UPPSALA
UNIVERSITET

A high performance procedure to generate feasible initial guesses for nonlinear optimization

Kajsa Norin, Oskar Wåglund

Project in Computational Science: Report

January 2015

PROJECT REPORT

Department of Information Technology

Abstract

In this project we develop a MATLAB-based algorithm for finding feasible initial guesses using low-discrepancy sequences to generate points. This is done by generating a Halton sequence and then analyzing whether the points found are feasible. Since Halton sequences show tendencies of correlation in higher dimensions, comparisons between permuted Halton and permuted Sobol sequences are made. Finally we develop an algorithm that generates points with the permuted Sobol sequence and seeks additional feasible points in the proximity of already found points using a normal distribution. To speed up the algorithm we implement the code in parallel.

Contents

1 Introduction

The scope of this project is to develop a MATLAB-based algorithm for finding feasible initial guesses by generating points using low-discrepancy sequences. We test two low-discrepancy sequences, the Halton and the Sobol sequence, as well as a permuted version of these two. Additionally, we examine normal distribution around already found feasible guesses. This is implemented in parallel.

1.1 Goal

Our goal is to investigate whether finding feasible initial guesses using low-discrepancy sequences is more efficient compared to the built-in function MultiStart in MATLAB's optimization toolbox. Another objective is to develop a prototype algorithm that can be used for high performance computing.

1.2 Finding initial feasible points

In many numerical optimization algorithms the user has to provide an initial feasible point, or at least a nearly feasible point, to initiate the search. In some applications, just finding a single feasible point can be acceptable as a solution to the problem. Therefore, the problem of seeking and finding an initial feasible point is of great importance.

When solving non-convex optimization problems, the solution depends heavily on the gradient and the Hessian, and also the location of the initial guess. In many applications optimization problems are highly complex, rendering function evaluation tedious and exceedingly computationally demanding. Developing a fast algorithm for finding feasible points is thus essential in these fields, and this is also where high performance computing comes in.

1.3 Applications

One of the applications for this project is the hot rolling process, which is the process where a piece of metal with a temperature above the recrystallization temperature passes through one or multiple pairs of rolls. This makes the metal thinner and gives it a uniform thickness. In addition to this, the inner structure of the metal changes. In the metal industry hot rolling is one of the dominating metal forming processes.

Another application area is model predictive control (MPC) which is widely used in the process industry. It is an advanced control method that is able to predict future states of the controlled system and to adjust the control signal based on that information. A crucial part of MPC is to numerically

optimize the control signal in each time step such that a cost function is minimized.

2 Theory

2.1 Optimization

In this report, the term 'optimization' refers to mathematical optimization, the technique of finding the global minimum or maximum value of a function. An optimization problem consists of an objective function, $f(x)$, which is the function to be minimized on a domain or subject to a set of constraints that limits the space where a solution can be found.

By convention, an optimization problem is defined in the following way:

$$\begin{aligned} &\min_{x} \ f(x) \\ \text{s.t } &G_i(x) = 0, \quad i = 1, \cdots, m \\ &H_j(x) \leq 0, \quad j = 1, \cdots, n \\ &x_l \leq x \leq x_u \end{aligned} \tag{1}$$

where $f(x)$, $G_i(x)$ and $H_j(x)$ are continuously differentiable non-convex functions. $G_i(x)$ and $H_j(x)$ represent equality and inequality constraints respectively. The convention is to define all optimization problems as minimization problems, however a maximization problem can easily be created by negating the objective function. The constraints can be either linear or non-linear. A point x is called a feasible point if it fulfills $G_i(x)$ and $H_j(x)$ and lies between the lower and upper bounds of x. The feasible point x belongs to the feasible set, D, which is the set that consists of all feasible points. A point x^* is called a local optimizer if $f(x^*) \leq f(x)$ for all x in a region around x^*, and a global optimizer if this holds in the entire feasible set D. The corresponding values of the objective function are called a local and a global optimum, respectively.

In order to classify optimization problems the characteristics of the objective function and the constraints are examined. A linear objective function and linear constraints render a problem that can be solved with the Simplex method, while non-linearity requires more advanced methods. Non-linear optimization problems are usually solved with methods using the so-called Karush-Kuhn-Tucker (KKT) conditions to verify optimality. The KKT conditions are defined as

$$\begin{aligned}
\nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla G_i(x) + \sum_{j=1}^{n} \mu_j \nabla H_j(x) = 0 \\
G_i(x) = 0, \quad i = 1, ..., m \\
H_j(x) \leq 0, \quad j = 1, ..., n \\
\mu_j \geq 0, \quad j = 1, ..., n \\
\mu_j H_j(x) = 0, \quad j = 1, ..., n
\end{aligned} \tag{2}$$

These conditions must be fulfilled for an optimal point x^*, and are first order necessary conditions for non-linear optimization problems with inequality constraints. They relate the gradient of the objective function to the gradient of the active constraints. The Lagrange multipliers, λ and μ, state whether the corresponding constraint is active or not. If all multipliers are zero, which means that all the constraints are non-active, it is easy to see that the KKT conditions represent the unconstrained first order necessary condition, which is that the gradient of the objective function is zero. [1]

2.2 Optimization methods

There exists multiple methods to find a solution to optimization problems, where some are more widespread than others. A couple of these methods are covered in this paragraph, more specifically the methods on which MATLAB's numerical optimization toolbox is based. For all of these methods the user has to provide the algorithm with an initial point x_0.

