NTNU

Norwegian University of
Science and Technology

A Parallelized Global Optimization Method for use in Parameter Estimation

Haakon Eng Holck

Chemical Engineering and Biotechnology
Submission date: June 2018
Supervisor: Nadav Skjøndal-Bar, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

Abstract

When modeling biological systems with a bottom-up approach, the system parameters need to be calibrated for the model behavior to match that of the physical system. The problem is typically presented in the form of an optimization problem, where the deviation between model outputs and experimental data is minimized as a function of the parameters. There are multiple methods available, but for some complicated models many either fail or take too long to find the correct solution.

In this thesis, the covariance matrix adaptation evolution strategy (CMA-ES), a stochastic global optimization method, is implemented for the purpose of parameter estimation on difficult problems. The CMA-ES is parallelized using message passing interface communication to reduce the computational time.

The method was tested on two models: An ODE model with known parameters and a hybrid DAE model with unknown parameters. The CMA-ES managed to find the correct parameters for the ODE model, albeit after a longer time than that needed by alternative methods included in MATLAB (trust-region-reflective and Levenberg-Marquardt). When it came to the hybrid DAE model, the CMA-ES minimized the model error, while the MATLAB methods that were tested were not applicable. However, the minimum found did not produce the desired model behavior. The most likely reasons for this are that the model was incomplete, the search space was badly defined or the objective function contained too much noise.

The results indicate that the parallel CMA-ES method is too slow for simpler model calibration problems, but that it may be a powerful tool for parameter estimation problems with noisy objective functions and long runtimes.

Sammendrag

Ved modellering av biologiske systemer ved en "bottom-up" tilnærming, må parametrene i systemet bli kalibrert slik at modellens atferd samsvarer med det fysikalske systemet. Dette er vanligvis presentert som et optimaliseringsproblem, hvor avviket mellom modellens output og eksperimentelle data minimeres som en funksjon av parametrene. Det er flere metoder som kan brukes for dette, men på kompliserte modeller er det mange som enten mislykkes eller bruker for lang tid på å finne den korrekte løsningen.

I denne oppgaven implementeres *covariance matrix adaptation evolution strategy* (CMA-ES), en stokastisk global optimaliseringsmetode. Formålet er å bruke den til parameterestimering på vanskelige problemer. For å redusere kjøretiden, blir CMA-ES parallellisert ved bruk av message passing interface-kommunikasjon.

Metoden ble testet på to modeller. En modell i form av et system differensialligninger med kjente parametere og en hybrid modell bestående av et differensial-algebraisk ligningssystem med ukjente parametere. CMA-ES fant de korrekte parameterverdiene for differensialligning-modellen, men med lengre kjøretid enn det som krevdes av trust-region-reflective og Levenberg-Marquardt metodene som er inkludert i MATLAB. Videre fant CMA-ES også et minimum for hybridmodellen. MATLAB-metodene som ble testet var uegnede for dette problemet. Minimumet som ble funnet produserte imidlertid ikke den ønskede modellatferden. De mest sannsynlige grunnene for dette er at feil i modellen, et dårlig definert søkeområde eller at det var for mye støy i målfunksjonen.

Resultatene indikerer at den parallelliserte CMA-ES metoden er for tidkrevende for enkle modelltilpasningsproblemer, men at den kan være et kraftig verktøy for problemer med støyete målfunksjoner og lange kjøretider.

Preface

This master's thesis was written during the spring of 2018, as the conclusion of the five year master's programme in Industrial Chemistry and Biotechnology at NTNU.

I wish to express my sincerest gratitude to my supervisor Professor Nadav Bar, for his guidance, and to John Floan, senior engineer at the NTNU IT Development Section, for his help with parallel programming. Thanks to Aud-Therese Tostrup for answering my questions relating to the RNAP model.

I am immensely grateful to my parents, Pedro, Dag and Fatbardha for their help and support.

Trondheim, June 2018
Haakon Eng Holck

Table of Contents

Abstract i

Sammendrag ii

Preface iii

Table of Contents v

List of Tables ix

List of Figures xi

Nomenclature xiii

1 Introduction 1
- 1.1 Problem Description . . . 1
- 1.2 Framework . . . 2
- 1.3 Structure of the Thesis . . . 3

2 Global Optimization 5
- 2.1 The Covariance Matrix Adaptation Evolution Strategy . . . 6
  - 2.1.1 Description . . . 6
- 2.2 Choosing Initial Mutation Strength . . . 10
- 2.3 Negative Recombination Weights for Step Adaptation . . . 10
- 2.4 Negative Recombination Weights for Covariance Adaptation . . . 11
- 2.5 Termination Criteria . . . 11
- 2.6 Boundary and Constraint Handling . . . 11

3 Parallelization 13
- 3.1 Structure of the Parallel Program . . . 13
- 3.2 MPI Functions . . . 17
- 3.3 Choice of Population Size . . . 17

4 Testing on a Delayed Negative Feedback Model **19**
4.1 Experimental Conditions . . . 21
4.2 Results . . . 22
4.2.1 Speedup . . . 22
4.2.2 Parameter Estimation . . . 23
4.3 Discussion . . . 27
4.3.1 The Speedup Test . . . 27
4.3.2 The Parameter Estimation . . . 27

5 Testing on an RNAP Hybrid Model **29**
5.1 The Model . . . 29
5.2 Constraints . . . 31
5.2.1 First Failure Mode: No Productive Yield . . . 31
5.2.2 Second Failure Mode: 100% Productive Yield . . . 33
5.2.3 Adding an Upper Bound . . . 34
5.3 Intensification: Improving the Accuracy of the Solution . . . 34
5.4 The Multistart Implementation . . . 35
5.4.1 Dividing Into Batches . . . 35
5.4.2 Intra-Batch MPI Communication . . . 36
5.4.3 Selecting Initial Points . . . 37
5.5 Conditions for Parameter Estimation . . . 38
5.6 Results . . . 39
5.6.1 Single CMA-ES Instance . . . 39
5.6.2 CMA-ES Multistart . . . 43
5.7 Discussion . . . 44

6 Further Discussion **47**

7 Conclusion **49**
7.1 Further Work . . . 50

Bibliography **51**

Appendix **i**

A Effects of Parallelization **i**

B The Results of a 10% Change of Parameter Values **v**

C Parallelized CMA-ES Code **xi**

D Multistart Code **xix**
D.1 CMAES_multistart.m . . . xix
D.2 myparsteps.m . . . xxvi
D.3 myreduction.m . . . xxvii
D.4 myspread.m . . . xxviii

E Results from the Multistart Run **xxxi**

List of Tables

4.1 The parameters used in the delayed negative feedback model 20
4.2 The search space in the delayed negative feedback problem 21

5.1 The solution of the RNAP parameter estimation 39

A.1 The test functions used for parallel benchmarking i

E.1 The parameters and corresponding fitnesses returned by the different batches in the multistart run xxxii

List of Figures

3.1 A flowchart describing the sequential CMA-ES algorithm 15
3.2 A flowchart describing the parallel CMA-ES algorithm 16

4.1 A bifurcation diagram illustrating the values of S for which the system oscillates 20
4.2 The cumulative distribution of runtimes in the speedup test 22
4.3 The cumulative distribution of runtimes in the speedup test, without the 10% slowest runs. 23
4.4 The cumulative distributions of fitness for the three optimization methods 24
4.5 The cumulative distributions of fitness for the three optimization methods, with limited x-axis 24
4.6 The cumulative distribution of runtimes for each of the three optimization methods 25
4.7 The parameters found using the three optimization methods, for fitness < 0.5 25
4.8 The resulting change in the error of the model when changing each of the parameters by 10% from the correct value 26

5.1 The fitnesses in the first failure mode, RNAP 32
5.2 The parameters in the first failure mode, RNAP 32
5.3 The fitnesses in the second failure mode, RNAP 33
5.4 The parameters in the first failure mode, RNAP 34
5.5 An example of where the starting points would be placed in a two-dimensional multistart problem with systematic distribution of starting points 38
5.6 The model output of the solution of the single-instance RNAP parameter estimation, with error bars 40
5.7 Fitness history of the single-instance RNAP parameter estimation 40
5.8 Parameter history of the single-instance parameter estimation 41
5.9 Step size between each generation in the single-instance parameter estimation 41

5.10 The model evaluation time for each generation in the single-instance parameter estimation 42
5.11 The mean values for the parameters in the ten solutions with fitnesses less than 0.2 43
5.12 The model output of the solution of the multistart RNAP parameter estimation, with error bars 44

A.1 The number of model equation sequences for 1 to 40 ranks when evaluating the Ackley function ii
A.2 The number of model equation sequences for 1 to 40 ranks when evaluating the Rastrigin function iii
A.3 The number of model equation sequences for 1 to 40 ranks when evaluating the Rosenbrock function iii
A.4 The number of model equation sequences for 1 to 40 ranks when evaluating the Sphere function iii

B.1 The model output when changing the value of k_3 by 10% vi
B.2 The model output when changing the value of k_4 by 10% vi
B.3 The model output when changing the value of K_{mk} by 10% vii
B.4 The model output when changing the value of K_{mp} by 10% vii
B.5 The model output when changing the value of S by 10% viii
B.6 The model output when changing the value of C_T by 10% viii
B.7 The model output when changing the value of E_T by 10% ix

Nomenclature

Metaphors from the theory of evolution are used to name some aspects of the algorithm, as is the convention with evolutionary algorithms. These terms are defined below. In addition, common terms in parallel programming are explained. The symbols in the symbol list are shown approximately in the order of appearance, with lines separating the symbols by section.

Evolutionary Metaphors used in the CMA-ES Algorithm

Generation: Iteration number.

Individual: A single sample in the parameter space, with a corresponding objective function value.

Population: The number of individuals in each generation.

Offspring: The population that is generated in a specific generation.

Fitness: The objective function value of an individual. Lower is better.

Recombination: Taking a weighted mean of the best individuals (by fitness) in each generation in order to generate better offspring.

Parent number: Number of individuals that are used for recombination.

Parallel Programming Terms

MPI: Message Passing Interface, a library standard for functions that perform communication between computation nodes (allowing for parallel programming). NMPI, an MPI library developed by the IT division at NTNU is used in this thesis.

Job: To run a parallel program on a computer cluster, the script has to be queued by specifying the number of ranks and cores to be used, in addition to the maximum time it should run. If the runtime exceeds the specified limit, the program is terminated.

Rank: The number of threads (or workers) that work in parallel: If a program uses eight ranks, then eight tasks can be performed simultaneously. One rank can contain multiple processor cores.

Core: A processor core is a single processing unit. Using multiple cores in one rank can speed up some processes.

Vector and Matrix Notation

Vectors and matrices are denoted by bold, upright letters. Vectors are lower case and matrices are upper case. As examples, $\mathbf{A}$ is a matrix, $\mathbf{a}$ is a vector and a is a scalar. $\mathbf{I}$ is always the identity matrix.

Abbreviations

CMA-ES - Covariance matrix adaptation evolution strategy

ODE - Ordinary differential equation

DAE - Differential algebraic equation

MPI - Message Passing Interface

HPC - High-performance computing

RNAP - RNA polymerase

APC - Anaphase promoting complex

PY - Productive yield

MES - Model evaluation sequence(s)

Symbol	Description	Units
N	Dimensionality of the optimization problem; number of parameters to be estimated	-
λ	Population; number of samples taken of the parameter space each generation	-
μ	Parent number; number of samples taken into consideration for recombination	-
$\mathbf{m}^{(g)}$	The mean of the multivariate normal distribution creating the offspring for generation g	-
$\mathbf{C}^{(g)}$	The covariance matrix of the multivariate normal distribution in generation g	-
$\mathbf{B}$	An orthogonal matrix consisting of the eigenvectors of $\mathbf{C}$	-
$\mathbf{D}$	A diagonal matrix consisting of the square roots of the eigenvalues of $\mathbf{C}$	-
$\sigma^{(g)}$	The mutation strength (also known as step length) in generation g	-
$\mathbf{x}_i^{(g)}$	Individual number i in generation g	-
$\mathbf{y}_i^{(g)}$	Mutation vector number i in generation g	-
$\mathbf{y}_m^{(g)}$	The (recombinated) mean mutation vector for generation g	-
$\mathbf{y}_{i:\lambda}^{(g)}$	The i-th best mutation when ranking by objective function value	-
w_i	Recombination weight number i	-
μ_{eff}	*Variance effective selection mass*, a (self-)tuning parameter	-
$\mathbf{p}_\sigma^{(g)}$	Evolution path used for adapting the step length for generation g	-
c_σ	Learning rate for $\mathbf{p}_\sigma$	-
$\mathbf{p}_c^{(g)}$	Evolution path used for adapting the covariance matrix for generation g	-
c_c	Learning rate for $\mathbf{p}_c$	-
c_1	The learning rate for the rank-one-update of $\mathbf{C}$	-
c_μ	The learning rate for the rank-μ-update of $\mathbf{C}$	-
d_σ	Damping parameter for the $\sigma^{(g)}$-update	-
λ_{seq}	Population size for sequential CMA-ES; used as minimal population size for parallel CMA-ES	-
R	Concentration of mitosis promoting factor	nM
E	Concentration of anaphase promoting complex (APC)	nM
E_P	Concentration of phospholyrated APC	nM
E_T	Total APC concentration ($E + E_P$)	nM
C	Concentration of Cdc20	nM
X	Concentration of a complex of APC and Cdc20	nM
k_1	First order rate constant	min^{-1}
k_2	Second order rate constant	$\text{nM}^{-1}\text{min}^{-1}$
k_3	Third order rate constant	min^{-1}

k_4	Fourth order rate constant	min^{-1}
k_5	Fifth order rate constant	min^{-1}
K_{mk}	Michaelis constant	$\text{n;}^{-1}\text{min}^{-1}$
K_{mp}	Michaelis constant	nM
K_d	Equilibrium constant	nM
S	Signal	nM
Q	Photosphase concentration	nM
C_T	Total Cdc20 concentration	nM
x	Position of the RNAP, measured in nucleotide lengths, L_n	L_n
v	Velocity	$\text{L}_\text{n}/\text{s}$
F_{TL}	The force from the trigger loop on the RNAP	$\text{L}_\text{n}/\text{s}^2$
F_{DNA}	The force applied on the RNAP when splitting the DNA base pairs	$\text{L}_\text{n}/\text{s}^2$
F_S	Stability	$\text{L}_\text{n}/\text{s}^2$
F_C	Force from the catalysis on the RNAP	$\text{L}_\text{n}/\text{s}^2$
t	Time	s
$k_{TL,A}$	Amplitude of the force from the trigger loop	$\text{L}_\text{n}/\text{s}^2$
$k_{TL,\omega}$	Frequency of the force from the trigger loop	s^{-1}
ϕ	Phase of the force from the trigger loop	-
k_C	Scaling parameter for F_C	-
k_{DNA}	Scaling parameter for F_{DNA}	-
k_S	Scaling parameter for F_S	-
PY	Productive yield	-

Chapter 1

Introduction

Understanding the behavior and interactions of living systems is desirable not only in academia, but in industry as well. For biotechnological industry, such insight is the basis of production. In order to optimize the production processes, knowledge is required on how the system reacts to different conditions.

The field of systems biology uses mathematical and computational models as a means for simulation. The models are usually built with a bottom-up approach where the underlying kinetics of the system are described mathematically [1]. These descriptions can be in the form of ordinary- or partial differential equations, differential algebraic equations, networks[1] and more. The models may contain random elements that emulate a stochastic behavior in the real system, or they may be deterministic. What the models have in common is that they all include parameters (for example kinetic- or rate constants) that affect the model behavior. Some parameters can be found in literature, but most parameter values are unknown, and need to be adjusted until the model can represent the real system adequately [3].

This thesis aims to parallelize a global optimization method. The method was implemented during a specialization project in the the fall semester of 2017. The goal is to develop a parallel optimization method that can be used as part of a toolbox for parameter estimation problems in systems biology.

1.1 Problem Description

Parameter estimation problems are a form of optimization problems. The goal is to minimize the model error, which is the difference between the behavior predicted by the model and the actual behavior of the physical system, as a function of the parameters. Usually, the objective function is on the form of a (weighted) sum of squared errors (SSE):

[1] Like Boolean networks or Petri Nets [2].

$$J(\mathbf{p}) = \sum_{i=1}^{N} \mathbf{e}_i(\mathbf{p})^T \mathbf{W} \mathbf{e}_i(\mathbf{p})$$

Where $\mathbf{p}$ is a vector of parameters, N is the number of experiments, $\mathbf{e}$ is the model error and $\mathbf{W}$ is a weighting matrix (for unweighted SSE, $\mathbf{W} = \mathbf{I}$). In order to find the $\mathbf{e}$ vector, the model needs to be solved using the parameters $\mathbf{p}$ to get the model outputs. The objective function is thus itself a function of the model, which makes for complex optimization problems: nonlinear, multimodal and non-smooth [2, 4]. This makes local optimization methods inefficient or even unusable, as they converge to the closest optimum, with no consideration of how good the optimum is. An alternative is a so-called multistart method, where multiple local solvers are run from initial points in a grid covering the search space, but it becomes inefficient as the amount of parameters increase.[2] Parameter estimation on a complex model can be a nontrivial problem in itself. In these cases, global optimization methods are an alternative.

Global optimization methods aim to find the lowest minimum of the entire search space, in problems where several local minima are available. The task is substantially more difficult than local optimization, and multiple methods are used without any clear "best" strategy or method.

Deterministic optimization methods such as Branch-and-Bound or Cutting Plane methods can guarantee convergence to the global optimum, but are limited in use and can be impractically slow with increasing dimensionalities [4].

On the other hand, stochastic optimization methods[3] such as evolutionary algorithms or swarm algorithms cannot guarantee that the solution found is the global optimum, but they are designed to efficiently navigate the search space, and are applicable to most optimization problems [3, 5, 6]. Due to the robustness and efficiency, a stochastic global method is used in this thesis.

1.2 Framework

The global optimization method implemented in this thesis is the covariance matrix adaptation evolution strategy (CMA-ES). It is an evolution strategy which is a subgroup of evolutionary algorithms, loosely inspired by the natural optimization process of evolution. The thesis considers the parallelization of the CMA-ES in order to run the method on multiple processor cores on computer clusters, to greatly reduce the required runtime.

The CMA-ES is, like other metaheuristic methods, a black box optimization method. The only information required about the optimization problem is obtained through sampling the objective function value in single points in the search space, and neither gradients nor hessians are used. The method is intended for continuous search spaces, although a mixed-integer version of the method exists [7]. Apart from this, no assumptions are needed about the objective function. It does not need to be convex, differentiable nor smooth [8].

This versatility means that the CMA-ES could be used for parameter estimation on any type of model, but in the thesis it will be tested on two parameter estimation prob-

[2] Known as Bellman's curse of dimensionality.

[3] Also known as metaheuristic methods.

lems from systems biology: One model with known parameters, consisting of an ordinary differential equation (ODE) and a hybrid model based on a differential algebraic equation (DAE). Although two real models are evaluated, model development (apart from parameter estimation) is not within the scope of this thesis.

An ODE is a system containing one or more equations in the form of

$$\dot{\mathbf{x}} = f(\mathbf{x}, t, \mathbf{p})$$

where $\mathbf{x}$ is a state vector, t is the time and $\mathbf{p}$ is a parameter vector that needs to be estimated.

A DAE includes additional algebraic equations, which means they are in the form of:

$$\begin{aligned} \dot{\mathbf{x}} &= f(\mathbf{x}, t, \mathbf{p}) \\ 0 &= g(\mathbf{x}, t, \mathbf{p}) \end{aligned}$$

where $g(\mathbf{x}, t, \mathbf{p})$ is an algebraic function.

The DAE model used in this case consists of multiple DAEs solved in sequence, with a random element in the initialization of each equation.

All parallel experiments are run on the Vilje cluster,[4] a system of 1404 computation nodes each consisting of two eight-core 2.6 GHz processors. The parallelization is performed using MPI (Message Passing Interface) calls for communication between computation nodes. This is implemented for MATLAB as mex compiled C programs by the HPC group at the NTNU-IT department.

1.3 Structure of the Thesis

The thesis is structured as follows: In Chapter 2, global optimization is explained further, and the mechanics of the CMA-ES algorithm is presented. In Chapter 3, parallel programming is introduced. In Chapter 4, the parallelized algorithm is used on a delayed negative feedback model with known parameters, and is compared with parameter estimation methods in MATLAB. In Chapter 5, the algorithm is used on a stochastic hybrid model of RNA polymerase backtracking, with unknown parameters. As a part of this problem, two additional features are developed and added to the CMA-ES: One allowing for an intensified search on stochastic models, and one allowing for multiple instances of the parallel CMA-ES to be run simultaneously from the same script. The results of the tests are further discussed in Chapter 6, and the concluding remarks are in Chapter 7.

[4] `https://www.hpc.ntnu.no/display/hpc/About+Vilje`

Chapter 2

Global Optimization

Global optimization is a field of optimization that deals with multimodal optimization problems, that is, problems with multiple local minima. This is in contrast to local optimization, which concerns itself with problems with a single minimum, which are significantly easier to solve. Unlike in local optimization, global optimization problems have no standard problem formulations (like that of linear programming, quadratic programming etc.) with universally accepted accompanying solution strategies, and no central idea comparable to that of local descent [9]. It is simple to prove local optimality for a given point in the feasible region, but determining if it is the global minimum on a multimodal problem is very difficult and time consuming. Because there is little consensus in the field, multiple methods and strategies are used, with overlapping applicabilities.

Neumayer (2004) [10] proposed four categories of global optimization methods, based on the rigorousness of the method.

- **Incomplete** methods, which are heuristic methods with good chances of finding the global optimum in a reasonable time. They are, however, not guaranteed to find the global optimum, and might return a local one instead.

- **Asymptotically complete** methods, which can guarantee to find the global minimum if allowed to run indefinitely long, but is unable to declare with certainty that a given solution is the global optimum.

- **Complete** methods, which can guarantee to find the global optimum after indefinitely long time and assuming exact computations (no rounding errors). In addition, complete methods can tell (within finite time) that an approximately global solution is found, within predefined tolerances.

- **Rigorous** methods, which are similar to complete methods, but able to find the global solution even with rounding errors.

Complete and rigorous methods (often called deterministic methods) are computationally expensive and not always applicable on every kind of problem, depending on the information available. For more difficult global optimization problems, with complex objective functions or high dimensionalities (many parameters), incomplete methods are a better choice.

Incomplete methods (more often called stochastic or metaheuristic methods) are optimization methods that use heuristics in order to efficiently explore the search space (*diversification*), without losing the capability to commit to improving a local (hopefully global) solution (*intensification*).[1] These methods are robust in regards to noisy, irregular objective functions,[2] and high dimensionalities. One such method is the covariance matrix adaptation evolution strategy (CMA-ES).

Stochastic methods have been applied successfully on multiple parameter estimation problems in systems biology [2].

2.1 The Covariance Matrix Adaptation Evolution Strategy

The covariance matrix adaptation evolution strategy (CMA-ES) is an evolution strategy, which is a subset of evolutionary algorithms.

