University of Pretoria

Department of Chemical Engineering

Masters dissertation

GPGPU-accelerated nonlinear state estimators:

Application to MPC-controlled bioreactor performance

Author:
Darren Craig Roos

Student Number:
u15041604

Co-Supervisor:
Mr. C Sandrock

University of Pretoria:
Chemical Engineering

Co-Supervisor:
Dr. JP de Villiers

University of Pretoria:
Electrical, Electronic and Computer Engineering

Co-Supervisor:
Ms. Esin Iplik

Mälardalen University:
Business, Society and Engineering

Submitted in partial fulfilment of the requirements for the degree
Master of Control Engineering in the Faculty of Chemical engineering, University of Pretoria

2021-05-17

Ethics statement

The author, whose name appears on the title page of this dissertation, has obtained, for the research described in this work, the applicable research ethics approval. The author declares that he/she has observed the ethical standards required in terms of the University of Pretoria's Code of ethics for researchers and the Policy guidelines for responsible research.

GPGPU-accelerated nonlinear state estimators:

Application to MPC-controlled bioreactor performance

Abstract

Practical control problems are subject to dealing with instrumentation noise and inaccurate models. These can be modelled as measurement and state noise, respectively. Nonlinear state estimators, for example a particle filter, can be used to mitigate these effects. However, they are usually computationally expensive which makes them impractical for industrial use. This text investigates using General Purpose Graphics Processing Units (GPGPU) to improve the performance particle and Gaussian sum filters by parallelizing their prediction, update and resampling steps. GPGPU accelerated filters are found to outperform non-accelerated filters as the number of particle increases. GPGPU acceleration also allows particle filters with $2^{19.5}$ particles to be used on systems with dynamic time constants on the order of 0.1 s and for Gaussian sum filters with $2^{18.5}$ particles to be used with time constants on the order of 1 s.

The filters are applied to a bioreactor system containing *R. Oryzae*, where MPC control is applied to the production phase fumaric acid and glucose concentrations. The bioreactor is modelled using results from Iplik (2017) and Swart (2019). It is found that the GPGPU filters' improved run times allow for more particles to be used which provides increased filter accuracy and thus better performance. This improved performance comes at the cost of consuming more energy. Thus, it is believed that the GPGPU implementations should be used for applications with complex dynamics/noise that require large numbers of particles and/or high sampling rates.

Contents

List of Figures

List of Tables

Nomenclature

The nomenclature contains symbols that are used but not defined in immediate surrounding text.

$p(X = x)$	The probability density function of a random variable X evaluated at x. It is written $p(x)$ when X is clear from the context
$p(x\|y)$	The conditional probability of X given y
$p(x_1, ..., x_n)$	The joint probability between multiple random variables $X_1, ..., X_n$
$\mathcal{N}(x\|\mu, \Sigma)$	A multivariate normal distribution with mean μ and covariance Σ
$\mathcal{N}_\Sigma(x\|\mu, \mathbf{\Sigma})$	A Gaussian sum distribution with means μ and covariances $\mathbf{\Sigma}$
P_{ISE}	Sum of the integral of the squared error between the set point and the process outputs
$q_{k\|j,i}$	A particle point estimate for the i'th particle at time step k given the measurement at the j'th time step
$w_{k\|j,i}$	A particle weight for the i'th particle at time step k given the measurement at the j'th time step
$\sigma^l_{k\|j,i}$	The l'th sigma point estimate for the i'th particle at time step k given the measurement at the j'th time step
$\delta(x)$	The Dirac delta function at x

1 Introduction

This chapter gives context to this work. It contains information about the project background, aims and scope.

1.1 Background

Industrial and chemical processes make use of automatic controllers to keep controlled variables at their set points. Correcting and maintaining set points costs energy. Thus, proper control is of paramount importance to the energy efficiency of a process.