Gradient based methods

Gradient based methods are methods that decide the search direction based on the gradient and the Hessian of the objective function at the current point. An example of a gradient based method is sequential quadratic programming. This method divides the given optimization problem into several quadratic programs, QPs, and finds the descent direction d by solving these. The step length for each direction is found by doing a line search and finally KKT conditions are used to verify that a minimum is reached. The QP is defined as

$$\begin{aligned}
\min_{d} \quad & \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d \\
s.t. \quad & \nabla G_i(x_k)^T d + G_i(x_k) = 0, \quad i = 1, ..., m \\
& \nabla H_j(x_k)^T d + H_i(x_k) \leq 0, \quad j = 1, ..., n
\end{aligned} \tag{3}$$

Direct search methods

Direct search methods, which are also called pattern search methods, are methods which do not require the gradient or the Hessian of the objective function to be computed. Instead, a direct search method searches the neighborhood of the current point and moves to a new point if it results in a lower value of the objective function. Since no information about the gradient is required, gradient search is often used for problems where the objective function is not differentiable.

2.3 Optimization methods in MATLAB

fmincon

`fmincon` is an optimization solver that, given an initial estimate to the solution, is capable of finding local minima of constrained nonlinear multivariable functions.[2] This is a gradient-based solver, which means that it uses the gradient and the Hessian to find the minimum. If a gradient or Hessian function is not supplied by the user, `fmincon` estimates them internally. The solver has four different algorithms to implement which are called `interior-point`, `trust-region-reflective`, `sqp` and `active-set`.

MultiStart

`MultiStart` uses several initial guesses and solves the problem using each point as an initial guess passed to `fmincon`. `MultiStart` runs all `fmincon` solvers and then returns the minimum of the found feasible points.

When defining a `MultiStart` solver there are several options that have to be considered, but for this project the focus is on `StartPointsToRun` and whether or not to use the default point generation algorithm that `MultiStart` provides. `StartPointsToRun` decides how `MultiStart` should select which points are feasible. The different settings are `all`, `bounds` and `bounds-ineqs`. With `all` the `MultiStart` solver will accept all points, while `bounds` and `bounds-ineqs` will discard the points that violate the bounds or bounds and inequality constraints respectively.

By default, `MultiStart` uses the random generator Mersenne Twister, which is a commonly used pseudorandom number generator, to generate points. In order to set custom starting points in `MultiStart` the MATLAB function `CustomStartPointSet` is used. `CustomStartPointSet` enables the user to provide its own starting points as a matrix with each row representing a coordinates of a point. The points are then used as initial guesses by adding them in the `run` object in `MultiStart`, `x = run(ms, problem, tpoints)`.

2.4 Low-discrepancy number generators

The discrepancy is a measure of how non-uniform a number sequence is. [3] When generating points for MultiStart it is undesired to provide points that are clustered together, i.e. with a high discrepancy, since they will probably converge to the same solution. To a certain extent this can be avoided by using low-discrepancy number generators. The generators pick points in a specified number of dimensions from a deterministic number sequence. This results in a set of points that is evenly distributed and covers the area better than a uniformly random set of points, see Figure 1. Low-discrepancy sequences can also be called quasi-random or subrandom sequences. There are several established number sequences that accomplish this, but for this project the Halton and Sobol sequences are investigated.

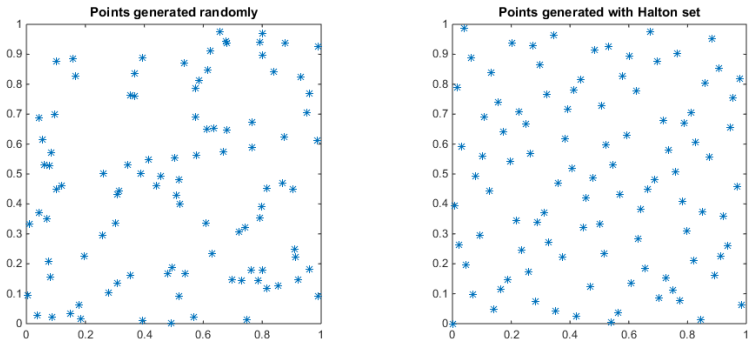

Figure 1: The points from the Halton set are more evenly distributed than the random set.

The Halton sequence

The Halton sequence, invented by J. Halton in 1960, is a low-discrepancy sequence that is defined by bases that consist of prime numbers. For each dimension a unique prime number is needed as a base. Usually the bases are simply selected as the first prime numbers. As an example a 3-dimensional Halton set would have the bases 2, 3 and 5.

The sequence is constructed by systematically evaluating fractions with the bases as denominators. For the next number in the sequence the numerator is increased by one, and if it reaches the denominator the denominator is divided by the base and the numerator is reset to one. If the numerator is an integer multiple of the base it is skipped to avoid recurring point components. To illustrate this the three first points of the 2-dimensional Halton sequence is $(\frac{1}{2}, \frac{1}{3})$, $(\frac{1}{4}, \frac{2}{3})$ and $(\frac{3}{4}, \frac{1}{9})$. Note that $\frac{2}{4}$ is skipped since $\frac{1}{2}$ is already used in the first point. [4]

In MATLAB the Halton sequence is created by the built-in `haltonset` function. The command `p = haltonset(d, 'Skip', s, 'Leap', l)` cre-

ates a `d`-dimensional Halton sequence. With `skip` the user can skip the first `s` values, and with `leap` the algorithm discards `l` values between each pick. The Halton sequence is then stored in the memory with the generator `net` that generates the first `q` points of the Halton sequence `p` with the command `pts = net(p, q)`.

