Applying Machine Learning to Reduce Systems of Ordinary Differential Equations

Using gradient descent and particle swarm optimisation to find ε-lumping matrices for systems of ordinary differential equations

Project Report

Group d503e20

Aalborg University
Computer Science
5th semester

Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Applying Machine Learning to Reduce Systems of Ordinary Differential Equations

Theme:
Theory-Driven Data Analysis and Modeling

Project Period:
Fall Semester 2020

Project Group:
Group d503e20

Participant(s):
Kristian Ødum Nielsen
Mark Feng van Kints
Nicklas Slorup Johansen
Rasmus Grønkjær Tollund

Supervisor(s):
Max Tschaikowski

Number of Pages: 73

Date of Completion:
March 22, 2021

Abstract:

A system of ordinary differential equations can be reduced by a lumping matrix. Lumping matrices are difficult to generate; therefore, this project utilises machine learning to find them. In particular, gradient descent and particle swarm optimisation (PSO) are utilised to minimise the mean squared error from an ε-lumping. Several gradient descent step size methods and PSO parameters are explored in order to maximise the chance of finding good lumping matrices.
The implementations of gradient descent and PSO are asserted on biochemical models in order to evaluate their performance. The results show that using machine learning generally does not outperform current tools for finding lumpings. Although, in some cases, machine learning provided the best lumpings.

Preface

This report was written by the project group 'd503e20' on the fifth term of Department of Computer Science at Aalborg University. The study period was September 2020 - January 2021 and the report was handed in December 18, 2020. The theme of the project is 'Theory-Driven Data Analysis and Modeling'.

This report will utilise the Vancouver reference style. Numbers within brackets refer to numbered entries in the bibliography. The first reference that appears in the text is the first entry in the bibliography. If the citation is placed after the last period in a section then this citation applies to the whole section.

We would like to extend our thanks to our supervisor Max Tschaikowski for excellent supervision during the project.

Aalborg University, March 22, 2021

Contents

Chapter 1

Introduction

Computational models that express systems that change through time can be formulated by ordinary differential equations (ODEs). An example of this is systems of chemical reactions such as signaling pathways, protein-protein interaction networks and genetic cascades [1].

Such systems contain information on a set of species and their chemical reactions. A model of these systems given by ODEs inform on how the species' quantity changes through time. Here, every species has a corresponding variable in the system of ODEs indicating the species quantity. Analysis or running simulations of a system of ODEs requires heavy computation if the model is complex. Therefore, an exact model reduction as a pre-analysis step is of the user's interest. This reduction takes a system of ODEs as input and outputs a smaller system of ODEs which still describes the behavior of the original model. Following the example of a system of chemical reactions, the aggregated model will contain fewer species. Additionally, the reduced system can give insights into the original system. Aggregatable structures, previously oblivious to the modeller, can show up after the reduction.

Given a system of ODEs, species which behaviour can be described as a sum, can be aggregated to a single variable instead. Consequently, an aggregated model becomes simpler and thus analysis and simulation less complex. This variable aggregation method is called lumping. The focus of this project is to utilise machine learning methods to find these lumpings.

In the second chapter of this report, Chapter 2 Problem Analysis, the problem of lumping and its utilisation is discussed. Herein, the problem statement is defined.

In Chapter 3 Preliminaries, the necessary considerations for defining a cost function and aspects related to it are discussed.

In Chapter 4 Gradient Descent, the first machine learning algorithm and associated methods and theory are described. Later in the chapter, the findings about the implementations are provided and analysed upon.

Likewise Chapter 5 Particle Swarm Optimisation, defines the general particle swarm optimisation algorithm. Additionally, the chapter carries out an optimisation of PSO parameters.

In Chapter 6 Evaluation, an analysis of the performance of some chosen algorithms from gradient descent and particle swarm optimisation is described. This is done by using them to aggregate biochemical models.

Next, the entire project is concluded in Chapter 7 Conclusion. Wherein, the fulfilment of the problem statement is evaluated.

Finally, additional ideas and optimisation considerations are described in Chapter 8 Future Work.

Chapter 2

Problem Analysis

To lump a system of ODEs, it is necessary to find out which ODEs can be aggregated together. While a guess might be viable for small systems, it will be infeasible for large-scale systems. This chapter will discuss the theory behind lumping and explore how to find a lumping for a given system of ODEs using machine learning.

2.1 SIR Model

The SIR model is a compartmental epidemiological model that describes species in three states: susceptible, infected and recovered. Each species has the associated ODEs

$$\begin{aligned} \dot{S} &= -\beta IS \\ \dot{I} &= \beta IS - \gamma I \\ \dot{R} &= \gamma I, \end{aligned} \tag{2.1}$$

where β is the contact rate and γ is the recovery rate [2]. $\dot{S}$, $\dot{I}$ and $\dot{R}$ in these ODEs represents the derivative of S, I and R with respect to time, respectively. The specifics of how and why this model works is not relevant to this project.

There also exists an expanded version of this model that tries to relax the assumption that all species are homogeneous. It introduces groupings of species and then assumes

that each group is homogeneous. This gives the new model

$$\begin{aligned} \dot{S}_k &= -S_k \sum_{l}^{K} \beta_k l I_l \\ \dot{I}_k &= S_k \sum_{l}^{K} \beta_k l I_l - \gamma_k I_k \\ \dot{R}_k &= \gamma_k I_k, \end{aligned} \tag{2.2}$$

where K is the amount of groupings and k is some grouping [3]. Each group has its own recovery rate and there are specific contact rates between each group.

The expanded version of SIR is a good candidate for evaluating lumpings because the complexity can easily be controlled by increasing K. Moreover, the recovery and contact rates can be randomly generated. Throughout the report, a SIR model with β_k and γ_k being random numbers uniformly distributed in $[0, 1]$ and $K = 3$ will be used to evaluate different machine learning algorithms. This model will be referred to as the evaluation SIR model.

2.2 Background

Given a system described by a set of ODEs, they can be represented as a function vector

$$\dot{x} = f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}, \tag{2.3}$$

where $\dot{x}$ is the derivative of x with respect to t (often time) and the functions $f_1, f_2, \cdots, f_n$ are the ODEs for $\dot{x}_1, \dot{x}_2, \cdots, \dot{x}_n$, respectively. [4]

A lumping matrix, M, is used reduce the dimensions of the system to $\hat{n}$ variables, where $\hat{n} < n$. Therefore, M is an $\hat{n} \times n$ matrix. The product of M and x yields the smaller system

$$y = Mx, \quad \hat{f}(y) = Mf(x), \tag{2.4}$$

where $\hat{f}$ is the lumped function vector.

There must exist a generalized right inverse such that $M\overline{M} = I_{\hat{n}}$ [4]. For full row

rank matrices the right inverse is calculated by [5]

$$\overline{M} = M^T(MM^T)^{-1}. \tag{2.5}$$

Intuitively, a lumping is possible when two or more variables' behaviour can be described by the behaviour of the sum of them. This is reflected by the entries in the lumping matrix describing the coefficient of each variable in the smaller system.

Finding the values of a lumping matrix that provides a successful lumping is not easy. It can be done with educated guesses with integer coefficients. However, this is infeasible for larger systems and the lumping matrix can even contain irrational numbers [4]. Therefore, it is in ones interest to create an algorithm for computing the values of M.

2.2.1 Exact Lumpability

Exact lumpability is when a lumped system completely describes the behavior of the original system. For a lumping to be exact, it must satisfy

$$Mf(x) = Mf(\overline{M}Mx) \tag{2.6}$$

for all x. [4]

Example

Given the system described by

$$\dot{x} = f(x) = \begin{bmatrix} x_1x_3 + x_3 \\ x_2x_3 + x_3 \\ x_3 \end{bmatrix}, \tag{2.7}$$

a lumping matrix

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

can be used to aggregate $f_1(x)$ and $f_2(x)$ together. In the new system, y is used instead of x as the variable, which is

$$y = Mx = \begin{bmatrix} x_1 + x_2 \\ x_3 \end{bmatrix} \tag{2.9}$$

and the lumped version of $f(x)$ is

$$\begin{aligned} Mf(x) &= \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1x_3 + x_3 \\ x_2x_3 + x_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1x_3 + x_3 + x_2x_3 + x_3 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} (x_1 + x_2)x_3 + 2x_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1y_2 + 2y_2 \\ y_2 \end{bmatrix}. \end{aligned} \tag{2.10}$$

This new system has aggregated x_1 and x_2 into a single variable y_1. For an exact lumping, the new system must be describable by only y variables. To prove that this lumping is an exact lumping, the right hand side of the equality, $Mf(\overline{M}Mx)$, is then shown to be equal to the left hand side. The right inverse of the M is given by

$$\overline{M} = \begin{bmatrix} a & 0 \\ 1-a & 0 \\ 0 & 1 \end{bmatrix}, \quad 0 \leq a \leq 1. \tag{2.11}$$

This can then be used to find

$$\overline{M}Mx = \begin{bmatrix} a & 0 \\ 1-a & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a \cdot x_1 + a \cdot x_2 \\ (1-a)x_1 + (1-a)x_2 \\ x_3 \end{bmatrix}. \tag{2.12}$$

Applying the function f on this yields

$$f(\overline{M}Mx) = \begin{bmatrix} a(x_1 + x_2)x_3 + x_3 \\ (1-a)(x_1 + x_2)x_3 + x_3 \\ x_3 \end{bmatrix}. \tag{2.13}$$

Finally, we have the reduced system $\hat{f}$ as

$$\begin{aligned} \hat{f}(y) = Mf(\overline{M}y) = Mf(\overline{M}Mx) &= \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a(x_1 + x_2)x_3 + x_3 \\ (1-a)(x_1 + x_2)x_3 + x_3 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} a(x_1 + x_2)x_3 + (1-a)(x_1 + x_2)x_3 + 2x_3 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} (x_1 + x_2)x_3 + 2x_3 \\ x_3 \end{bmatrix}, \end{aligned} \tag{2.14}$$

where a and $1 - a$ sum to 1. This new system is shown to be the same that was found in Equation 2.10. Therefore, it is an exact lumping.

2.2.2 ε-Lumpability

In some cases, it might not be possible to find an exact lumping matrix. In these cases, an ε-lumping might be sufficient instead.

ε-Lumpability is when a lumped system describes the original system within some ε error. Instead of equality between the systems, their relationship can be described as

$$\left\| Mf(x) - Mf(\overline{M}Mx) \right\|_2 \leq \varepsilon \tag{2.15}$$

for all x.

In this case, the smaller system is only an approximation of the original. If the Euclidean norm of the difference between the evaluations of the reduced system and the original system is smaller than a small ε error then the approximation might be good enough.

This presents an optimisation problem by continuously tweaking the lumping matrix such that the ε error is minimised. This problem is fit for machine learning. The problem is cast as a cost function optimisation problem. Therefore, the next section will focus on how to define such a cost function based on the ε error.

2.3 Cost Function

In Machine Learning, a cost function measures how well a model is performing on some given data. The cost function quantifies the difference between the expected values and the actual values as a single real number. As for finding a lumping matrix, it is sought to minimise the ε error of an ε-lumping. Therefore, the error will make the basis for the cost function.

Mean squared error (MSE) is a commonly used cost function that measures the average squared difference between the expected values and the actual values. If Y is a vector of the actual values, $\hat{Y}$ is a vector of the expected values and n is the length of the two vectors then the mean squared error can be written as

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2, \tag{2.16}$$

where $\frac{1}{n}\sum_{i=1}^{n}$ is the mean and $(Y_i - \hat{Y}_i)^2$ are the squared errors. Because MSE takes the

square of each term, it heavily weights the outliers more than the terms with a low error.

From Equation 2.15, it is sought to minimise the ε error in order to find a good lumping matrix. From here on, this equation will be referred to as

$$\theta(M, x) = \left\| Mf(x) - Mf(\overline{M}Mx) \right\|. \tag{2.17}$$

Ideally, the cost function would be the average for all x values for a given M. This would be the hypervolume of the lumping error function averaged over the hypervolume of x,

$$\mathcal{C}(M) = \frac{\int_{-k}^{k} \cdots \int_{-k}^{k} \theta(M, x)^2 dx_1 \cdots dx_n}{(2k)^n}, \tag{2.18}$$

where k is a very large number in which the integral is taken and n is the amount of dimensions. The complexity of this function renders it infeasible to compute. Therefore, it is desired to find an approximation instead. Here, Monte Carlo integration provides such approximation. Monte Carlo integration uses uniformly distributed random numbers to estimate an integral of a function [6].

Applying MSE to the lumping error function and using Monte Carlo integration by generating random values for x yields the cost function,

$$\mathcal{C}(M) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \theta(M, x)^2, \tag{2.19}$$

where $\mathcal{X}$ is a set of random vectors, $\mathcal{X} \subset \mathbb{R}^n$. In this version, the hypervolume size $(2k)^n$ cancels out. By increasing the cardinality of $\mathcal{X}$, the approximation of the cost function will approach the optimal cost function [7]. Furthermore, by using MSE the cost function will reward consistency over irregularity.

2.4 Purpose of Lumping

When lumping a system of ODEs, it is important to consider what the purpose of lumping the system is. Some models facilitate a total aggregation such that all species are described in a single variable. While this greatly improves simulation speed, the reduced system is incapable of providing any insight on specific species.

2.4.1 Guarantee of Lumping the SIR Model

For any SIR model, there always exist at least two ways of finding an exact lumping. First, for the SIR model, the sum of all ODEs is always zero. Therefore, if the system

is lumped into a single species by equal weighting of all species then it is simply a zero-function.

Secondly, any SIR model can be exactly lumped by lumping away the Recovery species. The reason for this is that Recovery is not used to calculate anything in the system of ODEs.

For the SIR model, the input is values for Susceptible, Infected and Recovered. These values are encapsulated symbolically by

$$\zeta = \begin{bmatrix} S \\ I \\ R \end{bmatrix}. \tag{2.20}$$

To prove that any SIR model can be exactly lumped this way, we show that the equation

$$Mf(\zeta) = Mf(\overline{M}M\zeta) \tag{2.21}$$

holds. Here, f is the system of ODEs described in Section 2.1 SIR Model.

First, we calculate $Mf(\zeta)$ by

$$Mf(\zeta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -\beta IS \\ \beta IS - \gamma I \\ \gamma I \end{bmatrix} = \begin{bmatrix} -\beta IS \\ \beta IS - \gamma I \end{bmatrix} \tag{2.22}$$

Here, the last row, γI, departures since it is the differential equation for Recovery.

Next, we calculate $Mf(\overline{M}M\zeta)$ by

$$Mf(\overline{M}M\zeta) = Mf\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \zeta \right) = Mf\left(\begin{bmatrix} S \\ I \\ 0 \end{bmatrix} \right) = \begin{bmatrix} -\beta IS \\ \beta IS - \gamma I \end{bmatrix} \tag{2.23}$$

Note, $\overline{M}M\zeta$ effectively removes Recovery by replacing it with a 0. Therefore, Recovery is no longer input to f. However, this has no influence on the reduced system since Recovery does not partake in any of the differential equations. Therefore, when calculating

$$Mf\left(\begin{bmatrix} S \\ I \\ 0 \end{bmatrix} \right), \tag{2.24}$$

it yields the same system as $Mf(\zeta)$.

2.4.2 Preserving Species

Consider an intention of analysing a single specific species of some SIR model where all other species are irrelevant. If this SIR model is simply lumped without further restrictions then the species to be analysed might get aggregated. The ODEs in the reduced system are sums of ODEs from the original system and addition is not bijective. Therefore, the values of the species in the original system are unobtainable if they have been aggregated.

The user is interested in preserving a species in order to analyse it in the reduced system. For this to happen, the lumping matrix must have a row that produces an exact copy of the species. Consider a system of ODEs

$$\dot{x} = f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_j(x) \\ \vdots \\ f_n(x) \end{bmatrix}, \tag{2.25}$$

where the species j with ODE $f_j(x)$ must be preserved. In order to preserve the species $f_j(x)$, the lumping matrix in Equation 2.26 is defined. Here, one row must have all zeroes except in the j-th column in order for the reduced system to preserve species j. The ordering of the rows in the lumping matrix makes no difference. Therefore we simply choose the first row.

$$M = \begin{bmatrix} 0 & 0 & \cdots & 1 & \cdots & 0 \\ m_{2,1} & m_{2,2} & \cdots & m_{2,j} & \cdots & m_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ m_{\hat{n},1} & m_{\hat{n},2} & \cdots & m_{\hat{n},j} & \cdots & m_{\hat{n},n} \end{bmatrix}. \tag{2.26}$$

By calculating the reduced system,

$$\hat{f}(y) = Mf(\overline{M}Mx) = \begin{bmatrix} f_j(\overline{M}Mx) \\ m_{2,1} \cdot f_1(\overline{M}Mx) + m_{2,2} \cdot f_2(\overline{M}Mx) + \cdots + m_{2,n} \cdot f_n(\overline{M}Mx) \\ \vdots \\ m_{\hat{n},1} \cdot f_1(\overline{M}Mx) + m_{\hat{n},2} \cdot f_2(\overline{M}Mx) + \cdots + m_{\hat{n},n} \cdot f_n(\overline{M}Mx) \end{bmatrix}, \tag{2.27}$$

it can be seen that the species j is present in the reduced system $\hat{f}(y)$ and thus it is preserved.

