GSOC 2019
Randomized Convex Optimization

Repouskos Panagiotis

August 17, 2019

Contents

Chapter 1

Randomized Cutting Plane Method

1.1 Linear Programming

A linear program of the form

$$\begin{aligned} &\text{minimize } c^T x \\ &\text{s.t. } Ax \leq b \end{aligned}$$

is to find in the polytope described by the inequalities $Ax \leq b$, the point x that minimizes $c^T x$. So the random walk will sample from a polytope and cutting planes will steadily reduce its volume.

1.1.1 Implementation

1.1.1.1 Finding an initial feasible point

To start the random walk, a initial point in the interior of the polytope is needed, or stated differently, a feasible solution of the linear program. Such a point can be acquired by solving the following linear program:

$$\begin{aligned} &\text{minimize } s \\ &\text{s.t. } Ax - b \leq s \end{aligned}$$

This is solved with the barrier method. Once some $s < 0$ is found, then the algorithm can stop; there is no need to solve the entire LP.

1.1.1.2 Random Walks

Three random walks are used. Ultimately, the most efficient walk for sampling turned out to be Hit and Run with coordinate directions.

1.1.1.2.1 Hit and Run with random directions In Hit and Run with random directions (RDH&R), starting from a point, a direction is randomly chosen and ray shooting is performed, to find the intersection points of the line defined by the initial point and the direction vector and the polytope. The next point is uniformly chosen on the segment defined by these two intersection points.

1.1.1.2.2 Hit and Run with coordinate directions The difference of Hit and Run with coordinate directions (CDH&R) from RDH&R, is that the the direction vector must be parallel to an axis. While RCH&R mixes better, but CDH&R is much cheaper, so we can allow more iterations. Specifically, by saving some information on the first ray shooting, which has complexity $O(md)$, the following ray shootings will cost $O(m)$.

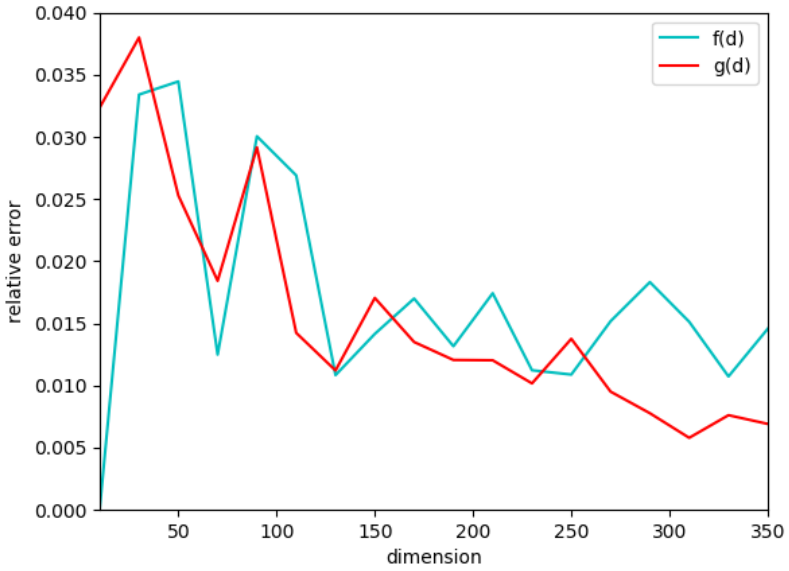

Figure 1.1: Coordinate HnR: sampling $g = 100 + d^2$, $f = 1000 + \sqrt{d} \cdot d$ points per phase.

1.1.1.2.3 Billiard Walk Billiard Walk [1], starting from a point, choose a direction vector and move towards it, for a specified distance. If the boundary of the polytope is reached, the trajectory reflects on the facet. This walk behaves well even for $O(1)$ sampled points per phase.

Note *W*hen the polytope becomes too skinny, the reflections are too many and the walk becomes too slow. So less reflections or smaller trajectory length? Too small and we don't move.

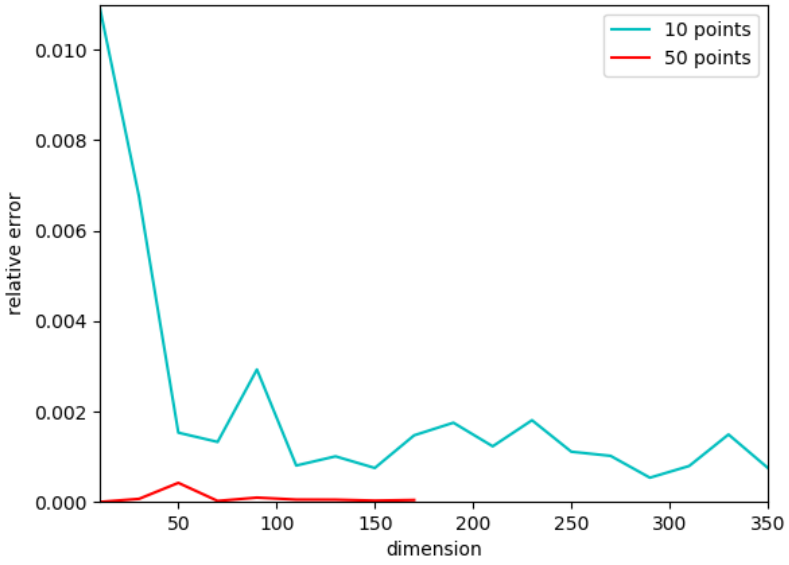

Figure 1.2: Billiard walk: sampling 10 and 50 points per phase.

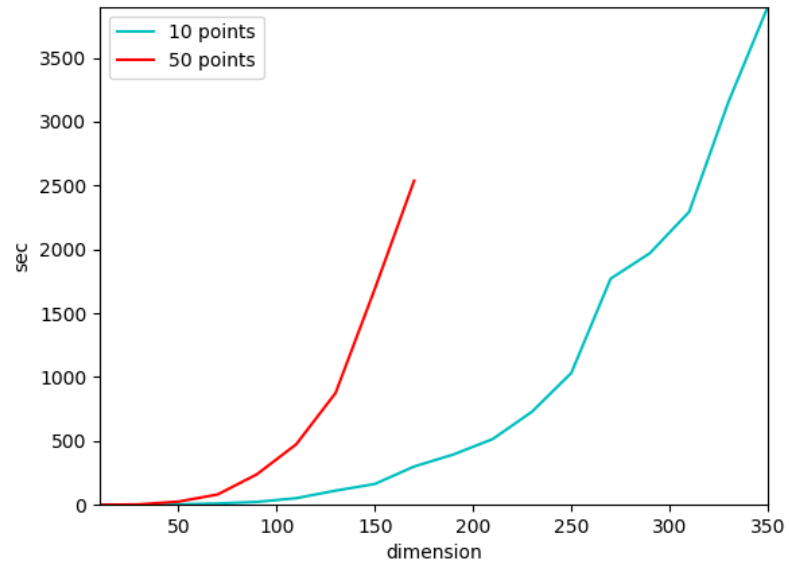

Figure 1.3: Billiard walk: sampling 10 and 50 points per phase.