Controllers are designed to use measurements from the system as well as a model of the system. Information from sensors is noisy (Seborg and Mellichamp, 2006), and it is possible to represent model inaccuracies as noise (Skogestad and Postlethwaite, 2005). State estimators aim to filter out these noise components and provide an estimate of the true state of the system. They are finding increasing industrial applications due to their ability to increase energy efficiency (Flemming and Sonner, 2006).

The pioneering work done by Kalman (1960) saw the creation of what is now known as the Kalman filter. The Kalman filter provided a closed form optimal solution to the state estimation problem when the system is linear and the noise is Gaussian. Since then much work has been done to find solutions to the more difficult cases where the assumptions do not hold.

The Extended Kalman Filter (EKF) developed by Gee *et al.* (1962) overcomes the requirement of the system to be linear by using a linear Taylor series approximation of the nonlinear model. The Unscented Kalman Filter (UKF) makes provisions for nonlinear systems by using weighted points which are propagated through the system's model and used to reconstruct a Gaussian estimation (Julier and Uhlmann, 2004). Quadrature approximation Kalman filters (QKF) use suitably chosen multi-dimensional quadrature

points of the random variable to approximate the expected value of the application of the non-linear system equation on the random variable (Arasarathnam *et al.*, 2007).

The greatest drawback to the use of the EKF, UKF and QKF is that the posterior estimate is always reformulated to be Gaussian. This is obviously not the case, since passing a Gaussian through a nonlinear function is unlikely to yield a Gaussian result. Gaussian sum filters represent the distributions as Gaussian mixtures, thus allowing any distribution to be approximated since any probability density function can be described as a (potentially infinite) sum of Gaussian distributions (Sorenson and Alspach, 1971). This has lead to Gaussian Sum Extended Kalman Filters (GS-EKF), Gaussian Sum Unscented Kalman Filters (GS-UKF), Gaussian Sum Quadrature approximation Kalman Filters (GS-QKF), and many other variations which can be applied to nonlinear systems with non-Gaussian noise.

Particle filters use weighted Monte Carlo samples to represent the distributions, thus also allowing their application to cases with nonlinear systems and non-Gaussian noise. The distribution is sampled randomly and weighted according to their probability and then the samples (called particles) are propagated through the system's model.

The disadvantage of particle filters and Gaussian sum filters is the curse of dimensionality: as the number of state dimensions increase, the number of samples/Gaussian distributions (both known as particles) needs to increase drastically to maintain the same accuracy. This either limits the size of the problem that can be solved or it limits the frequency of the control cycle. Both of these problems lead to less optimal use of energy resources in the system.

General Purpose computing on Graphical Processing Units (GPGPU) allow the same operation to be performed on different data simultaneously. This can be used to alleviate the aforementioned disadvantages, thereby improving energy efficiency for high dimensional systems. For example, in the particle filter's prediction step, all the particles need to be passed through the state transition function. This operation can be done on all the particles simultaneously as there are no dependencies between the data or the operation.

This work analyses the particle filter and Gaussian sum unscented Kalman filter for aspects that would receive significant speed-up from GPGPU. The accelerated and non-accelerated algorithms are implemented in Python and performance is measured on their application to a MPC controlled bioreactor model.

This project is done in the context of intradepartmental collaboration between the Process Modelling and Control group and the Bioreaction Engineering group within the chemical engineering faculty, as well as inter-university collaboration between the University of

Pretoria and Mälardalen University. It is in this context that the bioreactor model was chosen for investigation.

1.2 Deliverables

This work aims to identify aspects of the particle filter and Gaussian sum unscented Kalman filter for GPGPU acceleration. Next, it aims to deliver the accelerated and non-accelerated algorithms in Python. The open loop performance for the filters is investigated. The effect of GPGPU acceleration on the closed loop performance of the simulated bioreactor model is also investigated. To this end, the work aims to develop a bioreactor model for the purpose of closed loop simulation performance testing, using the results from Swart (2019). A model predictive controller is developed and used for control.

1.3 Research questions

1. Where could GPGPU be used in the nonlinear filtering algorithms to improve performance?
2. How much efficiency/performance can be gained by using GPGPU on these filters?
3. What effect does the aforementioned efficiency improvement have on the performance of a modelled bioreactor?