Evolution strategies is a group of flexible and effective global optimization methods, and is previously used with success in parameter estimation problems [2]. The CMA-ES method specifically is a state-of-the-art global optimization method that has proven to perform well on real-world search problems [11, 12, 13].

2.1.1 Description

The CMA-ES is a black box optimization method. The only information about the optimization problem is obtained through *selection*, evaluating the response from the objective function at specific points. This makes the method insensitive to the details of the objective function it evaluates, as the only interaction with the function is presenting a set of variables and looking at what comes out. On the other hand, the method cannot make use of additional information like gradients in order to solve the problem faster.

Like other stochastic global optimization methods, the CMA-ES has no way to verify whether a candidate solution is the global solution or not. However, as the objective function in a model calibration problem is the error between model outputs and the experimental data, the theoretical minimum has a value of zero. This presumes that the mathematical model accounts for every factor that affects the physical system (including noise in the measurements) and is as a result not possible in practice. On the other hand it does mean that the objective function value directly represents how well the model is fitted to the

[1] This is also known as *exploration* and *exploitation*.

[2] A "noisy" objective function is subject to random noise in the function evaluations, meaning that evaluating the same solution twice will return different objective function values.

physical system. It is up to the user whether to accept the solution, run the program again (perhaps with different initial conditions) or revisit the model itself. In any case, global optimization methods should always be run multiple times on problems with unknown optimal solutions.

In the following section, the basic mechanics of how the CMA-ES algorithm works is explained and the nomenclature of evolution strategies are introduced. The exact mathematical operations are presented afterwards.

The CMA-ES is an iterative method. Following the naming conventions, each iteration is called a *generation*, and the variables that are unique for a generation are marked by (g). In each generation, a number of random points (a *population*) are sampled from the parameter space. Each of the random points $\mathbf{x}_i^{(g+1)}$ is a candidate solution (an *individual*), and the individuals are generated around a specific point, $\mathbf{m}^{(g)}$ (the *mean*). Each of the individuals are evaluated (by evaluating the objective function) and then ranked in order of *fitness*, which is the objective function value for that individual. The mean is then moved to a more promising area, as indicated by to the best individuals of the generation ($\mathbf{m}^{(g)} \rightarrow \mathbf{m}^{(g+1)}$). Then, a new generation is generated around the new mean.

Unique to the CMA-ES is that the probability distribution for the randomly sampled points is changed in order to prioritize exploration in areas that are of interest (having better fitness) by adapting the covariance matrix, $\mathbf{C}$ of the distribution. One generation in the CMA-ES is described below.

0. **Initialization.** This is only done at the beginning of the algorithm, and is not part of the iterative process. Internal strategy parameters are initialized as a function of the dimensionality of the problem, N, and the population size, λ. The population size should be at minimum

$$\lambda \geq 4 + \lfloor 3 \log N \rfloor$$

1. **Generating offspring.** Select λ points from a normal distribution around the current mean according to:

$$\mathbf{x}_i^{(g+1)} \sim \sigma^{(g)} \mathcal{N}(\mathbf{m}^{(g)}, \mathbf{C}^{(g)}), \quad \text{for } i = 1, 2, ..., \lambda \tag{2.1}$$

The $\sim$ denotes that the statistical distribution is the same on the right- and the left-hand side of the equation. The mutation strength, $\sigma^{(g)}$ is a factor that scales the variation of the normal distribution (effectively, search range). $\mathbf{C}^{(g)}$ is the covariance matrix. It can be eigendecomposited to

$$\mathbf{C} = \mathbf{B}\mathbf{D}^2\mathbf{B}^{\mathrm{T}}$$

where $\mathbf{B}$ is a matrix of eigenvectors and $\mathbf{D}$ is a diagonal matrix of the square roots of the eigenvalues of $\mathbf{C}^{(g)}$

The actual implementation of Equation (2.1) is a bit indirect, as it first calculates the

mutation vectors $\mathbf{y}_i^{(g+1)}$, which are used further in the algorithm:

$$\mathbf{z}_i^{(g+1)} \sim \mathcal{N}(0, \mathbf{I})$$

$$\mathbf{y}_i^{(g+1)} = \mathbf{B}\mathbf{D}\mathbf{z}^{(g+1)} \qquad \sim \mathcal{N}(0, \mathbf{C}^{(g)}) \tag{2.2}$$

$$\mathbf{x}_i^{(g+1)} = \mathbf{m}^{(g)} + \sigma^{(g)}\mathbf{y}_i^{(g+1)} \qquad \sim \sigma^{(g)}\mathcal{N}(\mathbf{m}^{(g)}, \mathbf{C}^{(g)}) \tag{2.3}$$

2. **Selection and recombination.** Evaluate the fitness function value of each offspring, and put them in order from best to worst. Move the mean by taking a weighted mean of the mutation vectors, giving higher weights to the mutations corresponding with the offspring with best fitness. Only the best μ individuals are considered for the recombination ($\mu < \lambda$):

$$\mathbf{y}_m^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \tag{2.4}$$

where $\mathbf{y}_m^{(g+1)}$ is the weighted mean mutation vector, w_i is weight number i and $\mathbf{y}_{i:\lambda}^{(g+1)}$ is the i-th best ranked individual.

The mean is then moved by

$$\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + \sigma^{(g)}\mathbf{y}_m^{(g+1)} \tag{2.5}$$

This is equivalent to just setting $\mathbf{m}^{(g)}$ equal to the weighted mean of the sampled points ($\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)}$), apart from when incorporating negative weights, which is an alternative version of the method discussed in section 2.3.

3. **Update the evolution path.** The evolution path is a (exponentially smoothed) history of the preceding movement of the mean. When the mean has moved in one direction for multiple generations, the evolution path will bias the search of the next generations to look in the same direction. Two different kinds of evolution paths are calculated: The evolution path used for updating the step length ($\mathbf{p}_\sigma^{(g+1)}$) and the evolution path used for updating the covariance matrix ($\mathbf{p}_c^{(g+1)}$).

$$\mathbf{p}_\sigma^{(g+1)} = (1 - c_\sigma)\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\,\mathbf{C}^{(g)^{-\frac{1}{2}}}\mathbf{y}_m^{(g+1)} \tag{2.6}$$

$$\mathbf{p}_c^{(g+1)} = (1 - c_c)\mathbf{p}_c^{(g)} + h_\sigma\sqrt{c_c(2 - c_c)\mu_{eff}}\,\mathbf{y}_m^{(g+1)} \tag{2.7}$$

c_σ and c_c are strategy parameters (*learning rates*) deciding the rate of the exponential smoothing. μ_{eff} is also an internal strategy parameter.

The Heaviside function h_σ in Equation (2.7) prevents $\mathbf{p}_c$ from growing in cases where the mutation strength σ increases rapidly (typically in a problem where the initial mutation strength is set too low), which would cause the axes in $\mathbf{C}$ to grow too fast:

$$h_\sigma = \begin{cases} 1 & \text{if } \frac{||\mathbf{p}_\sigma^{(g+1)}||}{\sqrt{1-(1-c_\sigma)^{2(g+1)}}} < (1.4 + \frac{2}{n+1})\mathrm{E}||\mathcal{N}(0, \mathbf{I})|| \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

Equation (2.8) requires $\mathbf{p}_\sigma^{(g+1)}$, which means Equation (2.6) needs to be calculated first.

4. **Adapt the covariance matrix.** Like the evolution paths, $\mathbf{C}$ is exponentially smoothed. $\mathbf{C}$ is updated Using rank-μ-update and rank-one-update (from Hansen (2016) [14]), which represents the historical movement of the mean and the selection of the fittest individuals in the current generation, respectively.

$$\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu)\mathbf{C}^{(g)} + c_1 \left(\mathbf{p}_c^{(g+1)} {\mathbf{p}_c^{(g+1)}}^{\mathrm{T}} + \delta(h_\sigma)\mathbf{C}^{(g)} \right) + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} {\mathbf{y}_{i:\lambda}^{(g+1)}}^{\mathrm{T}} \quad (2.9)$$

   c_1 and c_μ are the learning rates for the rank-one-update and rank-μ-update, respectively. The $\delta(h_\sigma)\mathbf{C}$-term is a minor correction term for the rare cases where $h_\sigma = 0$.

$$\delta(h_\sigma) = (1 - h_\sigma)(2 - c_c)c_c \quad (2.10)$$

5. **Changing the mutation strength.** If the length of the evolution path $\mathbf{p}_\sigma^{(g+1)}$ is longer than the expected length of a vector in a system where there is no systematic movement of the mean ($\mathrm{E}||\mathcal{N}(0, \mathbf{I})||$),[3] then σ is increased. If it has moved less, σ is decreased.

$$\sigma^{(g+1)} = \sigma^{(g)} \exp\left(\frac{c_\sigma}{d_\sigma} \left(\frac{||\mathbf{p}_\sigma^{(g+1)}||}{\mathrm{E}||\mathcal{N}(0, \mathbf{I})||} - 1 \right) \right) \quad (2.11)$$

   d_σ is a dampening factor, another internal strategy parameter.

The way *diversification* and *intensification* is handled in the CMA-ES largely by adapting $\sigma^{(g)}$. At the beginning, the mutation strength is large, which allows for sampling of a large part of the search space. When the method reaches a promising area and the best individuals are generated close to $\mathbf{m}^{(g)}$, $\sigma^{(g)}$ is reduced. The search space is made smaller until only points in the neighborhood of the minimum is evaluated.

One of the strengths of the CMA-ES is the fact that it has very few inputs that need to be set by the user [12, 13]. Only two user inputs can affect the performance significantly: The initial parameter guess ($\mathbf{m}^{(0)}$) and the initial mutation strength ($\sigma^{(0)}$). Giving bad values for these can slow down the convergence time or even make it impossible to find the global minimum. Both of these inputs are unproblematic to set if an approximate search space is known: The initial guess can be picked randomly from the search space, and the step size can be set proportional to the size of the space. This is discussed in Section 2.2.

Two alternative (but compatible) uses of recombination weights can be used by the CMA-ES algorithm. Both work by including negative weights when calculating the weighted means, meaning that all λ offspring can be used to calculate the new mean and the covariance matrix, instead of just a subset $\mu < \lambda$. These implementations are discussed in Section 2.3 and 2.4.

[3] In other words, if the mean has moved further than it would in a case where all surrounding solutions have the same fitness.

The termination criteria for the method are implemented from Auger and Hansen (2005) [15], in which a restart functionality is introduced. The method is allowed to restart a set amount of times, increasing the population size each time. The restart functionality is intended for the sequential implementation of CMA-ES, and is unused in the parallelized version. This is described in Section 2.5.

2.2 Choosing Initial Mutation Strength

One of the factors that can greatly impact the performance of the CMA-ES method is the selection of the initial mutation strength ($\sigma^{(0)}$, also known as the step length). While $\sigma^{(g)}$ is adapted over generations, selecting too low of a value can make the method converge to a local minimum before the $\sigma^{(g)}$ can reach a size sufficient to be able to search through the entire search space. This happens if the initial mutation strength is so small that no individuals are generated outside of a local minimum.

A good value for the initial mutation strength is $0.3[a - b]$ where a and b are the approximate maximum and minimum values of the search space, respectively [14]. The fact that a and b are scalars implies that having parameters with greatly differing search space sizes (differing orders of magnitude) can negatively affect the performance. In those cases, it would be beneficial to rescale the parameters before starting the optimization.

2.3 Negative Recombination Weights for Step Adaptation

In principle, there is no reason why the weighted mean in Equation (2.4) needs to have weights that add up to 1. The difference between Equation (2.4) and $\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)}$ is that having weights that sum to more or less than 1 (for instance, if all the weights were multiplied by 2) will offset the directionality of the latter, while only changing the step length of the former. Consequently the sum in Equation (2.4) can be expanded to include all λ offspring:

$$\mathbf{y}_m^{(g+1)} = \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \tag{2.12}$$

Where the weights w_i for $i > \mu$ are negative, and decreasing as i increases. The negative weights are smaller (in absolute value) than the positive weights.

The purpose is to improve the recombination to not only push $\mathbf{m}^{(g+1)}$ *towards* areas with better fitness, but also to push it *away* from poor ones. This increases the amount of information used, which will hopefully[4] improve the movement of the mean. Using negative recombination weights this way is done in other kinds of evolution strategies, but is largely unexplored in CMA-ES [17].

The functionality is on by default, as it was found to provide a slight (but significant) increase to convergence rate during the specialization project. However, the positive impact might be lessened by using larger populations.

[4]But not necessarily. As mentioned by Arnold, D.V. (2008) [16] on this issue: "[...] the opposite of a bad direction is not always a good direction".

2.4 Negative Recombination Weights for Covariance Adaptation

Negative covariance adaptation weights is impemented according to the tutorial by Hansen, N (2016) [14]. The logic is the same as in Section 2.3, but applied to the weighted mean that is used in the covariance adaptation (Equation (2.9)), specifically the term

$$c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)^{\mathrm{T}}}$$

which is expanded to sum up to λ.

Extra care is needed to ensure that $\mathbf{C}^{(g+1)}$ stays positive definite. This is done by scaling the negative weights before using them to calculate $\mathbf{C}^{(g+1)}$. In each generation, a set of temporary weights w_i° are calculated according to:

$$w_i^\circ = \begin{cases} w_i & \text{if } w_i \geq 0 \\ w_i \cdot \frac{N}{||\mathbf{C}^{-\frac{1}{2}} \mathbf{y}_{i:\lambda}||^2} & \text{if } w_i < 0 \end{cases} \quad \text{for } i = 1, 2, ..., \lambda \tag{2.13}$$

This set of weights is then used to adapt $\mathbf{C}^{(g+1)}$.

The functionality is on by default, as it (as in Section 2.3) improves the convergence rate.

2.5 Termination Criteria

Termination criteria are taken from Auger and Hansen (2005) [15]. If any of the following events happen, the search is ended:

- If the best individuals in the last $10 + \lceil \frac{30n}{\lambda} \rceil$ generations have the exact same fitness value.
- If the range of fitness values for all individuals in the last generation as well as the best individuals of the last $10 + \lceil \frac{30n}{\lambda} \rceil$ generations is below a threshold `tolfun`.
- If the search space gets too small, below a certain value `tolX`.
- If one of the principal axis directions (the diagonal) of $\mathbf{C}^{(g)}$ is zero.
- If the condition number of $\mathbf{C}^{(g)}$ exceeds a certain value.

2.6 Boundary and Constraint Handling

The CMA-ES implementation does not contain any boundary handling by itself. Any boundaries or constraints are included in the model evaluation function called by the

method.[5] Constraints in parameter estimation problems are usually simple linear constraints of the type

$$\text{Lower Bound} < \mathbf{p} < \text{Higher Bound}$$

where $\mathbf{p}$ is a vector containing the parameters. In these cases, the boundaries can easily be implemented as a check before the model function is called:

if (ANY $\mathbf{p} <$ LoBound) **or** (ANY $\mathbf{p} >$ HiBound) **then**
  fitness $= a + ||\mathbf{p} - \mathbf{p}_{feasible}||$
  return fitness
else
  perform the model evaluation and calculate fitness as usual
end if

Where a is an integer value that is ensured to be larger than the worst fitness value of a feasible solution, and $||\mathbf{p} - \mathbf{p}_{feasible}||$ is the distance from the feasible parameter space, calculated as

$$||\mathbf{p} - \mathbf{p}_{feasible}|| = \sum_{p<\text{LoBound}} (p - \text{LoBound})^2 + \sum_{p>\text{HiBound}} (p - \text{HiBound})^2$$

Where p is an individual element in the vector $\mathbf{p}$.

This causes the individuals that are closest to the feasible area to be rated better than the ones further away, but still worse than any feasible solution. Recall from 2.1.1 that the magnitude of the differences in fitness values is unimportant: The CMA-ES algorithm only looks at the ranking[6] of the individuals, not by *how much* better one solution is. Therefore, a is usually set to 10^4, 10^5 or something similar. In a worst-case scenario, it can be set to infinity, but as a result ranking the infeasible points is no longer possible.

The role of constraints in parameter estimation problems is in this thesis assumed to be to help the optimization method to converge to the correct solution, by preventing the method from looking into unproductive parts of the parameter space.

Other problem specific constraints may be required, and can all be handled in the same way, by returning high fitness values for conditions that are unwanted. These conditions could be dependent on parameter values, model outputs or anything else measurable in the wrapper function.

[5] Typically, a wrapper function is called. The wrapper function calls the model, and calculates the fitness (by comparing the outputs of the model to the experimental data, which is included in the wrapper function).

[6] n-th best vs (n+1)-th best etc.

Chapter 3

Parallelization

Parallel programming allows for running parts of a program simultaneously (in parallel) on multiple processors, and is therefore a useful tool for large, time-consuming computational tasks that can be divided into smaller tasks that can be performed independently of each other. Depending on how much of the program that is parallelizable, the runtime of the program can be reduced to fractions of the original, non-parallel (*sequential*), runtime. Running a program in parallel is not as simple as running one sequentially, however: a parallel program requires communication between processors.

There are libraries with functions that perform such communication. MPI (Message Passing Interface) is a message passing library standard for communication between computation nodes [18]. It describes the syntax and routines for implementations of a message passing program in C, C++ or Fortran, standardized in order to make it easier to port existing code and run it on different machines [19]. In this thesis, a MPI library developed by the high-performance computing (HPC) group at the NTNU-IT department is used for parallel programming.

3.1 Structure of the Parallel Program

Running a program in parallel is performed as follows: The user specifies how many ranks (individual processes that run in parallel) that are required, and how many processors (cores) each rank should consist of. All ranks then run the same script, with the only difference between the ranks being that they are numbered: Each rank has a variable, `my_rank`, which is unique to that rank (the ranks are numbered from 0 and up). Thus, to make the ranks perform different tasks in parallel, the tasks need to be allocated to specific ranks. The function `parsteps` delegates the iterations of a for-loop between ranks. The `parsteps` function calculates which of the iterations each rank should perform, by comparing the number of iterations in the for-loop to the number of ranks available (`num_ranks`, a variable that is the same on all ranks).

After the ranks have performed the parallelized tasks, the variables containing the results of the tasks will be different between each rank. If those results are to be compared or

processed in aggregate, they need to be gathered on one rank. Typically this is the master rank, rank 0. After all ranks have sent the relevant data to the master rank (there is an MPI function for this), the master rank performs the relevant processing. If the other ranks need to know the results of these computations, the relevant data can be spread out from the master rank to all the others using another MPI function.

The theoretical speedup from parallelizing a program can be calculated from Amdahl's law [20], which is, paraphrased:

$$\text{Speedup} = \frac{1}{t_s + t_p/n}$$

where t_s is the fraction of the runtime that is used on sequential code (code that needs to be performed in order), t_p is the fraction of runtime that is used on parallelizable code (code that is independent of order and can thus be performed simultaneously as other parallelizable code) and n is the amount of processors available to perform in parallel. Some part of the program will always be sequential. Speedup is measured in how many times faster the program runs.

For Amdahl's law to be exact, the parallelizable tasks must have exactly the same runtime and the amount of parallelizable tasks must be divisible by n. In addition, the time spent on communication between processors must be negligible. In reality, the actual speedup will never reach the idealized number calculated by Amdahl's law.

Like the majority of global optimization algorithms, the CMA-ES algorithm is parallelizable. For each generation in the algorithm, a set of points (offspring) are sampled from the problem space around a certain point in parameter space, which is decided by the previous generation. Each individual in the new generation requires a model evaluation in order to determine its fitness. These model evaluations are only reliant on the parameters they take in, which are set for each individual. This means that after the offspring is generated, each individual can be evaluated independently, i.e. in parallel. When evaluating complex models with long runtimes, this is by far the most time intensive part of the program. There is a lot of time to gain by parallelizing this. Figures showing the flowcharts of the sequential vs the parallel programs are shown in figures 3.1 and 3.2.

Note that the parallelization is limited to each generation. The sequential code has to run after the parallel model evaluations have been performed before the next generation can start.

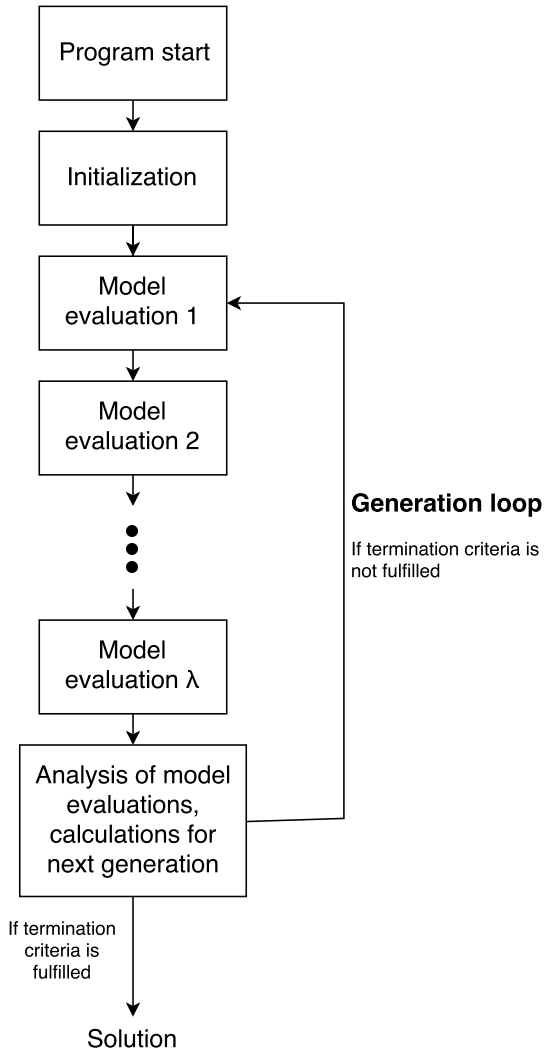

Figure 3.1: A flowchart describing the sequential CMA-ES algorithm, with a population size of λ.