1.1.1.3 Heuristics

1.1.1.3.1 Getting Random Points In the original algorithm, to produce one random point, a random walk was allowed a mixing time of w steps, so to produce N random points we required Nw steps. In this implementation, to take N random points, we allow the random walk a mixing time of 1 step (it's ok if we don't reach the uniform distribution), or equivalently, the random walk performs N steps and all its intermediate points are taken as samples. This, combined with the reduced cost of the ray shooting in CDH&R, offers a significant speedup.

1.1.1.3.2 1 Point Per Phase Another approach is to sample a single point P per phase. Let AB be the segment on which this point was chosen. Find which one of the end points minimizes the objective function; let it be A. Then cut the polytope at 70% of the segment PA and set the interior point at 85% of PA. This approach didn't offer the expected results; the random walk quickly got stuck and there were problems with numerical stability.

Note *P*erhaps, it makes sense as the polytope gets smaller to sample less points. But this would work on the assumption, that the sampled points are uniformly chosen, which doesn't hold.

1.1.1.3.3 Sampled Covariance Matrix

1.1.1.3.4 Escape Step - Walking To Chebyshev Center An idea for escaping a corner, in which the random walk is stuck, is to try to walk towards the center of the Chebyshev ball. To achieve this, a direction vector must be selected, as well as how far we will move.

Computing the Chebyshev ball amount to computing $\max\min\{b - Ax\}$, or equivalently $\min\max\{Ax - b\}$. Using the LogSumExp (LSE) function, which is a smooth approximation to the maximum function, we get:

$$LSE(Ax - b) = \log \sum_{i=0}^{m} exp(a_i x - b_i)$$

where a_i is the i-th row of A. The Hessian of this function is the softmax function. The direction vector d which we will use is:

$$d = A^T \cdot softmax(Ax - b)$$

Now we must find how far we must traverse so the maximum distance is minimized. This can be formulated as a 2 dimensional linear program and using Seidel's algorithm, we can solve it in $O(m)$.

This works fine if the polytope is round and it actually moves us to the center of the Chebyshev ball. If the polytope is skinny (which will be the case after some cutting planes are added), it needs many repetitions and sometimes it gets stuck.

1.1.1.3.5 Escape Step - Billiard Walk Another attempt to escape from a corner is the billiard walk. We tried selecting as direction vector the $d = A^T \cdot softmax(Ax - b)$ and a random direction, but it didn't manage to help us escape.

1.1.2 Testing

1.1.2.1 Comparing walks in 10 Dimensions

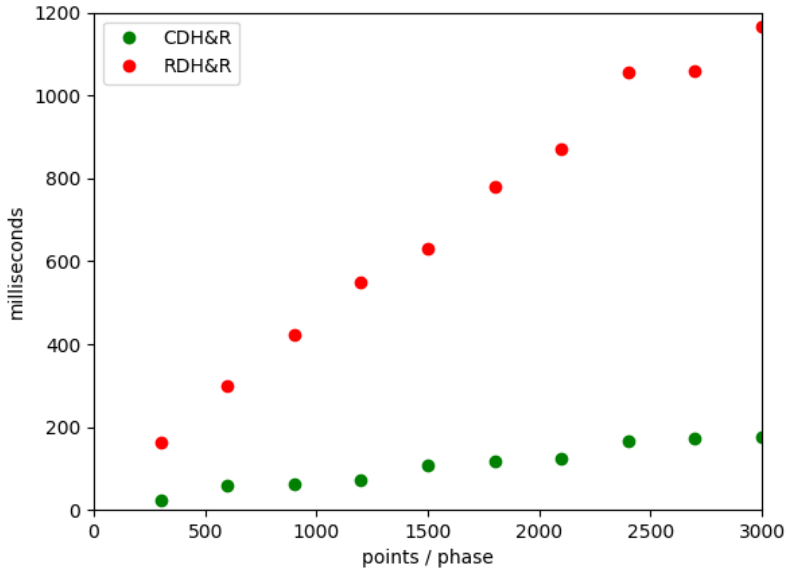

Figure 1.4: Time with respect to points sampled per phase

In this section we compare RDH&R to CDH&R in a randomly created polytope of 10 dimensions and 100 facets. The results are the average of three repetitions of every experiment.

In Figure 1.4 we see the speedup the use of CDH&R offers over RDH&R. But as we see in Figure 1.5, we don't lose accuracy.

1.1.2.2 Behavior of the algorithm in 250 Dimensions

In this section, we examine the behavior of the algorithm when used in a randomly created polytope with 1000 facets, in 250 dimensions. In the following results in this section, every experiment was performed only once, but the goal is to assert how close the algorithm comes to the optimal solution.

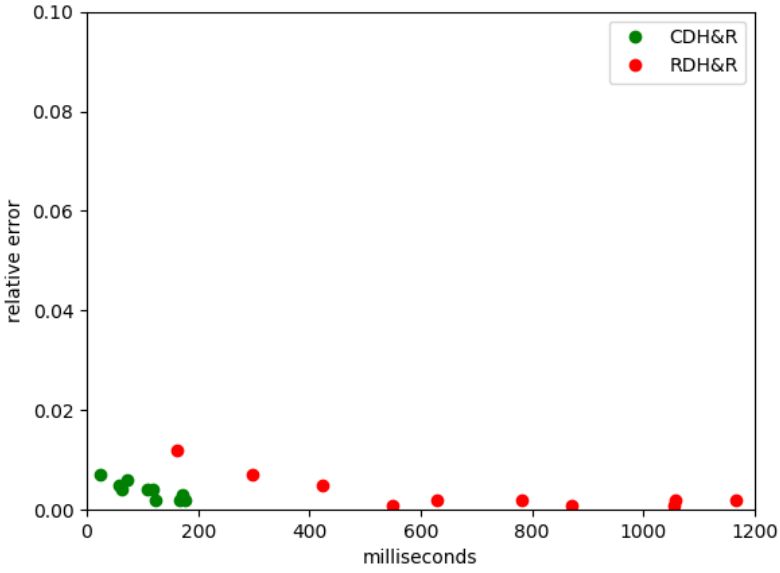

Figure 1.5: Relative error and time

In figure 1.6, we see the relative error of each run, with respect to the number of points allowed to sample per phase. The best relative error the algorithm achieves is just below 0.5%. Even allowing for more points does not bring a noticeable improvement.