1.4 Scope and deliverables

The limiting factors of this project are: cooperation — the project needed to allow for interdepartmental collaboration between the Process Modelling and Control group and the Bioreaction Engineering group, as well as the inter-university cooperation between the University of Pretoria and Mälardalen University; time — the project could only run for 10 months; availability — only the hardware available to Professor Pieter de Villiers at the University of Pretoria was used.

This work's results focus on the application specific results of the GPGPU accelerated filters to the performance of a bioreactor with a single type and instance of state and

measurement noise. For time reasons, the work also only aims to compare performance between GPU and CPU implementations, as opposed to other variables, for example the performance differences between numbers of particles. Further delimitations are clearly shown throughout the text as for example in Delimitation 1.1. They will frequently show the fixation of some experimental variable, for example the programming language.

Delimitation 1.1. Only Python: All code is written in Python. This includes GPGPU kernels which are compiled from Python code using the library `numba` and by using the CUDA enabled libraries `cupy` and `pytorch`. This allows for faster development in the limited time.

The following outputs are delivered: a code library containing GPGPU accelerated and non-accelerated implementations of a particle filter and a Gaussian sum unscented Kalman filter, a model predictive controller implementation, and a model of a bioreactor; a dissertation containing the relevant theory and background on the topic, as well as discussion of the results obtained by using the code library's simulations; and a journal article submission on the work of the GPGPU accelerated filters.

1.5 Implementation

This document outlines the theoretical and practical methods of the investigation. The results shown in this document are generated from a library of code found at `https://github.com/darren-roos/gpu_se`. Installation instructions for the source code can be found in the `README.md` file of the code library.

A cache of the results can be found at `https://github.com/darren-roos/picklejar`. The results available in the cache are from runs performed on a machine with an AMD Ryzen 5 2400G, 32 GB of RAM, using only one core with a clock speed of 3.2 GHz. As well as a GeForce GTX 1070 with 8 GB of on board memory, 1920 cores with a clock speed of 1.683 GHz. The machine was running Ubuntu 18.04.4 LTS with a Python 3.8.2 and CUDA 10.2 environment.

Further details about the Python environment can be found in the `environment.yml` file in the code repository.

Part I

Literature review

2 Theoretical background

This literature review consists of two parts. The first part introduces the relevant theory required for an understanding of the work presented in the rest of the dissertation. A basic and limited introduction of relevant topics in measure theory, topology, graph theory and probability theory is followed by a detailed discussion on dynamic Bayesian networks. The section concludes by reintroducing the reader to model predictive control and covering the relevant computer science theory; most notably, big O notation and the idea of parallelization. It contains mostly "low-order thinking" referencing that quotes sources without critical discussion.

The part concludes with a review of literature on the current state-of-the-art in GPGPU accelerated state estimators. This section aims to contain "high-order thinking" comparison of reference results and methodologies.

2.1 Measure theory and topology

This work introduces probability theory using the background of measure theory. Readers who are unfamiliar with measure theory, but are comfortable with probability theory should feel free to skip this section as well as later measure theory based derivations of common probability theory topics. All derivations directly linked to the application of state estimators are understandable for readers with an axiomatic probability theory background.

Measure theory aims to generalize concepts of length, area, and volume so that they can be applied on more general spaces. The following terms are defined: σ-algebra, measurable space, measure, measure space, topological space, and Borel σ-algebra.

Definition 2.1. σ-algebra A collection $\mathcal{S}$ of subsets on a set S is called a σ-algebra on S iff:

1. $\varnothing \in \mathcal{S}$
2. If $s \in \mathcal{S}$, then $S \setminus s \in \mathcal{S}$
3. If $s_i \in \mathcal{S}, i \in \mathbb{N}$, then $\bigcup_{i=0}^{\infty} s_i \in \mathcal{S}$

where $\varnothing$ is the empty set. The most trivial σ-algebras are $\mathcal{P}(S)$ (the power set on S) and $\{\varnothing, S\}$. Sets in $\mathcal{S}$ are called $\mathcal{S}$-measurable.