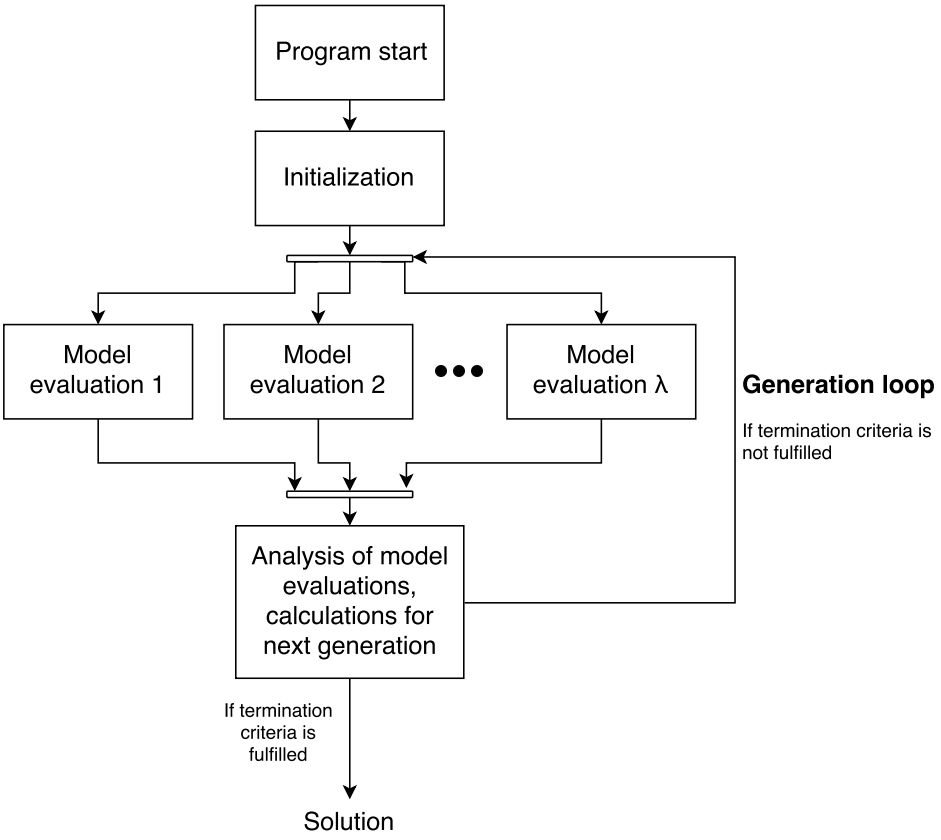

Figure 3.2: A flowchart describing the parallel CMA-ES algorithm with a population size of λ.

3.2 MPI Functions

The HPC group at the NTNU-IT department have provided an open source library of mex compiled MPI functions containing the standard MPI calls, called NMPI. In addition, they have created parallel functions that build upon NMPI, to allow for parallel programming without needing to know MPI, called Distributed MATLAB. Only a few of the functions are used in this project:

- `parsteps` allows a for-loop to be performed in parallel. This requires that each iteration in the loop is independent of the others. The role of the function in this program is to distribute the model evaluations between the different ranks, so that the evaluations can be performed in parallel as shown in Figure 3.2.

- `NMPI_Reduce` performs a reduction of a variable or array from all ranks. In this thesis, it is only used for one of its possible mathematical operations, i.e. addition. The values of an array are added together (element wise) from all ranks and the result is saved as an array on a specified rank (the *root*). This allows for collection of the results from each individual iteration in the parallel for-loop. The result of each iteration is saved to a corresponding element in a zeros-vector, which means that iteration number i will create a vector consisting of zeros except in element number i. When the vectors from all iterations are added together, the result is a vector containing the values of all iterations. The role of the function in this program is to collect the results after the parallel model evaluations.

- `NMPI_Bcast` spreads a variable or array from a specified rank (the *root*) to all other ranks. This is required when the master rank has performed the analysis of the model evaluations, in order to spread the relevant calculated variables to the other workers so that they can begin the next generation.

Combined, these allow for all communication between ranks required in the parallel CMA-ES method. The two NMPI functions above are also implemented in Distributed MATLAB, with the only difference (in use) being that the Distributed functions always uses the master rank as the root, while the NMPI allows the the root to be specified. In this project, the master rank is always used as the root, so the NMPI functions and Distributed functions for these operations have been used interchangeably during the coding.

3.3 Choice of Population Size

For the sequential implementation of the CMA-ES, a balance is needed when deciding the population size (λ):

- If λ is too large, the runtime of each generation becomes too long.

- If λ is too low, the estimation of the problem space becomes too inaccurate, reducing the convergence rate (measured as number of generations needed to converge).

A good value for λ in the sequential algorithm (λ_{seq}) is (from Hansen (2016) [14])

$$\lambda_{seq} = 4 + \lfloor 3 \ln N \rfloor \tag{3.1}$$

where N is the dimensionality of the problem. This is considered the minimum population size in order to get a sufficient sampling of the parameter space.

For parallelized implementations, the first concern (runtime of each generation due to the amount of function evaluations) gets reduced, as they can be performed simultaneously. If a program is running on exactly λ_{seq} ranks, the runtime of the model evaluations will be $\frac{1}{\lambda_{seq}}$ of that of the sequential runtime.

If the program is run on **more** than λ_{seq} ranks, the population size can be increased at no cost, improving the accuracy of the estimated gradients and hessians, and thus improving the convergence rate.

If the program is run on **less** than λ_{seq} ranks, each of the ranks will need to do more than one model evaluation in each generation, and the population will be set to

$$\begin{aligned} \min_{n} \quad & \lambda = n \cdot \text{Number of ranks} \\ \text{s.t.} \quad & \lambda \geq \lambda_{seq} \end{aligned} \tag{3.2}$$

where n is the amount of model evaluations each rank will need to perform in sequence for each generation.

The speedup in this case will be affected by both factors mentioned above: If the number of ranks is exactly $\frac{\lambda_{seq}}{n}$, then $\lambda = \lambda_{seq}$ and the runtime of the model evaluations will be $\frac{n}{\lambda_{seq}}$ of the sequential time. If the number of ranks is increased slightly, each rank must still do n evaluations, leaving the runtime for each generation unchanged. However, due to Equation (3.2), λ will be greater than λ_{seq} and the convergence rate will improve accordingly.

An example of how the effectiveness of the method is changed by increasing the amount of ranks is shown in Appendix A. It shows the diminishing returns from increasing the amount of ranks beyond λ_{seq}. The amount of ranks (and by extension, λ) used in this thesis is generally 32 unless otherwise specified.

Chapter 4

Testing on a Delayed Negative Feedback Model

The delayed negative feedback model is an oscillatory model common in systems biology. It was chosen as a trial model because the oscillations could present a challenge when calibrating the model. The model evaluated in this thesis is taken from System Modeling in Cellular Biology: From Concepts to Nuts and Bolts [21], and is a component of the cell cycle regulatory mechanism in eukaryotes:

$$\frac{dR}{dt} = k_1 S - k_2 X R \tag{4.1}$$

$$\frac{dE_P}{dt} = \frac{k_3 R(E_T - E_P)}{K_{mk} + E_T - E_P} - \frac{k_4 Q E_P}{K_{mp} + E_P} - k_5[E_P(C_T - X) - K_d X] \tag{4.2}$$

$$\frac{dX}{dt} = k_5[E_P(C_T - X) - K_d X] \tag{4.3}$$

Where R is concentration of mitosis promoting factor, E is conc. of anaphase promoting complex (APC), E_P is conc. of phosphorylated APC, E_T is total conc. of APC ($E + E_P$), C is conc. of Cdc20 and X is conc. of a complex of APC and Cdc20. The rest of the parameters are defined in Table 4.1.

The values for the parameters are the same as in System Modeling in Cellular Biology, with one important difference: According to the book the system is supposed to be oscillatory with $0.2 < S < 0.4$ (approximately). However, the model outputs did not reflect this. Multiplying S by a factor of 100 resulted in the correct behavior. The parameter values used are shown in Table 4.1, and the bifurcation diagram showing the values of S that result in a limit cycle is presented in Figure 4.1.

The fact that the system is only oscillatory for certain values of S can provide a challenge for parameter estimation when the initial guess starts outside of the area producing a limit cycle.

Table 4.1: The parameters used in the delayed negative feedback model, taken from System Modeling in Cellular Biology: From Concepts to Nuts and Bolts [21]. Note that the value for the signal, S, differs from the original, possibly due to an error in the book.

Parameter	Description	Value	Units
k_1	First order rate const.	1	min^{-1}
k_2	Second order rate const.	1	$\text{nM}^{-1}\text{min}^{-1}$
k_3	Third order rate const.	1	min^{-1}
k_4	Fourth order rate const.	1	min^{-1}
k_5	Fifth order rate const.	0.01	min^{-1}
K_{mk}	Michaelis const.	1	$\text{nM}^{-1}\text{min}^{-1}$
K_{mp}	Michaelis const.	1	nM
K_d	Equilibrium const.	50	nM
S	Signal	30	nM
Q	Photosphase conc.	100	nM
E_T	Total APC conc.	100	nM
C_T	Total Cdc20 conc.	1	nM

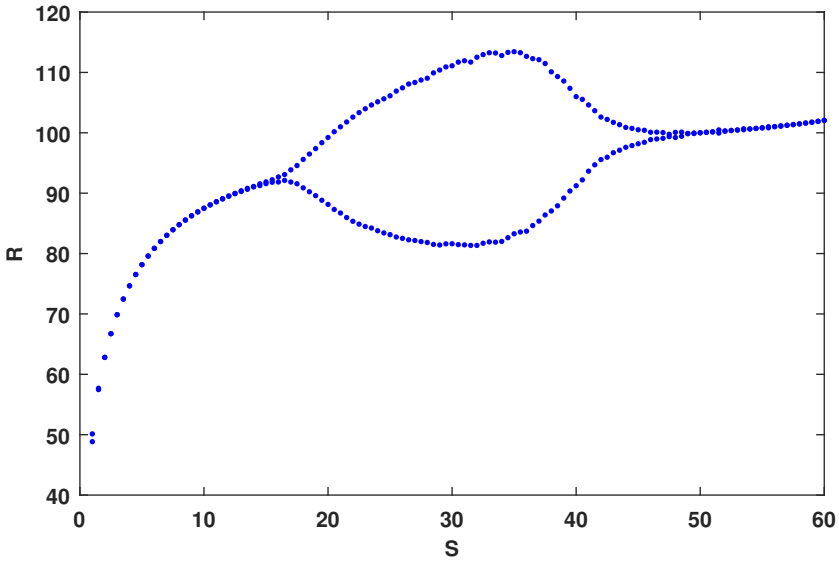

Figure 4.1: A bifurcation diagram showing the values of R (after the system has reached steasy state), as a function of S. Around $20 < S < 40$, the system oscillates, and the dots show both the minimum and maximum values of R in the stable limit cycle.

4.1 Experimental Conditions

Two types of experiments were performed using the delayed negative feedback model. The first experiment was a benchmark test, to check to what degree the parallelization lead to actual speedup on a model with relevant evaluation times. This is in contrast to the examples in Appendix A which only considers the reduction in evaluation sequences. The resulting impression of the speedup is exaggerated, as the CMA-ES contains sequential code as well.

In the speedup test, 30 generations of the CMA-ES algorithm[1] were run and timed 200 times on the model, for both a sequential and parallel implementation. The two implementations were run on the Vilje cluster, with one job using 1 rank, and another using 9 ranks,[2] with one core per rank. Each of the 200 runs started on a random initial point within the search space given by Table 4.2. The same random seeds were used in both jobs.

The parameter estimation problem was set up by first simulating the system with the correct parameters in order to create "experimental data" that could be used in the parameter estimation problem. 14 time steps were simulated, spaced by 3 seconds each. All three model outputs (R, X, E_p) were saved at each time step, resulting in a total of 42 data points, including the initial conditions ([0, 0, 0]).

After generating the experimental data, seven parameters were set to be unknown: k_3, k_4, K_{mk}, K_{mp}, S, C_T and E_T. The parameter estimation was subsequently performed 100 times, each time starting in a random point within the search space. The search space was limited within the boundaries shown in Table 4.2. In addition to the CMA-ES method, both methods utilized by the `lsqnonlin` function in MATLAB were used: Trust Region Reflective and Levenberg-Marquardt. Trust Region Reflective was used with the same bounds as specified in Table 4.2, while the Levenberg-Marquardt problem formulation was unbounded as this is a requirement for the Levenberg-Marquardt method.

The CMA-ES method was run on the Vilje cluster, using 32 ranks. The `lsqnonlin` methods were run on a desktop with an Intel(R) Core(TM) i5-6200U 2.30GHz quad core processor and 8192MB RAM.

Table 4.2: The search space in the delayed negative feedback problem.

Parameter	Lower bound	Upper bound
k_3	0	5
k_4	0	5
K_{mk}	0	5
K_{mp}	0	5
S	0	100
C_T	0	5
E_T	50	150

[1] The algorithm was hardcoded to terminate after 30 generations.

[2] The number 9 is chosen because the minimum population λ_{seq} for a 7 dimensional problem (which this is) is 9. This way, the population sizes are the same for the sequential and parallel jobs.

4.2 Results

The results of the speedup test is shown in Section 4.2.1, before the results of the actual parameter estimation is shown in Section 4.2.2. Cumulative plots are used to show how the runtimes and fitnesses are distributed over the 100 runs. When depicting runtimes, the y-axis shows the fraction of the 100 runs with a runtime less than the times at the x-axis. When depicting fitness, the y-axis shows the fraction of the runs which fitness less than the values at the x-axis.

4.2.1 Speedup

A cumulative plot showing the runtimes of 200 runs consisting of 30 iterations is shown in Figure 4.2. A few outliers with an order of magnitude 100-1000 times larger than the average means that a logarithmic x-axis is required. A plot excluding the 10% longest runtimes is shown in Figure 4.3.

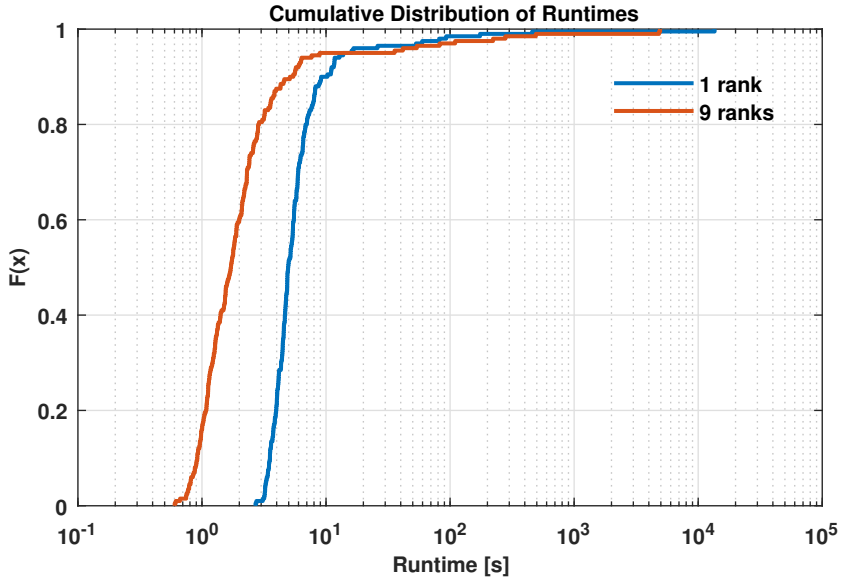

Figure 4.2: The cumulative distribution of the runtimes required to perform 30 generations of the CMA-ES algorithm on the delayed negative feedback model. Note that the x-axis is logarithmic.

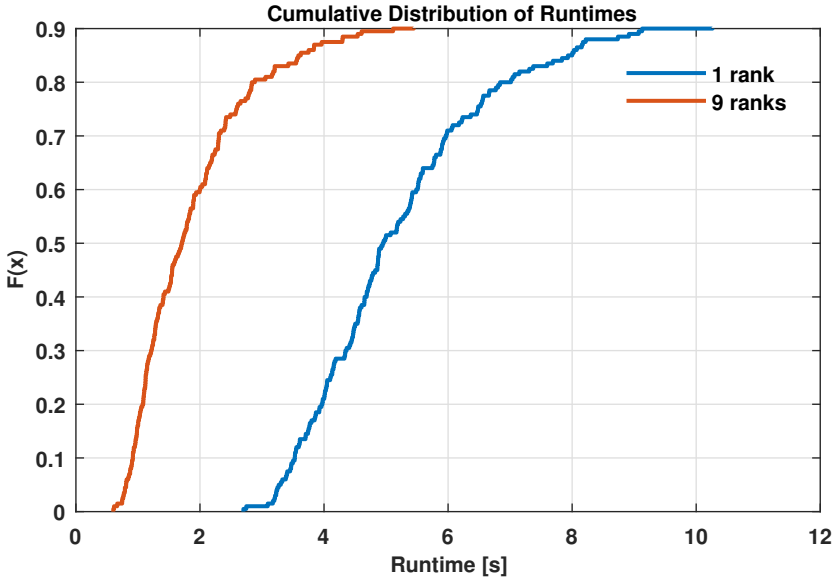

Figure 4.3: The cumulative distribution of the runtimes required to perform 30 generations of the CMA-ES method on the delayed negative feedback Model. This figure shows only the 90% fastest runs, in order to exclude outliers.

4.2.2 Parameter Estimation

In Figure 4.4, the cumulative distributions of the fitnesses resulting from 100 runs of each of the optimization methods are shown. The x-axis is logarithmic as there is a large difference in fitness between runs that managed to solve the problem and runs that did not. A zoomed-in version (with linear axes) only showing the lower fitnesses is shown in Figure 4.5. In Figure 4.6, the runtimes are shown, again with a logarithmic x-axis.

A box plot showing the distribution of parameter values found by solutions with good fitness (SSE < 0.5) is shown in Figure 4.7. The parameters are normalized (so that the correct value for all parameters would be 1) for visibility. The central marks of the box plots show the median values, with the boxes denoting the 25th and 75th percentiles. The whiskers show the most extreme values (excluding outliers). The $+$-signs mark outliers.

A sensitivity analysis was also performed, where each of the parameters were changed by 10% (from the correct values shown in Table 4.1), and the resulting increase in fitness was noted. The results of this is shown in Figure 4.8. Figures comparing the model output with and without the perturbed parameters are provided in Appendix B.

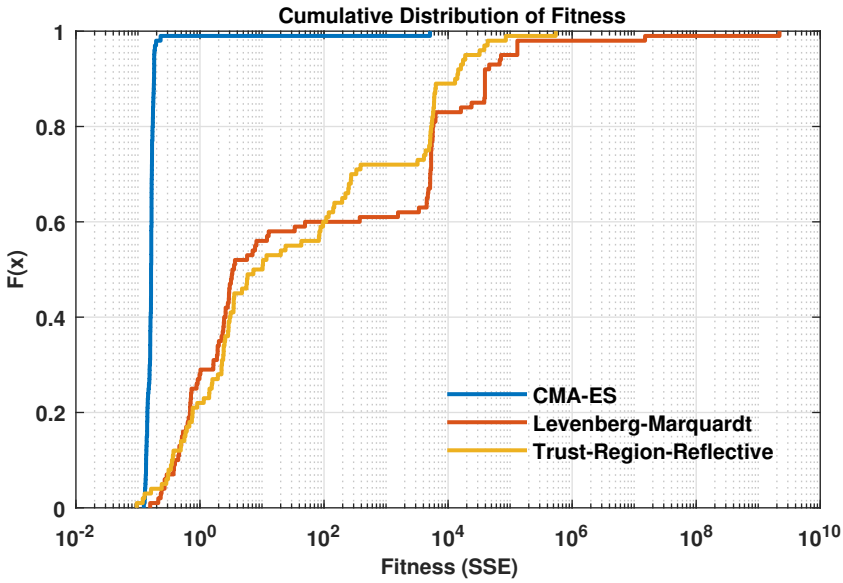

Figure 4.4: The cumulative distributions of fitness (measured by the sum of squared errors) for the three optimization methods. Note that the x-axis is logarithmic.

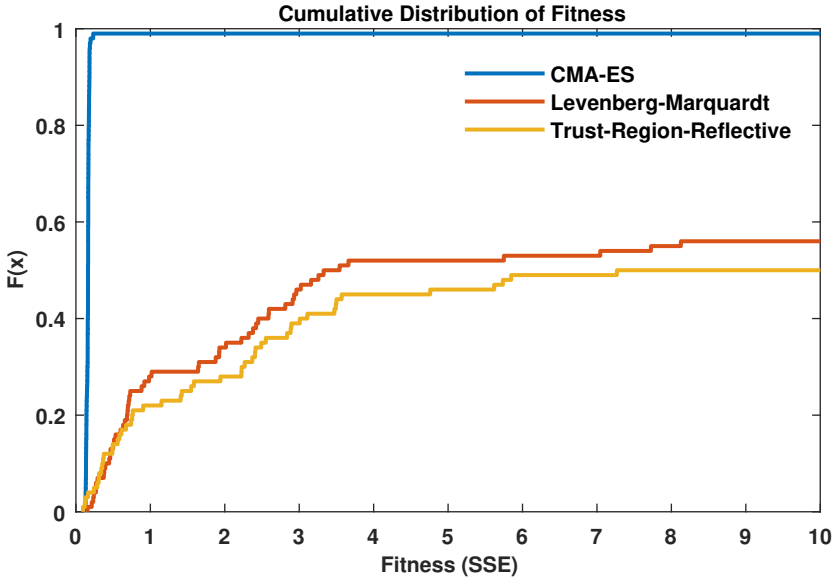

Figure 4.5: The cumulative distributions of fitness (measured by the sum of squared errors) for the three optimization methods. The lines do not reach unity as there are multiple solutions with fitnesses outside of the scope of the x-axis.

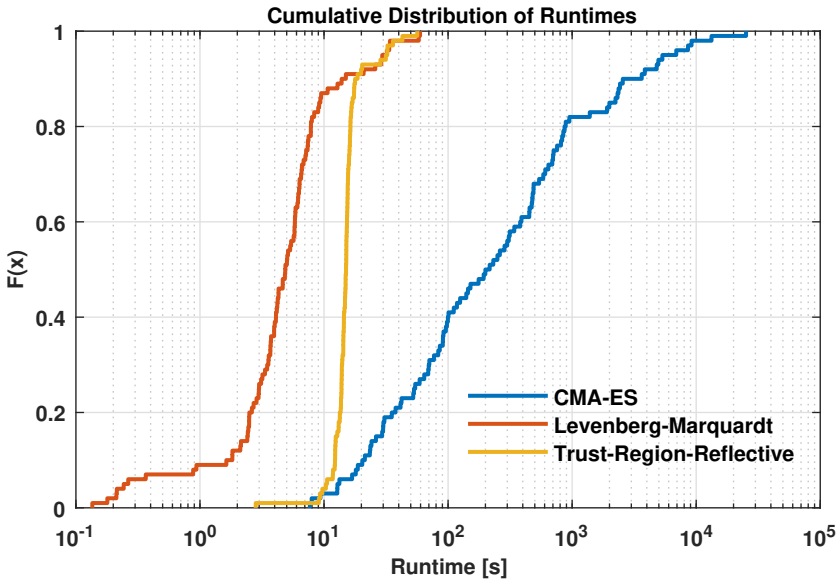

Figure 4.6: The cumulative distribution of runtimes for each of the three optimization methods. Note that the x-axis is logarithmic.