Figure 1.7 presents the Euclidean distance at each phase of the algorithm from the point that is the optimal solution, when allowing $25 \cdot 10^5$ points per phase. As expected, since the relative error doesn't reach zero, the distance doesn't reach zero either. Also, in the first 80 phases, the distance closes fast, while in the following steps, it remains about the same. This however, does not mean that there is necessarily no improvement to the objective function; two points may have equal distance from the optimal solution, but still provide very different approximations.

Lastly, in figure 1.8, we see the time of each execution of the algorithm, with respect to how many points we sample per phase.

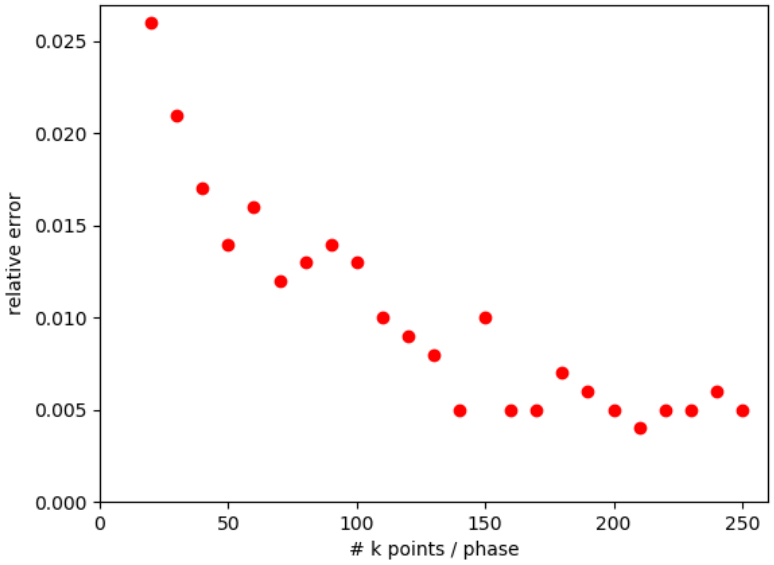

Figure 1.6: Relative error with respect to number of points sampled per phase

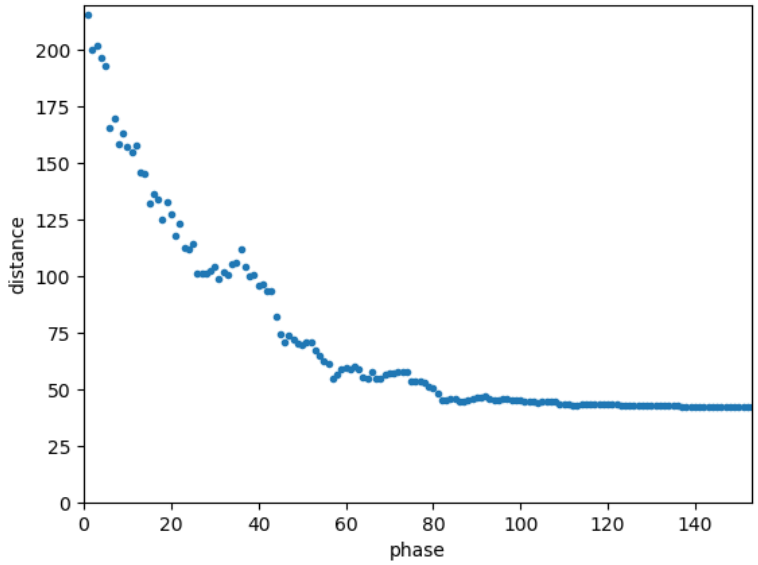

Figure 1.7: The distance from the optimal solution (vertex) per phase

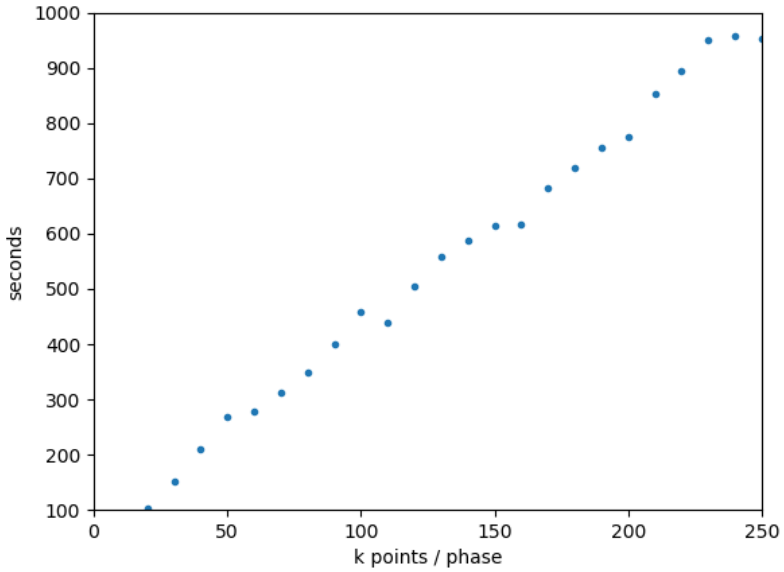

Figure 1.8: Time with respect to points sampled per phase

1.2 Linear Matrix Inequalities

In this part we discuss about linear matrix inequalities and semidefinite programming. In particular, we consider the problem

$$\begin{aligned} &\text{minimize } c^T x \\ &\text{s.t. } F(x) \preceq 0 \end{aligned}$$

where $F(x) = F_0 + \sum_{i=1}^{n} x_i F_i$, $F_i \in \mathbb{R}^{m \times m}$ are symmetrical matrices and $\preceq$ denotes negative semidefiniteness.

1.2.1 Implementation

1.2.1.1 Boundary Oracle

The following lemma is from [2]:

Lemma 1. Let $A \prec 0$ and $B = B^T$. Then the minimal and maximal values of $\lambda \in \mathbb{R}$ retaining the negative semidefiniteness of $A + \lambda B$ are:

$$\underline{\lambda} = \begin{cases} \max\limits_{\lambda_i < 0} \lambda_i \\ -\infty \qquad \text{if all } \lambda_i > 0 \end{cases}$$

and

$$\overline{\lambda} = \begin{cases} \min\limits_{\lambda_i > 0} \lambda_i \\ -\infty \qquad \text{if all } \lambda_i < 0 \end{cases}$$

where λ_i are the generalized eigenvalues of the matrices $(A, -B)$, i.e. $Au = -\lambda B u$.

