & Ostermeier 2001] [Hansen et al. 2001-2016]

candidate solutions sampled from *multivariate normal distribution*

\[
\mathcal{N}(X_t, \sigma_t^2 C_t)
\]

covariance matrix

state of the algorithm:

\[
\theta_t = (X_t, \sigma_t, C_t, p_t, p_t^\sigma) \in (\mathbb{R}^n \times \mathbb{R}^+ \times S(n, \mathbb{R}) \times \mathbb{R}^n \times \mathbb{R}^n)
\]
CMA-ES
Update of the mean

Sample candidate solutions

\[ X_{t+1}^i = X_t + \sigma_t C_t^{1/2} U_{t+1}^i, \quad i = 1, \ldots, \lambda \]

\[ \mathcal{N}(X_t, \sigma_t^2 C_t) \quad (U_{t+1}^i)_i \sim \mathcal{N}(0, I_d) \text{ i.i.d.} \]

Evaluate and rank solutions

\[ f \left( X_{t+1}^{1:\lambda} \right) \leq \ldots \leq f \left( X_{t+1}^{\lambda:\lambda} \right) \]

Recombine the mu-best solutions

\[ X_{t+1} = \sum_{i=1}^{\mu} w_i X_{t+1}^{i:\lambda} = X_t + \sigma_t \sum_{i=1}^{\mu} w_i C_t^{1/2} U_{t+1}^{i:\lambda} \]

\[ Y_{t+1} \]

\[ w_1 \geq w_2 \ldots \geq w_\mu > 0 \]

\[ \mu_w = \frac{1}{\sum w_i^2} \approx 0.3\lambda \]
CMA-ES
Update of the Covariance Matrix

\[ X_{t+1} = \sum_{i=1}^{\mu} w_i X_{t+1}^{i:\lambda} = X_t + \sigma_t \sum_{i=1}^{\mu} w_i C_t^{1/2} U_{t+1}^{i:\lambda} Y_{t+1} \]

\[ C_{t+1} = (1 - c_1 - c_\mu) C_t + c_1 p_{t+1}^c [p_{t+1}^c]^T + c_\mu \sum_{i=1}^{\mu} w_i C_t^{1/2} U_{t+1}^{i:\lambda} [C_t^{1/2} U_{t+1}^{i:\lambda}]^T \]

\[ p_{t+1}^c = (1 - c_c) p_t^c + \sqrt{1 - (1 - c_c)^2} \sqrt{\mu_w} Y_{t+1} \]

\( w_1 \geq w_2 \ldots \geq w_\mu > 0 \)
\( \mu_w = \frac{1}{\sum w_i^2} \approx 0.3\lambda \)
CMA-ES - Step-size Update(s)
Path Length Control

Measure the length of the \textit{evolution path}:
the pathway of the mean vector $m$ in the generation sequence

\begin{align*}
p_{t+1}^\sigma &= (1 - c_\sigma)p_t^c + \sqrt{1 - (1 - c_\sigma)^2} \sqrt{\mu_w}C_t^{-1/2}Y_{t+1} \\
\sigma_{t+1} &= \sigma_t \exp \left( \frac{c_\sigma}{d_\sigma} \left( \frac{\|p_\sigma\|}{E[\|\mathcal{N}(0, I_d)\|]} - 1 \right) \right)
\end{align*}
CMA-ES
Source Code Snippet

```matlab
% the next 40 lines contain the 20 lines of interesting code
while counteval < stopeval
    % Generate and evaluate lambda offspring
    for k=1:lambda,
        arx(:,k) = xmean + sigma * B * (D .* randn(N,1)); % m + sig * Normal(0,C)
        arfitness(k) = feval(strfitnessfct, arx(:,k)); % objective function call
        counteval = counteval+1;
    end

    % Sort by fitness and compute weighted mean into xmean
    [arfitness, arindex] = sort(arfitness); % minimization
    xold = xmean;
    xmean = arx(:,arindex(1:mu)) * weights; % recombination, new mean value

    % Cumulation: Update evolution paths
    ps = (1-cs)*ps ... + sqrt(cs*(2-cs)*mueff) * invsqrtd * (xmean-xold) / sigma;
    hsig = norm(ps)/sqrt(1-(1-cs)^2) * counteval/lambda) / chIN < 1.4 + 2/(N+1);
    pc = (1-cc)*pc ... + hsig * sqrt(1-(2-cc)*mueff) * (xmean-xold) / sigma;

    % Adapt covariance matrix C
    artmp = (1/sigma) * (arx(:,arindex(1:mu)) - repmat(xold,1,lambda));
    C = (1-cl-cmu) * C ... % regard old matrix + cl * (pc') ... % plus rank one update 
        + (1-hsig) * cc*(2-cc) + C ... % minor correction if hsig==0
        + cmu * artmp * diag(weights) * artmp'; % plus rank mu update

    % Adapt step size sigma
    sigma = sigma * exp((cs/damps)*(norm(ps)/chIN - 1));

    % Decomposition of C into B*diag(D.^2)*B' (diagonalization)
    if counteval - eigenable > lambda/(cl+cmu)/N/10 % to achieve O(N^2)
        eigenable = counteval;
        C = triu(C) + triu(C,1)'; % enforce symmetry
        [B,D] = eig(C); % eigen decomposition, B=normalized eigenvectors
    end

    invsqrtd = B * diag(D.^2) * B';
end
```
CMA-ES
Features

- Invariances
  - invariance to monotonic transformation
  - affine-invariant
- Learning of second order information
- Increasing lambda: more global search
- Parameter-less algorithm

\[ C_t \propto \text{inverse Hessian} \]
Performance on ill-conditioned Problems
Performance on Multi-modal Functions

multi-modal with global structure

multi-modal with weak global structure
Overall Performance

The graph illustrates the performance of different optimization algorithms across various function targets. The x-axis represents the logarithm of the number of function evaluations per dimension, while the y-axis shows the fraction of function-target pairs successfully evaluated. Different algorithms are represented by various line styles and colors:

- **best 2009**
- **BIPOP-CMA**
- **CMA pop10**
- **CMA defau**
- **ONEFIFTH**
- **SA adapt**
- **SA on bbo**

The targets are defined as:
- bbob f1-f24, 20-D
- 51 targets: 100.1e-08
- 15 instances
Thank you!