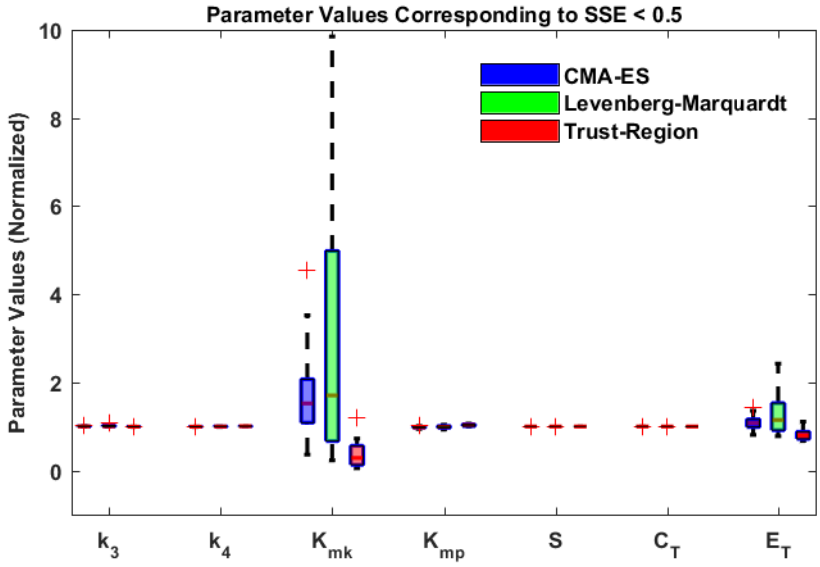

Figure 4.7: The parameters found using the three optimization methods, for fitness < 0.5. The parameters are normalized so that 1 is the correct value for all parameters. An outlier for K_{mk} found by the Levenberg-Marquardt method is outside of the scope, with a value of 18.

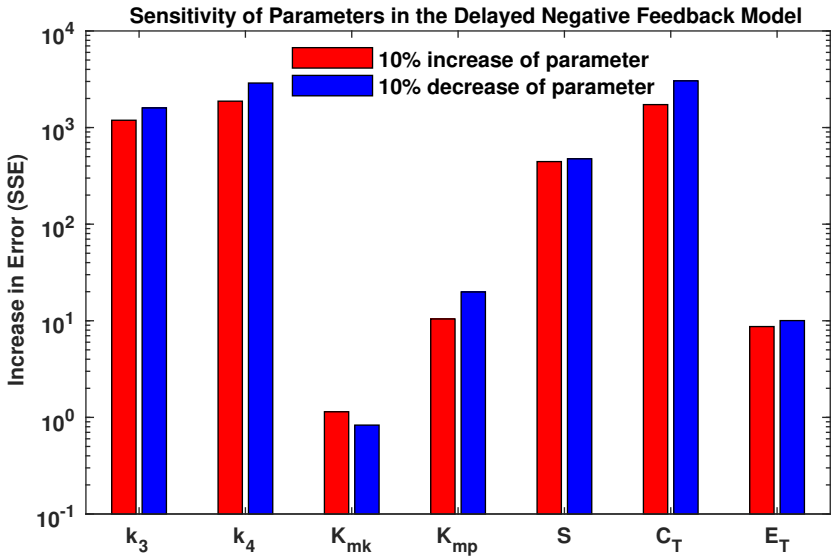

Figure 4.8: The resulting change in the error of the model (SSE) when changing each of the parameters by 10% from the correct value. Note that the y-axis is logarithmic: k_3, k_4, S and C_T are far more sensitive to change than K_{mk}, K_{mp} and E_T are.

4.3 Discussion

4.3.1 The Speedup Test

The results of the speedup test show a noticeable gain from parallelization, even though the model evaluations have a low runtime: 30 generations (which means $9 \cdot 30 = 270$ model evaluations in sequence for the sequential run) typically required less than 10 seconds. There were some massive outliers, however, where runs took 100-1000 times longer to finish. This indicates that there are some parameter combinations that massively slow down the ODE solver. These slow runs happened at different initial guesses between the sequential and parallel implementations. The same initial guess that resulted in an outlier for one implementation took an unremarkable amount of time for the other. This is because the CMA-ES algorithm is stochastic, which means that the two implementations differed between the exact parts of the search space they explored. This could have been avoided by rewriting the algorithm to use the same random seed each time.[3] Excluding the 10% slowest runs, the mean runtime of the parallelized version was 2.8 times faster than that of the sequential version.

The speedup test was performed with the assumption that the runtime of the 30 first generations are representative of the runtime of a generation in general. This assumption is probably acceptable (as long as outliers are excluded), as the 30 first generations are likely to have the worst candidate solutions of the parameter estimation problem. This specific model is quickly solved with the correct parameters, but can run slowly with very poor parameters, as mentioned above.

4.3.2 The Parameter Estimation

The CMA-ES algorithm managed to find the optimum with a much higher consistency than the MATLAB functions. The CMA-ES solved the problem for every initial starting point except one, albeit with a comparatively enormous runtime in some of the cases.[4] A way to handle this could be to add another termination criterion into the CMA-ES algorithm, ending the search after a set amount of time. There is no easy way to tell what this set amount of time should be however, as there is no obvious cutoff point present in the data (see Figure 4.6). The runtimes follow an approximately logarithmic distribution, where 40% of the runs converged after 100 seconds and 80% of the runs converged after 1000 seconds.

To chart the sensitivity to each of the parameters in the model, a test was performed where each of the parameters were increased and decreased by 10%. The resulting changes in model fitness are shown in Figure 4.8.

[3] The same random seed was used for intital guesses between the sequential and the parallel runs, but **not** in the CMA-ES that was subsequently used for these random seeds.

[4] The longest run of the CMA-ES algorithm took seven hours, 425 times as long as the longest runtime of Levenberg-Marquardt. Still, this run also managed to converge to the optimum.

Some of the parameters have a large variance. As shown in Figure 4.7, K_{mk} and E_T are generally inaccurate even at solutions with good fitness. The sensitivity analysis in Figure 4.8 shows that the model is the least sensitive to those parameters, but E_T is (according to this analysis) just barely less sensitive than K_{mp}, which is estimated relatively accurately. This type of univariate sensitivity analysis does not show the whole picture, as it is likely that covariances between the parameters improve the fitness of a solution where the univariate analysis would predict worse values. Still, the most sensitive parameters in this analysis have a very low variance.

Chapter 5

Testing on an RNAP Hybrid Model

5.1 The Model

The parallelized CMA-ES algorithm was used for estimating the parameters of a model describing the behavior of RNA polymerase (RNAP) when transcribing DNA strands. The model evaluated is from another thesis, and is a version of a model currently in development (Nadav Bar and Jørgen Skancke: Dynamic of RNA Polymerase and Backtracking (2018), to be submitted to BMC Bioinformatics). The model covers the phenomenon of backtracking (introduced in Nudler et al., 1997 [22]), in which the RNAP will "slide" backwards over the DNA it is currently transcribing. The model takes a DNA sequence as input, and simulates the trajectory of the RNAP enzyme. The model considers four forces that affect the movement of RNAP:

- A Brownian ratchet mechanism, enabled by the *trigger loop*, a component of the polymerase. This force is oscillatory.
- The force used to separate the DNA base pairs by breaking the hydrogen bonds of the nucleotides. This force is negative, pushing the RNAP backwards.
- The stability of the RNA/DNA hybrid. The transcribed RNA will be connected to the DNA for 8-9 base pairs before it is released. The more base pairs that are connected, the more stable the hybrid is, in addition, the stability depends on the exact base pairs.
- The catalytic force obtained from attaching the previous nucleotide to the RNA. This force is large initially, but decreases with time.

The model is a system of differential algebraic equations, which describes the translocation (movement) of the RNAP over a single base pair:

$$\dot{x} = v \tag{5.1}$$
$$\dot{v} = F_{TL} + F_{DNA} + F_S + F_C \tag{5.2}$$
$$\dot{F_C} = -20\ F_C, \quad F_C(t=0) = k_C\ f_1(NTP) \tag{5.3}$$

$$F_{TL} = k_{TL,A}\ \sin\left(t\ k_{TL,\omega} + \phi\right) \tag{5.4}$$
$$F_{DNA} = -k_{DNA}\ f_2(x, \text{DNA-DNA}) \tag{5.5}$$
$$F_S = k_S\ f_3(x, \text{RNA-DNA}) \tag{5.6}$$

where x is the position of the RNAP (each base pair has a length of 1), v is its velocity, F_{TL} is the force from the trigger loop, F_{DNA} is the force used (lost) to split the DNA base pairs, F_S is stability, F_C is force from the catalysis, t is time and $k_{TL,A}$ $k_{TL,\omega}$ and ϕ are the amplitude, frequency and phase of the trigger loop. $f_1(NTP)$ is a function dependent on the latest nucleotide added to the mRNA strand, $f_2(x, \text{DNA-DNA})$ is a function dependent on the position of RNAP as well as the DNA-DNA base pair it is currently separating, $f_3(x, \text{RNA-DNA})$ is a function dependent on the position of RNAP as well as the previous RNA-DNA base pairs transcribed.[1] k_C, k_{DNA} and k_S are scaling parameters.

The *hybrid* part of the model comes from the fact that the DAEs are chained. After the RNAP has reached the end of the current base pair, the velocity is reset to zero. The values of the functions in Equation (5.3), (5.5) and (5.6) are calculated anew and, after a slight delay, the movement begins again: The DAE system is solved for the next base pair. If the RNAP has not reached the end of the current base pair within 150 seconds, it is assumed that it never will, and that backtracking happens. The simulation is ended, and the transcription is counted as a failure.

Some elements of the model are random: The phase of the trigger loop, ϕ, is given by $\mathcal{N}(\frac{\pi}{2}, 1)$. In addition, one of the terms for calculating the delay before starting the next translocation is random[2] as well. Because of this randomness, the system has to be solved multiple times and then be averaged in order to determine the model output. The model output of interest is the *productive yield* (Hsu et al. (2006) [23]), which is the ratio of full-length RNA to the total RNA produced:

$$\text{PY} = \frac{Success}{Success + Backtrack} \tag{5.7}$$

A $Success$ is a case where a complete RNA strand was transcribed without backtracking, as opposed to the incomplete RNA strands that result from backtracking. In order to perform the parameter estimation, experimental data is provided, showing the productive yields corresponding to 30 DNA strands.

The parameters that were estimated in this thesis are $k_{TL,A}$, $k_{TL,\omega}$, k_{DNA}, k_C and k_S. The correct parameter values are unknown.

[1] The last nine base pairs as a maximum.

[2] Following a folded normal distribution, $|\mathcal{N}(0, \sigma)|$.

5.2 Constraints

The parameter estimation was first attempted almost unconstrained, on the assumption that the algorithm would converge to the global optimum even without variable bounds. Only a non-negativity constraint was included[3] in the first attempt as the parameters were supposed to be non-negative, and experience indicated that allowing the algorithm to solve the model with negative parameters can cause bad results: If the initial guess provides a bad combination of parameters, a solution that includes negative parameters can have a better fitness than many others. The non-negativity constraint turned out to be insufficient, as some local optima occupied a large part of the search space, making them easy to fall into. Two more constraints were added as additional failure modes (cases where the algorithm was stuck in a clearly unwanted situation) were discovered, and an upper bound was added as the last constraint.

5.2.1 First Failure Mode: No Productive Yield

Attempting the parameter estimation with only non-negativity constraints resulted in the algorithm steadily increasing the size of the parameters: The algorithm overlooked the relatively small part of the parameter space that produced some levels of backtracking, and instead moved into the (infinitely larger) parameter space that produced 100% backtracking (and correspondingly, 0% PY). Because the yields in the experimental data are low, the resulting fitness (sum of squared errors) from this was better than many other feasible solutions. Once the algorithm moved into this part of the parameter space, the candidate solution continued moving the same way, seeing as the fitness for all individuals were identical. To prevent this, a constraint was added in the wrapper function, checking if any successful transcriptions happened for any of the DNA strands. If absolutely none occurred, the fitness was set to a high value (10^5). Figures 5.1 and 5.2 show how the fitness is steady at 0.6217 (which is the SSE resulting from no yield) while the parameters diverge into increasing values. At generations 21-23, fitness values of less than 0.6217 are found, which means that these were feasible solutions featuring a low amount of productive yield. The following generations did not manage to find these kinds of solutions again. For the rest of the generations, the best solutions were those returning 0% PY.

[3] All constraints were incorporated as explained in Section 2.6.

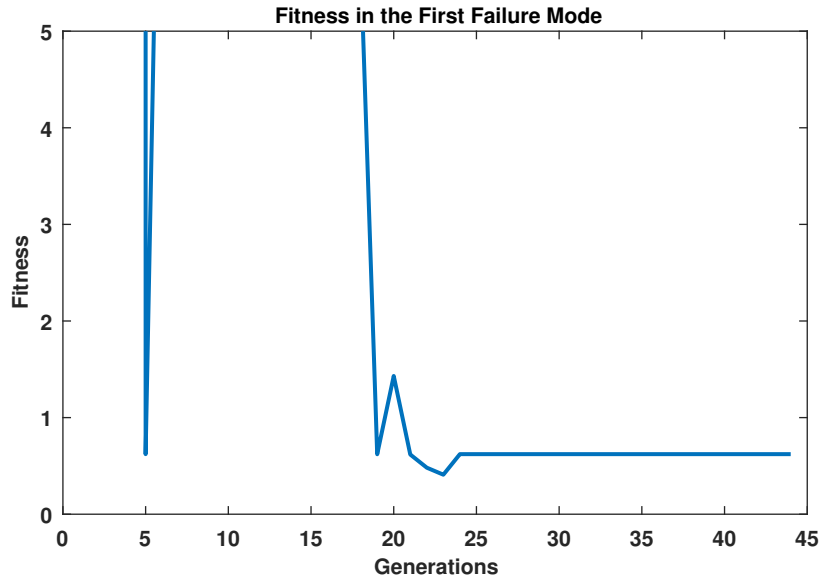

Figure 5.1: A plot showing the fitness values (of the best individual solution) of each generation in the first failure mode. The earlier values that are outside of the axis limits are a combination of solutions with negative parameters and feasible solutions with bad fitness.

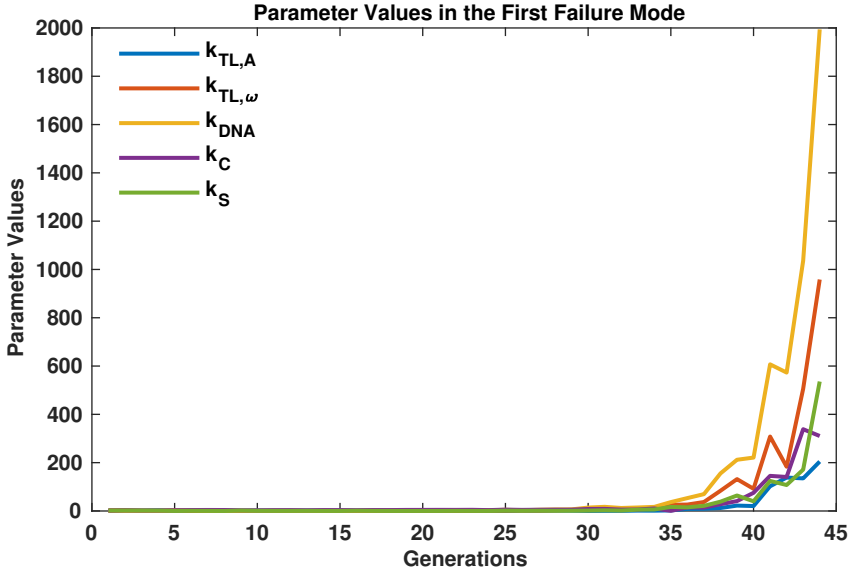

Figure 5.2: A plot showing the parameter values (of the best individual solution) of each generation in the first failure mode.

5.2.2 Second Failure Mode: 100% Productive Yield

After adding the 0% PY constraint another failure mode emerged, mirroring the prior: The algorithm found parameters giving 100% yield (0% backtrack rate), and once again diverged into ever-increasing parameter values. The fitness corresponding to 100% yield is significantly worse than that of 0% backtrack rate however: It is the worst fitness possible, excepting the fitness returned when breaking the constraints! To trigger this failure mode, the initial guess had to be in a region of the parameter space where all the candidate solutions either breached one of the constraints or returned 100% yield. As before, a constraint was added, setting the fitness to a high value (a higher value than the previous constraint, to more heavily discourage this solution than the 0% yield solution) if the simulation returned 100% yield for all DNA strands.

In Figure 5.3, it can be observed that the sum of squared errors (of the best individual in the generation) starts off at 23.3087, the fitness corresponding to 100% yield. Most likely, some amount of the individuals broke one of the constraints, giving the algorithm an initial direction to move; away from the constraint. At generation 5, an individual solution returns 4.8345, but this single incidence is not enough to move the algorithm into the feasible area. After this, the algorithm keeps expanding the search space as it moves away from the solutions that break one of the earlier constraints. The effect on the parameters can be seen in Figure 5.4. The program terminated at generation 21, because the 15 preceding generations had no difference in fitness (satisfying the first termination criterion in Section 2.6).

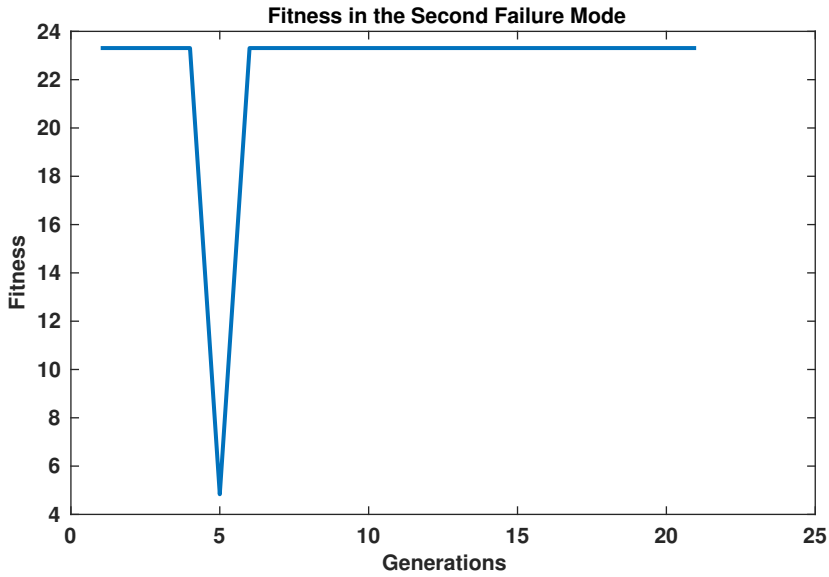

Figure 5.3: A plot showing the fitness values (of the best individual solution) of each generation in the second failure mode.

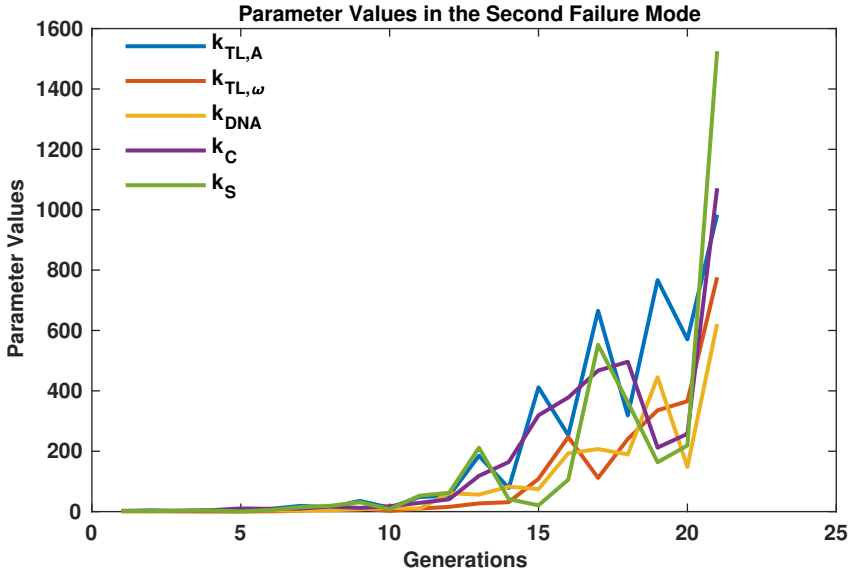

Figure 5.4: A plot showing the parameter values (of the best individual solution) of each generation in the second failure mode.

5.2.3 Adding an Upper Bound

After some time, enough parameter studies had been performed on the model to say with relative certainty that the parameters could be constrained between 0 and 3. This was useful to add even with all the other constraints already implemented, as there seemed to be a local minimum in the shape of a (perhaps infinitely long) "valley" where multiple solutions returned similar fitnesses as long as the ratios between the parameters stayed similar. Adding the upper bound reduced the amount of generations before a feasible solution was found.

5.3 Intensification: Improving the Accuracy of the Solution

After implementing the constraints, all parameter estimation runs converged to promising local minima. The size of these minima were relatively large for some of the parameters however, as the random element in the model means that subtle differences in fitness are near undetectable. The solution could only be improved to a certain point until the random noise made all solutions essentially the same. An attempted fix for getting past this "stochastic bedrock" was to increase the number of times the simulations were run, in order to reduce the uncertainty. Because this greatly increases the evaluation time for each generation, doing it in the early generations would be a waste of time: The *intensification*,

the increase in simulation evaluations,[4] should only happen when the fitness stagnates.

To decide when to intensify the search, a *t*-test is performed in order to see whether the change in fitness over the last few generations of the algorithm is statistically significant in the interval. If it is not, the search is intensified.

From the implementation of the termination conditions in section 2.6, the *generation history*, the amount of generations (looking backwards) to be considered when evaluating whether a termination criterion is satisfied is

$$\text{nGenHist} = 10 + \lceil 30\frac{N}{\lambda} \rceil$$

where N is the dimensionality of the problem and λ is the population size.

This generation history seemed too short for the purpose of evaluating the significance of the change in fitness, so empirically, a value of $\lfloor 1.5 \cdot \text{nGenHist} \rfloor$ was chosen instead.

When an intensification is triggered, the next intensification is barred from happening until another $\lfloor 1.5 \cdot \text{nGenHist} \rfloor$ generations have passed, in order to reset the data. Due to the behavior of the model, this is all but guaranteed to happen as soon as possible, as the improvement gained by the intensification happens fast and is not very large. The intensification functionality is included in the code in Appendix D.1.

5.4 The Multistart Implementation

When running preliminary parameter estimation attempts on the RNAP model, it became clear that multiple local minima existed, some of which the method would converge towards depending on the starting position. It is possible that, given enough generations, the method could eventually navigate out of the minimum (depending on how wide and "deep" it is), but the long evaluation times of the model means that this amount of generations is not available. Thus, multiple runs needed to be performed in order to be more secure that a given solution was optimal (or close to it). Because of the long runtime of the program, running multiple instances of the algorithm after one another would take an unreasonable amount of time. Either multiple jobs needed to be queued at the same time on the cluster, or one job needed to run multiple independent (but still parallelized) instances of the algorithm. The latter option was chosen.

5.4.1 Dividing Into Batches

To perform simultaneous parallel instances of the CMA-ES method in one job, the ranks first need to be divided into subgroups (hereby *batches*) in which all MPI communications are limited to the other ranks in the subgroup. Because each rank runs the same script individually, this can be done by using a regular for-loop with an if-statement:

[4] This increase in evaluations should coincide with the time when the focus of the optimization algorithm has switched from diversification to intensification. Thus, the intensification happens at both optimization algorithm level as well as model evaluation level.

for $i = 0$ **to** $(N_{Batches} - 1)$ **do**
  if (`my_rank` $\geq i \cdot$ `batch_size`) **and** (`my_rank` $< (i+1) \cdot$ `batch_size`) **then**
    run CMA-ES as batch number i
  end if
end for

The if-statement filters out the ranks within each batch. If a rank does not fulfill the criteria, it will continue in the for-loop until it does. Functionally, this is a parallel (outer) for-loop. The rank number, i is used further in order to keep track of which ranks belong to what batch.