The Sobol sequence

The Sobol sequence, invented by I. M. Sobol in 1967, is made up of a more regular pattern than the Halton sequence, but is harder to understand intuitively. The sequence forms a distinct pattern that looks almost like a grid in lower dimensions, but forms a pattern with lower discrepancy when used in higher dimensions. The n:th point in the Sobol sequence is generated by

$$x_{n,i} = x_{n-1,i} \oplus v_{k,i} \tag{4}$$

supposing that the first $n-1$ points in the sequence are already generated. Here $v_{k,i}$ is a number between 0 and 1, that is generated through a complex algorithm and $\oplus$ is the bitwise XOR-operator. [5]

Permutation

To avoid correlation in low-discrepancy sequences it is common to 'scramble' the sequence, where one uses permutations of the original sequence. For both the Halton and the Sobol sequences there is a built-in permutation implementation in MATLAB. For the Halton sequence the function is called `RR2` and is made up of a reverse-radix (bit-reversal) permutation. It numbers the elements in the sequence from 0 to $n-1$ and then reverses the binary representation of the numbers, creating a new scrambled sequence from these numbers. For the Sobol sequence the scramble implementation is called `MatousekAffineOwen` and it consists of a linear permutation in combination with a random digital shift. It is worth to note that if the user scrambles an already scrambled sequence it will result in a different point set due to the randomness in the scramble algorithm. [2]

2.5 Parallelism

parfor

`parfor` is MATLAB's built-in method to parallelize for-loops. The basic concept is the same as a for-loop, but when using `parfor` each worker performs an iteration independently of one another. When the number of workers is equal to the number of iterations in the for-loop each worker performs one iteration, but if there are more iterations than number of workers, the workers get assigned a set of iterations to compute.

fmincon

`fmincon` has a parallel mode for computing the gradients. This is activated by setting the option `UseParallel` to `true`.

MultiStart

`MultiStart` has a parallel implementation, where it distributes the trial points on the cores. Each worker then calls `fmincon` simultaneously. Parallelism in `MultiStart` is activated just as in `fmincon`, by setting `UseParallel` to `true`. `MultiStart` then runs on the preset number of workers or on the number of workers in the current parallel pool.

2.6 The Rosenbrock function

The Rosenbrock function is a non-convex function that is often used to test optimization algorithms for their performance. The global optimum is found at the bottom of a long flat valley which can be difficult for optimization algorithms to converge to. The multidimensional Rosenbrock function is defined as

$$f(\mathbf{x}) = f(x_1, x_2, ..., x_N) = \sum_{i=1}^{N/2} [100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2] \quad (5)$$

and has a global minimum at $(1, 1, 1, ..., 1)$. A constrained Rosenbrock function was used as a sample problem in this project.

3 Method

3.1 Testing environment

To test the programs they are run on the available servers at the Department of Information Technology at Uppsala University. The servers have the following specifications:

CPU: AMD Opteron (Bulldozer) 6274, 2,2 GHz, 16-core, dual socket
Memory: 120 GB
Operating system: Scientific Linux 6.5
MATLAB: version R2014b

All the results in this report are obtained within the above testing environment used in dedicated mode, which means that no other users are allowed access to the same system during testing.

3.2 Point generation

For all tests the point generation is run 100 times with 100 000 points for each case. The average times and feasible point counts are recorded.

Method 1: MultiStart

To get a reference of the current performance for these types of problems, we examine the generation of initial guesses in `MultiStart`. This is accomplished by using the `OutputFcn` option in `optimoptions` and providing a function which exits the `MultiStart` solver before the first iteration is made. This results in `MultiStart` selecting feasible points without performing any computations to actually solve the problem.

Method 2: MultiStart with Haltonset

A Halton low-discrepancy point set is generated using `haltonset` with `skip` and `leap` as random integers. `skip` lies between 0-1000 and `leap` lies between 0-100. To mimic the `bounds` option in MATLAB, the points are only generated between the lower and the upper bounds. To investigate the effect of `skip` and `leap` the range of them are increased to 0-1 000 000 and 0-10 000 respectively and the method is run again. When referring to "Method 2" the original run with low `skip` and `leap` is implied.

Method 3: Custom algorithm

To speed up the process of evaluating initial guesses a custom algorithm is written in MATLAB. Points are initially selected using points from the Halton sequence distributed between some predefined boundaries. The points are then evaluated in parallel and discarded if they do not satisfy the given constraints.

The parallelization starts by creating a pool of workers, letting them create a separate Halton sequence each and then check each point for feasibility. To ensure that identical sets of points are not created by different workers, `skip` is set individually for each worker to divide the sequence into intervals. After that, the feasible points are collected in a vector. No communication between the workers is necessary which speeds up the algorithm.