$\sigma(T)$ is the σ-algebra generated by a collection of subsets of S called T, and is defined as the smallest σ-algebra that contains all elements of T (Tao, 2011). A σ-algebra is a formal concept for a mathematical object that can be measured. For example, if parts of a piece of string were to be painted, one could measure the length of string that is painted. It is possible for none of the string to be coloured and for the painted length to be zero (corresponding to point one above). The length of the unpainted parts of the string is equal to the total length of the string minus the painted length. This corresponds to the second point from above. Finally, the third point corresponds to the case when non-contiguous parts of the string are painted. In this case, the total painted length would be the sum of all the painted sections.

The σ-algebra (painted parts of the string) and the set on which it is formed (the string itself) forms a measurable space. Formally, this is defined as:

Definition 2.2. Measurable space The tuple $(S, \mathcal{S})$ containing the set S and a σ-algebra $\mathcal{S}$ on S

The length measurement in the string example is an example of a real measure. Real measures are formally defined as:

Definition 2.3. Real measure (Bogachev, 2007) A function μ on a measurable space $(S, \mathcal{S})$ with the following properties:

1. $\mu : \mathcal{S} \to [0, \infty)$
2. $\mu(\varnothing) = 0$
3. $\mu\left(\bigcup_{i=0}^{\infty} A_i\right) = \sum_{i=0}^{\infty} \mu(A_i)$ for all countably infinite, pairwise disjoint collections $\{A_i : A_i \in \mathcal{S}\}$

A measurable space and a measure can be combined to form a measure space. The string example is an illustration of a measure space.

Definition 2.4. Measure spaces consist of a tuple $(S, \mathcal{S}, \mu)$, where μ is a measure on the measurable space $(S, \mathcal{S})$.

Measurable functions, of which random variables are an example, are abstractions of the concept of continuous functions in calculus.

Definition 2.5. Measurable functions are functions $f : X \to Y$, where X and Y are sets belonging to two measurable spaces (X, Σ) and (Y, T), such that

$$f^{-1}(E) \triangleq x \in X | f(x) \in E \in \Sigma \; \forall E \in T$$

Push-forward measures transfer a measure from one measurable space to another using a measurable function.

Definition 2.6. Push-forward measures Given two measurable spaces (X, Σ) and (Y, T); a measure $\mu : \Sigma \to [0, \infty]$; and a measurable function $f : X \to Y$, the push-forward measure is $\mu^* \triangleq \mu \circ f^{-1}$.

For the purposes of this text, we will use a specific category of measurable spaces known as topological spaces. They are the same as measurable spaces, but have the additional quality that they are also closed under infinite intersections.

Definition 2.7. Topological spaces are measurable spaces (S, T), where S is a set and T is a collection of subsets of S, with the following properties:

1. $\varnothing \in T$
2. $S \in T$
3. T is closed under infinite unions
4. T is closed under finite intersections

Sets in T are known as the open sets (Folland, 1999).

A base B of a topology T, is a collection of open sets such that $t = \bigcup b_i \quad \forall t \in T, \forall b_i \in B$. For example, a base on $\mathbb{R}^n$ is the collection of all open balls in $\mathbb{R}^n$.

Definition 2.8. Borel σ-algebra (Folland, 1999) For a topological space (S, T), $\mathcal{B} = \sigma(T)$ is defined as the Borel σ-algebra on the topological space. This is sometimes written $\mathcal{B}(S)$ when the open sets T are known or obvious from context. Elements of $\mathcal{B}(S)$ are called Borel sets of S.

For more detailed coverage please see Folland (1999), Bogachev (2007), or Tao (2011).