With the ranks divided into batches, the Distributed MATLAB functions needed to be redefined to limit MPI communication to ranks within the same batch.

5.4.2 Intra-Batch MPI Communication

With the batches functioning completely independently of each other, communication which would usually pass through the master rank (rank 0) needs to go to another substitute master rank. This rank is set to be the first rank in each batch, that is rank number $i \cdot$ `batch_size`.

The Distributed functions written by the HPC group at NTNU-IT use basic MPI functionalities which make some implicit assumptions that work against use in independent parallel instances:

- That the first (and master) rank in the process is rank 0 and that the other ranks involved in the process count up from there.
- That information should be spread to (and collected from) all ranks involved in the parallel job. This means that information would be spread outside of an individual batch.

Therefore, the Distributed functions needed to be edited in order to allow for temporary substitute master ranks, and to limit the communication to the other ranks within a batch. The problem was solved by making use of the batch size and batch number as arguments in the function calls.

The default `parsteps` function makes use of the `num_ranks` variable to determine the amount of available ranks, before dividing the iterations of the for-loop evenly between them. This code was simply rewritten by changing the number of ranks available to `batch_size`, and use the batch number in order to calculate the specific ranks that should be included in the for-loop.[5]

The new, custom `spread` and `reduction` functions take in the instance (*batch*) number the worker is a part of, as well the amount of workers in each batch (*batch size*) as input arguments. From this, the master rank for each batch is set to be the first rank of the batch, and the other ranks are instructed specifically to send (or receive) to (or from) that

[5]The default function assigned iterations to ranks 0 and up, which in this case would be incorrect for any batch except batch number 0.

rank. This way, there is no communication between each batch. These custom functions are built on simple `send` and `receive` NMPI functions, as these can specify sender and receiver rank. The code is shown in Appendix D.

Using point-to-point communication as done in these custom functions is less efficient than using the collective communication alternatives (which use the assumptions outlined above) that are used in the default functions. However, this time loss is negligible assuming the runtime between each instance of MPI communication is significant. This is very much the case when using the CMA-ES on the RNAP parameter estimation problem.

5.4.3 Selecting Initial Points

A concern when the algorithm converges towards multiple local minima is that the initial guess is going to affect what minimum the algorithm ends up in.[6] Thus, the independent CMA-ES instances should start at different areas in the problem space. Because of the relatively low dimensionality of the RNAP parameter estimation problem (five parameters), the starting positions could be provided systematically instead of randomly.

For each dimension, a low value and a high value is specified. Then, a CMA-ES instance is set to start on every possible permutation of high/low values. This means that a permutation matrix is created as follows (for a three-dimensional problem):

$$P_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.8}$$

Where 1 represents a high value and 0 represents a low value for a given parameter. Each row is a combination of parameters.

In the example case of Equation (5.8) there are 8 permutations, meaning that 8 independent instances of the algorithm are initiated. This scales up by 2^n, hence the five-dimensional RNAP problem will require $2^5 = 32$ instances. Keep in mind that each of these instances require a minimum of $4 + \lfloor 3 \ln 5 \rfloor = 8$ ranks in order to do all model evaluations simultaneously, which is very much desired. The high and low values were set to be $\frac{1}{4}$ and $\frac{3}{4}$ of the way through each dimension, with an initial search radius σ_0 of $\frac{1}{4}$. An example of how this looks is presented in Figure 5.5.

[6] Because of the random nature of the CMA-ES method, the method could theoretically end up in just about any of the minima from any initial position, but the initial position would heavily affect the likelihood of reaching a specific minimum.

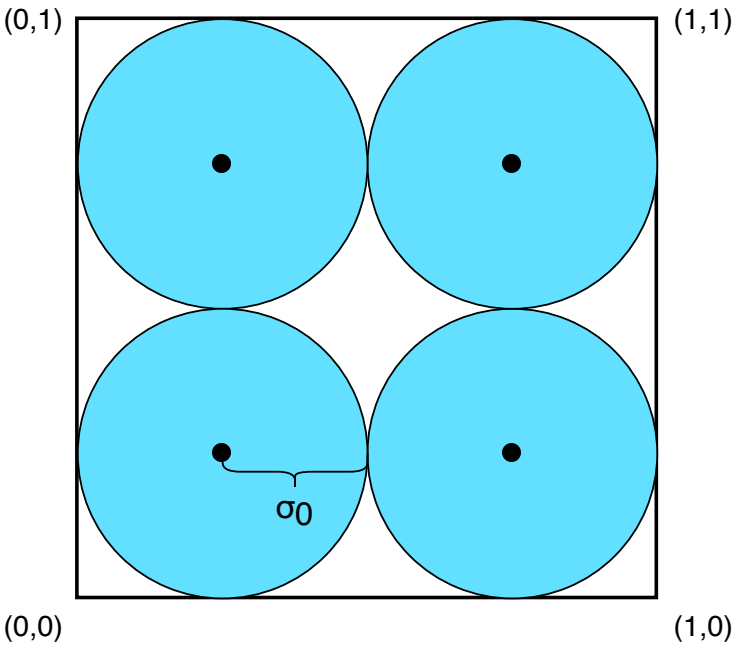

Figure 5.5: An example of where the four starting points would be placed in a two-dimensional problem where the search space is set between 0 and 1 for both dimensions. The initial step size σ_0 is chosen in order to cover as much of the space as possible with minimal overlap.

5.5 Conditions for Parameter Estimation

Because each model evaluation requires the model to be simulated multiple times (100 times, before intensification) for each of the 30 DNA strands, each evaluation takes a significant amount of time (generally over 10 minutes). The long evaluation times means that running a parallel method is very beneficial. In addition, the randomness also makes the problem space non-smooth and extremely noisy. The Levenberg-Marquardt and Trust Region Reflective methods in MATLAB do not work on this problem.

Two different strategies were used: A "single-start" and a multistart. The single-start consisted of a single, high-population CMA-ES instance with 32 ranks (and thus a population of 32).[7] The initial point was in the middle of the search space, at 1.5 for all parameters. The job was run for 40 hours.

The multistart strategy used 32 independent batches, each consisting of 8 ranks, resulting in a total of 256 ranks. The starting points were evenly distributed as explained in Section 5.4.3. The multistart job was also run for 40 hours.

[7] Preliminary attempts also used 64 ranks, with no discernible difference. 32 ranks were used subsequently as CPU hours on the Vilje cluster are a limited resource.

Both strategies used the intensification method explained in Section 5.3.

5.6 Results

Both jobs were terminated after 40 hours of runtime. The best solution was assumed to be the last solution found before termination. In all runs, intensification (from Section 5.3) happened twice, increasing the amount of simulations per model evaluation to 500 (times 30 DNA sequences).

5.6.1 Single CMA-ES Instance

The best parameters of the last generation in the single-start run are shown in Table 5.1. The fitness was calculated anew by performing 100 model evaluations using the best parameters, each model evaluation consisting of 500 simulations (per DNA sequence). The average output of the 100 model evaluations is shown in Figure 5.6, and compared to the experimental data. The error bars mark 95% confidence intervals for the productive yield of each DNA sequence.

In figures 5.7 to 5.9, the changes of the fitness and parameters of the best individual in each generation is shown. The step sizes are calculated as the distance (in parameter space) from the best solution in one generation to the next. Note that in these figures, only the best individual is considered. The parameter history (and subsequently, the step sizes) does not show the exact movement of the mean, although they are highly correlated, and would look very similar.

In Figure 5.10 the model evaluation time for each generation is shown. The evaluation times increase as intensification (Section 5.3) happens. The amount of simulations per evaluation are increased twice, first from 100 to 300, then to 500.

Table 5.1: The solution of the RNAP parameter estimation, from the single-instance CMA-ES run. The fitness is the mean fitness ($\pm$ two standard deviations) found after performing 100 runs of 500 simulations using these parameters.

Parameter	Value
$k_{TL,A}$	0.9514
$k_{TL,\omega}$	0.1077
k_{DNA}	2.7543
k_C	2.0898
k_S	0.8757
Fitness (SSE)	0.1759 ± 0.0264

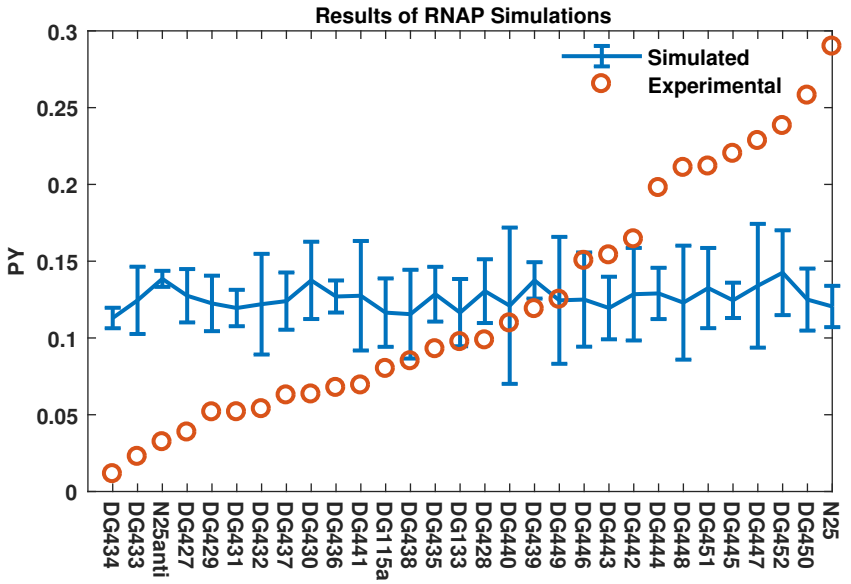

Figure 5.6: The resulting model output from using the parameters presented in Table 5.1. The x-axis consists of the different DNA sequences and is not a continuous variable. The error bars mark the 95% confidence interval.

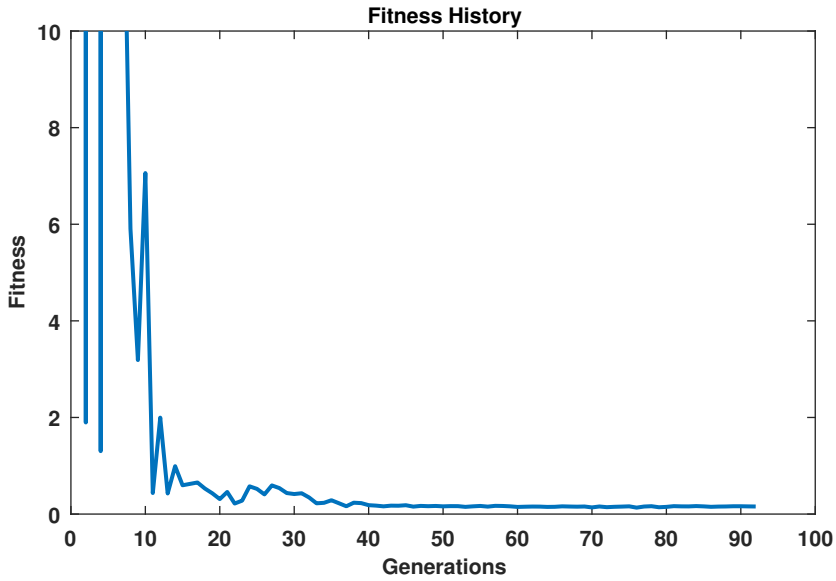

Figure 5.7: The fitness history (best fitness of each generation) of the single-instance parameter estimation. The fitness values outside of the scope of the y-axis are cases where the constraints were breached.

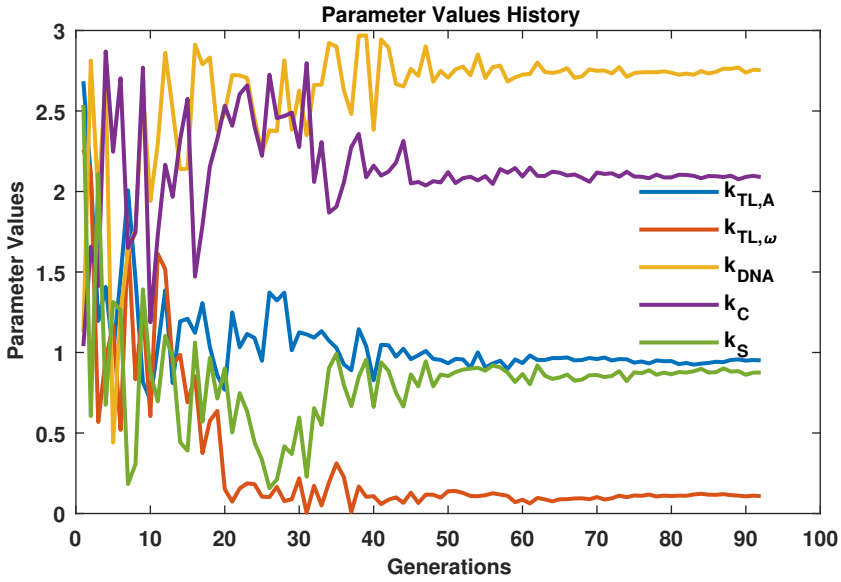

Figure 5.8: The parameter history (the parameters corresponding to the best individual of each generation) of the single-instance parameter estimation.

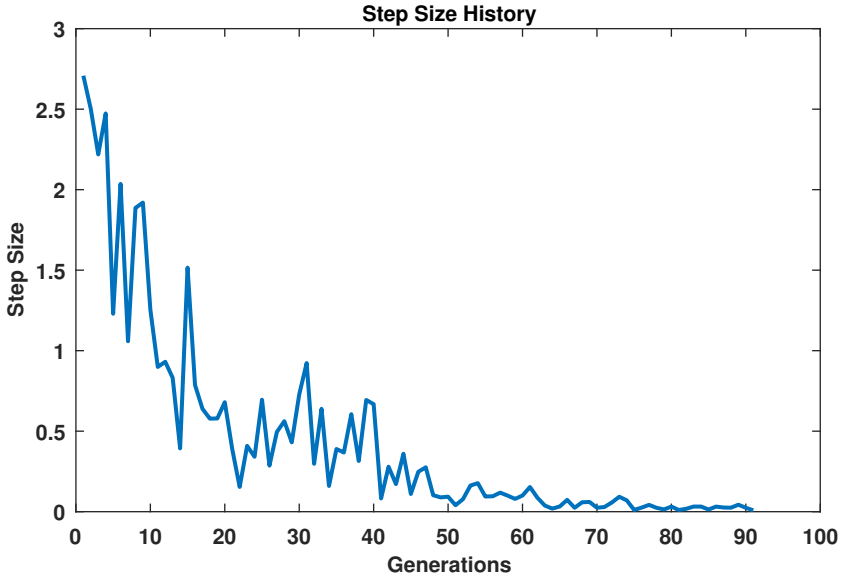

Figure 5.9: The step size between each generation in the single-instance parameter estimation, calculated as the distance between the best solution of one generation to the next.

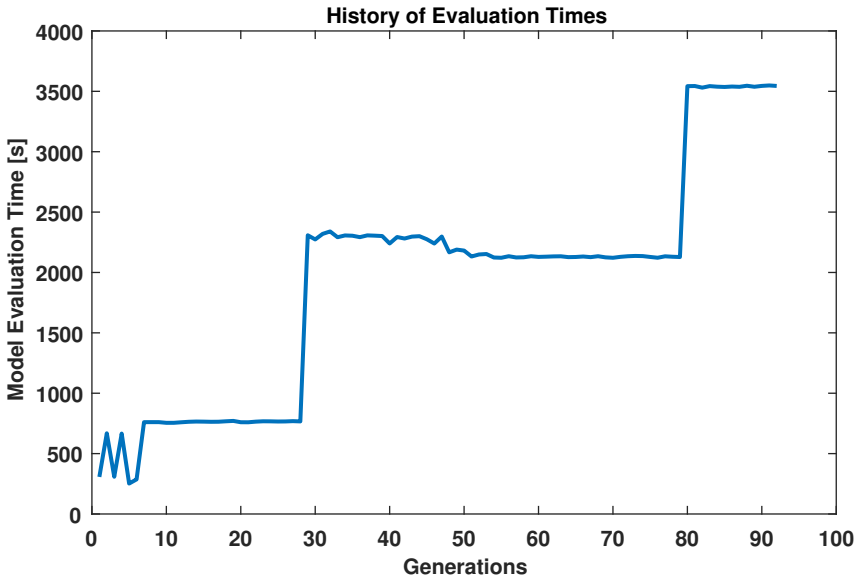

Figure 5.10: The model evaluation time for each generation in the single-instance parameter estimation. Intensification happens at generations 28 and 79, increasing the amount of simulations per evaluation from 100 to 300 and finally 500.

5.6.2 CMA-ES Multistart

The parameters and fitnesses found in each of the 32 CMA-ES instances is shown in Appendix E. Not all runs managed to find the minimum. For further analysis, the solutions with fitness over 0.2 has been discarded as unconverged. There were ten solutions with less than 0.2 fitness. A box plot showing the distribution of the parameter values in converged solutions is shown in Figure 5.11. The plot showing the model outputs of the solution with the best fitness is presented in Figure 5.12. All runs intensified twice.

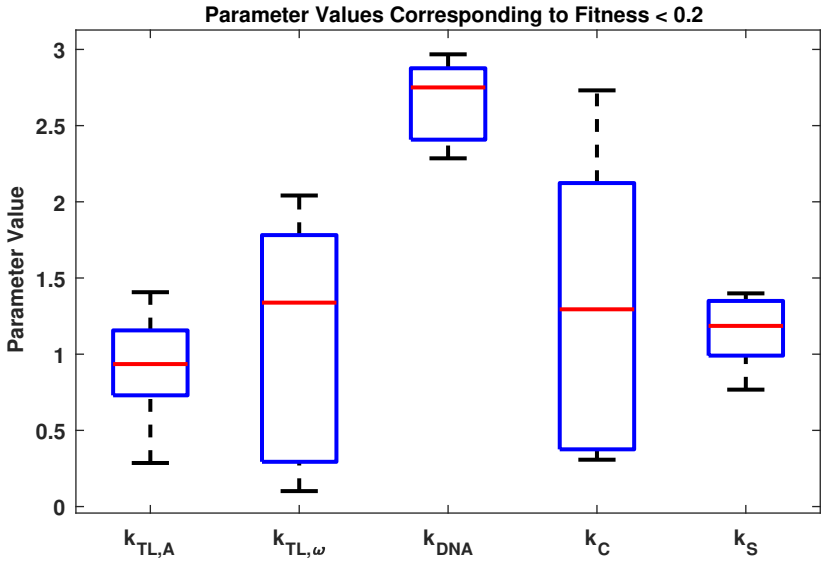

Figure 5.11: The mean values for the parameters in the ten solutions with fitnesses (SSE) less than 0.2. The error bars mark the 95% confidence intervals.

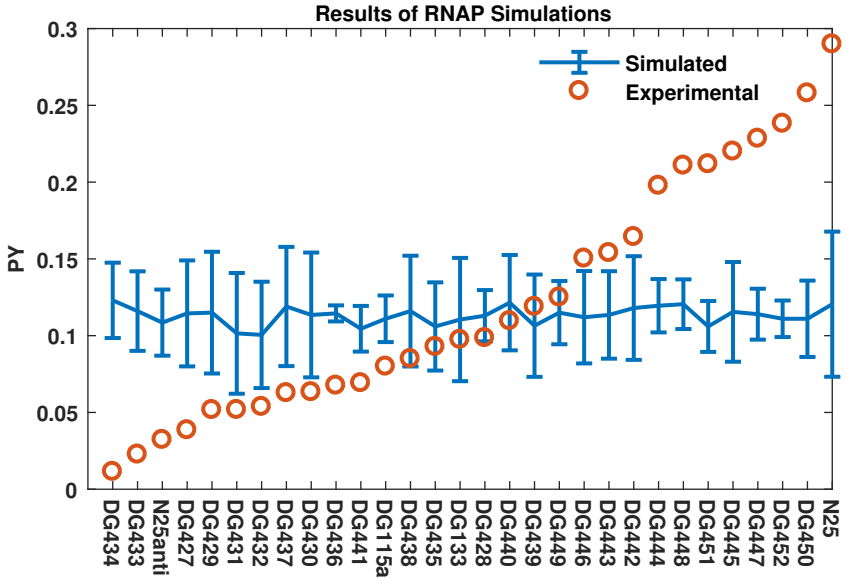

Figure 5.12: The resulting model output from using the parameters of the solution with the best fitness from the multistart run. The x-axis consists of the different DNA sequences, and is not a continuous variable. The error bars mark the 95% confidence interval.

5.7 Discussion

The CMA-ES has managed to find a minimum, as seen from Figures 5.7 to 5.9: The fitness and the parameter values stay stable in approximately 50 generations before termination. The best solutions from the multistart run indicate that this is the global minimum, as the best solutions have relatively similar values for three of the parameters. The two other parameters, $k_{TL,\omega}$ and k_C, have values spanning a large part of the search space, which indicates that the model is less sensitive to these parameters.

Even though what is likely to be the global minimum (within the constrained search space) is found, the model output does not follow the same correlation with the DNA sequences as the experimental data does (see Figure 5.6). There are two potential ways in which the model output could be improved, depending on what the cause of the mismatch is. The choices are either improving the model or finding a better solution.

It is possible that the model should be improved. The version of the RNAP model evaluated here does not take into account all types of forces affecting the RNAP, and it is very possible that emulating the experimental data is impossible using the model as it is. The fact that the model behavior does not correlate with the DNA sequences could indicate this. If a better solution existed with parameter values close to those found, there could still be expected some kind of systematic model behavior in response to the DNA sequences given as inputs.

There is also a possibility that a correct solution (in the sense that it produces the desired behavior) exists for the given model, either in a very small part of the search space or

far outside the search boundaries. If the random noise was too large even after the intensification had increased the amount of simulations, it could be too difficult to register the subtle differences in fitness that could lead to a solution which would result in the correct behavior. As can be seen in Figure 5.6, there is still a significant variance in the model outputs even with 500 simulations per model evaluation. The productive yields vary between 0.1 and 0.15 (approximately) for the same set of parameters, which results in a fitness with a 95% confidence interval of 0.1759 ± 0.0264, as shown in Table 5.1. If the boundaries of the search space were wrongly defined, the parameters could have values far larger than 3, which would increase the impact of some of the forces affecting the RNAP movement more than currently possible within the constraints.

As mentioned in Section 5.6, the solution selected as the "best" was the solution with the lowest fitness in the last generation. In a case where a completely deterministic model is used, the solution would simply be the one with the lowest fitness (out of any generation) as it would unquestionably be the best. However, due to the randomness inherent in the RNAP model, this is not a safe choice as it could be a statistical outlier for a solution with suboptimal parameters. Therefore, in the case of the RNAP model, the best solution was assumed to be the last solution found, as $\mathbf{m}^{(g)}$ should statistically move towards the best solutions as $g \to \infty$, in spite of noise in the individual model evaluations. When assessing the plots of the parameter changes (and step sizes), this assumption seems acceptable: The change in parameters is negligible for the last generations, which means that the solution found is not just a one-time lucky roll of the dice. Even though the solution is stable for approximately 50 generations, the solution of the last generation is likely to be the most correct, as the small adjustments of the last generations are more likely to be beneficial (or at worst, neutral) for the solution.