By studying reports on similar optimization problems [6, 7] and testing `haltonset` in MATLAB we find that Halton sequences perform poorly in high dimensional problems and that there are number sequences better suited to this situation. The reason for the poor performance of the Halton sequence is that the point components with large bases are highly correlated and not very 'random' at all. As an example, a dimension with the prime base 71, which is the 20th prime number, will take steps of 1/71, or 0.0141... See Figure 2 for an illustration of how points in the 19th and 20th dimension

behave for the first 1000 points of the Halton sequence. The Sobol sequence has previously been shown to perform better in higher dimensions and is therefore considered in this project as well. [8]

To investigate a few different number sequences, Method 3 is used with the Halton, scrambled Halton, Sobol and scrambled Sobol sequences. These methods are labeled

- Method 3a, Halton
- Method 3b, Scrambled Halton
- Method 3c, Sobol
- Method 3d, Scrambled Sobol

Each method is tested with 'high' and 'low' `skip` and `leap` to illustrate the effects of the two settings for each number sequence. As in Method 2, the low setting refers to `skip` as a random integer between 0-1000 and `leap` as a random integer between 0-100. The high setting refers to `skip` as a random integer between 0-10 000 and `leap` as a random integer between 0-1 000 000. When referring to method 3a, 3b, 3c and 3d the runs with high `skip` and `leap` are implied.

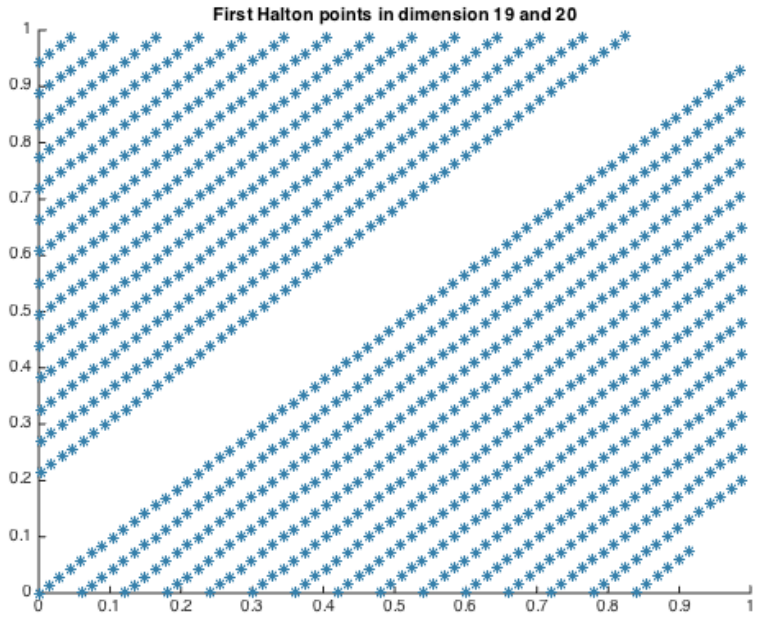

Figure 2: The 1000 first Halton generated points in dimension 19 and 20.

Method 4: Custom algorithm using scrambled Sobol and normal distribution

The final algorithm begins by using the approach of Method 3, however after a certain number of iterations the algorithm calculates a mean value for the positions of the points already found, and uses the mean value to calculate a standard deviation. The algorithm then generates normally distributed points in the proximity of previously found points to find more feasible points nearby. The pseudocode in Algorithm 1 illustrates the core functionality of the implementation. The complete code is found in Appendix B.

Data: The number of points to generate
The MultiStart problem
Result: An m*n matrix with m feasible points in n dimensions.
Each column contains one point coordinate and a row represents a point.

```
for M = 1 to loops do
    if no feasible points found or M is less than loops/2 then
        set seekFirst = true
    else
        set seekFirst = false
        compute mean values and standard deviations
        for normal distribution
    end
    for in parallel do
        if seekFirst then
            Generate Sobol sequence
        else
            Generate normally distributed points around feasible
            points
        end
        for all generated points do
            Check feasibility against linear and non-linear constraints
            Keep the feasible points
        end
    end
end
```

Algorithm 1: A basic description of the custom algorithm

4 Results

4.1 Comparison between the different methods

The two most interesting factors to compare are the CPU time a method takes for a given number of points and how many feasible points are found when generating 100 000 points. Table 1 shows these results for the different methods, with 99 samples for each method and CPU times with 16 workers. Note that Method 1 and Method 2 are using MultiStart which is unable to run in parallel. These methods give a reference of how quickly points are generated with MATLAB's built-in function and how many feasible points that are found.

*Table 1: The number of feasible points and CPU times for different methods with 16 workers. The results marked with * were obtained in serial.*

Method	Samples	Average number of feasible points	CPU time (s)
1	99	6.39	11.63*
2	99	6.12	12.35*
3a	99	6.07	1.20
3b	99	5.38	1.25
3c	99	6.02	1.13
3d	99	6.01	1.16
4	99	8.37	1.03

As previously stated, each method is tested with 'high' and 'low' `skip` and `leap`. Figure 3 shows two histograms, where part A shows the distribution of feasible points found using the Halton sequence with low `skip` and `leap`, and part B shows the distribution with high `skip` and `leap`. A more even distribution can be noted in part B.

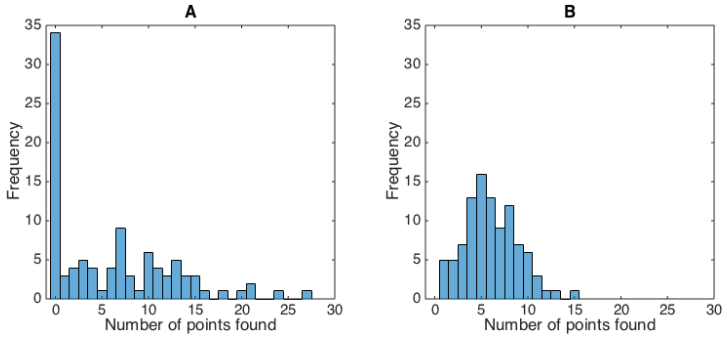

Figure 3: The distribution of feasible points found using the Halton sequence. Part A uses 'leap' and 'skip' from 0 to 100 and 0 to 1000 respectively, and Part B uses 'leap' and 'skip' from 0 to 10 000 and 0 to 1 000 000 respectively.