So using Lemma 1 to find the parameter λ s.t.

$$\begin{aligned} F(x + \lambda v) \preceq 0 \Rightarrow \\ F(x) + \lambda(F(v) - F_0) \preceq 0 \end{aligned}$$

we set $A = F(x)$ and $B = (F(v) - F_0)$. The endpoints of the segment are $x + \overline{\lambda} v$ and $x + \underline{\lambda} v$.

1.2.1.2 Stopping Criterion

As in Lps, we check the relative error between the estimations of two successive phases of the algorithm, but then instead of exiting, we evaluate F at the point we are. If the produced matrix is singular we stop, otherwise continue.

Note *I*f we don't check for singularity often, we may reach too close to the boundary of the spectrahedron and the boundary oracle may not work (due to stability). On the other hand, we may reach the boundary, but not at the optimal solution, so we must also check the relative errors.

1.2.1.3 Initial Point

1.2.1.3.1 Approach 1 From [3], to get an initial point, we solve the auxiliary problem:

$$\begin{aligned} &\text{minimize } \gamma \\ &\text{s.t. } F(x) \preceq \gamma I \end{aligned}$$

As a feasible point for this problem we can take $\{x = 0, \gamma = \max eig(F_0)\}$. If in the optimal solution $\{x^\star, \gamma^\star\}$, $\gamma^\star > 0$, then the problem is infeasible. Otherwise, we can get $x^\star$ as a feasible solution.

Note *A*s for polytopes, we can stop the method when we achieve $\gamma < 0$. If we let it run longer, it will output a point further in the spectrahedron.

This was solved with the barrier and Newton method [4]. The barrier function for a linear matrix inequality $A(x) = A_0 + x_1 A_1 + ... + x_m A_m \succ 0$ is:

$$\Phi(x) = -\ln \det(A(x))$$

with

$$\frac{\partial}{\partial x_i}\Phi = -\text{trace}\{[A(x)]^{-1}A_i\}$$

$$\frac{\partial}{\partial x_i \partial x_j}\Phi = \text{trace}\{[A(x)]^{-1}A_i[A(x)]^{-1}A_j\}$$

So, setting $A(x) = \gamma I - F(x)$, I use the Newton method to solve

$$minf(x) = \gamma - \mu \ln \det A(x)$$

Note *I* compute the inverse of the Hessian matrix, which is expensive and maybe unstable, but since I just run the method for few repetitions, till $\gamma < 0$ there is no difference.

1.2.1.3.2 Approach 2

[Not implemented] Another approach is using a constraint consensus method [5].

These methods need a feasibility vector, defined in the context of liner matrix inequalities as such:

$$d := d(x_0) = \frac{-f(x_0) \bigtriangledown f(x_0)}{|| \bigtriangledown f(x_0)||^2}$$

where $f(x) = \lambda_{min}(F(x)) \geq 0$. If $\{v_1, v_2, ..., v_m\}$ is the orthonormal set of eigenvalues of $F(x)$, then

$$\frac{\partial f(x_i)}{\partial x_i} = v_i^T F_i v_i$$

1.2.2 Testing

1.2.2.1 Generating Tests

Firstly, we can convert the LP problems to SDP problems. Furthermore, as proposed in [3], we create LMIs as such:

- $F_0 = -R_m \cdot R_m^\top - I$
- and for F_i
  - $M = R_{\frac{m}{2}} + R_{\frac{m}{2}}^\top$
  - $F_i = \begin{pmatrix} M & 0 \\ 0 & -M \end{pmatrix}$

where R_m is a randomly generated matrix of odd numbers, of dimension m.

1.2.2.2 Sampling in 2 Dimensions

These tests were performed to test the boundary oracle and the H&R algorithm for spectrahedra. 1000 points were sampled and the results were visualized in MATLAB. See figures 1.9, 1.10, 1.11, 1.12.

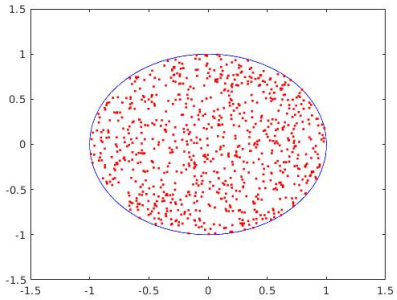

Figure 1.9: A sample of 1000 points in a circle.

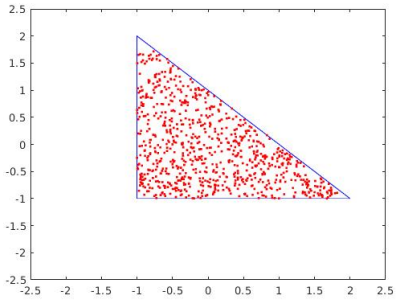

Figure 1.10: A sample of 1000 points in a translated/scaled simplex.

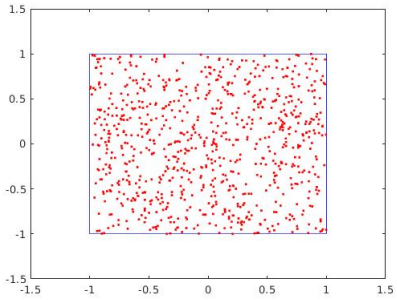

Figure 1.11: A sample of 1000 points in a square.

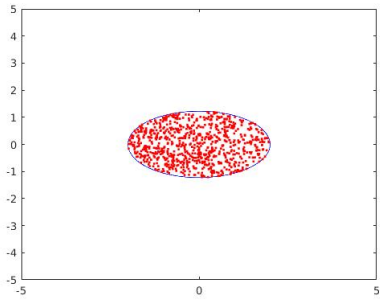

Figure 1.12: A sample of 1000 points in a ellipse.