Now, aggregations can occur by all the other rows of the lumping matrix to speed up simulations. Hereby, the species j is analysable in the reduced system and the system is smaller.

2.5 Problem Statement

As previously explained, a model can beneficially be lumped into a reduced model. A successfully lumped model provides an advantage in speeding up simulations and analysis. Additionally, a lumping might provide insight into the model.

As explained in Section 2.2 Background, guessing a lumping matrix for large systems is infeasible. Therefore, creating an algorithm, and more specifically a machine learning model, that finds a lumping matrix is of great interest. This leads to the problem statement:

> *How can machine learning be utilised to find a lumping matrix for an arbitrary system of ODEs?*

2.5.1 Subproblems

To solve the minimisation problem, we use unsupervised machine learning. Supervised learning is not feasible, as the correct lumping matrices are unknown for the vast majority of ODEs.

As previously explained, ε-lumpability provides a metric for error that fits nicely into a cost function in machine learning. Solving the minimisation of the cost function yields a lumping matrix with the lowest ε error.

For minimisation, we consider two machine learning methods. First, the gradient-descent method will be examined; a method that calculates and utilises the gradient of the cost function. The other approach is particle swarm optimisation which is a gradient-free method. These approaches define the following subproblems:

- How can gradient descent be used to minimise the ε error.
- How can particle swarm optimisation be used to minimise the ε error.

Chapter 3

Preliminaries

In this chapter, the necessary considerations before using cost function optimisation will be discussed. These problems center around the cost function and its Monte Carlo mean estimation. First, a problem of small values in the lumping matrix causing a small cost is addressed. Secondly, Monte Carlo integration is defined for calculating the cost function. Lastly, a problem of discontinuous functions is addressed.

3.1 Matrix Normalisation

The cost function should be a measure of how bad the lumping is. Two lumping matrices that perform equivalent lumpings should also yield the same cost. However, consider some matrix M and then a scaled version δM. The cost of δM is given by

$$\mathcal{C}(\delta M) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \left\| (\delta M) f(x) - (\delta M) f(\overline{(\delta M)}(\delta M) x) \right\|. \tag{3.1}$$

Then factoring out δ

$$\mathcal{C}(\delta M) = \delta \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \left\| M f(x) - M f(\overline{(\delta M)}(\delta M) x) \right\|. \tag{3.2}$$

Now, we can show that $\overline{(\delta M)}(\delta M) = \overline{M}M$ by using definition of right inverse

$$\begin{aligned}\overline{(\delta M)}\delta M &= \delta(\delta M^T)((\delta M)(\delta M)^T)^{-1}M \\ &= \delta^2 M^T(\delta^2 MM)^T)^{-1}M \\ &= \delta^2 M^T\delta^{-2}(MM^T)^{-1}M \\ &= M^T(MM^T)^{-1}M \\ &= \overline{M}M.\end{aligned} \tag{3.3}$$

Therefore, this will cancel out in the cost function which becomes

$$\mathcal{C}(\delta M) = \delta\frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}}\left\|Mf(x) - Mf(\overline{M}Mx)\right\|, \tag{3.4}$$

and then

$$\mathcal{C}(\delta M) = \delta\mathcal{C}(M). \tag{3.5}$$

This means that any scaling of M will also directly scale the cost function. This is a problem because one could simply choose any M and then scale it by a very small value to obtain a low cost. Therefore, the matrix must undergo some normalisation.

Initially this might be solved by using the euclidean norm of the matrix. However, this will allow the optimisation algorithm to choose the ODE that generates the least cost, and then increase the value of that row, while decreasing all the others. Scaling each row by some value $\delta_0, \delta_1, \cdots, \delta_n$ can be described as multiplying the matrix by the diagonal matrix

$$\Delta = \begin{bmatrix}\delta_0 & 0 & \cdots & 0 \\ 0 & \delta_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \delta_n\end{bmatrix}. \tag{3.6}$$

Similar to above, it can be shown that $\overline{\Delta M}\Delta M = \overline{M}M$. The proof is found in Appendix A Proof that $\overline{\Delta M}\Delta M = \overline{M}M$. This gives the cost function

$$\mathcal{C}(\Delta M) = \frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}}\left\|\Delta(Mf(x) - Mf(\overline{M}Mx))\right\|. \tag{3.7}$$

In this case, the cost function is not directly scaled by Δ since it is a matrix but each $\theta(M, x)$ is. This means that each row can be normalised without changing the actual lumping.

Furthermore, Li et al.[4] has shown that the lumping can be determined only by the

space spanned by the row vectors of the matrix. Therefore, the lumping matrix can be orthonormalised and still be an equivalent lumping.

By applying the normalisations from this section, the search space has been reduced from all real-valued matrices to all orthonormal matrices. While there are still multiple orthonormal matrices that span the same space, this at least reduces the different places to look. This also makes it easier to compare results of different experiments.

The orthonormalisation technique that will be used is modified Gram-Schmidt orthogonalisation. This is a modification of the Gram-Schmidt method in which the previous orthogonalised row-vectors are used to orthogonalise the next, instead of only the original vectors [8]. Additionally, each row is normalized by using the euclidean norm.

In Algorithm 3.1, the algorithm used can be seen. Here, m_{i*} denotes the i-th row of matrix M and $proj_u(v)$ is the projection of vector v onto u and is defined as

$$proj_u(v) = \frac{u \cdot v}{u \cdot u} u, \quad u \neq \vec{0}. \tag{3.8}$$

Algorithm 3.1: The algorithm for modified Gram-Schmidt orthonormalisation.

input :_An $s \times t$ matrix M to be orthonormalised.

output:_An $s \times t$ orthonormal matrix Q.

$q_{1*} \leftarrow \frac{m_{1*}}{||m_{1*}||}$
for $i \leftarrow 2$ **to** s **do**
  $w \leftarrow m_{i*}$
  for $k \leftarrow 1$ **to** i **do**
    $w \leftarrow w - proj_{q_{k*}}(w)$
  $q_{i*} \leftarrow \frac{w}{||w||}$
return Q

3.2 Monte Carlo Sampling

Numerical integration of multiple dimensions introduces a lot of numerical noise. To obtain the same error rate in n dimensions as N_1 samples gave in 1 dimension, it can be expected that

$$N_n = (N_1)^n \tag{3.9}$$

samples are needed. In other words, the sample size scales exponentially with the dimensionality of the function parameter [6]. This means that even if a single dimension could be done in as few as 100 samples then a dimension count of just 5 will require

10^{10} samples. Therefore, the model will scale poorly with dimension size.

As a means to counteract this, we attempted to use the VEGAS algorithm by Le Page [9, 10]. However, it was ultimately discarded because it was not found to be faster than the normal Monte Carlo. In Appendix B VEGAS, a description of how the algorithm works can be seen. The purpose of using VEGAS was to decrease the samples needed for a mean estimation by sampling more around points of larger value. These points have greater influence on the outcome and therefore their errors are more pronounced.

VEGAS did not end up being useful for two reasons. First, it requires a domain partitioning phase in order to find optimal partitions for sampling. It partitions each dimension into partitions with equal hypervolume. This phase is pretty slow because often it requires sampling the function at more points than simply doing the integral. This makes the overhead of the method quite large.

Secondly, for the partitioning to be effective, the function must be formed in such a way that the partitioning is useful. Exactly what this means requires explanation of more details of the algorithm. The important part is that for models like SIR, the VEGAS partitioning makes almost no difference because all of the partitions end up being almost the same size. In conclusion, VEGAS is abandoned and standard Monte Carlo method is still used.

3.3 Discontinuous Functions

Some models have discontinuous functions and therefore they are not integrable. Obviously, Monte Carlo integration does work for non-integrable functions. Often, this discontinuity lies outside of the models domain. However, when evaluating a lumping matrix the $\overline{M}Mx$ operation might transform x outside the domain of the model. Therefore, the evaluation of $f(\overline{M}Mx)$ as part of $\theta(M, x)$ can deviate a lot from the mean. This can prevent the integration from converging.

In Figure 3.1, a graph of the estimation of such a function can be seen. The y-axis shows the estimated integral so far for x amount of samples. Notice how the estimation is converging except for a few points that completely throw it off. It is infeasible to simply gather more samples as even 10 billion is not nearly enough to estimate the mean of this function.

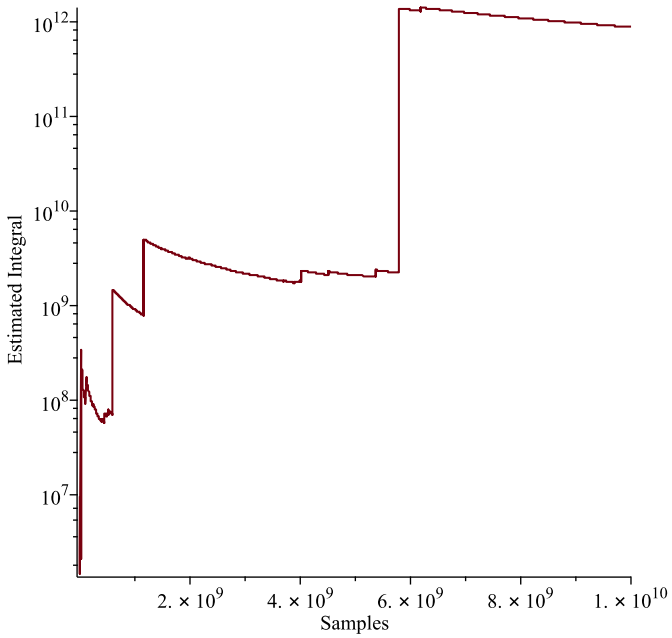

Figure 3.1: Estimated integral as a function of samples.

The most reasonable solution would be to ignore all $\bar{M}Mx$ that lie outside the domain of the model. However, for some lumping matrices this will cause a large amount, or all, of the sampling points to be discarded. If all sampling points are discarded, the mean estimation can never finish. Therefore, this is not viable.

Another approach is to apply a transformation on the function evaluations, such that enormous outliers do not have as much influence. To accomplish this, a modified logistic function is used. This function is

$$\sigma(x) = \frac{2}{1 + e^{-kx}} - 1, \tag{3.10}$$

where k is the growth rate which should be adjusted according to the expected range of the the cost. If chosen too low, it will squeeze the values too much and not provide meaningful results. The logistic function has been modified slightly, such that its range is $(-1; 1)$. Additionally, this modification allows for $\sigma(0) = 0$, which avoids unwanted transformation of small values.

The growth rate is found by calculating the trimmed mean of a small sample. This is done by taking 1000 samples and then calculating the mean of the 50% middle values. The growth rate need only be calculated once per model. The trimmed mean is used to avoid outliers. It cannot be used to calculate the cost in general, because it requires

storing each sample and sorting them. This overhead is too large when sampling many points.

This project will primarily focus on well-behaved functions. However, as the goal is to find a method that works in general, Section 6 Evaluation will also consider some arbitrary models.

Chapter 4

Gradient Descent

Gradient descent is an optimisation method for finding a local minimum of a differentiable function, and is often used to train machine learning models. The intuition is to find a minimum by travelling towards the negative gradient.

For a given multi-variable differentiable function, following the negative gradient always leads to a local minimum. The gradient of f for the point x, which is an n-vector, is computed with

$$\nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_0} f(x) \\ \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}, \tag{4.1}$$

where ∇f contains each of the partial derivatives in a vector [11]. To calculate the gradient of the cost function, the partial derivative of each element in the lumping matrix is used. This will be discussed in Section 4.1 Estimating Derivative.

When the gradient has been calculated, a step is taken towards the negative gradient [11]. There are multiple approaches to calculating the size of the step, these approaches will be explained in greater detail in Section 4.4.

4.1 Estimating Derivative

One way to find the gradient is to use numerical differentiation to compute the partial derivative of each element in the lumping matrix. This will introduce some noise which will be discussed in Section 4.2 Numerical Differentiation. The gradient of the cost

function is found by

$$\nabla \mathcal{C}(M) = \begin{bmatrix} \partial_{m_{1,1}} \mathcal{C}(M) & \partial_{m_{1,2}} \mathcal{C}(M) & \cdots & \partial_{m_{1,n}} \mathcal{C}(M) \\ \partial_{m_{2,1}} \mathcal{C}(M) & \partial_{m_{2,2}} \mathcal{C}(M) & \cdots & \partial_{m_{2,n}} \mathcal{C}(M) \\ \vdots & \vdots & \ddots & \vdots \\ \partial_{m_{\hat{n},1}} \mathcal{C}(M) & \partial_{m_{\hat{n},2}} \mathcal{C}(M) & \cdots & \partial_{m_{\hat{n},n}} \mathcal{C}(M) \end{bmatrix}. \tag{4.2}$$

Here we use ∂_a as shorthand for $\frac{\partial}{\partial a}$.

An estimation of the derivative of the cost function with respect to element i, j in the lumping matrix can be written using simple two-point numerical differential as

$$\partial_{m_{ij}} \mathcal{C}(M) \approx \frac{\left(\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \theta(M + h\mathbb{1}^{ij}, x)^2 \right) - \left(\frac{1}{|\mathcal{X}'|} \sum_{x \in \mathcal{X}'} \theta(M, x)^2 \right)}{h}, \tag{4.3}$$

where $\mathbb{1}^{ij}$ denotes the matrix where element i, j is 1 and all other elements are 0; and h denotes a very small positive number.

Another formula for computing the slope is to calculate the slope of a nearby secant line between two close adjacent points. This is the symmetric derivative

$$\frac{f(x+h) - f(x-h)}{2h}. \tag{4.4}$$

This formula is significantly more accurate as it cancels first-order errors [6]; giving a more accurate approximation of the tangent line. More on this in Section 4.2 Numerical Differentiation. Applying symmetric derivative to the cost function yields

$$\partial_{m_{ij}} \mathcal{C}(M) \approx \frac{\left(\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \theta(M + h\mathbb{1}^{ij}, x)^2 \right) - \left(\frac{1}{|\mathcal{X}'|} \sum_{x \in \mathcal{X}'} \theta(M - h\mathbb{1}^{ij}, x)^2 \right)}{2h}. \tag{4.5}$$

Both X and X' are sets of random vectors and thus might as well be the same. Additionally, it makes sense to use the same random values for both calculations of the slope to ensure uniformity. Therefore, the equation can be written as

$$\partial_{m_{ij}} \mathcal{C}(M) \approx \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \frac{\theta(M + h\mathbb{1}^{ij}, x)^2 - \theta(M - h\mathbb{1}^{ij}, x)^2}{2h}. \tag{4.6}$$

This will effectively calculate the mean slope of θ instead. This is then calculated for each element in M, which makes up the gradient of the cost function.

4.2 Numerical Differentiation

The cost function and its derivative is evaluated using numerical integration and differentiation. This introduces numerical noise and calculation errors. This section will discuss these and explain the countermeasures that have been used.

Recall Equation 4.6 for calculating the derivative of the cost function numerically. It has the problem with numerical noise from Monte Carlo integration as described in Section 3.2 Monte Carlo Sampling. Additionally, there are also the errors introduced by the differentiation itself. When estimating a derivative

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{h} \tag{4.7}$$

two types of errors occur: truncation and rounding [6].

Truncation comes from the higher terms in the Taylor expansion

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \cdots \\ f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) - \frac{1}{6}h^3 f'''(x) + \cdots . \end{aligned} \tag{4.8}$$

When subtraction these from each other and dividing by $2h$, the result is

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{1}{6}h^2 f'''(x) + \cdots . \tag{4.9}$$

Notice how the first order errors, $\frac{1}{2}h^2 f''(x)$, cancel out. This is what makes the symmetric derivative better than the normal one. The higher order terms are ignored because for small values of h their influence will diminish.