Figures 4 and 5 show the distribution of feasible points found using methods 3a-d, with low and high `skip` and `leap`, respectively. As in Figure 3, a more even distribution of points can be found in the cases where high `skip` and `leap` is used, especially when using the Halton sequence. These results are obtained by running the algorithms 100 times each.

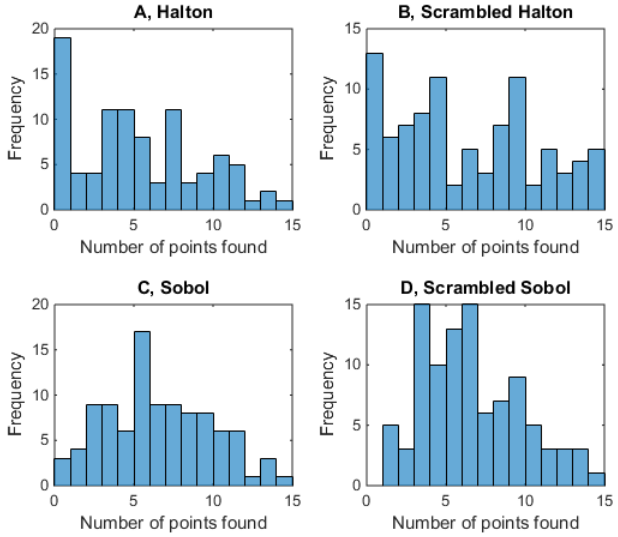

Figure 4: The distribution of feasible points found using different sequences with low skip and leap. Figure A and B uses the Halton sequence and the scrambled Halton sequence, respectively. Figure C and D uses the Sobol sequence and the scrambled Sobol sequence, respectively.

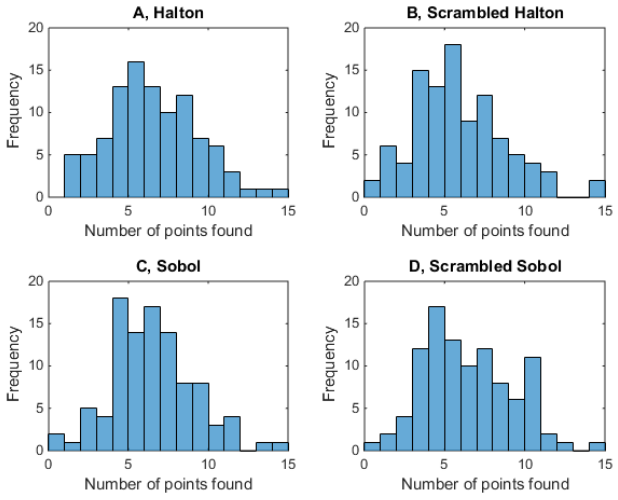

Figure 5: The distribution of feasible points found using different sequences with high skip and leap. Figure A and B uses the Halton sequence and the scrambled Halton sequence, respectively. Figure C and D uses the Sobol sequence and the scrambled Sobol sequence, respectively.

Figure 6 shows the average number of feasible points found using Methods 3a-d. These results are created from the average of 100 runs each for the different methods.

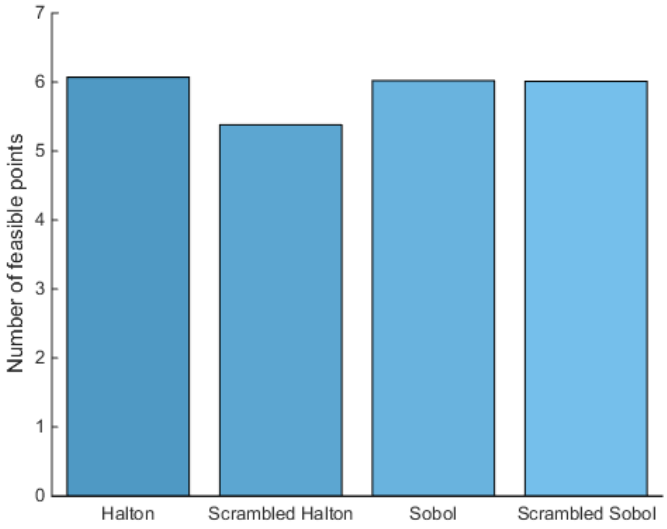

Figure 6: The average amount of feasible points found when using methods 3a-d with different number sequences.

Figure 7 shows the average CPU time when generating 100 000 points with Methods 3a-d.

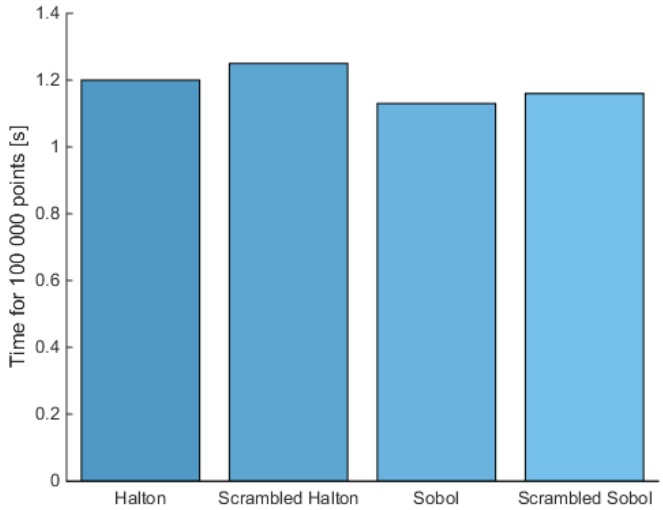

Figure 7: The average CPU times when using Method 3 with different number sequences.