2.2 Graph theory

This text discusses graphs as seen in Figure 2.1 and 2.2. Chartrand and P Zhang (2012) state that graphs are mathematical abstractions that are used to show connections and relationships between objects. Figure 2.1 and Figure 2.2 show labelled circles (letters in Figure 2.1 and numbers in Figure 2.1) called vertices. The lines or arrows that join vertices are called edges. Figure 2.1 shows edges that are arrows and thus have a direction, while and Figure 2.2 shows edges that do not have a direction.

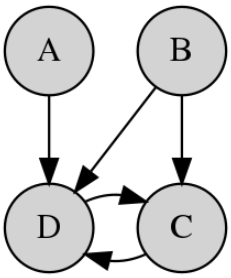

Figure 2.1: Directed graph.

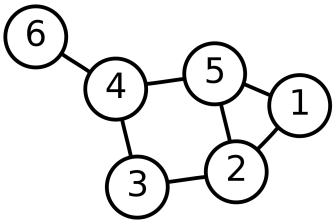

Figure 2.2: Undirected graph.

The following are taken from Trudeau (1993). Please refer to that text for more in depth coverage.

Definition 2.9. Graphs are discrete mathematical objects comprised of a pair of sets (V, E). A set of vertices V and a set edges E.

Definition 2.10. Directed/Undirected graph Directed graphs have a set of edges E that contains ordered pairs of vertices of the form (V_i, V_j). This means that the edge (V_i, V_j) is not the same as the edge (V_j, V_i) An undirected graph contains unordered pairs of vertices, such that the edge (V_i, V_j) is the same as the edge (V_j, V_i).

Definition 2.11. Paths P in a graph G are sequences of vertices $V_1, V_2, ..., V_n \in V$, such that $(V_i, V_j) \in E$ for each consecutive pair of vertices V_i and V_j in P.

Example 2.1. Paths Consider Figure 2.1. According to Definition 2.11, B, C, D, C is a path in this graph because (B, C), (C, D), and (D, C) are all edges in the graph. However, A, D, C, B is not a path because while there is an edge (B, C), there is no edge (C, B).

Definition 2.12. Directed Acyclic Graphs (DAG) A common subset of directed graphs are acyclic graphs. A graph is acyclic iff there exists no path P in G such that the first vertex in the path is the same as the last vertex in the path.

Common relationships between vertices arise when designing algorithms that operate on DAGs as described by West (1996). Defining these relationships is useful as they frequently simplify the algorithms.

Definition 2.13. Parents of a vertex V_i in a directed graph are all vertices V_j such that $(V_j, V_i) \in E$

Definition 2.14. Children of a vertex V_i in a directed graph are all vertices V_j such that $(V_i, V_j) \in E$

Definition 2.15. Ancestors in a DAG of a vertex V_i is the set of vertices that contains all vertices V_j that have a path from V_j to V_i.

Definition 2.16. Descendants in a DAG of V_i is the set of vertices that contains all vertices V_j that have a path from V_i to V_j.

2.3 Probability theory

Definition 2.17. Probability spaces (Durrett, 1996) are measure spaces defined by a tuple $(\Omega, \mathcal{F}, P)$. Ω is the sample space and contains the set of all possible outcomes. $\mathcal{F} \subseteq 2^{\Omega}$ is the σ-algebra on Ω containing the set of events. In addition to being a σ-algebra $\Omega \in \mathcal{F}$. $(\Omega, \mathcal{F})$ forms a measurable space, and P is a measure on that space, known as the probability measure. In addition to being a measure function, P must also satisfy:

1. $P : \mathcal{F} \to [0, 1]$
2. $P(\varnothing) = 0$
3. $P(\Omega) = 1$

Definition 2.18. Random variable Given a probability space $(\Omega, \mathcal{F}, P)$ and a measurable space $(E, \mathcal{E})$, a $(E, \mathcal{E})$-valued random variable is a measurable function $X : \Omega \to E$.

Papoulis (1965) states that random variables can be described by probability distributions, which are push-forward measures $\mu_X = P \circ X^{-1}$. Note that while $X : \Omega \to E$, that $X^{-1} : \mathcal{E} \to \mathcal{F}$. Without this, $\mu_X : \mathcal{E} \to \mathcal{F} \to [0, 1]$ would not be possible since $P : \mathcal{F} \to [0, 1]$.

