# `blackbox.py` — optimization of expensive black-box functions on multi-core processors

Paul Knysh

Last modified: February 2016

## 1 Introduction

Often times there is a need to find optimal parameters of complex object or process (geometry, chemical composition, operating conditions etc). It's not always easy or even possible to come up with a good analytical approach that solves the problem. Instead we can perform trial-and-error search using computer simulation. Performing such a search manually might be very tedious and inefficient, if feasible at all.

One can build a code interface (function/procedure) that has input — list of trial values, and output — some scalar function that represents cost/error and automatically calculated every time simulation is performed. From this point of view we are dealing with mathematical optimization. The problem however is that function doesn't have analytical form and, what is more crucial, is usually expensive (it can take many hours to evaluate it at a single point), and therefore called expensive black-box function.

Proposed here procedure allows to perform efficient optimization of expensive black-box functions using specialized methods. This procedure is designed to scale on multi-core CPUs (Intel Xeon Phi family carries up to 60 cores, Feb 2016). Resulting speedup is equal to a number of cores, which is achieved by always keeping every available core running its own independent function evaluation.

## 2 Overview of procedure

Our goal is to efficiently minimize a positive function $f$ defined in a box — each variable has its own independent range. To maximize $f$, we can simply apply procedure on $\frac{1}{f+1}$ or so. Our procedure has four main steps:

1. Rescaling of variables
2. Initial sampling
3. Rescaling of cost/error function
4. Subsequent iterations

All of these steps are described in detail in the following sections.

## 3 Rescaling of variables

Many aspects of our procedure are based on relative distance between sampled points so we want to be independent of scale (for example, two variables might be many orders of magnitude away from each other due to different nature). So first thing we do is we normalize each variable into a range of $[0, 1]$. For variable $x_i$ that is in range $[a_i, b_i]$ the following simple transformations are used:

$$\xi_i = \frac{x_i - a_i}{b_i - a_i}, \tag{1}$$

$$x_i = a_i + \xi_i(b_i - a_i). \tag{2}$$

In other words, we compress a box into a unit cube (and then vice versa), which simplifies both procedure and code.

## 4 Initial sampling

Our procedure is based on a response surface methodology — we replace given function with its fit (based on few evaluations) and optimize a fit instead, which is relatively cheap. A quality of response surface depends a lot on initial set of points that is used to build it. One natural idea is to introduce a uniform mesh for initial sampling. This is acceptable for lower dimensions and becomes problematic in higher dimensions. For example, in 4D space, mesh of size 2 needs $2^4 = 16$ evaluations (which might be not enough), while mesh of size 3 needs $3^4 = 81$ evaluations (which might be too much).

Another option is to generate random samples. This is also not going to work well — in this case points most probably will not cover space uniformly (Fig. 1a).

The problem of placing an arbitrary amount of points $n$ into a unit cube in a uniform manner can be solved by building so called Latin Hypercube (LH). Basically we introduce a uniform mesh of size $n$ and then place $n$ points at the nodes of this mesh in such a way, that there is exactly one point in each axis-aligned hyperplane containing it. As we see, $n$ can be chosen independently of dimension of the space.

Suppose we consider 2D space and we start with a diagonal placement (Fig. 1b), which is a LH. If we pick two random rows or columns and exchange them, we will obtain a new LH, which might be better than previous in terms of the spread of the points. Measure of spread (which we need to minimize) can be introduced like this:

$$S = \sum_{i}^{n} \sum_{j(>i)}^{n} \frac{1}{\|p_i - p_j\|}, \tag{3}$$

where $\|p_i - p_j\|$ is a distance between two points. If we repeat this iteratively many times (for example, 1000), we can get a much better placement (Fig. 1c).

Same exact concept is used in higher dimensions as well.

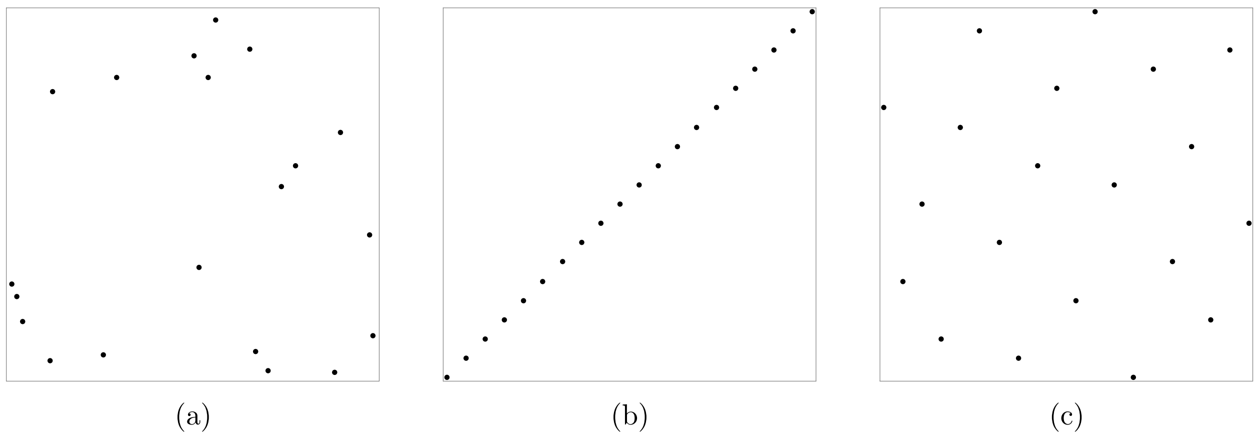

Figure 1: Examples of placing 20 points into a unit square. (a) — typical placement of random points (very nonuniform), (b) — initial (diagonal) LH, (c) — eventual LH (after 1000 shuffles)

## 5 Rescaling of cost/error function

After initial set is evaluated, we are having some values of cost/error at every point. These values are not necessarily useful as is (value of $1.3445e7$ might not say too much). So, it makes sense to rescale all values into range $[0, 1]$, where 1 is now obviously the worst value and 0 is obviously the best value.