4.2 Speedup in the parallel implementation

The resulting speedup curve of Algorithm 1 is shown in Figure 8, for 1 to 16 workers for a fixed problem size of 100 000 points. For a complete overview of the CPU times, see Appendix A.

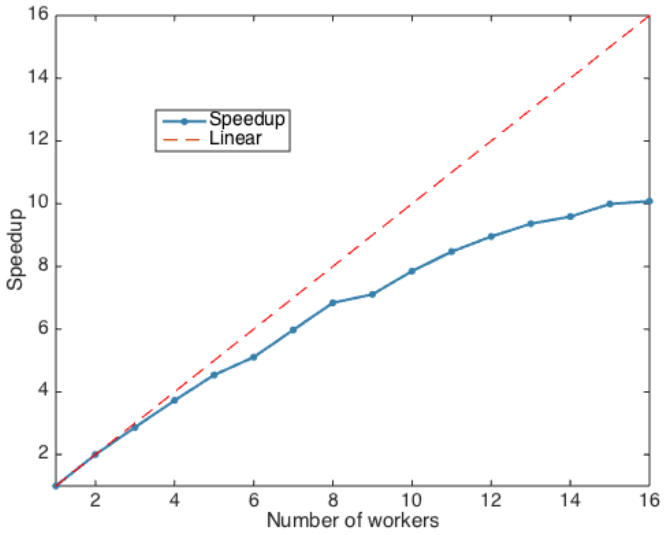

Figure 8: The speedup curve for 1 to 16 workers for a fixed problem size of 100 000 points. The speedup curve is compared to a dotted linear speedup curve.

4.3 Comparison between MultiStart and Algorithm 1

Figures 9 and 10 shows a comparison between MATLAB's MultiStart and our algorithm, Algorithm 1. Figure 9 shows the average number of feasible points found when generating 100 000 points, and Figure 10 shows the average CPU times taken for the same generation.

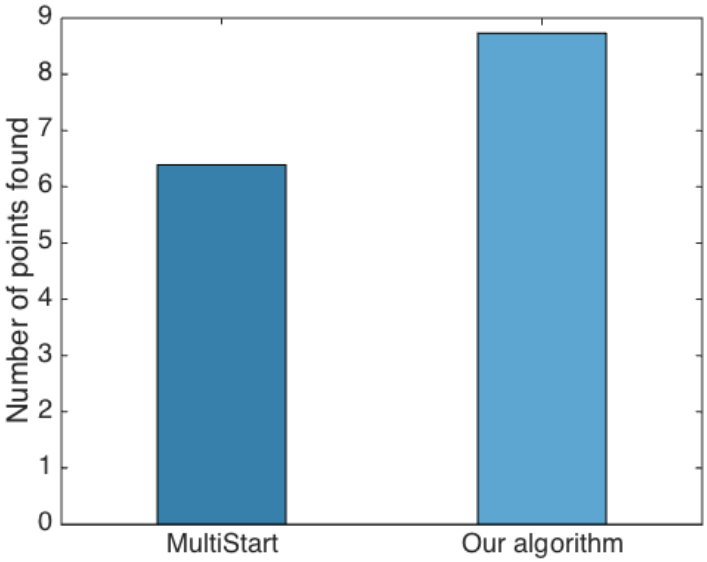

Figure 9: The average number of feasible points found using MultiStart and Algorithm 1.

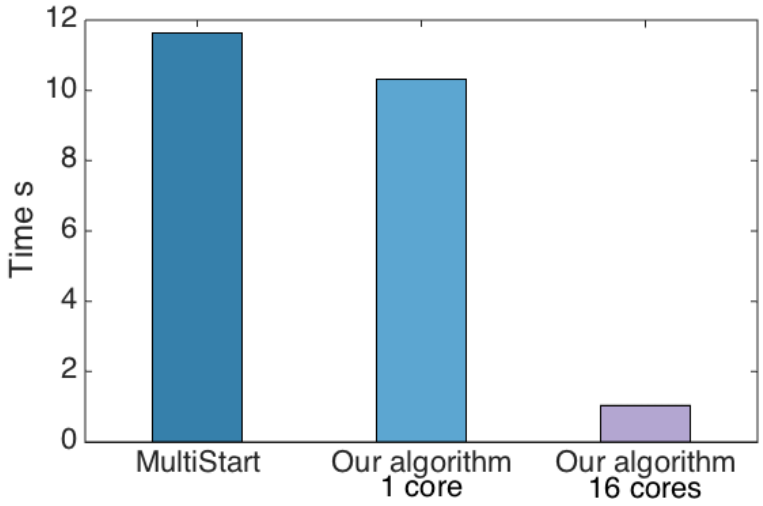

Figure 10: The average CPU times for MultiStart and Algorithm 1 using 1 and 16 cores.