The class of $(\mathbb{R}, \mathcal{B})$-valued random variables — also known as real valued random variables — are considered. The topology for the real space is selected to be the set of semi-infinite

intervals $T = \{(-\infty, x) : x \in \mathbb{R}\}$, and thus $\mathcal{B} = \sigma(T)$. In this text, the phrase "random variable" refers to a real valued random variable unless otherwise stated.

Definition 2.19. Cumulative Distribution Function (CDF) is an alternate way of defining a random variable. It is mathematically related to the probability distribution by

$$F_X(x) = \mu_X((-\infty, x]) \tag{2.1}$$

Definition 2.20. Probability density functions (PDF) also describe random variables:

$$f_X(x) = \frac{dF_x}{dx} \tag{2.2}$$

PDFs are often also written as $p(X = x)$ or most frequently $p(x)$. The last notation is the preferred notation for this text.

While the last notation is simple, it carries a lot of information behind it: It refers to the derivative of a CDF of a probability distribution from $-\infty$ to x, where a probability distribution is a push-forward measure of a probability measure P through the random variable X, which is a measurable real valued function on an event space Ω of a probability triple $(\Omega, \mathcal{F}, P)$.

2.3.1 Joints

The following discussion is taken from Papoulis (1965) and Blitzstein and Hwang (2014). Given the vector of random variables $X = [X_1, X_2, ..., X_n] : \Omega \to \mathbb{R}^n$, the joint probability measure is

$$P(X_1, X_2, ..., X_n) = P(X_1 \cap X_2 \cap ... \cap X_n) \tag{2.3}$$

When it is clear from the context, this text refers to both multidimensional and unidimensional random variables as random variables. The probability distribution is given by $\mu_X : \mathbb{R}^n \to \mathcal{F} \to [0, 1] = P \circ X^{-1}$. From this, the CDF

$$F_X(x_1, x_2, ..., x_n) = \mu_X((-\infty, x_1], ..., (-\infty, x_n]) \tag{2.4}$$

is similar to Equation 2.1. The PDF is given by

$$\begin{aligned} f_X(x_1, x_2, ..., x_n) &= p(x_1, x_2, ..., x_n) \\ &= p(x_{1:n}) \\ &= \frac{\partial^n F_X(x_1, ..., x_n)}{\partial x_1 \partial x_2 ... \partial x_n} \end{aligned} \tag{2.5}$$

Definition 2.21. Marginal distributions are probability distributions containing subsets of random variables, called the marginal variables. The other variables are said to have been marginalized out. Given the joint $p(x_{1:n})$, the marginal that has marginalized out x_1 is given by

$$p(x_{2:n}) = \int_{-\infty}^{\infty} p(x_{1:n}) dx_1 \tag{2.6}$$

similarly, the marginal of x_m, where $1 \leq m \leq n$ is given by

$$p(x_m) = \int_{-\infty}^{\infty} ... \int_{-\infty}^{\infty} p(x_{1:m-1}, x_{m+1:n}) dx_1 ... dx_{m-1} dx_{m+1} ... dx_n \tag{2.7}$$

Definition 2.22. Conditional probability distribution is the distribution of a joint distribution given information about one or more of the variables. For example, given the joint $p(x, y, z)$, the distribution of x and y *given* z can be written as $p(x, y|z)$. Mathematically, it is defined as

$$p(x, y|z) = \frac{p(x, y, z)}{p(z)} \tag{2.8}$$

Similarly,

$$p(x|y, z) = \frac{p(x, y, z)}{p(y, z)} \tag{2.9}$$

Theorem 2.1. Chain rule The joint $p(x_{1:n})$ can be factorized

$$p(x_{1:n}) = p(x_1)p(x_2|x_1)p(x_3|x_{1:2})...p(x_n|x_{1:n-1}) \tag{2.10}$$

Proof. Substituting in Definition 2.22 into Equation 2.10 gives

$$p(x_{1:n}) = p(x_1)\frac{p(x_{1:2})}{p(x_1)}\frac{p(x_{1:3})}{p(x_{1:2})}...\frac{p(x_{1:n})}{p(x_{1:n-1})} \tag{2.11}$$