Rounding errors are introduced due to the limited precision of the numerical representation used by the computer. When representing x, h and $x + h$, there is a risk of rounding errors due to this imprecision. In this case, x is randomly chosen and thus its exact value will be equal to its effective value. But if h is not chosen properly then x and $x + h$ might not *effectively* differ by h. This can be fixed by choosing h as

$$\begin{aligned} temp &\leftarrow x + h \\ h &\leftarrow temp - x. \end{aligned} \tag{4.10}$$

Since $temp$ is an exact representation using the initial guess of h, when subtracting x from it, a new value of h is achieved which ensures no rounding error [6].

In comparison to the error from Monte Carlo sampling, these rounding and truncation errors are minuscule.

4.3 Algorithm

In Algorithm 4.1, the basic generic gradient descent algorithm can be seen. The lumping matrix is initialised with random values. Then it will keep making descent steps until some satisfaction criteria is fulfilled, as seen on line 7. This can be a predetermined number of epochs, or until it has converged on a local minimum. In each step: first the gradient ∇ is calculated, then a step Δ is determined, and finally the matrix is updated. The gradient is calculated by using the method described in Section 4.1 Equation 4.6. The step length is determined by one of the methods described in Section 4.4.

Algorithm 4.1: Basic gradient descent algorithm.

input :_A function *cost*: $\mathbb{R}^{\hat{n}\times n} \to \mathbb{R}$

output:_A matrix M that minimises the *cost* function

1 initialise matrix M randomly such that $M_{ij} \in [-1, 1]$ for all i, j
2 **repeat**
3 **foreach** *index* i, j *in* M **do**
  `/* Calculate the gradient */`
4 $\nabla \leftarrow$ gradient(*cost*, M)
  `/* Determine a step */`
5 $\Delta \leftarrow$ step(M, ∇)
  `/* Update M by the chosen step */`
6 $M \leftarrow M + \Delta$
7 **until** *satisfaction criteria*
8 **return** M

4.4 Learning Step

The gradient descent method travels in the opposite direction of the gradient in order to converge into a local minimum. Here, a step size and direction must be calculated in order to specify how much to travel in a given iteration. [12]

There is a trade-off between a small step size and a large step size. A large step size might cause jumps over minima or even climb instead of descend. On the contrary, a small step size will more precisely and carefully descend towards the minimum at

the cost of slower convergence. Therefore, it is important to determine an appropriate learning step size.

4.4.1 Constant Step Size

The simplest form of step size is a constant step size. With this approach, gradient descent naively moves with the same proportion of the gradient all the time. Here, a naive learning rate is set, either by intuition or analysis.

A problem of constant step size is dealing with saddle points or in general points with a low gradient. In these points, the constant step size makes travelling with a low gradient very slow. A saddle point is a point where the gradient is zero but it is not a local minimum. This happens e.g. when in one direction the point is a minimum, but in the other it is a maximum. Such a point has a very small gradient and thus a constant step size will perform poorly here. This will lead to taking unnecessary many steps in order to leave the saddle point and continue searching for a local minimum.

4.4.2 Dynamic Step Size

Constant step size has some clear problems when encountering small gradients. Therefore, having a bigger step size at low gradients is beneficial. A simple implementation of this is to upscale the gradient if it is still going in the same direction as last iteration. If the gradient is still the same, it is an indication that the step was too small. If the gradient has changed direction then a minimum has been passed in the given direction and the step size is decreased. The algorithm for dynamic step size is shown in Algorithm 4.2. $\nabla_{i,j}^{t}$ is the gradient for element $m_{i,j}^{t}$ of the t-th iteration. $lr_{i,j}^{t}$ denotes the learning rate for this element. $\nabla_{i,j}^{t} \bullet \nabla_{i,j}^{t-1}$ is the dot product of the vectorised matrices, $\nabla_{i,j}^{t}$ and $\nabla_{i,j}^{t-1}$. This is defined as

$$A \bullet B \equiv vec(A) \cdot vec(B), \tag{4.11}$$

where $vec(A)$ is the vectorisation of A, an $n \times m$ matrix, by stacking its column vectors into a single column vector such that

$$vec(A) = \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{n,1} \\ a_{1,2} \\ \vdots \\ a_{m,n} \end{bmatrix}. \tag{4.12}$$

This vectorised matrix dot product is used as an indication of how similiar the direction of the two matrices are. Lastly, the element of the lumping matrix is updated with its respective learning rate.

Algorithm 4.2: Simple dynamic learning rate.

```
if ∇^t_{i,j} • ∇^{t-1}_{i,j} > 0 then
    lr^t_{i,j} ← lr^{t-1}_{i,j} · 1.2
else
    lr^t_{i,j} ← lr^{t-1}_{i,j} · 0.5
m^t_{i,j} ← m^{t-1}_{i,j} + lr^t_{i,j}
```

As a result of up-scaling the learning rates, this method should converge faster than constant step size. Furthermore, it should also converge more precisely as a result of down-scaling the learning rates when nearing the minimum.

4.4.3 Neural Network Methods

Neural networks often use gradient descent. Here, the implementation of gradient descent also requires a method for deciding the step size. Although, the problem of finding a lumping matrix is different to correcting weights in a neural network, the methods of neural network gradient descent might still be useful here.

There exist plenty methods for neural network gradient descent; a chosen few, that are close to state-of-the-art and sufficiently different, will attempted. These methods are RMSProp, Momentum and Adam which will be discussed in the following sections.

These algorithms are often implemented for neural networks for tuning weights and biases. For finding a lumping matrix each element $m_{i,j}$ will correspond a weight of the neural network and the bias will be left out.

RMSProp

A good method for gradient descent of neural networks is RMSProp [13]. The idea is to use a moving average of the gradient for each element in the lumping matrix to keep some momentum and smoothness. Thereby, maintaining velocity and direction such that the next step is similar to its previous step.

First, the mean square moving average of the gradient, denoted by $E(\nabla^2_{i,j})$, is calculated for an element in the lumping matrix

$$E(\nabla^2_{i,j})^t = \beta \cdot E(\nabla^2_{i,j})^{t-1} + (1-\beta) \cdot \left(\nabla^t_{i,j}\right)^2, \tag{4.13}$$

where $\nabla_{i,j}$ is the gradient of element $m_{i,j}$ of the lumping matrix, β denotes how much of the previous mean square is retained (usually set to $0,9$) and t is the iteration number. Note that superscripted t is not a power but the element in the t-th iteration. Next, the element of the lumping matrix is updated with

$$m_{i,j}^t = m_{i,j}^{t-1} - \frac{lr}{\sqrt{E(\nabla_{i,j}^2)^{t-1}} + \varepsilon} \cdot \nabla_{i,j}^{t-1}, \tag{4.14}$$

where lr is the learning rate and ε is a very small number, e.g. $1 \cdot 10^{-8}$, which is just to avoid division by 0.

The moving average dampens changes in the velocity. Therefore, if the gradient suddenly becomes very small, for example in a small and bad local minimum, it still makes a large step over it.

Momentum

The idea of the Momentum method [14] is also to maintain velocity over iterations. This should profit when encountering bad local minima as the momentum will cause it to jump over. The momentum will add up for elements that keep going in the same direction. This is useful in valleys that slowly go toward a local minimum. In these situations, if placed on the wall of the valley the negative gradient points toward the floor of the valley and only little toward the local minimum. But the gradient toward the minimum will be added up due to the momentum and thus dealing with valleys becomes better.

The velocity of momentum in iteration t is given by

$$v_{i,j}^t = \beta^t v_{i,j}^{t-1} + (1 - \beta^t)\nabla_{i,j}^t, \tag{4.15}$$

where

$$\beta^t = \beta^0 + \frac{\beta^f - \beta^0}{f} t, \tag{4.16}$$

where $\beta^0 \in [0;1)$ is the inital value and f the number of iterations, i.e. $\beta^f \in [0;1)$ is the desired final value. β is scaled in order to allow fast changes in velocity in the beginning and slower changes in the velocity in the later stages.

Subsequently, the lumping matrix elements are updated

$$m_{i,j}^t = m_{i,j}^{t-1} - lr \cdot v_{i,j}^t. \tag{4.17}$$

To make Momentum better in the early stages the velocities can be stabilised by

$$v_{i,j}^t \leftarrow \frac{v_{i,j}^t}{1-(\beta^t)^t}. \tag{4.18}$$

Here, the last superscript of $(\beta^t)^t$ is actually a power. Stabilising is done because the moving averages are biased towards 0 in the beginning. After some iterations $(\beta^t)^t \approx 0$ when $\beta \in [0;1)$ and the velocity is thus not stabilised any longer.

Adam

Another adaptive learning rate method is Adam [15]. It is similar to RMSProp in that it tries to accelerate learning. Additionally, it is similar to Momentum as it also keeps momentum. In some way, it tries to combine the two previous methods.

Adam utilises two moving averages, one for the gradient and one for the squared gradient. These are defined as

$$E(\nabla_{i,j})^t = \beta_1 E(\nabla_{i,j})^{t-1} + (1-\beta_1)\nabla_{i,j}^t \tag{4.19}$$

and

$$E(\nabla_{i,j}^2)^t = \beta_2 E(\nabla_{i,j}^2)^{t-1} + (1-\beta_2)\left(\nabla_{i,j}^t\right)^2, \tag{4.20}$$

where $\beta_1, \beta_2 \in [0,1]$ and usually have the values 0.9 and 0.999, respectively. Contrarily to momentum, β_1 and β_2 are constant. The moving averages are stabilised, like in the Momentum method, by

$$E(\nabla_{i,j})^t = \frac{E(\nabla_{i,j})^t}{1-\beta_1} \tag{4.21}$$

and

$$E(\nabla_{i,j}^2)^t = \frac{E(\nabla_{i,j}^2)^t}{1-\beta_2}. \tag{4.22}$$

Note that β_1 and β_2 are to the power of t.

Finally, the elements in the lumping matrix are updated

$$m_{i,j}^t = m_{i,j}^{t-1} - \frac{lr \cdot E(\nabla_{i,j})^t}{\sqrt{E(\nabla_{i,j}^2)^t + \varepsilon}}, \tag{4.23}$$

where ε is a very small number.

4.4.4 Line search

The idea of line search is to compute the optimal step size. This method aims to utilise the calculation of the gradient as much as possible at each iteration. This is an advantage as it is expensive to compute. The objective of line search is to find an $\alpha > 0$ that minimizes

$$\mathcal{C}(M + \alpha d), \tag{4.24}$$

where d is the direction matrix [16], in this case the negative gradient and M is the current lumping matrix. The intuition of line search is to take a single step directly into the minimum in this direction.

Computing the true global minimum is infeasible. Instead, a local minimum is found. It is necessary to find the local minimum in this direction with an iterative approach because the function that is minimised is unknown [16]. Since this is also a minimisation problem, it shares a lot of the problems with the original one. When finding α there is only a single dimension to minimise in and thus allows for simpler minimisation techniques. Minimisation can be done with and without using derivatives. Since computing the derivative is expensive derivative-free methods for minimising in the directions are explored.

Golden Section Search

Golden section search is a method for finding an extremum within an interval of the function, assuming the functions has exactly one minimum in the interval. By identifying 3 points $x_1 < x_2 < x_3$ that satisfy $f(x_1) > f(x_2) < f(x_3)$, it can be seen that a minimum must lie within x_1 and x_3. Then by probing a new point x_4, the value of $f(x_4)$ can determine which section the minimum can be found in. If x_4 is chosen between x_1 and x_2, then if $f(x_4) < f(x_2)$ a minimum must lie between x_1 and x_2, otherwise it must lie between x_2 and x_3. And similarly for when x_4 is between x_2 and x_3. This approach is used iteratively to limit the interval in which the minimum is located. [6]

Golden section search, as the name might suggest, uses the golden ratio to choose the spacing between the points. This can be shown to minimize the worst case possibility for choosing the interval. [6]

As the search section is in the direction of the negative gradient, the initial section size should be chosen such that the function has exactly one minimum in this interval. However, it is not easily calculated how big this section size should be, consequently this becomes a parameter that must be tuned.

In Algorithm 4.3, the algorithm for golden section search adapted to the line search

use-case can be seen. The initial search interval is given by β, which determines the largest distance away from the current point to consider. The two inner points c, d are then placed inside, with $\frac{b-a}{\varphi}$ distance to each outer bound. Given the value of the golden ratio is $\varphi \approx 1.7$, c will be the lower inner point and d the higher. The bounds are decreased until the interval is smaller than a given tolerance. At each iteration it is determined which inner point yields the smallest cost. Then the outer bound furthest from it is replaced by the other inner point. The largest inner point adopts the value of the smallest inner point, and then new point is chosen.

Algorithm 4.3: Golden section search algorithm to find step length.

input :_A cost function $\mathcal{C}$, a matrix M and a gradient ∇
output:_A scalar $\alpha \in [0, \beta]$ that minimizes $\mathcal{C}(M - \alpha\nabla)$

`/* Initialise the bounds */`
$a \leftarrow 0, \quad b \leftarrow \beta$
`/* Initialise inner points */`
$c \leftarrow b - \frac{b-a}{\varphi}, \quad d \leftarrow a + \frac{b-a}{\varphi}$
while $|c - d| > tolerance$ **do**
  `/* Decrease the bounds to the smallest of the two inner points */`
  if $\mathcal{C}(M - c\nabla) < \mathcal{C}(M - d\nabla)$ **then**
    `/* Adopt the new upper bound */`
    $b \leftarrow d$
    $d \leftarrow c$
    $c \leftarrow b - \frac{b-a}{\varphi}$
  else
    $a \leftarrow c$
    $c \leftarrow d$
    $d \leftarrow a + \frac{b-a}{\varphi}$
return $\frac{a+b}{2}$

Choosing a proper search section size is important to ensure that the function is reasonably convex within the section. We developed a simple method for choosing this section size β. Here, the next β after iteration i is given by

$$\beta_{i+1} = \gamma\beta_i + 2(1-\gamma)\alpha_i, \tag{4.25}$$

where α_i is the chosen point this iteration and γ is a constant adoption rate. For β_0 some initial value must be guessed. The intuition of this is that optimally $\beta_i = 2\alpha_i$, such

that α is in the middle of the interval. The γ factor is used to adjust the speed at which the new intervals are chosen, to avoid aberrant points to drastically decrease the section size.

In practice, a slight modification of the algorithm that memorises cost function evaluations is used, such that it is only calculated once for each point.

Quadratic Interpolation Optimisation

Quadratic interpolation optimisation is a method that finds an extremum within an interval for the function, f, assuming that the function has a minimum in that interval. Quadratic interpolation is used to generate the quadratic function that fits three points, $(x_1, f(x_1)), (x_2, f(x_2))$ and $(x_3, f(x_3))$. The three points are identified that satisfy $x_1 < x_2 < x_3$ and $f(x_1) > f(x_2) < f(x_3)$, meaning that the minimum lies between x_1 and x_3. Subsequently, the root x_r of the quadratic function is calculated. The value $f(x_r)$ is used to narrow down the section wherein the minimum can be found; similar to the golden section search approach. This method is applied iteratively find the minimum of f [17].

In Appendix C Root of Quadratic Interpolation, the derivations for finding the minimum of the quadratic function, $q(x)$, can be seen. In summary, $q(x)$ was differentiated and then solved for $q'(x_r) = 0$. Given the interval $x_1 < x_r < x_3$, x_r is given by

$$x_r = \frac{f(x_1)(x_2^2 - x_3^2) + f(x_2)(x_3^2 - x_1^2) + f(x_3)(x_1^2 - x_2^2)}{2[f(x_1)(x_2 - x_3) + f(x_2)(x_3 - x_1) + f(x_3)(x_1 - x_2)]}. \tag{4.26}$$

The value of $f(x_r)$ determines the interval for the next iteration in the same way as golden section search. This continues until a specified tolerance $|x_r - x_2| < \varepsilon$ is satisfied or for a predetermined number of iterations [17]. The size of the search section is chosen as shown in Equation 4.25.

In Algorithm 4.4, quadratic interpolation optimisation adapted to the line search use-case is shown. Same as golden section search, the search interval is given by β. Iteratively, the root of $q(x)$ given by x_r is calculated by Equation 4.26 and its cost, $\mathcal{C}(M - x_r\nabla)$, is compared to $\mathcal{C}(M - x_2\nabla)$ in order to determine if the minimum lies before or after x_r. Based on where the minimum lies the search section is given new bounds.

The search section is iteratively narrowed down until the difference between x_r and

x_2 is within a tolerance or *count* hits *maxIteration*.