Chapter 2

Simulated Annealing

The algorithm for simulated annealing for convex optimization from [6]:

Algorithm 1 Simulated Annealing for Convex Optimization

Input: n dimensionality, O_K boyndary oracle for K, c direction of minimization with $|c| = 1$, X_{INIT} starting point, R radius of ball containing K centered at X_{INIT}, R radius of ball contained in K centered at X_{INIT}, I number oh phases, k number of steps per walk, N number of samples per rounding
Output: X_I

1: $(X_0, V_0) \leftarrow$ UniformSample(X_{INIT}, O_K, R, r)
2: **for** $i = 1, 2, ..., I$ **do**
3: $T_i \leftarrow R(1 - \frac{1}{n})^i$
4: $X_i \leftarrow$ hit-and-run(e^{-cx/T_i}, O_K, V_{i-1}, X_{i-1}, k)
5: Update Covariance
6: For $j = 1$ to N: $X_i^j \leftarrow$ hit-and-run(e^{-cx/T_i}, O_K, V_{i-1}, X_{i-1}, k)
7: $V_i \leftarrow \frac{1}{N}\sum_j X_i^j (X_i^j)^\top - (\frac{1}{N}\sum_j X_i^j)(\frac{1}{N}\sum_j X_i^j)^\top$
8: **end for**

To sample with hit and run with the Boltzmann distribution:

- Pick a direction vector v according to n-dimensional normal distribution with mean 0 and covariance matrix V. Let l be the line though the current point in the direction v.
- Move to a random point on the intersection of l and K, with density proportional to the function $f = e^{-cx/T}$.

2.1 Linear Programming

We discuss the simulated annealing method for polytopes.

2.1.1 Implementation

2.1.1.1 Hit and Run with Boltzmann distribution

We already have the boundary oracle for polytopes, so:

1. we choose a direction v. Let l be the line though the current point in the direction v.
2. The oracle returns the intersection points of l and K, let them be A, B.
3. we choose a random point in the one dimensional chord AB with density proportional to the function $f = e^{-cx/T}$.

To achieve this, we use the exponential distribution.

density function (PDF)	$\lambda e^{-\lambda x}$, $\lambda > 0$, $x \geq 0$
distribution function (CDF)	$1 - \lambda e^{-\lambda x}$
quantile (QF)	$\frac{-\ln(1-p)}{\lambda}$, $0 \leq p < 1$

We also want the values to be within a specific range, wo we compute the truncated exponential in range $[a, b]$ as such [7]:

Algorithm 2 Truncated Exponential

Input: λ, a, b, u a uniform number in $(0, 1)$

1: $cdfA \leftarrow CDF(a, \lambda)$
2: $cdfb \leftarrow CDF(b, \lambda)$
3: **return** $QF(cdfA + u(cdfB - cdfA), \lambda)$

So, by writing $e^{-cx/T} = e^{-c(A-B)/T}$ and moving to the end of the segment that minimizes the objective function, let it be B, so $c(A - B) > 0$, we use the algorithm 2 to choose how much to move towards point A.

2.1.2 Heuristics

2.1.2.1 Original Algorithm

These two adjustments were made:

- The starting temperature instead of the radius of ball containing K centered at X_{INIT} is set to be the maximum distance of X_{INIT} from a facet.
- During the execution of the random walks (line 4 in algorithm 1), at each intermediate step, the points that minimize the objective function, on the boundary of the polytope, across the direction vector are kept, and the ”best” is returned at the end of the algorithm. This often is a better approximation.

- The covariance matrix is computed at the beginning and then only when the relative error between two successive estimation is less than 10^{-3}.

Note Numerical stability issue: As seen in the graphs below, when the random walk "sticks" to a corner and keeps moving towards it (not necessarily at the point giving the optimal solution), due to numerical stability the boundary oracle fails.

The following graphs represent executions of the algorithm for different numbers of samples for the covariance matrix r and walk lengths w, for two randomly created polytopes, in 10 (figure 2.1) and 50 dimensions (figure 2.2).

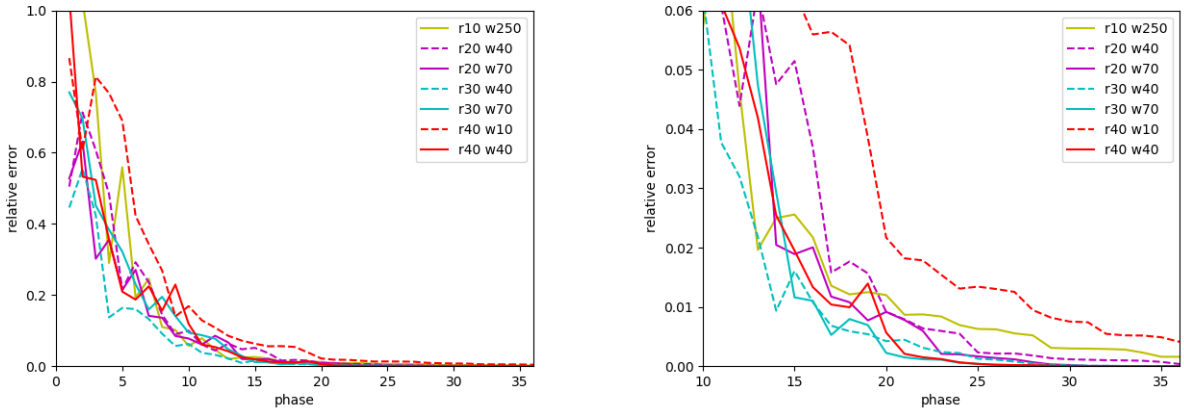

Figure 2.1: Execution of the original simulated annealing algorithm in 10 dimensions, for a randomly generated polytope.

2.1.2.2 Efficient Covariance Matrix

- Instead of sampling new points for the covariance matrix, a large walk length is chosen for the random walk and its intermediate points are used to compute the covariance matrix. The additional overhead is one vector addition per step of the random walk, while to compute the covariance matrix, 2 matrix multiplications, one matrix addition and a Cholesky decomposition are needed. See figures 2.3, 2.4.

- Since computing the covariance matrix is cheaper, we can try to compute it more frequently, i.d. compute i, when the relative error between two successive estimations is 0.1 and not 0.001. But in practice, it didn't make any difference.

Note Currently, I sum the points in each phase, but don't always compute covariance matrix. Perhaps compute covariance matrix more often.