The noise present in the fitness even when simulating each DNA sequence 500 times indicates that the intensification (from Section 5.3) increasing the amount of simulations by 200 each time is insufficient to find an exact minimum. The intensification happened twice during the process, for a total of 500 simulations per model evaluation. Looking at the amount of random noise still present, it seems like the number needs to be increased to a point where the evaluation times would make the runtime of the optimization process infeasibly long.[8] In this case, improving the model is the only feasible way to achieve the desired model behavior. As mentioned before, model development is not the focus of this thesis.

[8] At 500 simulations, one model evaluation required approximately 1 hour.

Chapter 6

Further Discussion

The CMA-ES method developed herein consistently solves the parameter estimation problem of the delayed negative feedback model, with a higher success rate than the Levenberg-Marquardt or trust-region-reflective methods used in MATLABs `lsqnonlin` function. However, this comes at the cost of significantly higher runtimes, even when running the CMA-ES in parallel on a computer cluster and the `lsqnonlin` methods on a laptop. This is a trait typical of stochastic global optimization methods, as the tradeoff between diversification and intensification means that the methods will converge slower than local methods, but with a higher chance of finding the global minimum.

The test on this model indicates that the CMA-ES is reliable for parameter estimation problems of this type.

The results from the RNAP model calibration were mixed in the sense that the parameters found did not make the model respond to inputs in the same way as the experimental data. However, a minimum was found, proving that the method manages to navigate noisy objective functions. If a correct solution exists for this RNAP model and it was not found, it is likely not due to the mechanics of the CMA-ES method, but rather because of limitations related to the choice of search space or an excessive amount of noise in the objective function. The fact that the CMA-ES is parallelized was critical, as the time otherwise needed to solve the problem would be impractically long.

In summary, the CMA-ES is more consistent than the `lsqnonlin` functions, and manages to solve more difficult problems, at the cost of comparatively longer computational time for simple problems. For problems with long evaluation times and noisy objective functions, the CMA-ES could be a powerful tool which should be explored further in a later project.

Chapter 7

Conclusion

In this thesis, a parallel implementation of the CMA-ES optimization method has been implemented and tested on two parameter estimation problems from systems biology. In addition to parallelizing the method, a set of functions were created in order to allow for independent instances of a parallelized CMA-ES process to run in parallel, in a multistart fashion. These functions can be used in other programs where separate groups of ranks work independently of each other, without needing to delve into lower level programming than MATLAB.

The first model calibration was performed on a delayed negative feedback ODE model with known parameters. In this case, the CMA-ES method managed to find the correct parameters more consistently than MATLABs `lsqnonlin` function, but with significantly longer runtimes. For a problem this simple, the method is unnecessarily complicated and resource intensive.

The second model calibration was performed on a DAE model with a stochastic element, simulating the movement of RNA polymerase during transcription. The correct parameters for this model were unknown. The `lsqnonlin` functions in MATLAB were unusable for this problem. Due to the random element in the model, the model evaluations had to be performed by simulating the model multiple times in order to use the average of the simulations as the model output. Starting at 100 simulations per model evaluation, the high amount of noise still present in the objective function required an increase in simulations when approaching a local minimum. A functionality was added to the CMA-ES algorithm which increased the amount of simulations after the improvement in the fitness values for each generation seemed to have stagnated.

The CMA-ES managed to minimize the model error, although the minimum found spanned a wide area in parameter space and did not lead to satisfactory model behavior. This is most likely due to either errors in the model or the amount of noise still present in the objective function after increasing the amount of simulations to 500. The boundaries of the search space could also be too narrow.

The parallelized CMA-ES method seems to potentially have a niche in difficult parameter estimation problems, where the objective function is multimodal or noisy and the model evaluations take too long to be performed sequentially. A prime example of this type of problem is the RNAP model evaluated in this thesis.

7.1 Further Work

The CMA-ES should be tested on additional models with noisy objective functions, preferably with known parameters, in order to gain more knowledge about the speed and accuracy the method achieves on such models.

Bibliography

[1] F. J. Bruggeman and H. V. Westerhoff, "The nature of systems biology," *TRENDS in Microbiology*, vol. 15, no. 1, pp. 45–50, 2007.

[2] J. Sun, J. Garibaldi, and C. Hodgman, "Parameter Estimation Using Metaheuristics in Systems Biology: A Comprehensive Review," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, no. 1, pp. 185–202, 2012.

[3] M. Ashyraliyev, Y. Fomekong-Nanfack, J. A. Kaandorp, and J. G. Blom, "Systems biology: parameter estimation for biochemical models," *FEBS Journal*, vol. 276, no. 4, pp. 886–902, 2009. [Online]. Available: http://dx.doi.org/10.1111/j.1742-4658.2008.06844.x

[4] C. G. Moles, P. Mendes, and J. R. Banga, "Parameter Estimation in Biochemical Pathways: A Comparison of Global Optimization Methods," *Genome Research*, vol. 13, no. 11, pp. 2467–2474, 2003. [Online]. Available: http://genome.cshlp.org/content/13/11/2467.abstract

[5] J. R. Banga, "Optimization in computational systems biology," *BMC Systems Biology*, vol. 2, no. 1, p. 47, 2008.

[6] C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 9 2003.

[7] N. Hansen, "A CMA-ES for mixed-integer nonlinear optimization," Ph.D. dissertation, INRIA, 2011.

[8] T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber, "Exponential Natural Evolution Strategies," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 393–400. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830557

[9] A. Zhigljavsky and A. Zilinskas, *Stochastic global optimization*. Springer Science & Business Media, 2007, vol. 9.

[10] A. Neumaier, “Complete search in continuous global optimization and constraint satisfaction,” *Acta numerica*, vol. 13, pp. 271–369, 2004.

[11] N. Hansen and A. Ostermeier, “Completely Derandomized Self-Adaptation in Evolution Strategies,” *Evolutionary Computation*, vol. 9, no. 2, pp. 159–195, 6 2001.

[12] N. Hansen and S. Kern, “Evaluating the CMA evolution strategy on multimodal test functions,” in *PPSN*, vol. 8. Springer, 2004, pp. 282–291.

[13] N. Hansen, “The CMA Evolution Strategy: A Comparing Review,” in *Towards a New Evolutionary Computation: Advances in the Estimation of Distribution Algorithms*, J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 75–102.

[14] ——, “The CMA Evolution Strategy: A Tutorial,” *CoRR*, vol. abs/1604.0, 2016. [Online]. Available: http://arxiv.org/abs/1604.00772

[15] A. Auger and N. Hansen, “A restart CMA evolution strategy with increasing population size,” in *2005 IEEE Congress on Evolutionary Computation*, vol. 2, 2005, pp. 1769–1776.

[16] D. V. Arnold, “Optimal Weighted Recombination,” in *Foundations of Genetic Algorithms: 8th International Workshop, FOGA 2005, Aizu-Wakamatsu City, Japan, January 5 - 9 , 2005, Revised Selected Papers*, A. H. Wright, M. D. Vose, K. A. De Jong, and L. M. Schmitt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 215–237.

[17] D. V. Arnold and D. C. S. Van Wart, “Cumulative Step Length Adaptation for Evolution Strategies Using Negative Recombination Weights,” in *Applications of Evolutionary Computing: EvoWorkshops 2008: EvoCOMNET, EvoFIN, EvoHOT, EvoIASP, EvoMUSART, EvoNUM, EvoSTOC, and EvoTransLog, Naples, Italy, March 26-28, 2008. Proceedings*, M. Giacobini, A. Brabazon, S. Cagnoni, G. A. Di Caro, R. Drechsler, A. Ekárt, A. I. Esparcia-Alcázar, M. Farooq, A. Fink, J. McCormack, M. O’Neill, J. Romero, F. Rothlauf, G. Squillero, A. c. Uyar, and S. Yang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 545–554.

[18] D. Walker and J. Dongarra, “MPI: a standard message passing interface,” *Supercomputer*, vol. 12, pp. 56–68, 1996.

[19] W. D. Gropp, W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[20] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[21] Z. Szallasi, J. Stelling, and V. Periwal, “System modeling in cellular biology,” 2006.

[22] E. Nudler, A. Mustaev, A. Goldfarb, and E. Lukhtanov, “The RNADNA Hybrid Maintains the Register of Transcription by Preventing Backtracking of RNA Polymerase,” *Cell*, vol. 89, no. 1, pp. 33–41, 1997.

[23] L. M. Hsu, I. M. Cobb, J. R. Ozmore, M. Khoo, G. Nahm, L. Xia, Y. Bao, and C. Ahn, "Initial transcribed sequence mutations specifically affect promoter escape properties," *Biochemistry*, vol. 45, no. 29, pp. 8841–8854, 2006.

Appendix A

Effects of Parallelization

To illustrate the effects of parallelizing and choice of population size, four test functions are evaluated on different dimensionalities, using different amounts of ranks in parallel.

These functions are standard test functions in optimization benchmarking the Ackley, Rastrigin, Rosenbrock and Sphere test functions.

The Ackley and Rastrigin functions are multimodal (multiple local minima), while the

Table A.1: The test functions used for benchmarking. n is the amount of dimensions.

Name	Function
Ackley	$f(x) = 20(1 - \exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2})) - \exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)) - \exp(1)$
Rastrigin	$f(x) = 10n + \sum_{i=1}^{n}[x_i^2 - 10\cos(2\pi x_i)]$
Rosenbrock	$f(x) = \sum_{i=1}^{n-1}[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$
Sphere	$f(x) = \sum_{i=1}^{n} x_i^2$

Rastrigin and Sphere functions are unimodal. From prior experience, the CMA-ES method navigates the Ackley and Sphere functions easily, while the Rosenbrock and Rastrigin requires more evaluations.

The amount of ranks working in parallel were 1, 8, 16, 24, 32 and 40. The single rank run is used to contrast the performance of the sequential implementation with the performance of the parallel implementation.

Due to the simplicity of the test functions, little to no actual time is saved by parallelization on these problems. Each function call has a runtime on the order of sub-milliseconds. When coupled with the small added computational overhead from the MPI processes, the parallelization is useless. In these test cases, the runtime is mostly affected by the (unavoidable) sequential parts of the code, more specifically the eigendecompositon of the covariance matrix. This is a computationally expensive operation, especially at higher

dimensionalities.

All of the aforementioned effects are negligible when evaluating a real model where the runtime for each model evaluation is several orders of magnitude longer than that of the rest of the code. However, in this specific case the runtime would be a poor indicator of the achieved speedup of the program. Instead, the number of *model evaluation sequences* (hereby MES) is used instead. For each MES, one parallelized set of model evaluations are performed. Because these model evaluations are performed in parallel, it can be expected that one MES takes a set amount of time no matter how many model evaluations each sequence consists of. In other words, each MES should take approximately the same amount of time no matter how many ranks that are used (when evaluating the same model). As a result, the runtime of the program should be proportional to the amount of MES needed to converge.

In figures A.1 to A.4, the mean amount of model equation sequences at different dimensionalities is shown for experiments running 1 to 40 ranks. Due to the extreme difference between the performance of the single worker run and the parallelized runs, the figures are showing plots of the performance with and without the single worker run, for the sake of visibility.

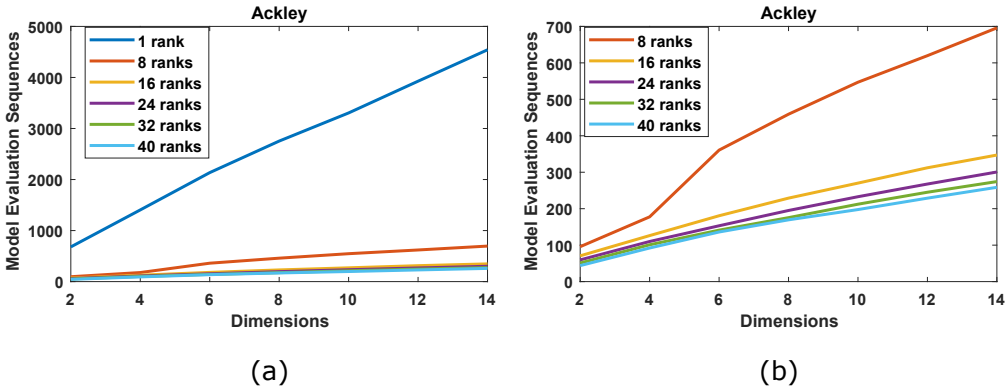

Figure A.1: The number of model equation sequences at different dimensionalities for 1 to 40 ranks when evaluating the Ackley function. Figure (b) shows only the parallelized runs.

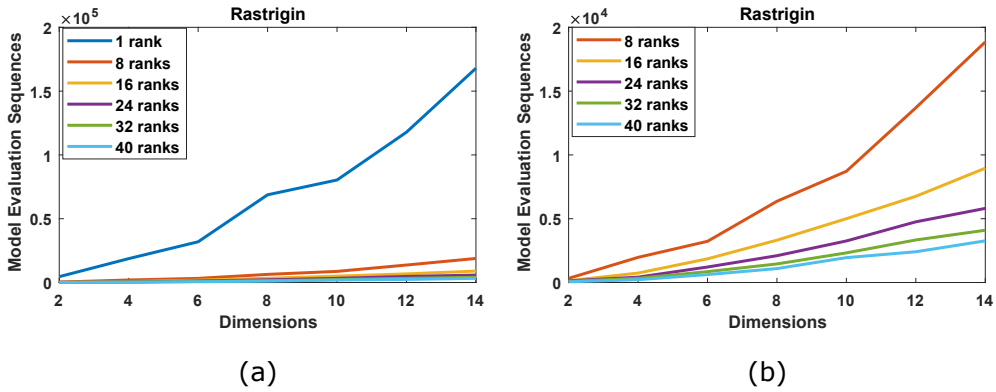

Figure A.2: The number of model equation sequences at different dimensionalities for 1 to 40 ranks when evaluating the Rastrigin function. Figure (b) shows only the parallelized runs.

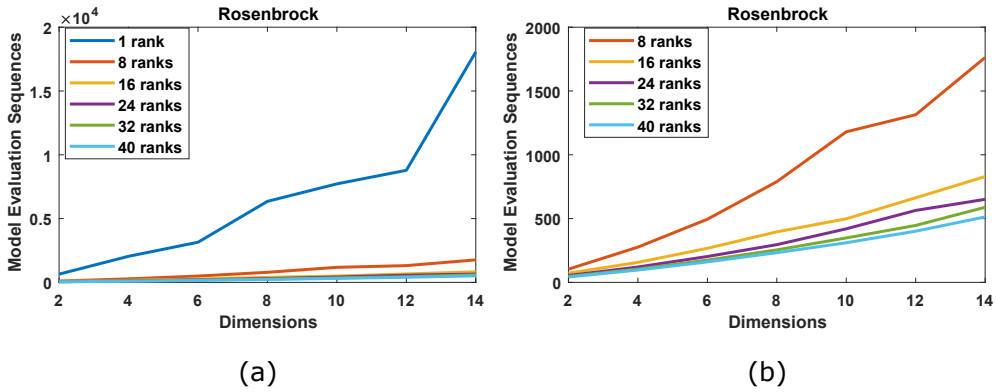

Figure A.3: The number of model equation sequences at different dimensionalities for 1 to 40 ranks when evaluating the Rosenbrock function. Figure (b) shows only the parallelized runs.

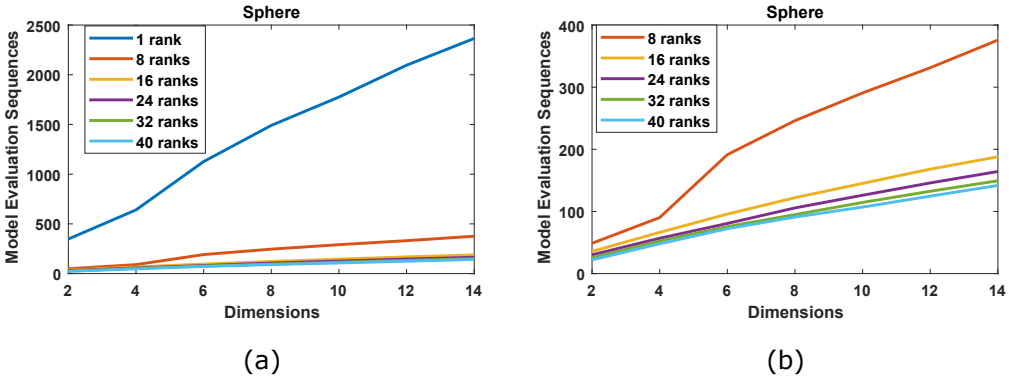

Figure A.4: The number of model equation sequences at different dimensionalities for 1 to 40 ranks when evaluating the Sphere function. Figure (b) shows only the parallelized runs.

From the results, it can be seen that parallelizing does result in a large speed-up in the sense of reducing the amount of time spent on model evaluations. In the case of 1 rank, the number of model evaluation sequences are in the thousands, while the parallel implementations only need some hundreds of MES. The number of MES decreases as the amount of ranks are increased, but with diminishing returns. This is due to the fact that increasing the amount of ranks only increases the population size in each generation. As mentioned in Section 3.3, increasing the population size improves the accuracy of the algorithm's estimation of the search space around the current point, allowing the algorithm to make better moves and converge faster. At large population sizes, the benefits of increasing the size further lessens.

The 8-rank run displays a less consistent level of performance compared to the ones using more ranks. This is due to the fact that 8 ranks are too few to consistently have a higher amount of ranks than λ_{seq}. When going from 4 dimensions to 6, the population size is increased from 8 to 16, (and consequently, the number of MES per generation is doubled) as λ_{seq} goes from 8 to 9 (following Equation (3.2)). This is most noticeable on the Ackley and Sphere functions: These functions are easily solved by low populations and they get little benefit from the improved fidelity provided by higher populations, so they get a spike in MES at the increase from 4 to 6 generations.

For the Rosenbrock and Rastrigin functions, the increase in population size is more beneficial, as these are difficult objective functions to navigate, making the increased information from the bigger population sizes more useful. In fact, for the notoriously difficult Rastrigin function, the jump from 8 to 16 offspring is beneficial, with the performance at 6 dimensions being disproportionately good (see Figure A.2). This makes sense as the largest gap between λ and λ_{seq} is at 6 dimensions with 8 ranks.[1]

[1]The population used at 6 dimensions (when using 8 ranks) is 16, while the population that heuristically should be enough, λ_{seq}, at this point is 9. The Rastrigin function is too difficult for λ_{seq} to be sufficient. For this function, higher populations is better. At 6 dimensions, λ/λ_{seq} is disproportionately large compared to the other dimensions. Thus the disproportionately good performance.

Appendix B

The Results of a 10% Change of Parameter Values

The results of running the delayed negative feedback model with perturbed parameters is shown in Figures B.1 to B.7, comparing the model outputs to those of the correct parameters. The blue line corresponds to a 10% decrease in parameter value, and the red line corresponds to a 10% increase.

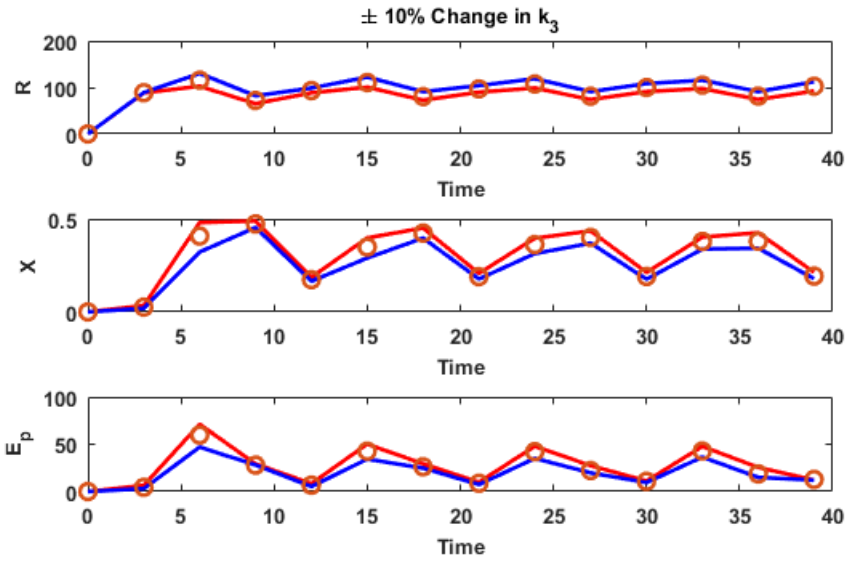

Figure B.1: The model outputs with a 10% change in the value of k_3. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase.

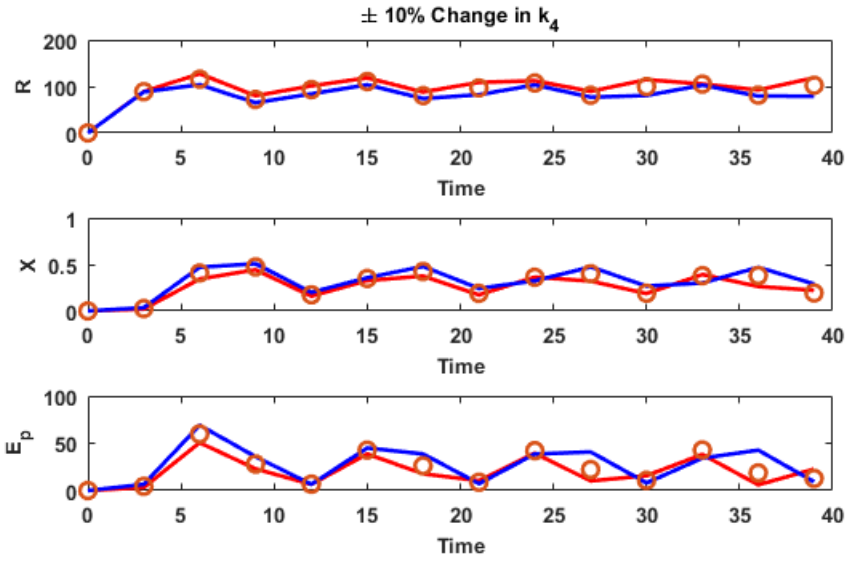

Figure B.2: The model outputs with a 10% change in the value of k_4. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase.