Also, sometimes we will have a region with extremely high values of cost/error function. These peaks might affect overall response surface and we don't want to keep them. In this case we can choose a threshold value $t$, for example based on some percentage of best points from initial set. Then scaled function can be defined like this:

$$f^* = \begin{cases} \frac{f}{t}, \, f < t \\ 1, \, f \geq t \end{cases} . \tag{4}$$

## 6 Subsequent iterations

### 6.1 RBF + CORS

Once initial sampling is done, we can build a response surface using, for example, radial basis functions (RBF) [1] with cubic terms. Since now, every time we sample new points, we update the fit.

Subsequent iterations are performed in accordance with somewhat modified CORS algorithm [2] — we minimize current fit and at the same time require next point to be not closer than $r$ from each of previously sampled points. Essentially, we place balls of radius $r$ around each of previously sampled points and require the next point to minimize the current fit and be outside of every ball. This prevents us from being stuck in a local minimum.

Depending on value of $r$ we can have a different density $\rho$ of these balls in a unit cube. Since the volume of a unit cube is 1, we can write:

$$\rho < Nv, \tag{5}$$

where $N$ is amount of previously sampled points and $v$ is a volume of each ball and inequality means that real density is less — some balls are going to intersect with each other, some are going to have some fraction of volume outside of a cube.

The volume of $d$-dimensional ball is equal to $v_1 r^d$, where $v_1$ is a volume of a ball with radius 1. The current amount of balls $N$ is equal to $n + i - 1$, where $n$ is amount of initial points and $i$ is a number of current subsequent iteration (counted from 1). Then:

$$\rho < (n + i - 1) v_1 r^d, \tag{6}$$

$$r > \left( \frac{\rho}{(n + i - 1) v_1} \right)^{1/d}. \tag{7}$$

Volume $v_1$ can be written in a few ways, for example, using gamma function:

$$v_1 = \frac{\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2} + 1\right)}. \tag{8}$$

There are different strategies on choice of $\rho$. Cyclic density was used in [2]. Here, aiming to explore first, then refine, we are using the following expression:

$$\rho = \rho_0 \left( \frac{m - i}{m - 1} \right)^p, \tag{9}$$

where $\rho_0$ is initial density, $m$ is a total number of subsequent iterations, $p$ is a rate of density decay. If $p = 1$, then density linearly decays with iterations, if $0 < p < 1$ — slower than linearly, if $p > 1$ — faster than linearly.

So, equation (7) together with (8) and (9) allows to find current radius $r$ on every subsequent iteration.

## 6.2 Minimizing fit

For minimization of a current fit we are populating a large number of random points (normally $> 5000$) in a unit cube. Then we create a loop over every single one and select the best candidate.

This way may sound inefficient, but computational time spent here is nothing in comparison to evaluations of expensive functions . Also, there is no obvious alternative way to minimize the fit under distance constraint — balls are going to intersect and create all sorts of features that are hard to explore with gradient-based method or so. Finally, we don't need to be perfectly accurate — any reasonable point will still improve response surface, which is what we need eventually.

## 6.3 Space rescaling

Often times global minimum is sitting on a bottom of a valley-like feature. This can cause all sorts of issues when we are building a RBF-fit. One way to avoid this is to estimate (roughly) the shape of the feature, find its principal components and then stretch the space accordingly.

We first build a regular RBF-fit and then populate a large numbers of random points in a unit cube. Then we select some percentage of best points (normally 5% of total number) based on their RBF-values. These points form a cloud that approximates the shape of the feature. Then we find a covariance matrix of this cloud:

$$C = \mathrm{cov}(\mathrm{cloud}). \tag{10}$$

Then we find eigenvalues/eigenvectors of covariance matrix:

$$\lambda_i, m_i = \mathrm{eig}(C). \tag{11}$$

Then we find scaling matrix:

$$T = \begin{pmatrix} m_1/\sqrt{\lambda_1} \\ m_2/\sqrt{\lambda_2} \\ \vdots \\ m_d/\sqrt{\lambda_d} \end{pmatrix}. \tag{12}$$

Matrix $T$ can be divided by one of its norms to get rid of big/small numbers. Then we update the following RBF expressions:

$$s_n(x) = \sum_{i=1}^{n} \lambda_i \phi(\|T(x - x_i)\|) + b^T \cdot x + a, \tag{13}$$

$$\Phi_{ij} = \phi(\|T(x_i - x_j)\|). \tag{14}$$

It's enough to repeat described above procedure few times during subsequent iterations (for example, once every 5 iterations).

## 7 Using multiple cores

Described procedure uses multiple cores. This is done by dividing samples on initial/subsequent stage into batches (size of batch is equal to number of cores used) that are evaluated simultaneously.

This provides a speedup, that is equal to the amount of cores available.

## References

[1] Kenneth Holmström. An adaptive radial basis algorithm (ARBF) for expensive black-box global optimization. *Journal of Global Optimization*, 41(3):447–464, 2008.

[2] Rommel G Regis and Christine A Shoemaker. Constrained global optimization of expensive black box functions using radial basis functions. *Journal of Global Optimization*, 31(1):153–171, 2005.