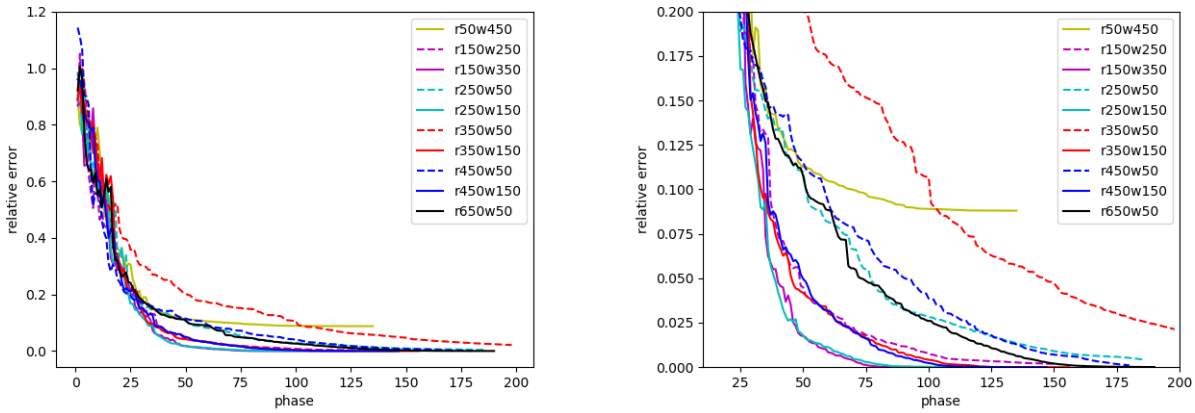

Figure 2.2: Execution of the original simulated annealing algorithm in 50 dimensions, for a randomly generated polytope.

2.1.2.3 Temperature Schedule

Lemma . From [6]:

$$E[c \cdot X] \leq nT + \min_K cx$$

Keeping this lemma in mind:

- Starting from a low temperature (0.01) doesn't work (figures 2.5, 2.6, 2.7).
- The temperature doesn't need to go below $\frac{\epsilon}{n}$, where $\epsilon = E[c \cdot X] - \min_K cx$ is a bound to the error and n the dimensions (figure 2.8). It also makes sense intuitively, because with too small a T, the random walk sampling from the exponential distribution, will probably get stuck.

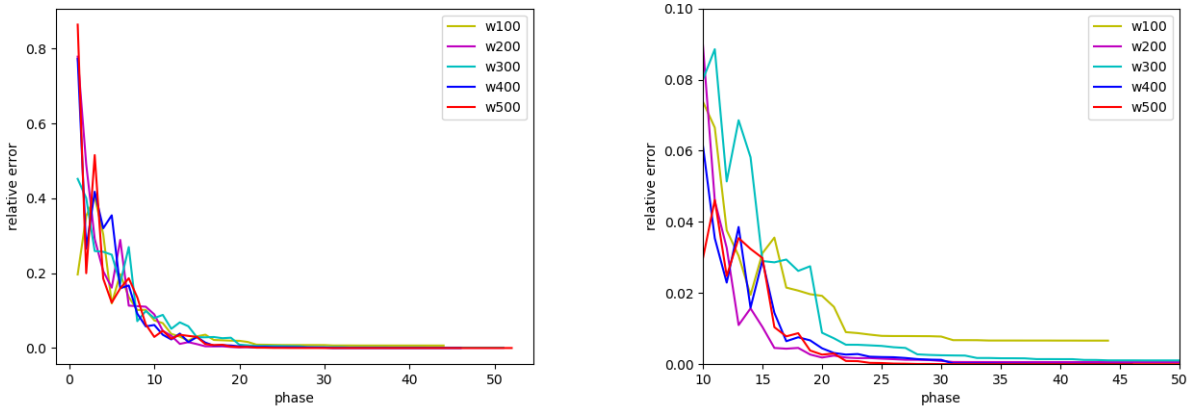

Figure 2.3: Computing the covariance matrix with the intermediate points of the random walk, in 10 dimensions.

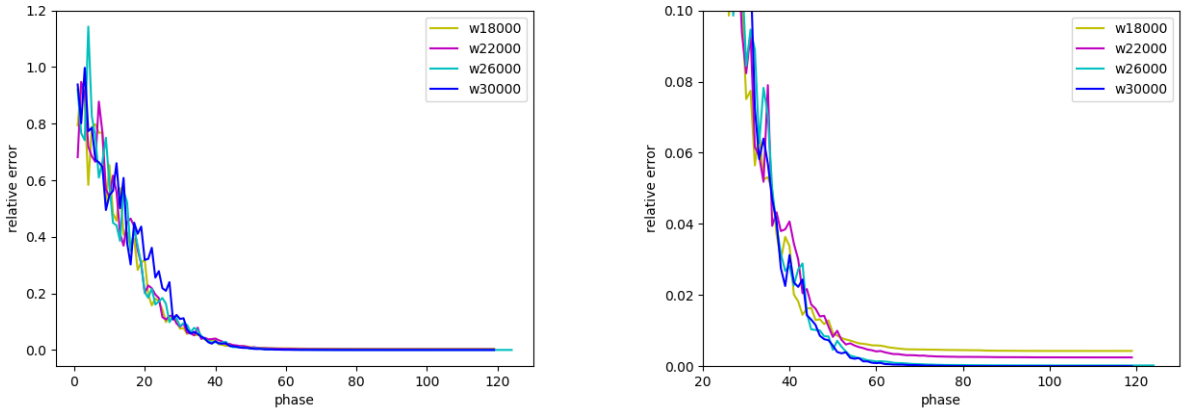

Figure 2.4: Computing the covariance matrix with the intermediate points of the random walk, in 50 dimensions.

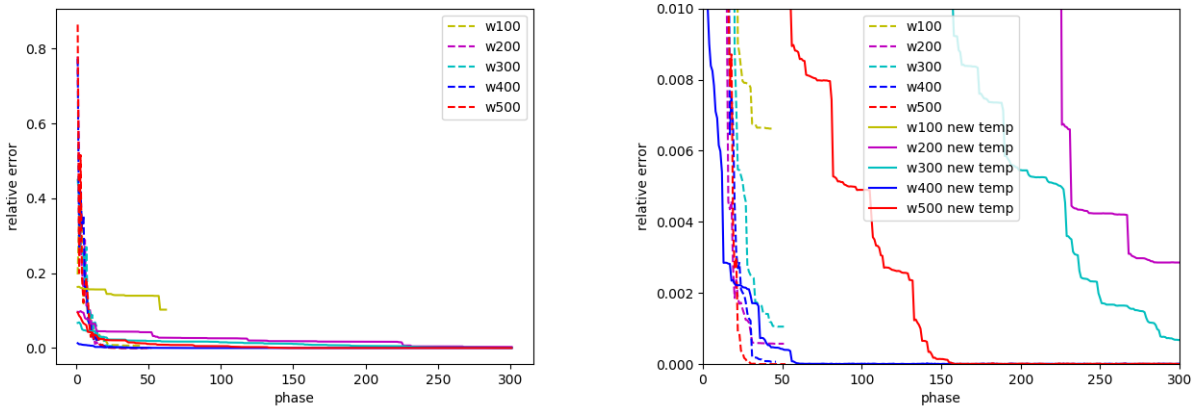

Figure 2.5: Starting the temperature at 0.01, in 10 dimensions.