Algorithm 4.4: Quadratic interpolation algorithm to find step length.

input :_A cost function $\mathcal{C}$, a matrix M and a gradient ∇
output:_A scalar $\alpha \in [0, \beta]$ that minimizes $\mathcal{C}(M - \alpha\nabla)$

```
/* Initialise the initial points                                    */
```
1 $x_1 \leftarrow 0, \quad x_2 \leftarrow \frac{\beta}{2}, \quad x_3 \leftarrow \beta$
```
/* Calculate the root with Equation 4.26                            */
```
2 $x_r \leftarrow calcRootOfQuadraticFunction(x_1, x_2, x_3)$
3 $count \leftarrow 0$
4 **repeat**
```
    /* Decrease the interval for the search space                   */
```
5 **if** $\mathcal{C}(M - x_r\nabla) < \mathcal{C}(M - x_2\nabla)$ **then**
6 **if** $d < b$ **then**
7 $x_3 \leftarrow x_2$
8 $x_2 \leftarrow x_r$
9 **else**
10 $x_1 \leftarrow x_2$
11 $x_2 \leftarrow x_r$
12 **else**
13 **if** $d < b$ **then**
14 $x_1 \leftarrow x_r$
15 **else**
16 $x_3 \leftarrow x_r$
17 $count \leftarrow count + 1$
18 $x_r \leftarrow calcRootOfQuadraticFunction(x_1, x_2, x_3)$
19 **until** $|x_2 - x_r| \leq tolerance \vee count \geq maxIteration$
20 **return** x_2

4.5 Findings

This section will present the findings of the different implementations of gradient descent. More specifically, the implementations of the different step size methods that were discussed in Section 4.4 Learning Step. First, the step size methods, which dynamically decide a step size, will be analysed. This includes dynamic step size and the methods derived from neural networks. Next, the line search methods using golden

section search and quadratic interpolation optimisation will be analysed.

To quickly sum up the methods before delving into the details, all methods are capable of finding good lumpings and perform quite similarly. Although, the methods derived from neural networks require dynamic scaling of the learning rate in order to achieve a good cost.

4.5.1 Probability Density Function

Initially, an experiment is defined as a complete run of an algorithm, i.e. from a random starting matrix until the given algorithm no longer improves the cost.

To visualise the performance of different optimisation methods, a probability density function (PDF) will be used. Its purpose is to show the likelihood of converging at some cost. To create the probability density functions, 500 experiments of the given algorithm was run with the same parameter values on the evaluation SIR model as described in Section 2.1 SIR Model. For each experiment, the cost is logged which is used to create a PDF.

If the cost of a given lumping matrix is below 10^{-8} then it is considered an exact lumping. Continuing to lower the cost and terminating closer to an exact lumping might require many additional epochs but will provide very small changes to the lumping matrix. Additionally, the information obtained on the model when continuing below the given threshold is minuscule. Also, values lower than the threshold become more prone to numerical noise and an exact lumping might require irrational numbers which cannot be represented by a double in a computer.

4.5.2 Dynamic Step Size Methods

This section will analyse the methods which dynamically decide a step size. This includes dynamic step size, RMSProp, Adam and momentum.

Constant Step Size

The purpose of constant step size is to have a reference method for the other methods. Figure 4.1 shows the PDF for constant step size. This graph shows that the method is not capable of finding any good minimum at all and that it is most likely to converge at a cost around 10^5.

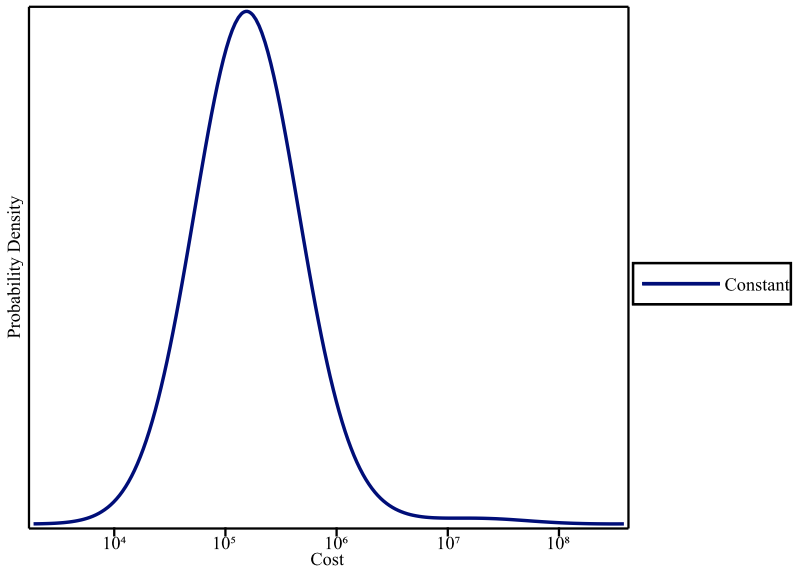

Figure 4.1: PDF of the cost after gradient descent with simple constant step size method with a learning rate of 0.003.

Dynamic Step Size

Despite its simplicity, the dynamic step size performs very well on the evaluation SIR model. Figure 4.2 shows two typical runs for dynamic step size; one that converges at a good cost and one at a bad cost. This shows that dynamic step size is capable of converging into a very good local minimum with a cost just short of $10^{-}20$. Additionally, it converges rather quickly at around ~250 epochs, and achieves a cost below 10^{-8} before 200 epochs, which is considered an exact lumping. Although, this is not the case every time; as the other line demonstrates. The important finding here is that dynamic step size is able to find a good lumping matrix.

Neural Network Methods

RMSProp, Adam and Momentum all perform rather poorly on the evaluation SIR model. Figure 4.3 shows the PDFs of RMSProp, Adam, Momentum and the constant step size method. Here, it is clear that the methods are only on par with the constant step size method and does not provide any exact lumpings.

Lowering the Learning Rate

So far only the dynamic learning rate method is capable of providing an exact lumping. This fact possibly arises from its dynamic scaling of its learning rates, whereas, the other

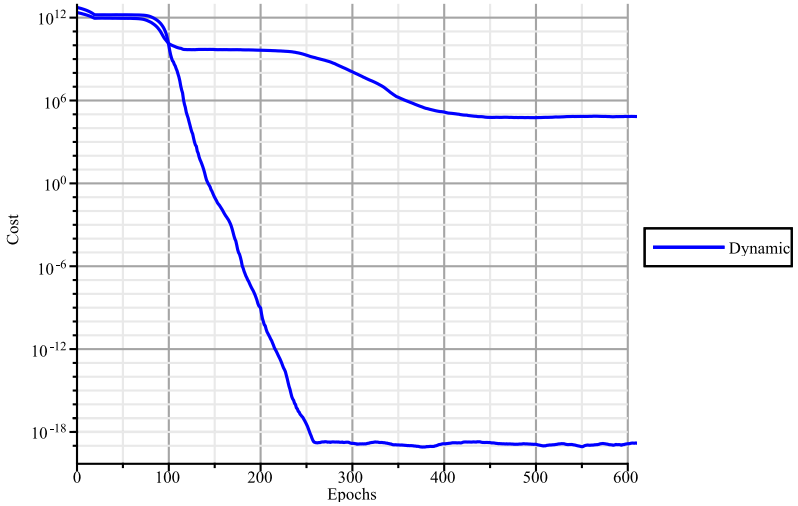

Figure 4.2: Plot of two gradient descent runs using simple dynamic step size method.

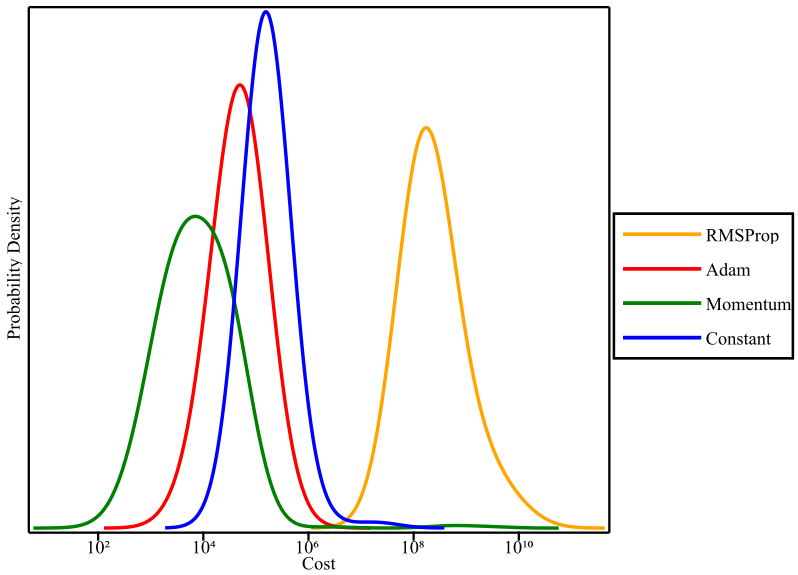

Figure 4.3: PDFs of constant step size, RMSProp, Adam and Momentum; all with a learning rate of 0.003.

Figure 4.4: Constant step size method with dynamic scaling of learning rate. Initial learning rate is 0.003

methods have a constant learning rate.

A small learning rate is required to avoid jumping over a minimum of an element in the lumping matrix. Dynamic step size scales the learning rate if the sign of the gradient has changed from the previous iteration; indicating a jump over a minimum. After scaling the learning rate, a smaller step is taken in the opposite direction; now more accurately approaching the minimum.

Despite a learning rate of 0.003 already appears to be low, it is not low enough to accurately explore the lowest point of a local minimum. Figure 4.4 shows a run of a modified version of the constant step size method. The modification is that after the cost has plateaued, the learning rate is lowered; $lr = lr \cdot 0.1$. The cost is deemed to have plateaued if the all-time best cost has not changed in 100 iterations.

Dynamically lowering the learning rate after plateaus on the neural network methods RMSProp, Adam and Momentum also allows them to provide an exact lumping matrix. When the modified constant step size method finds an exact lumping with a cost less than 10^{-19}, the learning rate is around 10^{-16}. Initialising the method with such small learning rates would cause it to take an infeasible amount of time before converging. Additionally, the method will be susceptible to bad local minimums. Therefore, the method must scale the learning rate dynamically.

4.5.3 Combining Dynamic Step Size Method with Neural Network Methods

This section will attempt to combine the dynamic step size method with the neural network methods. As lowering the learning rate when the cost plateaus unlocks a better cost, it is expected that incorporating dynamic scaling directly into the methods enables them to find a good cost.

Basically, for each neural network method, a modified version is created. First, it performs dynamic scaling of the learning rate and subsequently performs the method specific calculations for deciding a step size and direction.

Table 4.1 shows the ratios of costs better than 10^{-8} after converging. More specifically, it shows that MRMSProp and Mmomentum (the prefix M stands for modified) perform slightly better than the standard dynamic step size method. Oppositely, MAdam performs worse than the standard dynamic step size method.

Method	Ratio of Exact Lumpings
Dynamic Step Size	0.375
MRMSProp	0.389
MAdam	0.228
MMomentum	0.414

Table 4.1: The ratio of exact lumpings for methods that scale learning rates dynamically. The evaluation SIR model is used.

4.5.4 Line Search

Figure 4.5 shows a plot of the cost function for some lumping matrix M in the direction of the negative slope as a function of α. This example is for a very simple ODE system. Here, it can clearly be seen that there is very steep gradient around the 0.155 mark. Therefore, it is easy to jump over it. With line search, the step size is narrowed down to right around this point, and a single step is taken right into the minimum in this direction.

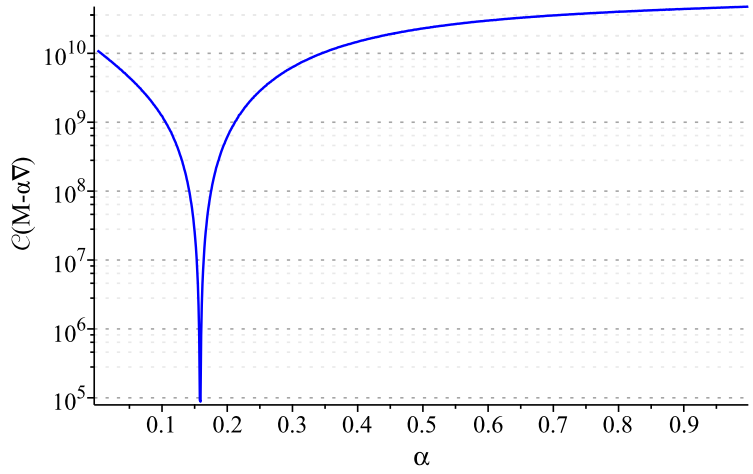

Figure 4.5: Cost function as a function of α.

Comparatively, a fixed step size will only take a step in this direction. It is therefore unlikely that it will hit that small interval, instead of taking either a too large or a too small step.

However, it was found that line search has a similar problem. Choosing a reasonably sized search interval is important. When chosen too small, the target minimum lies outside and thus a too small step is made. When chosen too large, several hills and valleys might be included, and instead of finding the adjacent minimum, one farther away can be found. This other minimum might be worse; therefore, it risks jumping to an unfavorable position. When jumping in this fashion, the slope is no longer followed, and therefore it can no longer be guaranteed to converge to a local minimum.

Calculating the gradient requires evaluating the slope of the cost function $n\hat{n}$ times, where n is the size of the original system and $\hat{n}$ is the size of the reduced system. Since $\hat{n} < n$, the derivative is an $\mathcal{O}(n^2)$ operation.

Golden Section Search

The complexity of golden section search depends on the interval of the search space and tolerance[1]. Golden section search evaluates the cost function several times. Each iteration of golden section search reduces the interval of the search space to $\frac{1}{\varphi}$ of the previous. Therefore, the number of evaluations needed to achieve a given precision p is

$$\#\text{evaluations} = \frac{log(p) - log(\beta)}{-log(\varphi)}. \tag{4.27}$$

[1]The algorithm terminates when its search interval is less than the tolerance.

In other words, the number of evaluations scales logarithmically with p and β[2]. This is not dependent on n, and will therefore not scale with the problem size.

Quadratic Interpolation Optimisation

Quadratic interpolation optimisation evaluates the cost function several times. In some cases, reductions of the interval of the search space can be so minuscule that it takes disproportionately long time before reaching the tolerance. Therefore, it is required to specify a maximum number of interval reductions in order to stop the iterations from taking too long.

Comparison and Drawback

When comparing the two line search methods, the tolerance is set to 10^{-5} and the maximum number of iterations is set to 50 for quadratic interpolation optimisation. Golden section search and quadratic interpolation optimisation share a similar approach to gradient descent and therefore their ratio of exact lumpings are also very similar. As shown on Table 4.2, on average golden section search plateaus 8 seconds faster than quadratic interpolation for all the experiments. However, the average time for the two methods is almost identical for the experiments in which the methods found exact lumpings.

Method	Time	Time at Exact Lumpings
Golden Section Search	71.9s	113.0s
Quadratic Interpolation	79.7s	113.4s

Table 4.2: Average time to plateau for all experiments and average time for experiments with an exact lumping. The evaluation SIR model is used.

The drawback of line search is that it is very susceptible to the noise on the cost function. When the cost has little change in the interval of the search space, the noise causes comparisons of points to become unreliable, thus leading the algorithm to an incorrect minimum. Consequently, line search cannot converge to the exact minimum.

[2] β is the size of the initial search section.

Chapter 5

Particle Swarm Optimisation

Particle swarm optimisation (PSO) is a derivative-free optimisation algorithm. The algorithm is initialised with a number of particles that compose the swarm. The particles' initial positions are random within a given search space. The swarm moves around in the search space where each particle attempts to find a local minimum. The intuition of PSO is to have the particles be affected by each other in order to find a good minimum.

The strength of PSO lies in the fact that multiple minima can be explored. Given long enough execution time and enough particles, the algorithm is capable of finding many minima, if they exist. If multiple minima are found, it is more likely that one of them is a global minimum, but it is not guaranteed [18].

5.1 Basic Algorithm

In the basic version of particle swarm optimization, each step is made according to a velocity. At each iteration the new velocity v is calculated for each particle p [19]

$$p.v_{ij} \leftarrow \omega p.v_{ij} + \phi_p r_p(p.best_{ij} - p.x_{ij}) + \phi_g r_g(g_{ij} - p.x_{ij}), \tag{5.1}$$

where $p.x_{ij}$ is the index ij of the position of the particle p. Factors r_p and r_g are random values in the interval [0;1]. $p.best$ and g are the best known positions of particle p and the swarm, respectively. The equation consist of 3 parts, weighted by the parameters ω, ϕ_p and ϕ_g. These parameters have great influence on the performance on the algorithm and the selection of these parameters will be discussed in Section 5.2 Parameter Optimisation. The first part is the influence of the previous velocity, $p.v_{ij}$. The second part attracts the particle to its own best known position. The last part attracts the particle to the swarm's best position. Finally, the particle's new position is updated using the

calculated velocity,

$$p.x \leftarrow p.x + p.v. \tag{5.2}$$

With many iterations, all particles will be drawn to local minima and are likely to find multiple minima and perhaps the global minimum. Although, the algorithm cannot guarantee to find any minimum at all [18].