which clearly shows how terms cancel to give the desired result □

Theorem 2.2. Bayes' theorem allows a prior distribution $p(x)$ to be updated given new information y, where the support that y provides for x is known. This allows the posterior $p(x|y)$ to be calculated, without having to know the joint. It is formally stated as

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{2.12}$$

Proof.

$$\begin{aligned} p(x,y) &= p(x,y) \\ p(x|y)p(y) &= p(y|x)p(x) \end{aligned} \tag{2.13}$$

□

Definition 2.23. Independence implies that a joint distribution factorizes as the product of the marginal distributions of the individual variables

$$p(x_{1:n}) = \prod_{i=1}^{n} p(x_i) \tag{2.14}$$

Definition 2.24. Conditional independence of a joint works the same way as independence, but with a certain given random variable z.

$$p(x_{1:n}|z) = \prod_{i=1}^{n} p(x_i|z) \tag{2.15}$$

2.3.2 Moments

Definition 2.25. Expectation of a function of a random vector (Jaynes *et al.*, 2003) shows that the expectation $\mathbb{E}(\cdot)$ of a function $f(X) : \mathbb{R}^n \to \mathbb{R}$ of a random variable X, is given by

$$\begin{aligned} \mathbb{E}(f(X)) &= \int_{\Omega} f(X(\omega))dP(\omega) \\ &= \int_{\mathbb{R}} f(x)dF_X(x) \\ &= \int_{\mathbb{R}} f(x)p(x)dx \end{aligned} \tag{2.16}$$

where the second equality is found using the Law of the Unconscious Statistician (LOTUS), which allows the variable of integration to be changed using a push-forward measure. In this case the push-forward measure is the probability distribution μ_X. The third equality is found by using a change of variables with the PDF in Equation 2.2.

Definition 2.26. Covariance measures how much two random variables vary against one another. Jaynes *et al.* (2003) show that if both variables tend to have larger and smaller values at the same points, then the covariance is positive. Similarly, if they have their large and small values at opposite points, then the covariance is negative. The covariance matrix is calculated for two random column vectors X and Y as

$$\text{cov}(X,Y) = \mathbb{E}(XY^T) - E(X)E(Y)^T \tag{2.17}$$

Moments (sometimes called raw moments) are quantitative measures of the shape of a function. The n-th moment of a random variable about a point c is defined to be

$$\mu_n(X) = \mathbb{E}((X - c)^n) \tag{2.18}$$

If $c = 0$ and $n = 1$ then $\mu_1 = \mathbb{E}(X) = \mu$ which is the mean of the random variable. Central moments are measures of the shape of a random variable about the mean. Thus, the n-th central moment is defined as

$$\mu'_n = \mathbb{E}((X - \mathbb{E}(X))^n) \tag{2.19}$$

The second central moment is called the variance and is also equal to the covariance of a random variable with respect to itself.

Definition 2.27. Gaussian random variables or normally distributed random variables are d-dimensional real-valued random vectors that have a PDF of the form

$$\begin{aligned} p(x|\mu, \Sigma) &= \frac{\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma(x-\mu)\right)}{\sqrt{(2\pi)^d \det(\Sigma)}} \\ &= \mathcal{N}(x|\mu, \Sigma) \end{aligned} \tag{2.20}$$

where μ is called the mean/average and is also the first moment, and Σ is called the covariance matrix/covariance and is also the second central moment.

A random variable X described by a PDF that is Gaussian with mean μ and covariance Σ is written $X \sim \mathcal{N}(\mu, \Sigma)$.