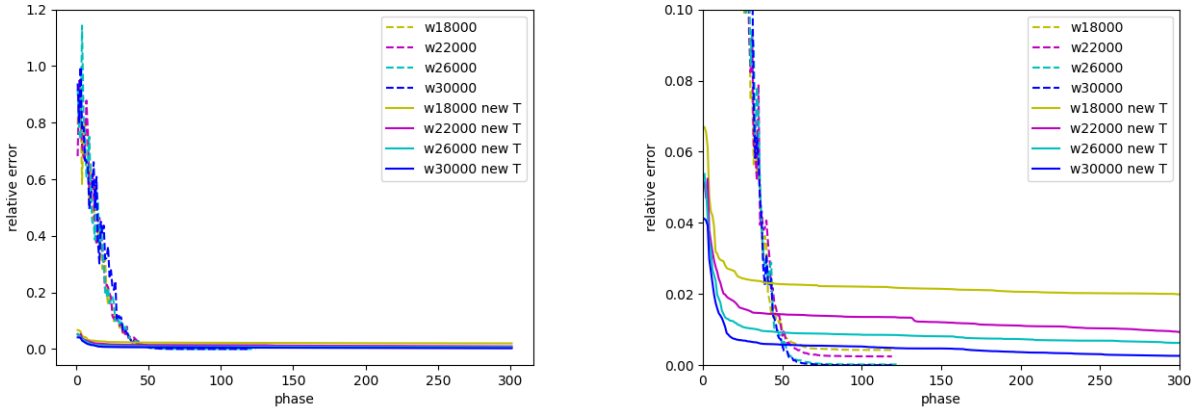

Figure 2.6: Starting the temperature at 0.01, in 50 dimensions.

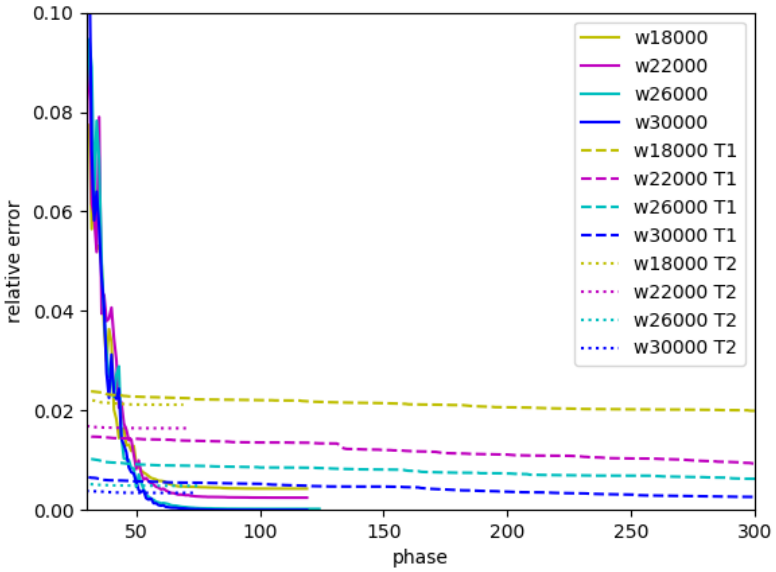

Figure 2.7: Starting the temperature at 0.01, in 50 dimensions.The lower bound for temperature is $10^{-3}/50$ (T1) and 10^{-6} (T2).

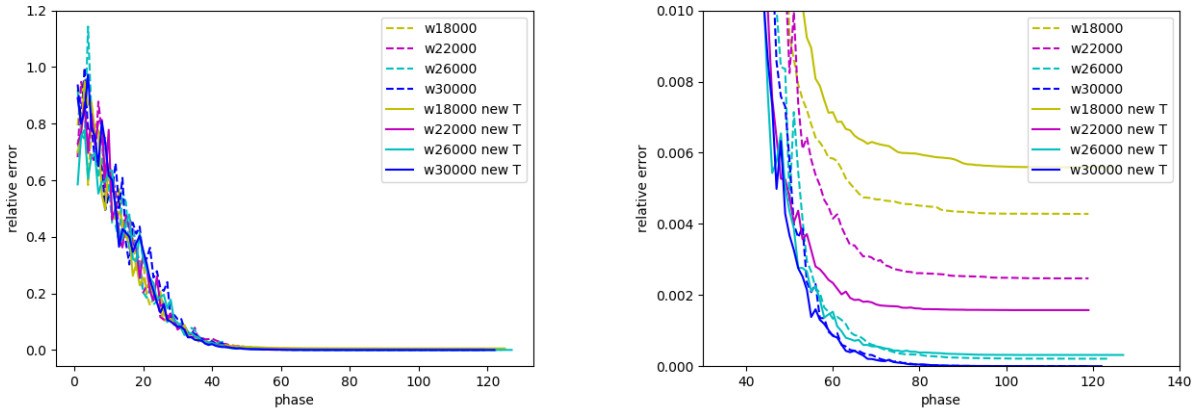

Figure 2.8: The lowest bound for temperature is 10^{-6} (T2).

2.1.2.4 Set Direction Vectors

From the paper [8], but with walk length 1:

- The starting point for the random walk, is the arithmetic mean of the intermediate points of the random walk from the previous phase.
- Instead of computing the covariance matrix, the direction vectors of the random walk will be uniformly chosen in $\{Y_i \mid Y_i = X_i - X\}$, where X_i are the intermediate points of the random walk during the previous phase and X the mean.

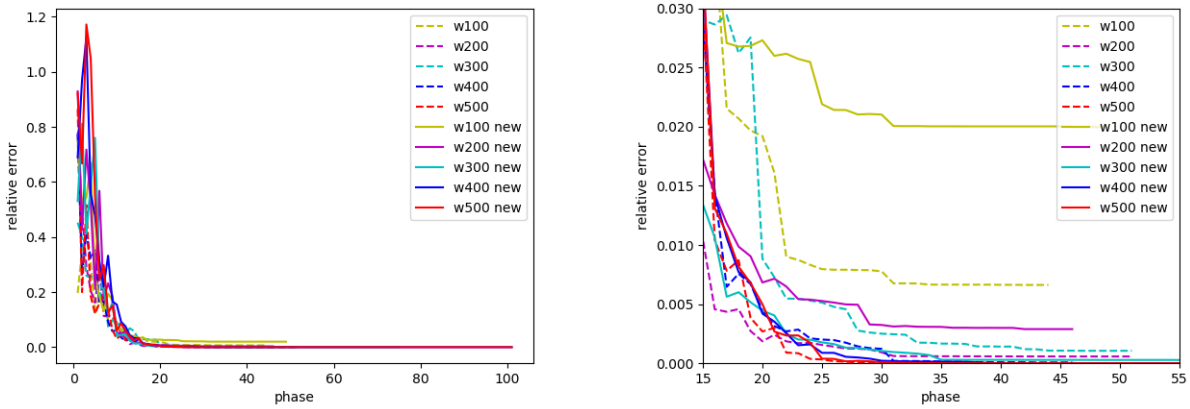

Figure 2.9: Testing the set directions heuristic in in 10 dimensions.