5 Discussion

As seen in Figure 8, when running the implementation on 16 workers the parallel efficiency is a bit more than 50% which is an acceptable, but not ideal, result. A speedup of almost 10 is still a huge increase in performance even if the affected part only makes up a small piece of the complete op-

timization problem. We believe that the performance loss is due to the merging of point sets when the program comes back from the parallel part.

We can see in Figure 3 that increasing the skip and leap variables in the Halton sequence makes the generation of points more evenly distributed, which to some extent can substitute permuting the algorithm. It is however worth noting that the Halton sequence still shows repetitive tendencies in higher dimensions.

There seems to be several reasons for choosing the Sobol sequence over the Halton sequence. Generating points using the Sobol sequence is slightly faster. This is expected since the Sobol sequence uses a very efficient bitwise operation to compute a new point while the Halton sequence uses time consuming arithmetics. The point distribution also seems to be more even, especially when skip and leap are set to low values. Even if the different sequences produce approximately the same amount of feasible points on average, it is desirable to use a sequence that performs more evenly. In higher dimensions this becomes even more important.

To further increase the number of found feasible points, Algorithm 1 uses a normal distribution to focus the search of new points around already found feasible points. The way it is used can however be considered further. It is difficult to set the standard deviation dynamically to fit all problems and a normal distribution is by definition centered around a point, which may lead to an uneven distribution of feasible points.

During this project we found that the main reason for the program being slow was that it is written in MATLAB. For example, we know that MultiStart can run in parallel but we cannot know how it is parallelised, therefore much of our work was dedicated to guessing and testing how MATLAB works 'under the hood'. Using MultiStart means using a program that is a black box.

This leads us to believe that in order to truly speed up the optimization process the code should be written in C or any other lower level programming language, where one has full control over how the functions work. MATLAB is very user-friendly but suffers a lot from performance issues which become evident when the parallel implementations are introduced. Porting to C may however be time-consuming and was not a part of the scope of this project.

In this project we have investigated the Halton and Sobol sequences, and the permuted versions of these, and this is what we base our results on. It is however clear from our literature study that there are other ways of generating low-discrepancy sequences that might be more efficient in higher dimensions, such as the Faure and Niederreiter sequences and Latin hypercube sampling, and also other ways of permuting the sequences. This is not something that we have had enough time and resources to look into, but it might be considered for future work. To further establish the results in this report it is recommended to run the methods on a computer cluster. Another possibility of improving this algorithm could be to introduce slack

variables to the inequality constraints and make the search for an initial point an optimization problem of its own. By minimizing the slack variables it may be possible to find a feasible initial point and use it as a basis for finding more feasible points.

When testing our algorithm we use only the non-linear constraints that were given to us initially in the project. For the results to be more reliable one has to test multiple sets of constraints which also could be addressed in a future project.

6 Conclusions

Compared to the native MultiStart algorithm in MATLAB our implementation performs very well. The CPU time is shorter and in terms of points found it produces a better result than MultiStart. The task of finding the very first feasible point is still a problem and should be explored further, possibly by using the slack method previously mentioned in "Discussion".

After having done this project, it is our general recommendation to permute low-discrepancy sequences, to 'scramble' them. In this report, we show that the Halton sequence exhibits correlation in higher dimensions. We are also establishing that permutations increase the chance of finding feasible points due to a better distribution, while still not being computationally heavy. When dealing with high dimensional problems it is better to use the Sobol rather than the Halton sequence, as the scrambled Sobol sequence is a good candidate for generating low-discrepancy points in higher dimensions.

Acknowledgement

We would like to thank our supervisors Kateryna Mishchenko and Maya Neytcheva for their valuable support throughout the project.

References

[1] Kuhn, H. and Tucker, A. W. 1951. *Proceedings of 2nd Berkeley Symposium.* Berkeley: University of California Press. pp. 481-492.

[2] Mathworks. (2014). *Global Optimization Toolbox: User's Guide (r2014b).* Retrieved January 2, 2015 from *www.mathworks.com/help/pdf_doc/gads/gads_tb.pdf*

[3] Kocis, L. and Whiten, W. 1997. *Computational investigations of low-discrepancy sequences.* ACM Transactions on Mathematical Software (TOMS) Volume 23, Issue 2, pp 266-294.

[4] Ökten, G., Shah, M. and Goncharov, Y. 2012. *Random and Deterministic Digit Permutations of the Halton Sequence.* Springer Proceedings in Mathematics & Statistics Volume 23, 2012, pp 609-622.

[5] Krykova, I. 2003. Thesis, *Evaluating of path-dependent securities with low discrepancy methods.* Faculty of the Worcester polytechnic institute.

[6] Hess, S., and Polak, J. 2003. *An alternative method to the scrambled Halton sequence for removing correlation between standard Halton sequences in high dimensions.* ERSA conference papers ersa03p406, European Regional Science Association.

[7] Hess, S., Train, K. and Polak, J. 2006. *On the use of a Modified Latin Hypercube Sampling (MHLS) method in the estimation of a Mixed Logit Model for vehicle choice.* Transportation Research Part B: Methodological, Elsevier, vol 40(2), pages 147-163, February.

[8] Morokoff, W. and Caflisch, R. 1995. *Quasi-Monte Carlo Integration.* Journal of Computational Physics, pp 218-230.

Appendices

A

Table 2: CPU times for 1 to 16 cores when generating 100 000 points with Algorithm 1.