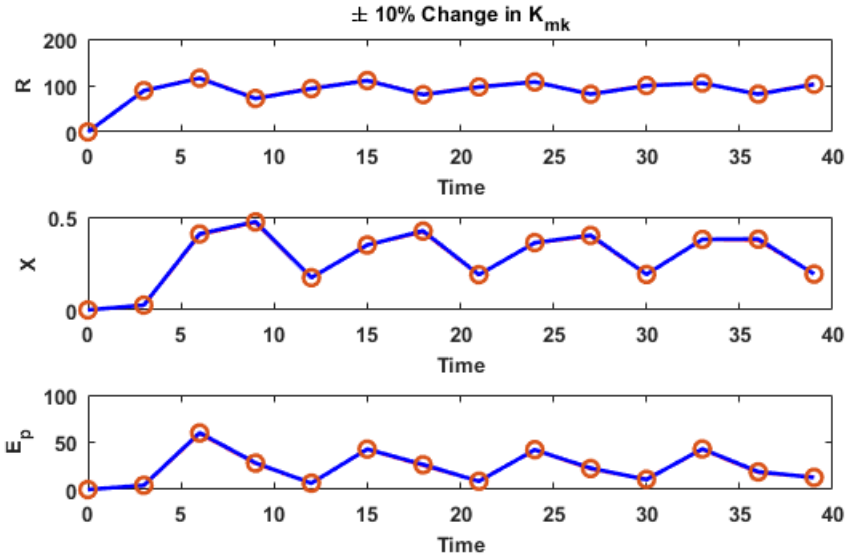

Figure B.3: The model outputs with a 10% change in the value of K_{mk}. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase (the lines overlap).

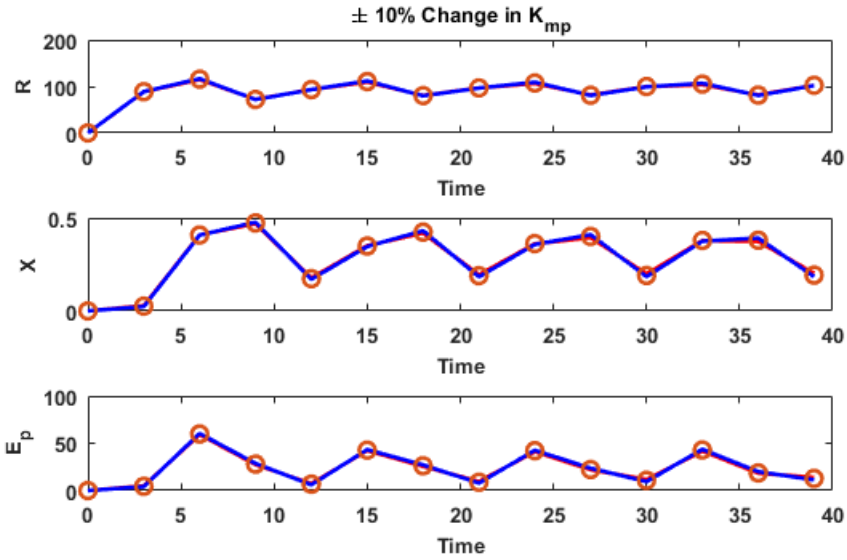

Figure B.4: The model outputs with a 10% change in the value of K_{mp}. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase (the lines overlap).

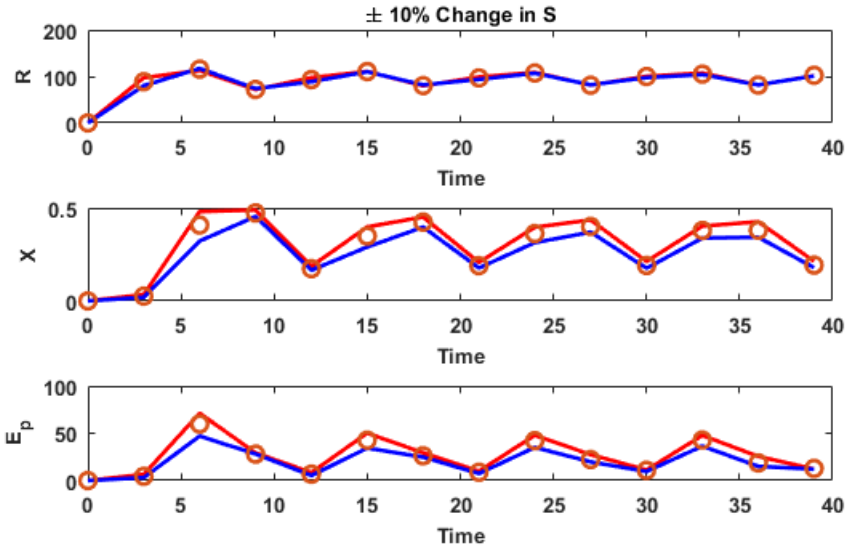

Figure B.5: The model outputs with a 10% change in the value of S. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase.

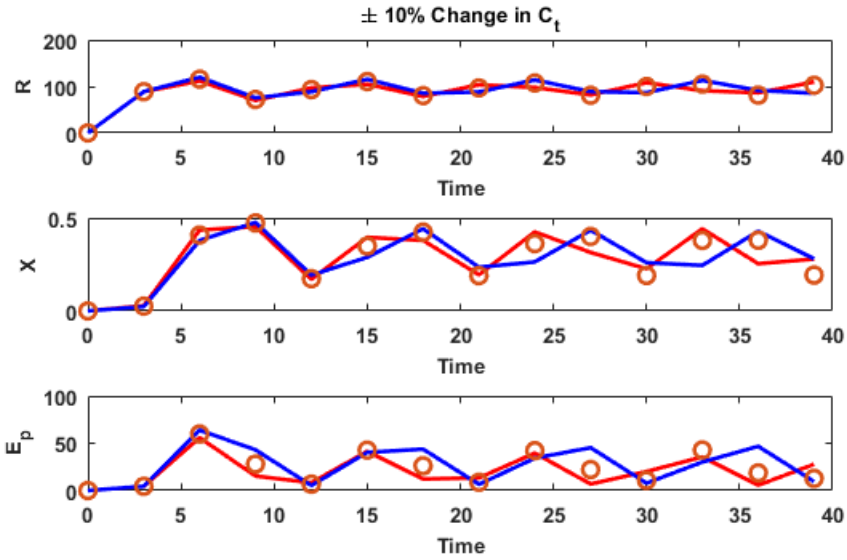

Figure B.6: The model outputs with a 10% change in the value of C_T. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase.

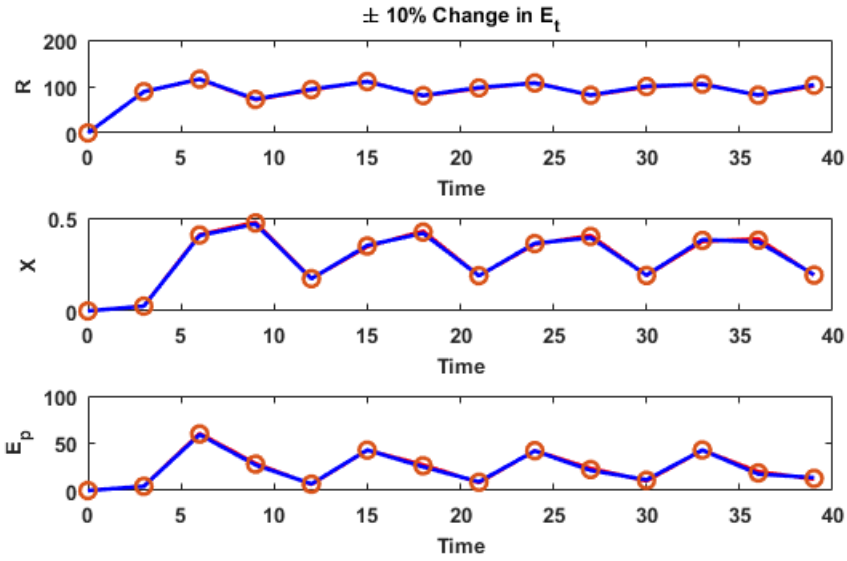

Figure B.7: The model outputs with a 10% change in the value of E_T. The circles denote the model output of the correct parameters. The blue line corresponds to a decrease in parameter value, and the red line corresponds to an increase (the lines overlap).

Appendix C

Parallelized CMA-ES Code

```
function [solution, fitnessHist, solutionHist, runtime]=CMAES(initP, modelfct, my_rank, nu
% A parallel CMA-ES implementation intended for parameter estimation.
% Returns solution (struct of fitness and parameters of the best solution),
% the histories of the solutions and their corresponding fitnesses as well
% as the runtime of the algorithm. A workspace file is saved at each
% evaluation, for cases where the algorithm is unable to terminate
% (typically due to time constraints when running on computer clusters).

% The underlying functionality is taken from Hansen, N. "The CMA Evolution
% Strategy: A Tutorial". Termination criteria is taken from A. Auger and
% N. Hansen, "A restart CMA evolution strategy with increasing population
% size".

% The function takes the following inputs:
% initP: Vector containing initial guesses for parameters
% modelfct: A wrapper function that takes the parameters as input, runs the
%   model and returns a fitness function (e.g. sum of squared error or
%   similar).
% my_rank: MPI variable, containing the (integer) rank of the worker.
% num_ranks: MPI variable, containing the total amount of workers.

% Additional options:
% 'filename': Name of the saved workspace. NOTE: '.mat' is appended
%   automatically. Defaults to CMAES_WS.
% 'stopfitness': The objective function value for which the method is
%   considered converged. Defaults to 1e-10.
% 'adaptiveStep': True or false, to toggle the adaptive step method, which
%   uses additional negative weights to shift the mean.
% 'adaptiveCov': True or false, to toggle the use of additional
%   negative weights to adapt the covariance matrix. Recommended on.
% 'pmin' and 'pmax': The approximate bounds of the parameter space, used to
%   calculate the initial step size. Can be on the form of vectors with
%   individual values, or scalars. Will improve the performance/avoid
%   local minima better with a good value. NOTE: Constraint handling is NOT
%   included in this script as it is easier and more flexible to implement
```

```
%   in the wrapper function (modelfct). The given bounds are NOT
%   constraints, but rather scale.
% 'nRestarts': Number of restarts. The population size increases for each
%   restart. If a solution is found that gives a fitness value below
%   'stopfitness', the loops are terminated immediately. Defaults
%   to 0 restarts, and is more useful for sequential versions of the CMAES
%   method.
% 'popfactor': Factor with which the population increases each restart.
%   Defaults to 2. Between 2 and 5 is good.

%%  INITIALIZATION
inputs = inputParser; % Instance inputParser class for varargin
inputs.addParameter('filename','CMAES_WS',@(x) ischar(x) || isstring(x));
inputs.addParameter('stopfitness',1e-10,@(x) isnumeric(x)); % stop if fitness < stopfitnes
inputs.addParameter('adaptiveStep',1,@(x) x == 0 || x == 1); % Toggle adaptive step
inputs.addParameter('adaptiveCov',1,@(x) x == 0 || x == 1); % Toggle adaptive cov
inputs.addParameter('pmin',NaN,@(x) isnumeric(x));
inputs.addParameter('pmax',NaN,@(x) isnumeric(x));
inputs.addParameter('nRestarts',0,@(x) isnumeric(x)); % Number of restarts
inputs.addParameter('popfactor',2,@(x) isnumeric(x) && x > 0); % Factor for
                                        % population increase per restart
inputs.parse(varargin{:});

filename = inputs.Results.filename;
stopfitness = inputs.Results.stopfitness;
adaptiveStep = inputs.Results.adaptiveStep;
adaptiveCov = inputs.Results.adaptiveCov;
pmin = inputs.Results.pmin;
pmax = inputs.Results.pmax;
nRestarts = inputs.Results.nRestarts;
popfactor = inputs.Results.popfactor;

Master_rank = 0;    % Master_rank is an MPI variable, but it is always 0,
                    % and can therefore be defined here instead of having
                    % to be used as an imput argument.

% Initialize solution struct
solution = struct;
for i = 1:nRestarts+1
    solution(i).fitness = inf;
    solution(i).params = zeros(length(initP),1);
end

% Initialize counters
restartcount = 0;   % Number of restarts, given as output
restartNum = 0;     % Internal count of restarts
totalEvals = 0;     % Total number of evaluations over all restarts
p = zeros(length(initP),1);
fitness = inf;      % Initializing fitness
runtime = 0;        % Initializing runtime
totalRuntime = tic;
%% BEGIN RESTART LOOP
while restartNum <= nRestarts
restartflag = 0;

% INPUT PARAMETERS
N = length(initP); % Number of dimensions/parameters to estimate
```

```
if isrow(initP) % initial parameter guess
    xmean = initP';
else
    xmean = initP;
end

% coordinate wise standard deviation (step-size)
if (pmin < pmax)
    sigma = 0.3*mean(pmax - pmin);
else
    sigma = 0.5; % Decent for search spaces of magnitude 10^0 to 10^1
end
alpha_cov = 2;



% STRATEGY PARAMETER SETTING: SELECTION
if num_ranks < 4+floor(3*log(N))
    lambda = ceil((4+floor(3*log(N)))/num_ranks)*num_ranks ...% population size, offspring
        *popfactor^restartNum;    % Increase population size each restart
else
    lambda = num_ranks*popfactor^restartNum;
end

mu = floor(lambda/2);

wTmp = log((lambda+1)/2) - log(1:lambda); %Initial (un-normalized) weights


mueff = sum(wTmp(1:mu))^2 / sum(wTmp(1:mu).^2);%Effective selection mass
muNegeff = sum(wTmp(mu+1:lambda))^2 / sum(wTmp(mu+1:lambda).^2);

c1 = alpha_cov / ((N + 1.3)^2 + mueff);   %Learning rate for rank-one
                                            %update of C
cmu = min([1 - c1, alpha_cov * (mueff - 2 + 1/mueff) / ...
    ((N + 2)^2 + alpha_cov*mueff/2)]);     %Learning rate for rank-mu
                                            %update of C

alpha_mu = 1 + c1/cmu;                        %The alphas are multipliers
alpha_mu_eff = 1 + 2*muNegeff/(mueff + 2);    %for scaling the negative
alpha_posDef = (1 - c1 - cmu)/(N*cmu);       %weights

posWeights = (1/sum(wTmp(wTmp>0))) * wTmp(wTmp>0)';
negWeights = (min([alpha_mu, alpha_mu_eff, alpha_posDef]) /...
                (abs(sum(wTmp(wTmp<0))))) * wTmp(wTmp<=0)';


switch adaptiveStep % Whether to use negative weights for adjusting mean or not
    case 0
        nMeanWeights = mu;
        meanWeights = posWeights;
    case 1
        nMeanWeights = lambda;
        meanWeights = [posWeights; negWeights./10];
end

switch adaptiveCov % Whether to use negative weights for adjusting C or not
```

```
    case 0
        nCovWeights = mu;
        covWeights = posWeights;
    case 1
        nCovWeights = lambda; % The negative weights are defined in the
                              % generation loop
end

% STRATEGY PARAMETER SETTING: ADAPTATION
cc = (4+mueff/N) / (N+4 + 2*mueff/N); % time constant for cumulation for C
cs = (mueff+2)/(N+mueff+5); % t-const for cumulation for sigma control
damps = 1 + 2*max(0, sqrt((mueff-1)/(N+1))-1) + cs; % damping for sigma

% INITIALIZE DYNAMIC (INTERNAL) STRATEGY PARAMETERS AND CONSTANTS
pc = zeros(N,1); % evolution path for C
ps = zeros(N,1); % evolution path for sigma
B = eye(N); % B defines the coordinate system
D = eye(N); % diagonal matrix D defines the scaling
C = B*D*(B*D)'; % covariance matrix

% INITIALIZING STOPPING CRITERIA-RELATED VARIABLES AND ARRAYS
stopeval = 1000 * lambda; % maximum amount of evaluations. Will rarely be
                          %triggered as the other termination criteria
                          %will be triggered first.
tolfun = 1e-13;    % lowest tolerance of range of objective function values
fitnessHist = [];  % objective function value history
solutionHist = [];  % solution history (unused, just interesting to have)
evaluationTimes = [];   % Array of evaluation times. Also just interesting.
nGenHist = 10 + ceil(30*N/lambda); % number of generations to consider
tolX = 1e-12;       % Lowest tolerance of search space
condNum = 1e14; % Maximum condition number in covariance matrix
bestfitness = inf;

eigeneval = 0;
chiN = N^0.5*(1-1/(4*N)+1/(21*N^2)); % expectation of ||N(0,I)||


counteval = 0;

for i = parsteps(1:num_ranks) % Initializing random seed
    t=clock();
    seed=t(6) * 1000 * my_rank;
    rng(seed);
end

% First generation
arz = randn(N,lambda); % standard normally distributed vector
arx = xmean + sigma * (B*D * arz); % add mutation

%% GENERATION LOOP
while counteval < stopeval
% GENERATE AND EVALUATE OFFSPRING
% Resetting the variables for reduction() reasons
arfitness = zeros(1,lambda);
evaltimes = zeros(1,lambda);

for k = parsteps(1:lambda)
```

```
    evaltimer = tic;
    arfitness(k) = feval(modelfct, arx(:,k)); % evaluate model
    evaltimes(k) = toc(evaltimer);
end

arfitness = reduction('+',arfitness);
arfitness = arfitness';


evaluationTimes(:,end+1) = reduction('+',evaltimes);
counteval = counteval+lambda;
totalEvals = totalEvals+lambda;

% SEQUENTIAL OPERATIONS
if my_rank == Master_rank   % Master_rank
    % Sort by fitness and compute weighted mean into xmean
    [arfitness, arindex] = sort(arfitness); % minimization
    zmean = arz(:,arindex(1:nMeanWeights))*meanWeights; % == D^-1*B'*(xmean-xold)/sigma
    xmean = xmean + sigma * (B*D * zmean); % recombination

    % Cumulation: Update evolution paths
    ps = (1-cs)*ps + (sqrt(cs*(2-cs)*mueff)) * (B * zmean);
    hsig = norm(ps)/sqrt(1-(1-cs)^(2*counteval/lambda))/chiN < 1.4+2/(N+1);
    pc = (1-cc)*pc + hsig * sqrt(cc*(2-cc)*mueff) * (B*D*zmean);

    if adaptiveCov == 1 % Rescaling vector lengths associated with negative weights
        covWeights = [posWeights; ...
            negWeights*N/norm(B*inv(D)*B'* ...
            (B*D * arz(:,arindex(1:lambda))))^2];
    end

    % Adapt covariance matrix C
    C = (1-c1-cmu) * C ... % old matrix
        + c1 * (pc*pc' ... % plus rank one update
        + (1-hsig) * cc*(2-cc) * C) ... % minor correction
        + cmu ... % plus rank mu update
        * (B*D*arz(:,arindex(1:nCovWeights))) ...
        * diag(covWeights) * (B*D*arz(:,arindex(1:nCovWeights)))';

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

    % Update B and D from C
    if counteval - eigeneval > lambda/(c1+cmu)/N/10 % to achieve O(N^2)
        eigeneval = counteval;
        C=triu(C)+triu(C,1)'; % enforce symmetry
        [B,D] = eig(C); % eigen decomposition, B==normalized eigenvectors
        D = diag(sqrt(diag(D))); % D contains standard deviations now
    end

    if arfitness(1) < solution(restartNum+1).fitness % Update best solution
        solution(restartNum+1).fitness = arfitness(1);
        solution(restartNum+1).params = arx(:, arindex(1));
    end


    % UPDATE FITNESS HISTORY
```

```
    fitnessHist(end+1) = arfitness(1);          % fitness history
    solutionHist(:,end+1) = arx(:,arindex(1)); % solution history (unused,
                                                %just interesting to have)

    % STOPPING/RESTARTING CRITERIA, from Auger & Hansen (2005)
    nGen = counteval/lambda;
    if nGen > nGenHist
        if range(fitnessHist(end-nGenHist:end-1)) == 0 || ... % Zero range
                isnan(range(fitnessHist(end-nGenHist:end-1))) % If all solutions are inf
            restartflag = 1;
        elseif range([fitnessHist(end-nGenHist:end-1), arfitness]) < tolfun
            restartflag = 1;    % Function value range too small
        elseif all([sigma*abs(pc), sigma*sqrt(diag(C))]) < tolX
            restartflag = 1;    % Search space too small
        elseif xmean == xmean + 0.1*sigma*B(:,1+floor(mod(nGen,N))) ...
                *D(1+floor(mod(nGen,N)),1+floor(mod(nGen,N)))
            restartflag = 1;    % Zero movement in a principal direction
        elseif xmean == xmean + 0.2*sigma*sqrt(diag(C))
            restartflag = 1;    % Zero movent in all principal directions?
                                % I don't know how this criterion will
                                % ever be satisfied without already
                                % satisfying the prior criterion. Left in
                                % because it's in Auger & Hansen (2005)
        elseif cond(C) > condNum
            restartflag = 1;    % Condition number too large
        end
    end



    % Escape flat fitness
    if arfitness(1) == arfitness(ceil(0.7*lambda))
        sigma = sigma * exp(0.2+cs/damps);
    end
    bestfitness = arfitness(1);

    arz = randn(N,lambda); % standard normally distributed vector
    arx = xmean + sigma * (B*D * arz); % add mutation

    save([filename,'.mat']); % Save workspace

end % if, sequential code on master rank

% Spread relevant variables to all ranks
bestfitness = NMPI_Bcast(bestfitness,1,Master_rank);
restartflag = NMPI_Bcast(restartflag,1,Master_rank);

arx1d = reshape(arx, N*lambda, 1); % Convert to 1D array
arx1d = NMPI_Bcast(arx1d,N*lambda,Master_rank);
arx = reshape(arx1d, [N,lambda]);

% Break, if fitness is good enough, or termination criteria is fulfilled
if bestfitness <= stopfitness
    restartNum = nRestarts + 1; % End restart loop
    break;
elseif restartflag == 1
    restartcount = restartcount + 1;
```

```
    restartNum = restartNum + 1; % End generaton loop, allow for restarts
    break;
end

end % while, generation loop

end % while, restart loop

runtime = toc(totalRuntime);
end
```