The pseudocode for a basic Particle Swarm Optimisation algorithm can be seen in Algorithm 5.1. A swarm is initialised with a specified number of particles, where each particle's position is uniformly given a random matrix within the search-space and so is its velocity. The initial best known position for each particle and the swarm is then set, which is determined by its cost, given function $\mathcal{C}$.

On line 11, the velocity for particle p is modified with Equation 5.1. If the velocity is higher or lower than the threshold v_{max}, the velocity is then capped. After all indices in the velocity for a particle is updated, the particle's position is updated. If the new position, $p.x$, is better than the current best position for the particle, $p.best$, or the swarm's best position, g, they are updated. This is repeated until a criteria is satisfied, which can be a predetermined number of iterations or until the particles have converged to a

minimum.

Algorithm 5.1: Basic particle swarm optimisation algorithm

input :_A function $\mathcal{C}$: $\mathbb{R}^{\hat{n}\times n} \rightarrow \mathbb{R}$

output:_The swarm's best position g that minimizes the cost function

initialise swarm with S particle where $S \in \mathbb{Z}$
foreach *particle* p **do**
    initialise $p.x \in \mathbb{R}^{\hat{n}\times n}$ where each $p.x_{ij}$ is uniformly random in $[b_{lo}, b_{up}]$
    initialise $p.v \in \mathbb{R}^{\hat{n}\times n}$ where each $p.v_{ij}$ uniformly in $[-|b_{up} - b_{lo}|, |b_{up} - b_{lo}|]$
    $p.best \leftarrow p.x$
initialise g with the position of the particle with the lowest cost
while *Condition is not fulfilled* **do**
    foreach *particle* p **do**
        foreach *index* ij **in** $p.v$ **do**
            initialise $r_p, r_g \in [0, 1]$
            $v \leftarrow \omega p.v_{ij} + \phi_p r_p(p.best_{ij} - p.x_{ij}) + \phi_g r_g(g_{ij} - p.x_{ij})$
            $p.v_{ij} \leftarrow max(-v_{max}, min(v_{max}, v))$
        $p.x \leftarrow p.x + p.v$
        if $\mathcal{C}(p.best) < \mathcal{C}(p.x)$ **then**
            $p.best \leftarrow p.x$
            if $\mathcal{C}(g) < \mathcal{C}(p.x)$ **then**
                $g \leftarrow p.x$
return g

5.2 Parameter Optimisation

The PSO algorithm contains multiple parameters that can be fine-tuned. These include the amount of particles, S, in the swarm, the inertia weight ω, the two acceleration coefficients ϕ_p and ϕ_g and the maximum velocity v_{max}. This section will discuss the different parameters and the findings of optimal parameters for the evaluation SIR model mentioned in Section 2.1 SIR Model. This section contains a number of probability density function (PDF) graphs that are created from 500 experiments of PSO with the same parameter values as explained in Section 4.5.1 Probability Density Function.

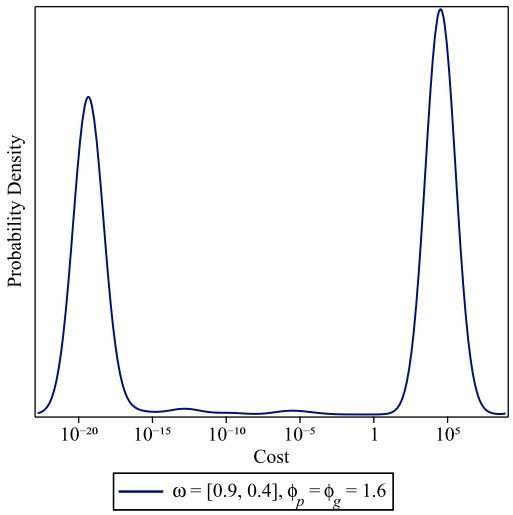

Figure 5.1: PDF of the cost after reaching a local minimum with $\omega = [0.9, 0.4]$.

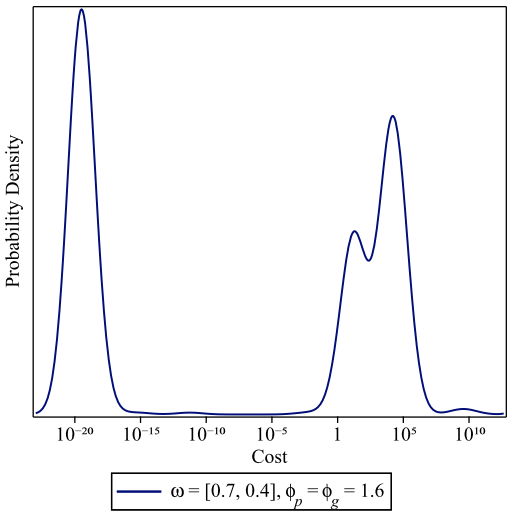

Figure 5.2: PDF of the cost after reaching a local minimum with $\omega = [0.7, 0.4]$.

Inertia Weight

The inertia weight, ω, controls how much the previous velocity influences the current velocity. Consequently, a larger inertia weight promotes global exploration while a smaller inertia weight promotes local exploitation. Instead of a constant inertia weight, a linearly decreasing weight (LDW) is commonly used to have high exploration in the beginning and high exploitation in the end [19]. The new inertia weight, using LDW, can be calculated by

$$\omega_t = \omega_{max} - \frac{\omega_{max} - \omega_{min}}{T_{max}} t, \tag{5.3}$$

where t is the current iteration, T_{max} is the maximum iteration count and ω_{max} and ω_{min} are the maximum and minimum inertia weight, respectively. The values of ω_{max} and ω_{min} are typically set to 0.9 and 0.4, respectively [19]. The intuition of LDW is to start with a high inertia weight to facilitate global exploration to find a minimum closer to the global minimum. The inertia weight is then decreased at each iteration, which will result in the particles exploiting a minimum.

To find an optimal pair of inertia weight parameters, the evaluation SIR model was run for different values for ω_{max} and ω_{min}. Here it was found that the PDF of the cost, after each experiment had reached a local minimum, was similar for experiments where $\omega_{max} > 0.7$, as shown in Figures 5.1 and 5.2. However, when $\omega_{max} = 0.7$, the local minimum was on average found after 1415 iterations, but when $\omega_{max} = 0.9$, it on average took 2102 iterations to find a local minimum. When the value of ω_{max} and ω_{min}

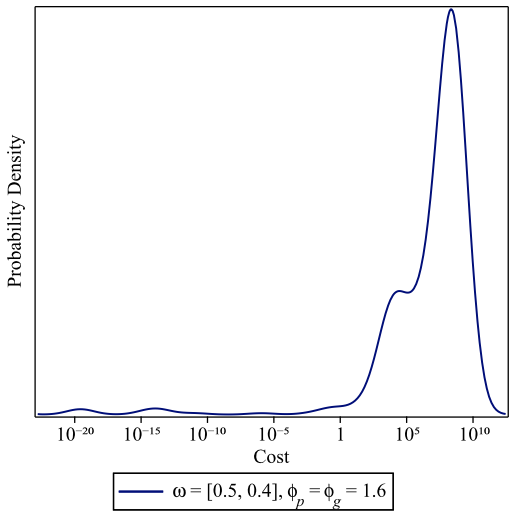

Figure 5.3: PDF of the cost after reaching a local minimum with $\omega = [0.5, 0.4]$.

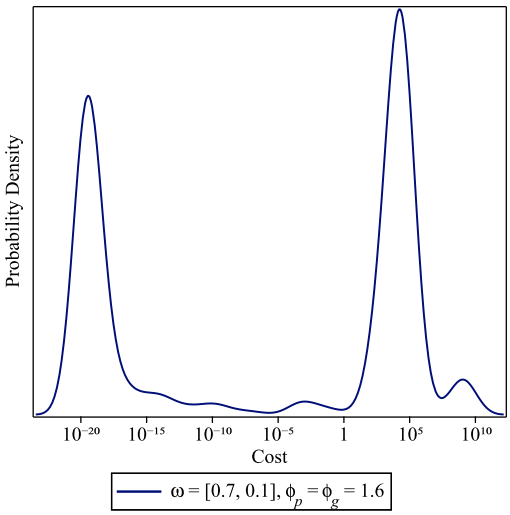

Figure 5.4: PDF of the cost after reaching a local minimum with $\omega = [0.7, 0.1]$.

were set below 0.7 and 0.4, respectively, it was observed that PSO would more frequently get stuck in an unfavorable local minimum, as shown in Figures 5.3 and 5.4. Therefore, the values 0.7 and 0.4 were chosen as default values for ω_{max} and ω_{min}, respectively.

Acceleration Coefficients

The acceleration coefficients ϕ_p and ϕ_g represent the acceleration towards the personal best and global best, respectively. Others have a posteriori found good behaviour with acceleration coefficients $\phi_p = \phi_g = 1.496$ and $\phi_p = \phi_g = 2.05$ [20][21]. Yan et al. [19] have tested decreasing both acceleration coefficients linearly, but they observed that fixed acceleration constants provided better results.

Initially, the values $\phi_p = \phi_g = 2.05$ were tested for the acceleration coefficients. Using these values, it was found that the acceleration coefficients were way too high and that they would always find an unfavorable local minimum as shown in Figure 5.5. After testing numerous different values, it was found that a good pair of acceleration coefficients was around $\phi_p = \phi_g = 1.6$, which is shown in Figure 5.2. For pairs of acceleration coefficients below 1.6, PSO was found to more frequently get stuck in bad local minima as shown in Figure 5.6.

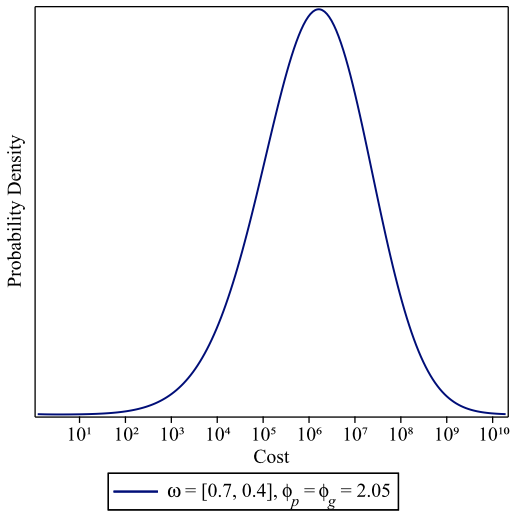

Figure 5.5: PDF of the cost after reaching a local minimum with $\phi_p = \phi_g = 2.05$.

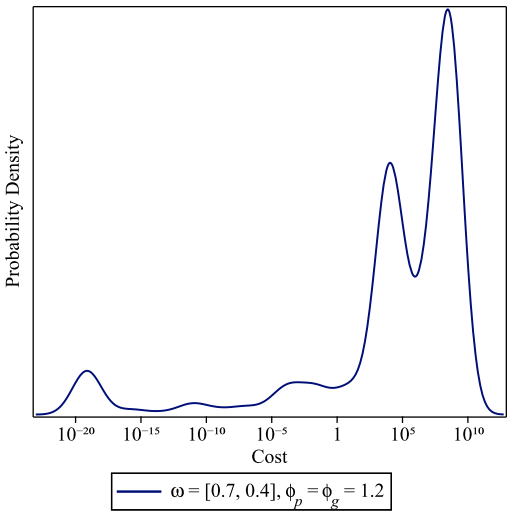

Figure 5.6: PDF of the cost after reaching a local minimum with $\phi_p = \phi_g = 1.2$.

Number of Particles

In a paper on parameter selection of PSO, Yan et al. [19] concludes that if the number of particles, S, is more than 50, they have a smaller impact on PSO. However, if the number of particle is less than 50, the number of particles will greatly influence the performance of the PSO.

After testing different values of S, it was observed that increasing the number of particles from 10 to 20 greatly improved the PDF of the cost as shown in Figures 5.7 and 5.8. However, doubling the amount of particles also causes the evaluation time of PSO to double. Additionally, increasing the number of particles beyond 20 had a much smaller improvement on the PDF of the cost. This can be seen in Figure 5.9, where 40 particles is only a slight improvement compared to 20 particles. The amount of particles is thus a decision between faster evaluation speed and a higher probability for finding a good minimum.

Maximum Velocity

The maximum velocity v_{max} ensures that the particles do not diverge from the search space [19]. If the maximum velocity is too high, the particle might jump over potentially good local minima. On the other hand, if the maximum velocity is too small the particles will move slowly, and consequently they will not explore much. Eberhart et al. [22] has found an optimal maximum velocity to be 10-20% of the search space.

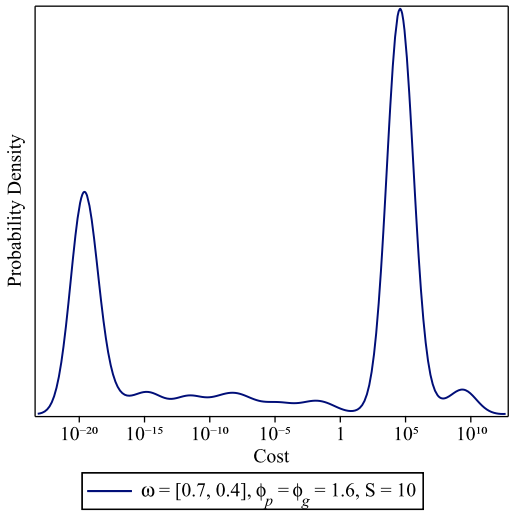

Figure 5.7: PDF of the cost after reaching a local minimum with $S = 10$.

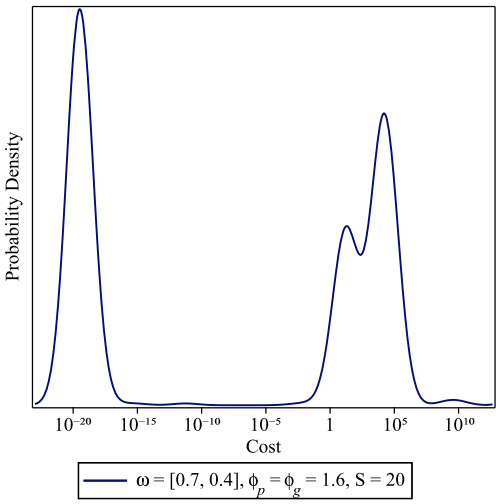

Figure 5.8: PDF of the cost after reaching a local minimum with $S = 20$.

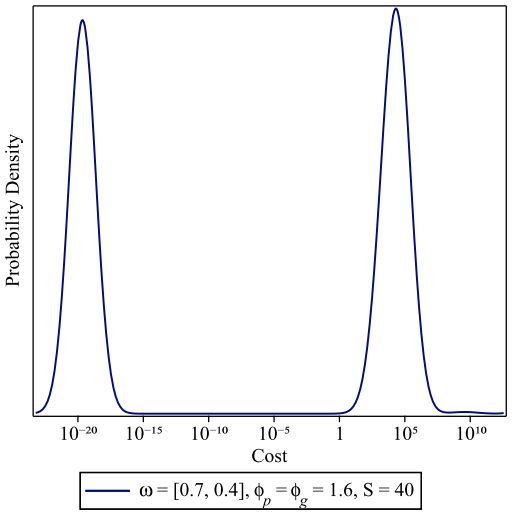

Figure 5.9: PDF of the cost after reaching a local minimum with $S = 40$.

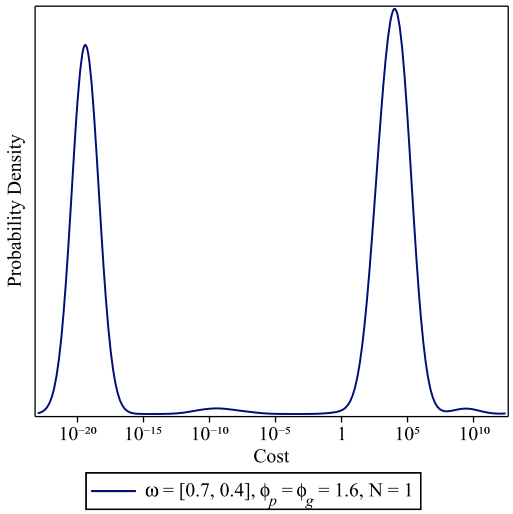

Figure 5.10: PDF of the cost after reaching a local minimum with $N = 1$.