Number of cores	1	2	3	4	5	6	7	8
CPU time (s)	10.32	5.15	3.60	2.77	2.27	2.02	1.73	1.51

Number of cores	9	10	11	12	13	14	15	16
CPU time (s)	1.45	1.31	1.22	1.15	1.10	1.08	1.03	1.02

B

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Feasible point generator
%
% This algorithm finds one or more feasible points using
% a scrambled Sobol sequence and then generates additional
% points around the first points using a normal distribution.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function newPts = pointGenerator2(numPoints, problem, varargin)

%Open a parallel pool if one doesn't already exist
try
    pool = parpool;
catch err
    pool = gcp;
end

nargin = size(varargin, 1);
if(nargin > 0)
    maxTime = varargin{1};
end

Aeq = problem.Aeq;
beq = problem.beq;
Aineq = problem.Aineq;
bineq = problem.bineq;
nonlincon = problem.nonlcon;
lb = problem.lb;
ub = problem.ub;
dim = max(size(problem.x0));

%Initialize the resulting new points as empty
newPts = zeros(dim, 0);

%Seed the random generator with Mersenne Twister
%Make sure that we use different seeds
rng(mod(tic, 2^32), 'twister');

workers = pool.NumWorkers;
majorLoops = 100;
N = floor(numPoints/majorLoops);

%Make sure that the number of points are evenly divided by the workers
N = N - mod(N, workers);

%How many points each worker should investigate
interval = N/workers;
```

```
hasLinearConstraints = true;
hasNonlinearConstraints = true;

%Check if there are linear inequality constraints
if(isempty(Aineq))
    hasLinearConstraints = false;
    disp('No linear constraints');

elseif(size(Aineq, 2) ~= dim)
    disp('Wrong dimensions of A!');
    return;
end

%Check if there are non-linear inequality constraints
if(isempty(nonlincon))
    hasNonlinearConstraints = false;
    disp('No nonlinear constraints');
end

%Check if bounds are set incorrectly
if(~isscalar(lb) && ~isempty(lb) && size(lb, 1) ~= dim)
    disp('Bad bounds!');
    return;
end

if(~isscalar(ub) && ~isempty(ub) && size(ub, 1) ~= dim)
    disp('Bad bounds!');
    return;
end

%Check if bounds are set and if not set them arbitrarily
if(isempty(lb) && isempty(ub))
    lb = -ones(dim,1);
    ub = ones(dim,1);
end

%At least one of the bounds is not empty
if(isscalar(lb))
    lb = lb*ones(dim, 1);
end
if(isscalar(ub))
    ub = ub*ones(dim, 1);
end

if(isempty(lb))
    lb = (ub-3);
elseif(isempty(ub))
    ub = (lb+3);
end

means = zeros(dim, 1);
stdev = zeros(dim, 1);

tic;
%Major loop
```

```
for major=1:majorLoops
    %If no points have been found yet, use Sobol sequence
    if(size(newPts, 2) == 0)
        seekFirst = true;
    %If we have found a point we can focus on that area
    else
        seekFirst = false;
        %Get mean and Stdev for the normal distribution
        maxima = max(newPts, [], 2)';
        minima = min(newPts, [], 2)';
        means = 0.5*(maxima+minima);

        defaultStdev  = 3; %Consider setting this dynamically
        dynamicStdev = maxima-minima;
        stdev = max(defaultStdev, dynamicStdev);
    end

    disp(' ');
    disp(['**Iteration ' num2str(major) ': Found '...
         num2str(size(newPts,2))' points so far!**']);

    parfor i = 1:workers
        if(seekFirst)
            %Create Sobol set
            %Select the skip and leap randomly
            skip = randi(1e6);
            leap = randi(1e4);

            %Define the Sobol sequence
            p = sobolset(dim, 'Skip', skip+(i-1)*interval*leap,...
                'Leap', leap);
            p = scramble(p, 'MatousekAffineOwen');
            %Generate points from the sequence
            pts = net(p, interval)';

            for j = 1:dim
                pts(j, :) = lb(j) + (ub(j) - lb(j))*pts(j, :);
            end
        else
            %Generate normally distributed points around the points
            %that are already found
            pts = mvnrnd(means, stdev, interval)';
        end
        %Extract the feasible points in parallel with respect to the
        %constraints of the problem

        %Indices says which points are feasible
        %A 1 means feasible and 0 means not feasible
        indices = false(1, size(pts, 2));

        %Loop through each point and set indices(i) to 1 if the point
        %is feasible
        for j = 1:size(pts, 2)
            point = pts(:, j);

            % Non-linear constraints
            if(hasNonlinearConstraints)
```

```
                nonlinconValues = nonlincon(point);
                satisfiedConstraints = (nonlinconValues < ...
                    zeros(size(nonlinconValues)));
                if(~all(satisfiedConstraints))
                    continue; %continue to next point
                end
            end

            % Linear constraints
            if(hasLinearConstraints)
                satisfiedConstraints = (Aineq*point < bineq);
                if(~all(satisfiedConstraints))
                    continue; %continue to next point
                end
            end
            indices(j) = true;
        end

        %Get the indices for the points which are feasible
        ind = find(indices);

        %Merge the feasible points
        newPts = [newPts, pts(:, ind)];
        disp(['Core ' num2str(i) ' found ' ...
            num2str(size(ind, 2)) ' points!']);
    end
    time = toc;
    if(exist('maxTime', 'var') && time > maxTime)
        break;
    end
    estTime = time/major*majorLoops;
    disp(['Estimated time left: ' num2str(estTime-time) ' seconds.']);

end
disp(' ');
disp('The search is complete!');
```