Appendix D

Multistart Code

D.1 CMAES_multistart.m

This code utilizes both the intensification and the multistart functionalities. The outputs of the function are irrelevant, as the method will rarely have the time to terminate, and the workspaces that are saved every generation is analyzed directly instead.

```
function [p, fitness, details]=CMAES_multistart(initP, modelfct, ...
    my_rank, batch_size, batch_number, initSigma, varargin)
% A multistart CMA-ES implementation intended for parameter estimation.
% Returns solution (parameters), fitness value for solution and runtime
% (given in number of function evaluations).

% The underlying functionality is taken from Hansen, N. "The CMA Evolution
% Strategy: A Tutorial". Termination criteria is taken from A. Auger and
% N. Hansen, "A restart CMA evolution strategy with increasing population
% size".

% The function takes the following inputs:
% initP: Vector containing initial guesses for parameters
% modelfct: A wrapper function that takes the parameters as input, runs the
%   model and returns a fitness function (e.g. sum of squared error or
%   similar).
% my_rank: MPI variable, containing the (integer) rank of the worker.
% batch_size: Number of ranks in the batch
% batch_number: Which batch this rank is part of
% initSigma: Initial step size

% Additional options:
% 'filename': Name of the saved workspace. NOTE: '.mat' is appended
%   automatically. Defaults to CMAES_WS.
% 'stopfitness': The objective function value for which the method is
%   considered converged. Defaults to 1e-10.
% 'adaptiveStep': True or false, to toggle the adaptive step method, which
%   uses additional negative weights to shift the mean.
% 'adaptiveCov': True or false, to toggle the use of additional
```

```
%   negative weights to adapt the covariance matrix. Recommended on.
% 'pmin' and 'pmax': The approximate bounds of the parameter space, used to
%   calculate the initial step size. Can be on the form of vectors with
%   individual values, or scalars. Will improve the performance/avoid
%   local minima better with a good value. NOTE: Constraint handling is NOT
%   included in this script as it is easier and more flexible to implement
%   in the wrapper function (modelfct). The given bounds are NOT
%   constraints, but rather scale.
% 'nRestarts': Number of restarts. The population size increases for each
%   restart. If a solution is found that gives a fitness value below
%   'stopfitness', the loops are terminated immediately. Defaults
%   to 0 restarts, and is more useful for sequential versions of the CMAES
%   method.
% 'popfactor': Factor with which the population increases each restart.
%   Defaults to 2. Between 2 and 5 is good.

%%  INITIALIZATION
inputs = inputParser; % Instance inputParser class for varargin
inputs.addParameter('filename','CMAES_WS',@(x) ischar(x) || isstring(x));
inputs.addParameter('stopfitness',1e-10,@(x) isnumeric(x)); % stop if fitness < stopfitnes
inputs.addParameter('adaptiveStep',1,@(x) x == 0 || x == 1); % Toggle adaptive step
inputs.addParameter('adaptiveCov',1,@(x) x == 0 || x == 1); % Toggle adaptive cov
inputs.addParameter('pmin',NaN,@(x) isnumeric(x));
inputs.addParameter('pmax',NaN,@(x) isnumeric(x));
inputs.addParameter('nRestarts',0,@(x) isnumeric(x)); % Number of restarts
inputs.addParameter('popfactor',2,@(x) isnumeric(x) && x > 0); % Factor for
                                        % population increase per restart
inputs.parse(varargin{:});

filename = inputs.Results.filename;
stopfitness = inputs.Results.stopfitness;
adaptiveStep = inputs.Results.adaptiveStep;
adaptiveCov = inputs.Results.adaptiveCov;
pmin = inputs.Results.pmin;
pmax = inputs.Results.pmax;
nRestarts = inputs.Results.nRestarts;
popfactor = inputs.Results.popfactor;

% Initialize solution struct
solution = struct;
for i = 1:nRestarts+1
    solution(i).fitness = inf;
    solution(i).params = zeros(length(initP),1);
end

% Initialize counters
restartcount = 0;   % Number of restarts, given as output
restartNum = 0;     % Internal count of restarts
totalEvals = 0;     % Total number of evaluations over all restarts
p = zeros(length(initP),1);
fitness = inf;      % Initializing fitness
runtime = 0;        % Initializing runtime
tic
%% BEGIN RESTART LOOP
while restartNum <= nRestarts
restartflag = 0;
intensificationCounter = 0;
```

```
% INPUT PARAMETERS
N = length(initP); % Number of dimensions/parameters to estimate
if isrow(initP) % initial parameter guess
    xmean = initP';
else
    xmean = initP;
end

% coordinate wise standard deviation (step-size)
sigma = initSigma;
alpha_cov = 2;



% STRATEGY PARAMETER SETTING: SELECTION
if batch_size < 4+floor(3*log(N))
    lambda = ceil((4+floor(3*log(N)))/batch_size)*batch_size ...% population size, offspri
        *popfactor^restartNum;    % Increase population size each restart
else
    lambda = batch_size*popfactor^restartNum;
end

mu = floor(lambda/2);

wTmp = log((lambda+1)/2) - log(1:lambda); %Initial (un-normalized) weights


mueff = sum(wTmp(1:mu))^2 / sum(wTmp(1:mu).^2);%Effective selection mass
muNegeff = sum(wTmp(mu+1:lambda))^2 / sum(wTmp(mu+1:lambda).^2);

c1 = alpha_cov / ((N + 1.3)^2 + mueff);   %Learning rate for rank-one
                                            %update of C
cmu = min([1 - c1, alpha_cov * (mueff - 2 + 1/mueff) / ...
    ((N + 2)^2 + alpha_cov*mueff/2)]);     %Learning rate for rank-mu
                                            %update of C

alpha_mu = 1 + c1/cmu;                        %The alphas are multipliers
alpha_mu_eff = 1 + 2*muNegeff/(mueff + 2);    %for scaling the negative
alpha_posDef = (1 - c1 - cmu)/(N*cmu);       %weights

posWeights = (1/sum(wTmp(wTmp>0))) * wTmp(wTmp>0)';
negWeights = (min([alpha_mu, alpha_mu_eff, alpha_posDef]) /...
                (abs(sum(wTmp(wTmp<0))))) * wTmp(wTmp<=0)';


switch adaptiveStep % Whether to use negative weights for adjusting mean or not
    case 0
        nMeanWeights = mu;
        meanWeights = posWeights;
    case 1
        nMeanWeights = lambda;
        meanWeights = [posWeights; negWeights./10];
end

switch adaptiveCov % Whether to use negative weights for adjusting C or not
    case 0
```

```
        nCovWeights = mu;
        covWeights = posWeights;
    case 1
        nCovWeights = lambda; % The negative weights are defined in the
                              % generation loop
end

% STRATEGY PARAMETER SETTING: ADAPTATION
cc = (4+mueff/N) / (N+4 + 2*mueff/N); % time constant for cumulation for C
cs = (mueff+2)/(N+mueff+5); % t-const for cumulation for sigma control
damps = 1 + 2*max(0, sqrt((mueff-1)/(N+1))-1) + cs; % damping for sigma

% INITIALIZE DYNAMIC (INTERNAL) STRATEGY PARAMETERS AND CONSTANTS
pc = zeros(N,1); % evolution path for C
ps = zeros(N,1); % evolution path for sigma
B = eye(N); % B defines the coordinate system
D = eye(N); % diagonal matrix D defines the scaling
C = B*D*(B*D)'; % covariance matrix

% INITIALIZING STOPPING CRITERIA
stopeval = 300*32; % Should not be necessary, but an alternative is
            % 1000*N^2*popfactor^restartNum;
tolfun = 1e-13;    % lowest tolerance of range of objective function values
fitnessHist = [];  % objective function value history
solutionHist = [];  % solution history (unused, just interesting to have)
evaluationTimes = [];   % Array of evaluation times. Also just interesting.
nGenHist = 10 + ceil(30*N/lambda); % number of generations to consider
tolX = 1e-12;
condNum = 1e14; % Maximum condition number in covariance matrix
bestfitness = inf;
intensificationGeneration = [0]; % Keeping track of when intensification
                                 % happens, for later analysis

eigeneval = 0;
chiN = N^0.5*(1-1/(4*N)+1/(21*N^2)); % expectation of ||N(0,I)||


counteval = 0;

for i = myparsteps(1:lambda,my_rank,batch_size,batch_number) % Initializing seed
    t=clock();

    seed=t(6) * 1000 * my_rank;  % Seed with the second part of the clock array.

    rng(seed);
end

%% GENERATION LOOP
while counteval < stopeval
% GENERATE AND EVALUATE OFFSPRING
% Resetting the variables for reduction() reasons
arz = zeros(N,lambda);
arx = arz;
arfitness = zeros(1,lambda);
evaltimes = zeros(1,lambda);

for k = myparsteps(1:lambda,my_rank,batch_size,batch_number)
```

```
    evaltimer = tic;
    arz(:,k) = randn(N,1); % standard normally distributed vector
    arx(:,k) = xmean + sigma * (B*D * arz(:,k)); % add mutation

    arfitness(k) = feval(modelfct, arx(:,k), intensificationCounter); % evaluate model
    evaltimes(k) = toc(evaltimer);
end
arfitness = myreduction(batch_size,batch_number,arfitness);
arfitness = arfitness';

arz1d=reshape( arz , N*lambda , 1 ); % reshape to 1d array
arz1d=myreduction (batch_size,batch_number, arz1d);  % reduce all elements
arz = reshape(arz1d,[N,lambda]); % shape back to 2d array

arx1d=reshape( arx , N*lambda , 1 ); % reshape to 1d array:
arx1d=myreduction (batch_size,batch_number,arx1d);  % reduce all elements
arx = reshape(arx1d,[N,lambda]); % shape back to 2d array

evaluationTimes(:,end+1) = myreduction(batch_size,batch_number,evaltimes);

counteval = counteval+lambda;
totalEvals = totalEvals+lambda;

% SEQUENTIAL OPERATIONS
if my_rank == batch_size*batch_number   % Master_rank
    % Sort by fitness and compute weighted mean into xmean
    [arfitness, arindex] = sort(arfitness); % minimization
    zmean = arz(:,arindex(1:nMeanWeights))*meanWeights; % == D^-1*B'*(xmean-xold)/sigma
    xmean = xmean + sigma * (B*D * zmean); % recombination



    % Cumulation: Update evolution paths
    ps = (1-cs)*ps + (sqrt(cs*(2-cs)*mueff)) * (B * zmean);
    hsig = norm(ps)/sqrt(1-(1-cs)^(2*counteval/lambda))/chiN < 1.4+2/(N+1);
    pc = (1-cc)*pc + hsig * sqrt(cc*(2-cc)*mueff) * (B*D*zmean);

    if adaptiveCov == 1 % Rescaling vector lengths associated with negative weights
        covWeights = [posWeights; ...
            negWeights*N/norm(B*inv(D)*B'* ...
            (B*D * arz(:,arindex(1:lambda))))^2];
    end

    % Adapt covariance matrix C
    C = (1-c1-cmu) * C ... % old matrix
        + c1 * (pc*pc' ... % plus rank one update
        + (1-hsig) * cc*(2-cc) * C) ... % minor correction
        + cmu ... % plus rank mu update
        * (B*D*arz(:,arindex(1:nCovWeights))) ...
        * diag(covWeights) * (B*D*arz(:,arindex(1:nCovWeights)))';

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

    % Update B and D from C
    if counteval - eigeneval > lambda/(c1+cmu)/N/10 % to achieve O(N^2)
        eigeneval = counteval;
```

```
    C=triu(C)+triu(C,1)'; % enforce symmetry
    [B,D] = eig(C); % eigen decomposition, B==normalized eigenvectors
    D = diag(sqrt(diag(D))); % D contains standard deviations now
end

if arfitness(1) < solution(restartNum+1).fitness % Update best solution
    solution(restartNum+1).fitness = arfitness(1);
    solution(restartNum+1).params = arx(:, arindex(1));
end


% UPDATE FITNESS HISTORY
fitnessHist(end+1) = arfitness(1);          % fitness history
solutionHist(:,end+1) = arx(:,arindex(1));  % solution history
                                %(unused, just interesting to have)

% STOPPING/RESTARTING CRITERIA
nGen = counteval/lambda;
if nGen > nGenHist
    if range(fitnessHist(end-nGenHist+1:end)) == 0 || ...
            isnan(range(fitnessHist(end-nGenHist+1:end))) % If all solutions are inf
        restartflag = 1;
    elseif range([fitnessHist(end-nGenHist+1:end), arfitness]) < tolfun
        restartflag = 1;
    elseif all([sigma*abs(pc), sigma*sqrt(diag(C))]) < tolX
        restartflag = 1;
    elseif xmean == xmean + 0.1*sigma*B(:,1+floor(mod(nGen,N))) ...
            *D(1+floor(mod(nGen,N)),1+floor(mod(nGen,N)))
        restartflag = 1;
    elseif xmean == xmean + 0.2*sigma*sqrt(diag(C))
        restartflag = 1;
    elseif cond(C) > condNum
        restartflag = 1;
    end
end


% Intensification: Following good coding practice, this is taken from an
% earlier script for model fitting: A first order regression is
% performed on the fitness history, and if the slope is insignificant
% according to a t-test, the search is intensified. The computational
% cost of doing this operation every generation is negligible compared
% to the cost of regular model evaluations.

if (intensificationGeneration(end) + floor(1.5*nGenHist) < nGen) && ...
        (~any(fitnessHist(end-floor(1.5*nGenHist)+1:end) == inf))
    y = fitnessHist(end-floor(1.5*nGenHist)+1:end)';
    X = [ones(floor(1.5*nGenHist),1) (1:floor(1.5*nGenHist))'];
    [n_X,p_X] = size(X);
    b = X\y;
    e = X*b - y;
    SSE = e'*e; % Sum of squared errors
    DOF_SSE = n_X-p_X;  % Degrees of freedom for sum of squared errors
    MSE = SSE/DOF_SSE; % Mean Square Error
    covb = MSE * inv(X'*X); % Covariance matrix for b

    ttest = tinv(0.975,DOF_SSE);
```

```
        b_u = b + sqrt(diag(covb))*ttest; % Upper confidence interval
        if b_u(2) > 0
            intensificationCounter = intensificationCounter + 1;
            intensificationGeneration(intensificationCounter+1) = nGen;
        end
    end


    % Escape flat fitness
    if arfitness(1) == arfitness(ceil(0.7*lambda))
        sigma = sigma * exp(0.2+cs/damps);
    end
    bestfitness = arfitness(1);

    save([filename,'.mat']);

end % if, sequential code on master rank

% Spread relevant variables to all ranks

xmean = myspread(batch_size, batch_number, xmean);
sigma = myspread(batch_size, batch_number, sigma);
bestfitness = myspread(batch_size, batch_number, bestfitness);
restartflag = myspread(batch_size, batch_number, restartflag);
intensificationCounter = myspread(batch_size, batch_number, intensificationCounter);

B1d = reshape(B, N*N, 1); % Convert to 1D array
B1d = myspread(batch_size, batch_number, B1d);
B = reshape(B1d, [N,N]);

D1d = reshape(D, N*N, 1); % Convert to 1D array
D1d = myspread(batch_size, batch_number, D1d);
D = reshape(D1d, [N,N]);

 % Break, if fitness is good enough
if bestfitness <= stopfitness
    restartNum = nRestarts + 1; % End restart loop
    break;
elseif restartflag == 1
    restartcount = restartcount + 1;
    restartNum = restartNum + 1;
    break;
end

end % while, generation loop

end % while, restart loop

runtime = toc;
%% End, manage outputs


if temp_my_rank == temp_master_rank
    [~, ind] = sort([solution.fitness]);    % Find the best solution

    fitness = solution(ind(1)).fitness; % Return the best fitness value
    p = solution(ind(1)).params;        % Return the best solution
```

```
end % if, master rank
fitness = myspread(batch_size, batch_number, fitness);
p = myspread(batch_size, batch_number, p);

details.runtime = runtime;      % Return runtime
details.evals = totalEvals;     % Return total evaluations
details.restarts = restartcount;% Return total restarts
details.nRanks = batch_size;     % Return number of ranks
details.parEvaluations = totalEvals/batch_size;  % Return number of parallel
                                                 % evaluation "chunks"

end
```

D.2 myparsteps.m

```
function psteps=myparsteps(isteps,my_rank,batch_size,batch_number)

% Edited by Haakon Eng Holck in 2018, for use in double parallel for
% loops: MPI communication restricted to a subgroup of MPI_COMM_WORLD.
% Allows for independent subgroups ("batches") of ranks to perform parsteps
% (e.g. limiting a batch consisting of ranks 8-15 to perform an individual
% parallel loop)

   istart=isteps(1);
   iend=isteps(length(isteps));
   lsteps=((iend-istart+1)/batch_size);
   if lsteps<1.3
      lsteps=1;
   else
      lsteps=ceil(lsteps);
   end

      if my_rank==batch_number*batch_size
         pstart=istart;
         pend=lsteps;
      else
         pstart=(my_rank-batch_number*batch_size)*lsteps+1;
         if (my_rank-batch_number*batch_size)==batch_size-1
             pend=iend;
         else
             pend=((my_rank-batch_number*batch_size)+1)*lsteps;
         end
      end %if
      if pstart>iend||pend>iend
          pstart=1;
          pend=0;
      end
      psteps=pstart:pend;

end%function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% NMPI
% John Floan
% NTNU - IT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2015 NORWEGIAN UNIVERISTY OF SCIENCE AND TECHNOLOGY
%
% IN NO EVENT SHALL MIT BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
% SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF
% THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF MIT HAS BEEN ADVISED OF THE
% POSSIBILITY OF SUCH DAMAGE.
%
% NTNU SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTIES INCLUDING,
% BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS
% FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
%
% THIS SOFTWARE IS PROVIDED "AS IS," NTNU HAS NO OBLIGATION TO PROVIDE
% MAINTENANCE, SUPPORT, UPDATE, ENHANCEMENTS, OR MODIFICATIONS.
%
```

D.3 myreduction.m

```
function array=myreduction(batch_size, batch_number, inarray)
% :Reduction function for parallel
% Reduce the variable(s) varargin with operator operatorin
% Operators: '+','*','MAX','MIN'
% NTNU John Floan.
% Build on NMPI.
% Copyright 2015 Norwegian University of Science and Technology

% Edited by Haakon Eng Holck in 2018, for use in double parallel for
% loops: MPI communication restricted to a subgroup of MPI_COMM_WORLD.
% Now only uses '+' operation, as the intended purpose is to collect
% information saved in array cells together in a single array on the
% (functional) master rank.
% Uses point-to-point communication which is slightly inefficient, but
% good enough for the purpose of this function.

global my_rank

  nvar=length(inarray);

  if nvar==1
     array=inarray;
     temparray = 0;
  else
     array=zeros(nvar,1);
     temparray = array;
     for ii=1:nvar
           array(ii)=inarray(ii);
     end
   end
  for k = 1:batch_size-1
      if my_rank == batch_size*batch_number+k
          NMPI_Send(array,nvar,batch_size*batch_number);
      elseif my_rank == batch_size*batch_number
```

```
          temparray = NMPI_Recv(nvar,batch_size*batch_number+k);
          array = array+temparray;
      end
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% NMPI
% John Floan
% NTNU - IT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2015 NORWEGIAN UNIVERISTY OF SCIENCE AND TECHNOLOGY
%
% IN NO EVENT SHALL MIT BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
% SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF
% THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF MIT HAS BEEN ADVISED OF THE
% POSSIBILITY OF SUCH DAMAGE.
%
% NTNU SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTIES INCLUDING,
% BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS
% FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
%
% THIS SOFTWARE IS PROVIDED "AS IS," NTNU HAS NO OBLIGATION TO PROVIDE
% MAINTENANCE, SUPPORT, UPDATE, ENHANCEMENTS, OR MODIFICATIONS.
```

D.4 myspread.m

```
function array=myspread(batch_size, batch_number, inarray)

% Edited by Haakon Eng Holck in 2018, for use in double parallel for
% loops: MPI communication restricted to a subgroup of MPI_COMM_WORLD.
% Spreads array to all ranks in same subgroup ("batch").
% Uses point-to-point communication which is slightly inefficient, but
% good enough for the purpose of this function.

global my_rank

  nvar=length(inarray);

  if nvar==1
     array=inarray;
     temparray = 0;
  else
     array=zeros(nvar,1);
     temparray = array;
     for ii=1:nvar
           array(ii)=inarray(ii);
     end
   end
  for k = 1:batch_size-1
      if my_rank == batch_size*batch_number
          NMPI_Send(array,nvar,batch_size*batch_number+k);
      elseif my_rank == batch_size*batch_number+k
          array = NMPI_Recv(nvar,batch_size*batch_number);
      end
```

```
  end
end
  %array=NMPI_Bcast(array,length(array),Master_rank);%,my_rank);
%   if nvar==1
%      varargout{1}=outarray;
%   else
%
%      for ii=1:nvar
%            varargout{ii}=outarray(ii);
%      end
%   end
%
% end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% NMPI
% John Floan
% NTNU - IT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2015 NORWEGIAN UNIVERISTY OF SCIENCE AND TECHNOLOGY
%
% IN NO EVENT SHALL NTNU BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
% SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF
% THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF NTNU HAS BEEN ADVISED OF THE
% POSSIBILITY OF SUCH DAMAGE.
%
% NTNU SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTIES INCLUDING,
% BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS
% FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
%
% THIS SOFTWARE IS PROVIDED "AS IS," NTNU HAS NO OBLIGATION TO PROVIDE
% MAINTENANCE, SUPPORT, UPDATE, ENHANCEMENTS, OR MODIFICATIONS.
```

Appendix E

Results from the Multistart Run

Table E.1 shows the parameters and corresponding fitness for each of the batches in the multistart run on the RNAP model. The fitnesses are unnaturally low, as they are the fitnesses of the best (and therefore, the "luckiest") individual in the last generation. Evaluating the same solutions again is likely to lead to slightly higher values.

Table E.1: The parameters and corresponding fitnesses returned by the different batches in the multistart run. Batch 26 did not manage to find a solution that did not break a constraint.

Batch	$k_{TL,A}$	$k_{TL,\omega}$	k_{DNA}	k_C	kS	Fitness
1	1.0334	1.3735	2.8047	2.4268	1.1668	0.1674
2	2.1000	0.0427	2.5683	1.3702	0.0546	0.3060
3	0.4927	0.9304	1.7706	0.4743	0.5960	0.3959
4	0.7300	1.3300	2.6622	1.8589	1.3494	0.1626
5	1.2235	2.0414	2.9678	1.2797	1.3558	0.1597
6	1.5388	1.5330	1.9517	1.8740	0.1753	0.6068
7	1.6161	1.2867	2.3408	1.6909	0.3582	0.6066
8	0.6223	0.1822	2.2853	2.1230	0.8589	0.1728
9	1.6508	1.8113	2.7695	1.4047	0.6228	0.3965
10	2.5006	0.6539	2.2531	1.7968	0.0434	0.5670
11	1.9835	0.6712	2.6028	1.6712	0.2073	0.6123
12	0.2860	0.6269	2.3178	0.3348	1.3990	0.1663
13	1.1561	1.7927	2.4078	2.7313	0.7673	0.1699
14	2.5875	0.8870	2.8338	2.5556	0.1446	0.6128
15	0.8368	0.1015	2.7050	0.3075	0.9906	0.1606
16	1.7972	0.1154	2.4145	2.3881	0.1267	0.2995
17	0.9521	1.2744	2.3736	0.2525	0.7664	0.3900
18	0.3036	0.6943	0.9976	1.2618	0.2730	0.8150
19	2.0519	1.8913	2.7138	0.6303	0.3565	0.6152
20	2.2349	1.7247	2.5760	1.9084	0.2037	0.6075
21	0.8022	0.3223	2.2295	0.9583	0.6294	0.4287
22	0.6139	0.4889	1.6458	2.3352	0.3301	0.6072
23	1.9416	0.1914	2.7904	1.5682	0.2334	0.4194
24	0.7450	0.2937	2.7962	0.3756	1.2265	0.1529
25	2.2231	0.4386	1.9869	1.4197	0.0447	0.5292
26	1.5596	0.8310	2.5309	2.1881	1.6911	N/A
27	2.2879	1.7330	2.7464	2.3769	0.2195	0.6057
28	1.4068	1.7815	2.9596	1.3092	1.0780	0.1646
29	1.0609	1.3466	2.8764	0.9752	1.2045	0.1782
30	1.7236	0.5930	2.2970	1.1815	0.1734	0.6116
31	0.8889	0.6109	2.5931	0.8995	0.8727	0.3824
32	1.3678	1.4658	2.8599	0.3311	0.8752	0.3783