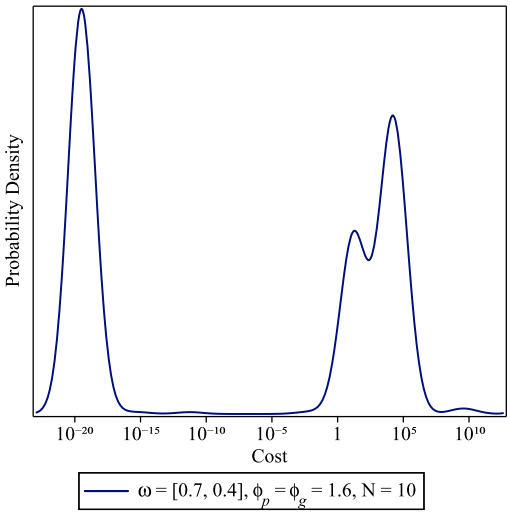

Figure 5.11: PDF of the cost after reaching a local minimum with $N = 10$.

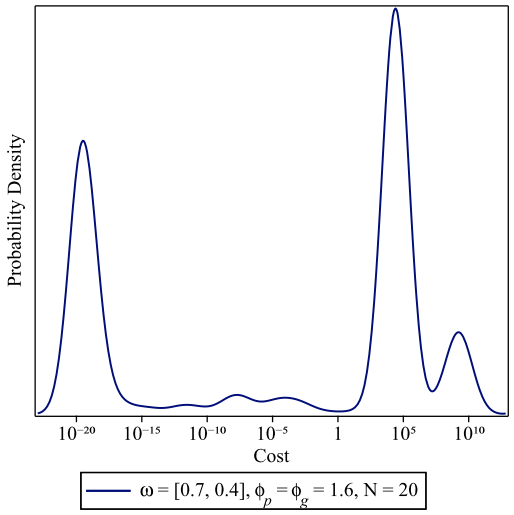

Figure 5.12: PDF of the cost after reaching a local minimum with $N = 20$.

When choosing a good value of the maximum velocity, the maximum velocity was found by

$$V_{max} = \frac{ub - lb}{N}, \tag{5.4}$$

where N is a constant and ub and lb are the upper and lower bound of the search space, respectively. After running PSO for different values of N, it was found that values of N higher than 10 and lower than 2 resulted in unfavorable outcomes. This can be seen in Figures 5.10, 5.11 and 5.12.

5.3 Findings

The parameter values found to make PSO perform well on the evaluation SIR model with 3 groupings in Section 5.2 Parameter Optimisation, were also good when increasing the number of groupings of the SIR model. Therefore, it is assumed that the parameter values will also perform well on other models.

The calibration of the parameters yielded no combination of parameter configurations that guaranteed an exact lumping. Therefore, it is expected no such combination exists.

5.3.1 Other PSO algorithms

After running the general PSO algorithm, it was found that all the particles tended to end up in the same local minimum. Therefore, it is of interest to change the algorithm in such a way that the particles are more likely to find different local minima, as this would potentially lead to a higher probability of finding a good local minimum. Therefore, a local-best topology PSO was implemented, which only allows each particle to get affected by its k-nearest neighbours instead of it being affected solely by the global best [23]. Using this algorithm, the new velocity of a particle is then

$$p.v_{ij} \leftarrow \omega p.v_{ij} + \phi_p r_p(p.best_{ij} - p.x_{ij}) + \phi_l r_l(p.l_{ij} - p.x_{ij}), \tag{5.5}$$

where $p.l_{ij}$ is the index ij of the local best of the particle p and its neighbors. This, however, was observed to more commonly find a bad local minimum than the global-best topology PSO algorithm as shown on Figure 5.13.

A PSO algorithm combining the global-best and local-best topology algorithms was also implemented, where the new velocity for a particle is

$$p.v_{ij} \leftarrow \omega p.v_{ij} + \phi_p r_p(p.best_{ij} - p.x_{ij}) + \phi_l r_l(p.l_{ij} - p.x_{ij}) + \phi_g r_g(g_{ij} - p.x_{ij}). \tag{5.6}$$

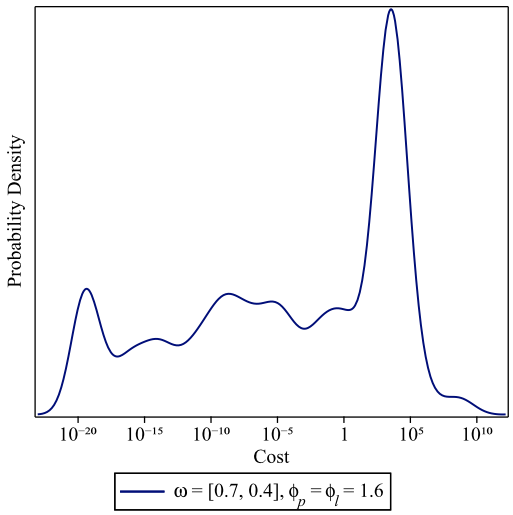

Figure 5.13: Probability distribution of the cost after reaching a local minimum with the local-best topology algorithm.

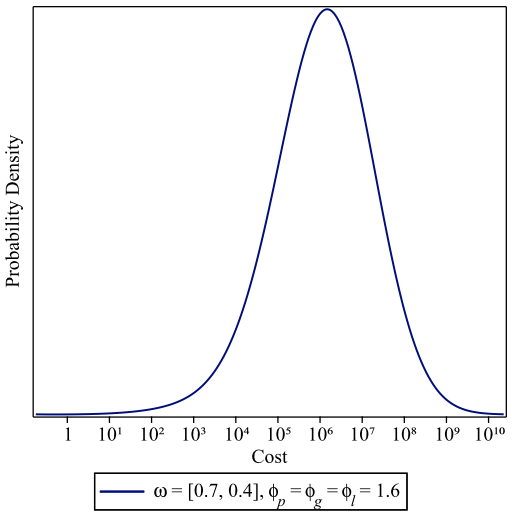

Figure 5.14: Probability distribution of the cost after reaching a local minimum with a combination of global-best and local-best.

However, this algorithm yielded worse results than both the local-best and global-best topology PSO algorithms and it was very unlikely to find a good local minimum at all as shown on Figure 5.14.

Chapter 6

Evaluation

This section will compare the different optimisation algorithms from Chapter 4 Gradient Descent and Chapter 5 Particle Swarm Optimisation on a variety of biochemical models. First, different metrics for evaluating the performance of the different optimisation algorithms will be discussed. Secondly, the biochemical models that are used for the evaluation will be introduced. Lastly, the findings of running the different optimisation algorithms on the biochemical models will be presented and discussed.

Choice of Algorithms to Evaluate

Ideally, all algorithms would be tested on all available models to accurately assert their effectiveness. In reality, this takes far too long. Therefore, a candidate from each distinct category of cost optimisation algorithms is chosen. These are: gradient descent with MMomentum, gradient descent with golden section line search and general particle swarm optimisation.

From gradient descent with dynamic step size, the MMomentum algorithm is chosen as it on average produced the best cost. For line search, golden section search is picked. This is because quadratic interpolation optimisation is not guaranteed to reduce the search space within a given amount of iterations. The general PSO algorithm is chosen without any local-best modifications, as these were found to be unfavourable. The values of the parameters used in PSO for evaluation are those from the calibration in Section 5.2 Parameter Optimisation.

6.1 Metrics

To evaluate the quality of the different optimisation algorithms, three different metrics are considered, each measuring some different aspect of an experiment.

The first metric is the geometric mean of the best cost from 100 experiments. The geometric mean is found by taking the arithmetic mean of the logarithm of all the values and finally applying the antilogarithm [24]. This metric clearly indicates the algorithm's ability of being able to find low valued costs. The geometric mean is used instead of the arithmetic mean because it is less affected by extreme values [25]. Consider an example of finding the mean of a good cost 10^{-8} and a bad cost 10^{8}. By using the arithmetic mean, the bad cost highly dominates the good cost and the mean is $5 \cdot 10^{7}$. Whereas, the geometric mean is less biased towards the bad costs and in this example the geometric mean of the two costs is 1.

The formula for calculating the geometric mean of the best cost is given by

$$v = 10^{a}, \quad a = \frac{1}{N} \sum_{i=0}^{N} log_{10}(best_i), \tag{6.1}$$

where $best_i$ is the best cost from i-th experiment and N is the number of experiments.

The second metric is *average time to converge* which is used to measure the speed of the algorithms.

The last metric that is considered is the ratio of exact lumpings. This metric shows the ratio of experiments with a cost that is less than 10^{-8}. Why the specific number of 10^{-8} is chosen was explained in Section 4.5.1 Probability Density Function.

Table 6.1 shows how the evaluation tables will look. This table contains no real data and is simply used for clearance. To evaluate the algorithms, 100 experiments are conducted and metrics are calculated based on the results.

The metrics column in Table 6.1 shows four specifiers. The first specifier is the start cost which shows the cost before the given algorithm begins executing. This is used to give a rough idea of how much the cost has been reduced. Note, this metric is specific to the model and not the algorithm; hence only a single value is present. Next, the metric of average cost after converging is specified. Here, the geometric mean is used to calculate the average. Thirdly, the metric of time is simply specifying how many seconds the algorithm needed in order to converge. Lastly, the ratio of exact lumpings is showing how often the algorithm was able to find a cost less than 10^{-8}.

Model	Metrics	PSO	Line Search	MMomentum
Model A	Start Cost	$1.50 \cdot 10^{6}$		
	Cost	$1.00 \cdot 10^{-1}$	$2.00 \cdot 10^{-2}$	$4.00 \cdot 10^{-4}$
	Time	1.0s	2.0s	4.0s
	Exact Rate	0	0.25	0.50

Table 6.1: Explanatory table. The data within this specific table is made-up and contain no material of findings.

6.2 Biochemical Models

In this section, the machine learning algorithms will be evaluated on biochemical models. These models were fetched from ERODE [26], which is a tool for "evaluation and reduction of stochastic reaction networks and differential equations". ERODE supplies collections of ODE systems for reduction. We will use models from `TCS_CMSB2020` [27]. We have assured through ERODE that all models can be exactly lumped.

First, the machine learning algorithms will be evaluated on small models with multiple runs. Then, they will be evaluated on larger models with only a few experiments to test how well they scale.

6.2.1 Small Models

In this section, six small polynomial models and four small arbitrary models will be tested on the chosen machine learning algorithms.

Polynomial models have only polynomial ODEs, which ensures that their integral is straightforward to compute. On the other hand, the arbitrary models can be discontinuous and therefore a logistic function will be utilised to centralise the outliers of function evaluations; as explained in Section 3.3 Discontinuous Functions.

The chosen models can be seen in Table 6.2. The SP prefix in the names indicates that the models are small polynomial models while the SA prefix indicates small arbitrary models. The amount of species is equivalent to the dimensionality of the ODE system.

Name	Model Id	Species
SP0	BIOMD0000000077	9
SP1	BIOMD0000000108	12
SP2	BIOMD0000000125	6
SP3	BIOMD0000000189	19
SP4	BIOMD0000000459	4
SP5	BIOMD0000000104	6
SA0	BIOMD0000000005	9
SA1	BIOMD0000000010	8
SA2	BIOMD0000000107	16
SA3	BIOMD0000000143	22

Table 6.2: Chosen biochemical models.

As an example of these ODEs, the model SP4 is shown on Equation 6.2, where $a, b, ..., g$ are constants. These constants have actual values in the real model but, here, they are omitted for clarity. By analysing this equation, a couple of things become apparent. Since $f_1(x) = 0$, there is no change in this species; it will remain with its initial value at any time. Also, many of the terms in the species cancel out. Hence, if all ODEs are summed, the result is just $ax_1 - bx_2$, which describes the overall change in 'mass' of the system. This is equivalent to the lumping matrix $\begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix}$. However, this exact lumping is probably not useful since one cannot extract any information about the original species. Therefore, it will not be attempted to lump these models to just one dimension because this is almost trivial for many of the systems.

$$\frac{dx}{dt} = f(x) = \begin{bmatrix} 0 \\ ax_1 - (b+c+g)x_2 + ex_3 + dx_3x_4 + fx_4 \\ cx_2 - ex_3 - dx_3x_4 \\ gx_2 - fx_4 \end{bmatrix} \tag{6.2}$$

Polynomial Models

In Table 6.3, the results of running 100 experiments on each small polynomial model with each algorithm is shown. All the data from each model was evaluated on the same machine. However, to decrease the time to evaluate the models, they were spread out over several machines. This is not a problem, because the complexity and dimensionality

of each model varies and thus they cannot be compared anyways. All the models are reduced by a single dimension.

Model	Metrics	PSO	Line Search	MMomentum
SP0	Start Cost	$1.22 \cdot 10^{17}$		
	Cost	$1.83 \cdot 10^{7}$	$3.92 \cdot 10^{13}$	$6.97 \cdot 10^{12}$
	Time	70.0s	18.5s	40.6s
	Exact Rate	0	0	0
SP1	Start Cost	$2.57 \cdot 10^{25}$		
	Cost	$2.59 \cdot 10^{20}$	$4.35 \cdot 10^{20}$	$3.43 \cdot 10^{20}$
	Time	11.2s	14.3s	111.1s
	Exact Rate	0	0	0
SP2	Start Cost	$8.6 \cdot 10^{3}$		
	Cost	$9.06 \cdot 10^{-8}$	$6.42 \cdot 10^{-9}$	$5.81 \cdot 10^{-9}$
	Time	9.9s	4.1s	10.7s
	Exact Rate	0.81	1.00	1.00
SP3	Start Cost	276.86		
	Cost	$1.25 \cdot 10^{-4}$	$2.05 \cdot 10^{-3}$	$1.49 \cdot 10^{-4}$
	Time	86.1s	818.1s	500.0s
	Exact Rate	0	0	0
SP4	Start Cost	$6.17 \cdot 10^{6}$		
	Cost	$5.28 \cdot 10^{-4}$	10.67	1.62
	Time	15.5s	0.9s	26.1s
	Exact Rate	0.38	0	0.04
SP5	Start Cost	$2.95 \cdot 10^{6}$		
	Cost	$1.32 \cdot 10^{-8}$	$1.44 \cdot 10^{-6}$	$1.85 \cdot 10^{-8}$
	Time	14.3s	18.4s	12.5s
	Exact Rate	0.98	0.84	0.98

Table 6.3: Results for small polynomial models. Each model is run for 100 experiments with each algorithm.

For the models SP0, SP1 and SP3, the algorithms were not able to find a single exact lumping. For the models SP0 and SP1, the achieved cost is much too large to be useful. Since the cost is the mean squared error, it can be expected that model SP0 with PSO will give errors on the order of 10^4. However, model SP3 will only give errors of

order 10^{-2}, which may be small enough to be acceptable. For model SP2, line search and Mmomentum were able to find an exact lumping for every single experiment, while PSO only found an exact lumping in 81% of the experiments. For the model SP4, PSO and Mmomentum were able to find exact lumpings with rates of 0.38 and 0.04, respectively. Line search, however, was not able to find an exact lumping for this model. Finally, for model SP5, PSO, line search and Mmomentum found an exact lumping in 98, 84 and 98 of the experiments, respectively.

Of the three algorithms PSO and Mmomentum are the best performers. Line search consistently achieves a worse cost than both PSO and Mmomentum. When considering the time spent, it is also necessary to adjust for the cost achieved. While line search is occasionally faster, it is only because it plateaus at a much larger cost.

By examining the models SP1 and SP3, where PSO and Mmomentum have almost reached the same cost, it can be seen that Mmomentum is significantly slower for the larger models. None of them generally outperform the others, for model SP5 the performance was very similar, for model SP2 Mmomentum is much more consistent in finding the exact lumpings but the opposite is the case for model SP4.

Arbitrary Models

Table 6.4 shows the results of running 100 experiments on the arbitrary models with each algorithm. For the arbitrary models SA0, SA1, SA2 and SA3, the algorithms were only able to slightly reduce the cost from the starting cost and it was not possible to find any exact lumping.

Conclusively, the different algorithms find costs that are very similar of value. However, Mmomentum generally takes a lot longer to find a local minimum than the two other algorithms; especially on the larger models.