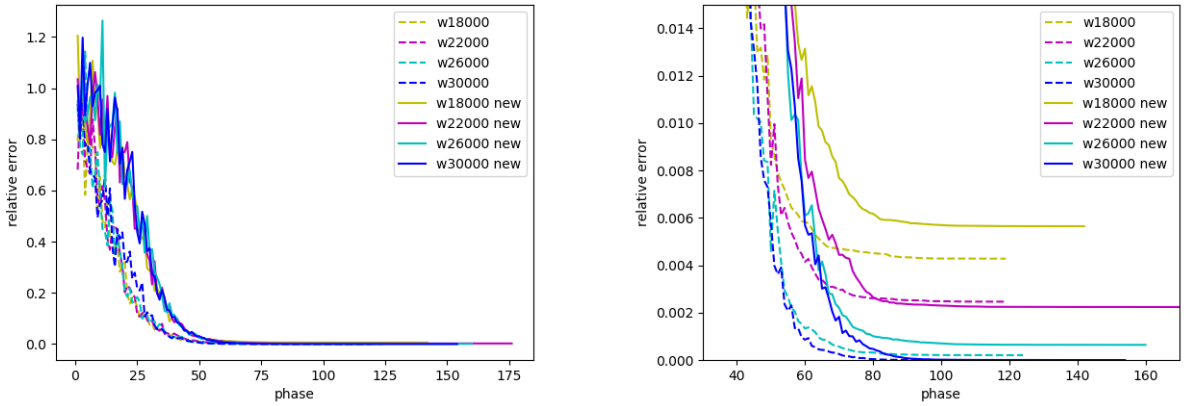

Figure 2.10: Testing the set directions heuristic in in 10 dimensions.

2.1.3 Experiments and Final Version

The latest version of the algorithm is:

- The starting temperature instead of the radius of ball containing K centered at X_{INIT} is set to be the maximum distance of X_{INIT} from a facet.
- The length of the random walk is set to 1, and we sample many points per phase.
- IN order to compute the covariance matrix, the points sampled (the previous bullet) are used.
- During the execution of the random walks (line 4 in algorithm 1), the points that minimize the objective function, on the boundary of the polytope, across the direction vector are kept, and the ”best” is returned at the end of the algorithm. This often is a better approximation.
- The covariance matrix is computed at the beginning and then only when the relative error between two successive estimation is less than 10^{-3}.
- Each time, $1000 + \sqrt{d} \cdot d$ points are used to compute the covariance matrix.
- The temperature doesn't go below $\frac{\epsilon}{n}$, where $\epsilon = E[c \cdot X] - \min_K cx$ is a bound to the error and n the dimensions.
- A window of size $5 + \sqrt{d}$ is used to determine when to stop: when the relative error between the first and last entries is below 10^{-6}.
- A window of size $1000 + \sqrt{d} \cdot d^2$ is used to determine how many points to sample per phase: until the relative error between the first and last entries is below 10^{-5}.

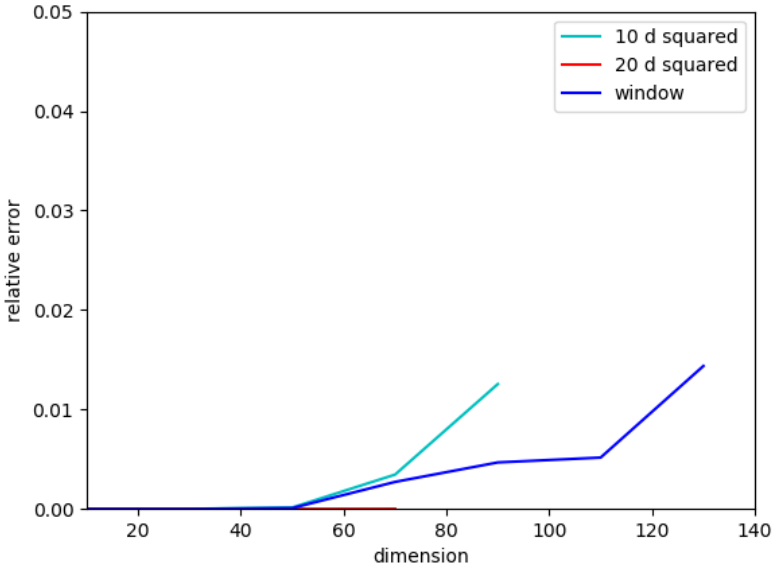

Figure 2.11: Final version of the algorithm: sampling $10d^2$, $20d^2$ points and keeping a window of size $1000 + \sqrt{d} \cdot d^2$.

Bibliography

[1] Boris Polyak. Billiard walk - a new sampling algorithm for control and optimization. pages 6123–6128, 08 2014.

[2] Boris Polyak and Pavel Shcherbakov. The d-decomposition technique for linear matrix inequalities. *Automation and Remote Control*, 67:1847–1861, 11 2006.

[3] F. Dabbene, P. S. Shcherbakov, and B. T. Polyak. A randomized cutting plane method with probabilistic geometric convergence. *SIAM J. on Optimization*, 20(6):3185–3207, October 2010.

[4] Y E Nesterov and A S Nemirovsky. Interior Point Polynomial Methods in Convex Programming~: Theory and Algorithms. 24(3):97–105, 1993.

[5] Shafiu Jibrin and James Swift. Constraint consensus methods for finding strictly feasible points of linear matrix inequalities. *Journal of Optimization*, 2015:1–16, 01 2015.

[6] Adam Tauman Kalai and Santosh Vempala. Simulated Annealing for Convex Optimization. *Mathematics of Operations Research*, 31(2):253–266, May 2006.

[7] RGeode source: R/rexptr.R.

[8] Riley Badenbroek and Etienne de Klerk. Simulated annealing with hit-and-run for convex optimization: rigorous complexity analysis and practical perspectives for copositive programming. *arXiv:1907.02368 [math]*, July 2019. arXiv: 1907.02368.