Model	Metrics	PSO	Line Search	MMomentum
SA0	Start Cost	$1.35 \cdot 10^8$		
	Cost	$8.90 \cdot 10^6$	$3.77 \cdot 10^7$	$2.33 \cdot 10^7$
	Time	27.5s	3.2s	59.3s
	Exact Rate	0	0	0
SA1	Start Cost	3.44		
	Cost	$4.44 \cdot 10^{-2}$	$5.4 \cdot 10^{-2}$	$4.23 \cdot 10^{-2}$
	Time	38.5s	44.6s	84.6s
	Exact Rate	0	0	0
SA2	Start Cost	$2.49 \cdot 10^4$		
	Cost	91.6	55.85	16.71
	Time	368.2s	1042.5s	1482.5s
	Exact Rate	0	0	0
SA3	Start Cost	$5.40 \cdot 10^8$		
	Cost	$1.56 \cdot 10^7$	$2.06 \cdot 10^7$	$1.48 \cdot 10^7$
	Time	49.3s	91.1s	1827.7s
	Exact Rate	0	0	0

Table 6.4: Results for the small arbitrary models. Each model is run for 100 experiments with each algorithm.

Increasing Reductions

Table 6.5 shows the results from lumping the same model, SP2, with incremental reduction sizes. First, the model was lumped with a reduction of size one and the reduction size was incremented until the reduced model was of one dimension. These tests were carried out on the same machine to facilitate comparison between the reductions.

Model	Metrics	PSO	Line Search	MMomentum
SP2 - 1 reduction	Start Cost	$8.6 \cdot 10^3$		
	Cost	$9.06 \cdot 10^{-8}$	$6.42 \cdot 10^{-9}$	$5.81 \cdot 10^{-9}$
	Time	9.9s	4.1s	10.7s
	Exact Rate	0.81	1.00	1.00
SP2 - 2 reduction	Start Cost	$1.6 \cdot 10^4$		
	Cost	$9.30 \cdot 10^{-6}$	$2.06 \cdot 10^{-4}$	$1.29 \cdot 10^{-6}$
	Time	15.0s	35.0s	199.5s
	Exact Rate	0.53	0.4	0.58
SP2 - 3 reduction	Start Cost	$2.00 \cdot 10^4$		
	Cost	$6.45 \cdot 10^{-2}$	$2.13 \cdot 10^{-1}$	$2.50 \cdot 10^{-6}$
	Time	19.1s	47.0s	394.6s
	Exact Rate	0.03	0.03	0.55
SP2 - 4 reduction	Start Cost	$1.77 \cdot 10^4$		
	Cost	$2.17 \cdot 10^{-3}$	$4.64 \cdot 10^{-2}$	$3.81 \cdot 10^{-5}$
	Time	11.3s	43.7s	336.8s
	Exact Rate	0.25	0.12	0.38
SP2 - 5 reduction	Start Cost	$1.13 \cdot 10^4$		
	Cost	$3.67 \cdot 10^{-7}$	$7.71 \cdot 10^{-3}$	$8.37 \cdot 10^{-6}$
	Time	9.5s	20.4s	101.4s
	Exact Rate	0.66	0.29	0.42

Table 6.5: Results from the SP2 model with different sizes of reduction. Each is run for 100 experiments with each algorithm.

The first row in Table 6.5, shows SP2 with a reduction of 1. Here, a good cost was generally easy to find for all algorithms. Mmomentum and line search were always able to find an exact lumping, whereas PSO was only able to find an exact lumping with a rate of 0.81.

Moving on to a reduction of 2, the rate of exact lumpings declined to just above 0.5 for PSO and Mmomentum. Here, line search is inferior with a rate of 0.4. Compared to the results from the reduction of 1, there is an increase in the time spent for line search and Mmomentum by roughly 9 and 20 times, respectively. Contrarily, PSO is only 1.5 times slower.

Reducing the model by 3 dimensions provided some interesting results in that Mmomentum now greatly outperforms the two other algorithms with respect to the ratio of

finding an exact lumping. As a drawback, the ratio of good lumpings is accompanied by the algorithm taking a lot longer time to plateau.

Continuing with a reduction of 4, the most interesting finding is that PSO's and line search's ratio of exact lumpings increased. Oppositely, Mmomentum found fewer exact lumpings than with a reduction of three. Although, it still showed the best ratio of good lumpings of the three algorithms.

Lastly, the model was reduced by 5 dimensions; meaning it was described by a single species. Here, PSO outperformed Mmomentum, and line search continued being the worst of the bunch. Another finding is that the time to find a minimum greatly declined for all three algorithms.

6.2.2 Big Models

In this section, the three algorithms will be tested on larger polynomial models in order to see how they perform on higher dimensions. The chosen models can be seen in Table 6.6.

Name	Model Id	Species
BP0	BIOMD0000000504	75
BP1	BIOMD0000000334	74
BP2	BIOMD0000000332	78

Table 6.6: Chosen big biochemical models.

When performing the experiments with the algorithms, it was found that calculating the gradient for line search and Mmomentum took way too long. A single iteration when running line search and Mmomentum took over 5 minutes. Thus, an experiment on a model for one of these two algorithms would require an entire day. Therefore, PSO was the only algorithm that finished the experiments.

Table 6.7 shows the results of 20 experiments on each of the three big models using PSO. It was observed that PSO was not able to find a single exact lumping. Furthermore, it was found that PSO, on average, reduced the starting cost by around a factor of 100.

Model	Start Cost	Cost	Time	Exact Rate
BP0	$1.36 \cdot 10^4$	10.77	654.3s	0
BP1	$1.77 \cdot 10^6$	$1.78 \cdot 10^4$	2356.2s	0
BP2	$2.09 \cdot 10^6$	$2.31 \cdot 10^4$	2532.0s	0

Table 6.7: Results for the big polynomial models. Each model is run for 20 experiments with PSO.

6.3 Discussion

In this section, the results presented previously this chapter will be discussed. The three algorithms will be discussed and compared within the different metrics. Additionally, this section covers the topics of acceptable error values and choosing a reduction size. Lastly, the performance of our algorithms are compared to ERODE.

6.3.1 Line Search

In terms of finding exact lumpings, line search was the poorest performer of the algorithms. This is likely caused by the fact that it makes no attempt to avoid bad local minima. Wherever it starts, it will simply follow the negative gradient until the minimum is hit. In contrast, PSO explores a much larger search space, because all the particles are initially randomly distributed. Likewise, Mmomentum maintains some of the velocity of the previous step; therefore, it can jump over small local minima.

6.3.2 MMomentum

The main drawback of Mmomentum is that it is slow on larger models. This is because each epoch requires calculating the gradient. As previously mentioned, this requires $n\hat{n}$ Monte Carlo mean estimations of the derivative. This means that MMomentum's time consumption scales with the size of the system. For large systems, Mmomentum basically spends all of its computation time on calculating the gradient.

While line search also has to calculate the gradient, it utilises the gradient more by minimising the function in this direction. Therefore, it has to do fewer calculations of the gradient and therefore in terms of time consumption line search is better than MMomentum for large systems.

6.3.3 Particle Swarm Optimisation

Particle Swarm Optimisation was the fastest of the three algorithms; especially on the larger models. This is caused by the fact that it does not calculate the gradient on each iteration, which line search and Mmomentum need to.

Comparing PSO with MMomentum, it is difficult to say which one is the best performer in terms of finding exact lumpings. Often, one is out-performing the other but just as often it is out-performed by the other. For model SP2 with a reduction of four, MMomentum has the best ratio of exact lumpings. However, with a reduction of five, PSO provides the best. Therefore, in terms of the ratio of exact lumpings, there is no clear best algorithm.

The fact that PSO is competitive with Mmomentum in terms of finding exact lumpings is caused by the high amount of particles. Mmomentum and line search have a single particle that takes precise steps in the steepest direction but consume a lot of time from calculating the gradient. On the other hand, PSO takes less precise steps but consumes much less time.

6.3.4 Choosing Reduction Size

The amount of dimensions one chooses to reduce has great effect on the results. For some models, reducing by fewer dimensions can be easier because there will be more opportunities for a good lumping. If, for example, a model could be reduced by lumping $\dot{x}_1$ and $\dot{x}_2$, but also $\dot{x}_3$ and $\dot{x}_4$, there are more opportunities for a good lumping with a reduction of 1 compared to 2. However, as the reduction size increases, the search space decreases. This is because the lumping matrix has size $n \times \hat{n}$ and thus the size of the reduced system, $\hat{n}$, has great influence on the size of the search space.

In Section 6.2.1 Increasing Reductions, the results show that the execution time is largest when $\hat{n} \approx \frac{1}{2}n$ for model SP2. This is of course only the case because this particular model has at least one exact lumping for all sizes of reduction. For models where the possible reductions are not expected to be greater than $\frac{1}{2}n$, this will not have any effect.

In Section 2.4.2 Preserving Species, it was described how one can 'lock' the rows of a matrix to ensure that species are preserved in the reduced system. Locking a row has a lot of influence on how the system can be lumped. It does not mean an additional row simply is added to the reduced system. Consider model SP4 that is shown in Equation 6.2. If the first row is locked to $\begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix}$, i.e. preserving the second species, it is not possible to find a good lumping. This is because the second species is dependent on the product of x_3 and x_4, meaning these cannot be lumped. This leaves only x_1, which is

not enough for a lumping.

6.3.5 Comparison to ERODE

As previously mentioned, the ERODE software can also find reduced systems. For most models tested in Section 6.2 Biochemical Models, our algorithms did not perform as well as ERODE. However, our algorithms performed better on a few of the models. ERODE was unable to find any reduction for model SP4, yet our algorithms found exact lumpings with reduction 1, 2 and 3. Similarly for model SP2, ERODE found a reduction of 2, while our approach found exact lumpings with reductions up till 5.

Conclusively, ERODE performs better than any of our three algorithms. Although, our algorithms found exact lumpings for a model, for which ERODE could not, there exists potential use of machine learning to find lumpings.

Chapter 7

Conclusion

This chapter concludes on the problem statement and its subproblems. As given in Section 2.5, the problem statement is

> *How can machine learning be utilised to find a lumping matrix for an arbitrary system of ODEs?*

The problem of finding lumping matrices was cast as a cost function optimisation problem. This cost function measures the mean squared error of the reduced system by using the Monte Carlo integral technique. It was discussed which criteria the lumping matrix must uphold for the optimisation to be valid, namely orthonormalisation. This ensures that the row-vectors are linearly independent and that minimisation is not achieved by simply making the matrix small. Additionally, a method for dealing with discontinuous models by using a logistic function that squeezes enormous values was proposed.

With these preliminary considerations in order, the two chosen methods for cost function optimisation were ready to be considered. This was represented through two subproblems. The first being:

- How can gradient descent be used to minimise the ε error?

In gradient descent, the main concern was to calculate the step that should be taken based on the gradient. Several methods were analysed and separated into two main categories, namely, dynamic step size and line search. The dynamic step size methods focus on using previous steps to determine the size of the next step. Contrarily, line search attempts to minimise the function in the direction of the gradient with each step. Of these methods, Mmontentum and golden section search were found to work the best for dynamic step size and line search, respectively.

Mmomentum is a modification of momentum where learning rates of dimensions are scaled dynamically. The intuition of momentum is to maintain velocity to skip over small local minima.

Golden section search is a method for finding extrema of single dimensional functions. It works by narrowing the search interval down in section sizes of the golden ratio. The main advantage of golden section search over the other line search method, quadratic interpolation, is that it takes a definite amount of iterations before terminating.

It was found that line search is faster than Mmomentum at converging because it utilises the gradient more. However, line search makes no attempt to avoid local minima and therefore it did not find as good lumpings as Mmomentum.

The second subproblem is:

- How can particle swarm optimisation be used to minimise the ε error?

Particle swarm optimisation (PSO) was used as a derivative-free alternative. The main advantage of PSO is that it spreads out particles over the search space. Hereby, the chance of finding a better minimum is increased. PSO works by the particles, at each iteration, moving a bit towards the global best found point in addition to their own best position and a bit of random velocity from the previous iteration.

Mmomentum, line search and PSO were evaluated on biochemical models. Line search was clearly outperformed by both Mmomentum and PSO in most of the tests. The results show that for most models none of the algorithms are effective, and for larger models it became infeasible to calculate the derivative. However, for some models, they were able to find a higher reduction than ERODE (a program for finding reductions). Furthermore, it is possible to lock specific species and still yield exact lumpings. Therefore, there exist potential for using machine learning to find lumpings.

Chapter 8

Future Work

This chapter will discuss possible further exploration into the topic.

Improving PSO Convergence Time

The main downside of PSO is that it can be quite slow at exploiting minima. Even though a particle might be on a slope towards a good minimum, it will take a long time to descend because it does not have the gradient to instruct which way to go. Therefore, a single particle only really descends by random chance.

It was briefly attempted to use line search for the best particle, allowing it to descend much faster (in terms of fewer iterations). Unfortunately, this did not seem to work out of the box. In the first iteration, the best particle is unlikely to be on a slope towards a particularly good minimum; evident by the fact that line search rarely found good minima. Since this particle has done line search, it will now be in a much better position than the other particles. Given this initial boost, it will continue to be the best particle until it converges to a minimum. This means that the particle's initial small lead is magnified so much that the algorithm essentially becomes line search.

In this project, this option was not explored because initial testing made it seem unfavorable. However, since experimenting with the models, the drawback of PSO's slow descent has become more apparent.

Learning Maximum Reduction

In this project, the reduction is predetermined by the user. The given machine learning algorithm only attempts to make that specific reduction. However, in practice this reduction amount is not likely to be known in advance. Therefore, it would be favorable

if the algorithm found the maximum reduction. This could be achieved by simply attempting reduction starting from 1 until it is no longer possible. This approach would be very inefficient.

In Section 3.1 Matrix Normalisation, it was described how the lumping matrix is orthonormalised. This operation prevents the matrix from becoming linearly dependent. Before this was introduced, we observed that the matrix would often become linearly dependent. By relaxing this orthogonalisation constraint, one could catch when rows have become linearly dependent and remove all but one of them — thereby increasing the reduction amount.

References

[1] Isabel Cristina Pérez-Verona, Mirco Tribastone, and Andrea Vandin. "A Large-Scale Assessment of Exact Model Reduction in the BioModels Repository". In: *Computational Methods in Systems Biology*. Ed. by Luca Bortolussi and Guido Sanguinetti. Cham: Springer International Publishing, 2019, pp. 248–265. ISBN: 978-3-030-31304-3.

[2] Tiberiu Harko, Francisco S.N. Lobo, and M.K. Mak. "Exact analytical solutions of the Susceptible-Infected-Recovered (SIR) epidemic model and of the SIR model with equal death and birth rates". In: *Applied Mathematics and Computation* 236 (2014), pp. 184–194. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2014.03.030. URL: http://www.sciencedirect.com/science/article/pii/S009630031400383X.

[3] Raymond Keith Watson. "On an epidemic in a stratified population". In: *Journal of Applied Probability* 9.3 (1972), pp. 659–666.

[4] Genyuan Li and Herschel Rabitz. "A general analysis of exact lumping in chemical kinetics". In: *Chemical Engineering Science* 44.6 (1989), pp. 1413–1430. ISSN: 0009-2509. DOI: https://doi.org/10.1016/0009-2509(89)85014-6. URL: http://www.sciencedirect.com/science/article/pii/0009250989850146.

[5] A Rao. *Linear algebra*. New Delhi: Hindustan, 2000. ISBN: 978-81-85931-26-5.

[6] William Press. *Numerical recipes : the art of scientific computing*. Cambridge, UK New York: Cambridge University Press, 2007. ISBN: 9780521880688.

[7] Matt Pharr and Greg Humphreys. "Chapter Thirteen - Monte Carlo Integration I: Basic Concepts". eng. In: *Physically Based Rendering*. Second Edition. Elsevier Inc, 2010, pp. 636–676. ISBN: 0123750792.

[8] Walter Hoffmann. "Iterative algorithms for Gram-Schmidt orthogonalization". In: *Computing* 41.4 (1989), pp. 335–348.

[9] G Peter Lepage. "A new algorithm for adaptive multidimensional integration". In: *Journal of Computational Physics* 27.2 (1978), pp. 192–203.

[10] G Peter Lepage. *VEGAS-An adaptive multi-dimensional integration program*. Tech. rep. 1980.

[11] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents, 2nd Edition*. Cambridge University Press, 2017.

[12] Nikhil Buduma and Nicholas Locascio. *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. 1st_ed. O'Reilly Media, 2017. ISBN: 9781491925614.

[13] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[14] Ning Qian. "On the Momentum Term in Gradient Descent Learning Algorithms". In: *Neural Netw.* 12.1 (Jan. 1999), pp. 145–151. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(98)00116-6. URL: https://doi.org/10.1016/S0893-6080(98)00116-6.

[15] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[16] Jorge Nocedal and Stephen J Wright. "Line search methods". In: *Numerical optimization* (2006), pp. 30–65.

[17] T. Zhang and Y. Zhao. "The Root-Finding Algorithm of Three-Point Quadratic Interpolation of the Nonlinear Equation". In: *2012 Fifth International Conference on Information and Computing Science*. 2012, pp. 27–29. DOI: 10.1109/ICIC.2012.64.

[18] R. Brits, A.P. Engelbrecht, and F. van den Bergh. "Locating multiple optima using particle swarm optimization". In: *Applied Mathematics and Computation* 189.2 (2007), pp. 1859–1883. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2006.12.066. URL: http://www.sciencedirect.com/science/article/pii/S0096300306017826.

[19] Yan He, Wei Ma, and Ji Zhang. "The Parameters Selection of PSO Algorithm influencing On performance of Fault Diagnosis". In: *MATEC Web of Conferences* 63 (Jan. 2016), p. 02019. DOI: 10.1051/matecconf/20166302019.

[20] Russ C Eberhart and Yuhui Shi. "Comparing inertia weights and constriction factors in particle swarm optimization". In: *Proceedings of the 2000 congress on evolutionary computation. CEC00 (Cat. No. 00TH8512)*. Vol._1. IEEE. 2000, pp. 84–88.

[21] Saptarshi Sengupta, Sanchita Basak, and Richard II. "Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives". In: (Apr. 2018). DOI: 10.3390/make1010010.

[22] Eberhart and Yuhui Shi. "Particle swarm optimization: developments, applications and resources". In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*. Vol._1. 2001, 81–86 vol. 1. DOI: 10.1109/CEC.2001.934374.

[23] Huanqing Cui et al. "Parameter Selection and Performance Comparison of Particle Swarm Optimization in Sensor Networks Localization". In: *Sensors* 17 (Mar. 2017), p. 487. DOI: 10.3390/s17030487.

[24] Donald McAlister. "XIII. The law of the geometric mean". In: *Proceedings of the Royal Society of London* 29.196-199 (1879), pp. 367–376.

[25] D. Clark-Carter. "Measures of Central Tendency". In: *International Encyclopedia of Education (Third Edition)*. Ed. by Penelope Peterson, Eva Baker, and Barry McGaw. Third Edition. Oxford: Elsevier, 2010, pp. 264–266. ISBN: 978-0-08-044894-7. DOI: https://doi.org/10.1016/B978-0-08-044894-7.01343-9. URL: http://www.sciencedirect.com/science/article/pii/B9780080448947013439.

[26] *ERODE*. https://www.erode.eu/. Accessed: 16-12-2020.

[27] *ERODE model TCS_CMSB2020*. https://www.erode.eu/models/TCS_CMSB2020.zip. Accessd: 16-12-2020.

[28] Wenyu Sun and Ya-Xiang Yuan. "Line Search". In: *Optimization Theory and Methods: Nonlinear Programming*. Boston, MA: Springer US, 2006, pp. 89–98. ISBN: 978-0-387-24976-6. DOI: 10.1007/0-387-24976-1_2. URL: https://doi.org/10.1007/0-387-24976-1_2.

Appendix A

Proof that $\overline{\Delta M}\Delta M = \bar{M}M$

M must be an $n \times m, n < m$ such that it has a right inverse, and Δ must be $n \times n$ and have a regular inverse. First, we rewrite right inverse to

$$\overline{\Delta M}\Delta M = (\Delta M)^T(\Delta M(\Delta M)^T)^{-1}\Delta M. \tag{A.1}$$

Then we use the property $(AB)^T = B^T A^T$

$$= M^T\Delta^T(\Delta M M^T\Delta^T)^{-1}\Delta M. \tag{A.2}$$

Now, we use the property $(AB)^{-1} = B^{-1}A^{-1}$ when A and B are invertible;

$$= M^T\Delta^T\Delta^{-T}(MM^T)^{-1}\Delta^{-1}\Delta M. \tag{A.3}$$

Finally, we cancel out Δ and rewrite to right inverse notation;

$$= M^T(MM^T)^{-1}M = \bar{M}M. \tag{A.4}$$

Appendix B

VEGAS

To reduce the needed samples importance sampling is used. When performing the Monte Carlo integration for the cost function the x-values used are taken from some distribution. In the basic case, this can simply be a uniform distribution. However, to improve this the distribution can be dependent on the value of the function at each point. The idea is to sample more thoroughly at points that yield a larger value, because these will have a larger influence on the integral. E.g. if at some point the functions is evaluated to $f(x_1) = 10$, and at another point it is evaluated to $f(x_2) = 1000$, then error in estimation around the second point will have a much larger influence on the average than in the first.

An algorithm used to accomplish this is VEGAS [10]. The basic idea of VEGAS is to partition the distribution of points on which the function is evaluated, such that each partition is equally likely but areas with greater values will have higher partition density [10].

Consider the function f, and the definite integral over some n-dimensional hypervolume Ω

$$I = \int_{\Omega} f(x) d^n x, \tag{B.1}$$

where x is a vector in Ω. To explain the algorithm, a 1-dimensional example will be used. On Figure B.1 the example function can be seen, the goal here is to calculate the green marked area.

The VEGAS algorithm is executed in two stages. The first stage optimally partitions the function, such that each partition has the same hypervolume. The probability of choosing each partition is always equal. This means that values that are in smaller partitions are more likely to be chosen. Afterwards, the definite integral is computed by first uniformly choosing a partitioning, then uniformly choosing a point inside this

partitioning and evaluating the function at this point. This is repeated until the variance is acceptable.

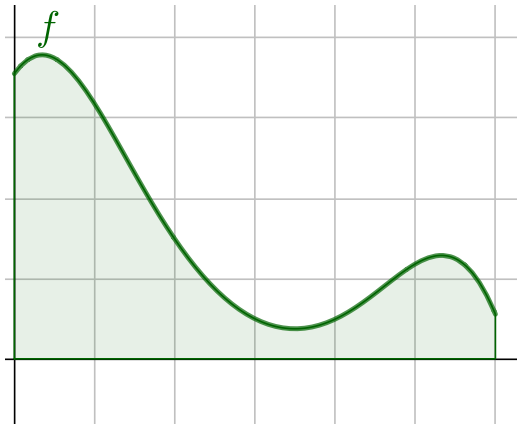

Figure B.1

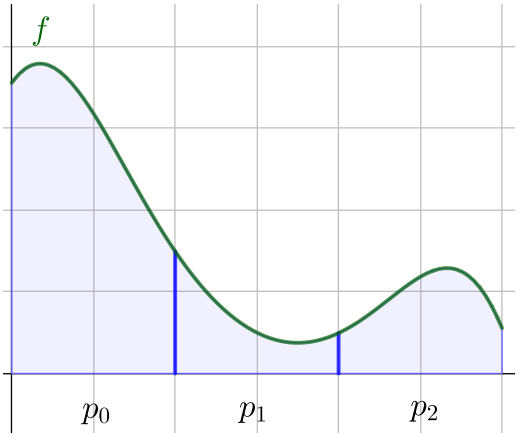

Figure B.2: Initial partitions evenly distributed.

Initially, all the partitions are the same size. To simplify the partitioning, it is done for each dimension separately, and therefore the hypervolume must be a hypercuboid. Each partition is continuous and is not overlapping with any other partition. Each element x_i in the vector x has N partitions $p_{i_0}, p_{i_1}, \cdots, p_{i_N}$ of equal size that sum to the size of the hypervolume in that direction. On Figure B.2 the initial partitioning for the example can be seen. [10]

In order to find an optimal partitioning subdivision and merging is done iteratively. Each partition is subdivided into subpartitions dependent on the size of the hypervolume in that partition. Each subpartition in a partition are of equal size. Then consecutive new subpartitions are merged back into N new partitions. Hereby, partitions with large hypervolume, i.e many subpartition, are shrunk and likewise partitions with small hypervolume are expanded.

First, the size of the function in each partition is estimated, notated $\bar{f}_{i_j}$. The function is evaluated at a set of points X to estimate the size of the function in each partition in each dimension. X is generated by uniformly choosing a partition for each dimension and then uniformly choosing a point in these partition. The chosen partitions are saved, such that a set X' with points only inside some partition can easily be restored. The function values for each evaluation in each partition are averaged as

$$\bar{f}_{i_j} = \frac{1}{|X'|} \sum_{x \in X'} |f(x)|, \quad X' = \{x \in X : x_i \in p_{i_j}\}, \tag{B.2}$$

where X' is the subset of points for which x_i is in partition j; i.e. $\bar{f}_{i_j}$ is the average value of all the function evaluations that are in the partition p_{i_j}. [10]

These estimates of the size of f in each partition is then used to determine the amount of subdivisions for that partition. Since the goal is to have the same hypervolume under the function in each partition, each partition gets subdivision proportional with the ratio of its hypervolume and the total hypervolume for all partition. Each partition is subdivided in to $m_{i_j} + 1$ partition with

$$m_{i_j} = K \frac{\bar{f}_{i_j} p_{i_j}}{\sum_{l=0}^{N} \bar{f}_{i_l} p_{i_l}}, \tag{B.3}$$

where K is some constant describing how many subdivisions should happen on each iteration. On Figure B.3 the subpartitions for the example can be seen in the red lines. [9]

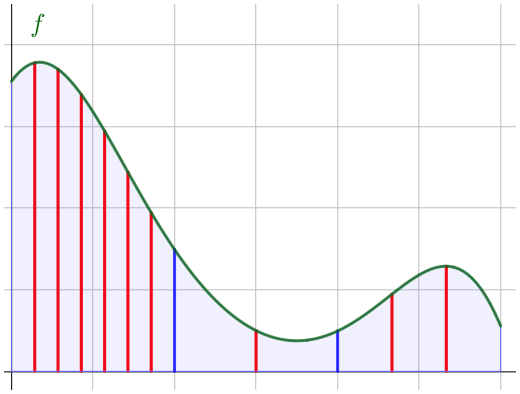

Figure B.3: Subpartitions created with $K = 9$ on the example function. Each subpartition is evenly distributed inside its partition.

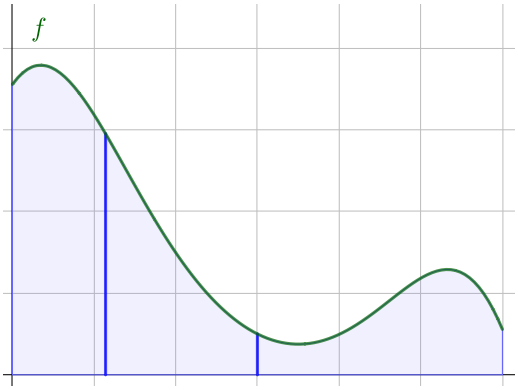

Figure B.4: New partitions gained from merging every 4 consecutive subpartitions on Figure B.3.

Now these new partitions must be merged back into N partitions. This can be done by simply merging $\frac{\#partitions}{N}$ consecutive partitions. From Equation B.3 it can be seen that K new subpartitions are created. Therefore, by choosing K such that N divides K the consecutive partitions to merge will always be an integer. On Figure B.4 the new partitions after merging on the example can be seen. Observe that the areas in each partition are now closer to being equal.

By doing this iteratively the partitions will become better and better. An optimal partitioning has been reached when $\forall_{j,l} m_{i_j} = m_{i_l}$; i.e. when all partitions are subdivided the same amount. When this is achieved the definite integrals in each partition are

approximately equal, and therefore another iteration accomplishes nothing. On Figure B.5 the final partitioning of the example can be seen.

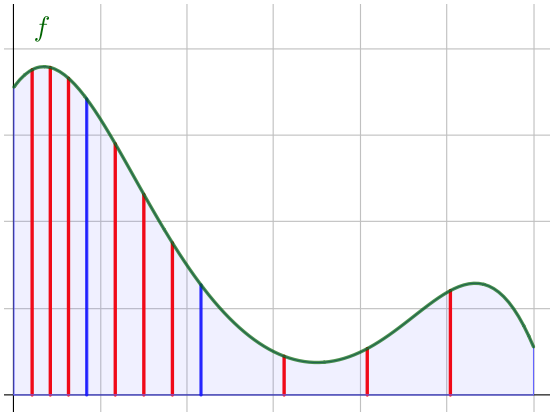

Figure B.5: The optimal partitioning of the example, indicated by the equal distribution of subpartitions.

With an optimal partitioning, the integral can be approximated by

$$I \approx \frac{N^n}{M} \sum_h^M f(p_h(x))|p_h|, \tag{B.4}$$

where p_h is a partition chosen uniformly random from each dimension, $|p_h|$ is the size of the partition, $p_h(x)$ is a uniformly random vector in partition p_h and Ω is the size of the hypervolume that is integrated over. Intuitively, each function evaluation is weighted by the size of the hypervolume it is evaluated in. This equation is derived from the general [9]

$$I \approx \frac{1}{M} \sum_x \frac{f(x)}{P(x)}, \tag{B.5}$$

where $P(x)$ is the probability density of x. The probability density for n-dimensions is

$$P(x) = \frac{1}{N^n \frac{|p_x|}{\Omega}}. \tag{B.6}$$

Inserting this into Equation B.5

$$I \approx \frac{\Omega}{M} \sum_h^M \frac{f(x)N^n|p_h|}{\Omega} = \Omega \frac{N^n}{M\Omega} \sum_h^M f(x)|p_h| = \frac{N^n}{M} \sum_h^M f(x)|p_h|. \tag{B.7}$$

The fraction before the sum might look a bit suspicious, but noticing that $\frac{\Omega}{N^n}$ is the average size of the partitions, $\frac{M\Omega}{N^n}$ is then an estimation of the total space that is integrated

over. The reciprocal comes up in this equation. Therefore, when multiplying with Ω it cancels out.

For the cost function, we just want to estimate the average value of f, therefore we divide the estimated integral by the volume that was integrated over, Ω, to obtain

$$\mathcal{C}(M) = \frac{N^n}{M\Omega} \sum_{h}^{M} \theta(M, x)|p_h|. \tag{B.8}$$

Appendix C

Root of Quadratic Interpolation

To determine the root of the quadratic function, which is derived from quadratic interpolation given three points, the polynomial interpolation formula is used. The polynomial interpolation formula, $p_n(x)$, is used to approximate curves with polynomial functions [28], where n is the degree of the polynomial.

Polynomial interpolation of degree n, requires $n+1$ points with their respective function evaluations. Given a set of points $x_1, \cdots, x_{n+1}$ that satisfying the condition $x_1 < x_2 < \cdots < x_{n+1}$, polynomial interpolation can determine $p_n(x)$ for any x within the interval of x_1 to x_{n+1}. The interpolated polynomial is described with the Lagrange form

$$p_n(x) = \sum_{j=1}^{n+1} f(x_j) \cdot L_j(x), \tag{C.1}$$

where n is the degree of the polynomial, f is the function to approximate and

$$L_j(x) = \prod_{i=1, i\neq j}^{n+1} \frac{x - x_i}{x_j - x_i}. \tag{C.2}$$

For a quadratic function where $n = 2$, we get

$$\begin{aligned} p_2(x) = f(x_1)\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + f(x_2)\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \\ + f(x_3)\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}. \end{aligned} \tag{C.3}$$

This is then differentiated to

$$p_2{}'(x) = f(x_1)\frac{(x-x_2)+(x-x_3)}{(x_1-x_2)(x_1-x_3)} + f(x_2)\frac{(x-x_1)+(x-x_3)}{(x_2-x_1)(x_2-x_3)} + f(x_3)\frac{(x-x_1)+(x-x_2)}{(x_3-x_1)(x_3-x_2)}. \tag{C.4}$$

The root of the quadratic function is found by letting $p_2{}'(x_r) = 0$. Both sides are multiplied by $(x_1-x_2)(x_2-x_3)(x_3-x_1)$ to simplify the equation, ending up with

$$f(x_1)(x_3-x_2)(2x_r-x_2-x_3) + f(x_2)(x_1-x_3)(2x-x_3-x_1) + f(x_3)(x_2-x_1)(2x_r-x_1-x_2) = 0 \tag{C.5}$$

which can be simplified into

$$2x_r[f(x_1)(x_3-x_2)+f(x_2)(x_1-x_3)+f(x_3)(x_2-x_1)] - f(x_1)(x_3^2-x_2^2)+f(x_2)(x_1^2-x_3^2)+f(x_3)(x_2^2-x_1^2) = 0. \tag{C.6}$$

Then x_r is isolated to

$$x_r = \frac{f(x_1)(x_2^2-x_3^2)+f(x_2)(x_3^2-x_1^2)+f(x_3)(x_1^2-x_2^2)}{2[f(x_1)(x_2-x_3)+f(x_2)(x_3-x_1)+f(x_3)(x_1-x_2)]} \tag{C.7}$$

which is the root of the quadratic function. In our report is $p_2(x)$ referred to as $q(x)$.