2.3.3 Stochastic processes

Gallager (2013) defines a stochastic process to be a collection of related random variables. The random variables are indexed by some set known as the index set. Often this set is used to indicate time, which leads to the interpretation that a stochastic process models the evolution of a probability distribution over time. There are numerous examples of stochastic processes in science and engineering, for example the Poisson process and the Wiener process.

Definition 2.28. Stochastic processes are sequences of random variables from the same probability space $(\Omega, \mathcal{F}, P)$ to the same measure space, known as the state space

(only the real space $(\mathbb{R}^n, \mathcal{B})$ is considered in this text), indexed by some set T. This is denoted $\{X_t\}_{t\in T}$

Stochastic processes are often grouped by their properties. Frequent qualities used are the cardinality of the index set, cardinality of the state space, and relationship between variables in the sequence. The cardinality generally refers to whether the space is discrete or continuous. Due to the index set frequently representing time, discrete time stochastic processes refer to processes with an index set containing a finite or countably infinite number of elements. Conversely, if the index set contains an infinite number of elements, the process is said to be a continuous time stochastic process. Similarly, discrete-valued and continuous-valued processes refer to stochastic processes with finite (or countably infinite) or infinite state spaces, respectively. Differences between sequences with different relationships between elements generally form different classes of random variables.

Definition 2.29. Markov property (Durrett, 2012) A property of a stochastic process where X_t is conditionally independent of X_s given X_r, where $s < r < t$. This means that the current state contains all the required information about the future behaviour of the stochastic process.

Discrete time processes with the Markov property are called Markov chains, while continuous time processes with the Markov property are called Markov processes.

Definition 2.30. Hidden Markov Model (HMM) (Bishop, 2013) Consider two stochastic processes $\{X_t\}_{t\in T}$ and $\{Y_t\}_{t\in T}$ on the same probability space $(\Omega, \mathcal{F}, P)$ and indexed by the same set T. (Note: they need not map to real spaces of the same dimension as each other). The pair $(\{X_t\}, \{Y_t\})_{t\in T}$ is called a HMM iff

1. $\{X_t\}_{t\in T}$ is an unobservable Markov chain with the transition PDF $p(x_t|x_{t-1})$
2. Y_n is conditionally independent on all previous states $X_{1:n-1}$ given X_n.
   $p(y_n|x_{1:n}) = p(y_n|x_n)$ is known as the emission/observation PDF

2.4 Dynamic Bayesian network

This discussion is taken from Bishop (2013). Dynamic Bayesian Networks (DBNs) result from the combination of probability theory and graph theory. This text uses them as a useful tool to help the reader gain a better understanding of HMMs and the probability calculations that follow from their filtering.

Definition 2.31. Bayesian networks are probabilistic graphical models that represent conditional probability relationships between random variables using DAGs. In a Bayesian network with variables $x_{1:n}$ where the joint would normally factorize using the chain rule (see Equation 2.10), the Bayesian network specifies the factorization

$$p(x_{1:n}) = \prod_{i=1}^{n} p(x_i|Parents(x_i)) \tag{2.21}$$

where *Parents(x_i)* represents a function that returns all random variables that are parents of x_i in the DAG. Often, Bayesian networks are sparse, thus this factorization is frequently simpler than the chain rule alternative.

A recurring problem arises when using Bayesian networks of how to determine conditional independence relationships. A method known as d-separation is the most common method used to determine this (Koski and Noble, 2011). There are various ways of defining and describing d-separation. The method presented here comes from Scutari and Denis (2014).

Definition 2.32. Conditional independence in Bayesian networks between a set of nodes $\mathcal{X}$ and $\mathcal{Y}$ given a set of nodes $\mathcal{Z}$ implies $\exists z \in \mathcal{Z}$ such that

$$p(x, y|z) = p(x|z)p(y|z)\ \forall x \in \mathcal{X}\ \forall y \in \mathcal{Y} \tag{2.22}$$

It is said that $\mathcal{Z}$ d-separates $\mathcal{X}$ and $\mathcal{Y}$.

Definition 2.33. Colliding node Consider any three consecutive nodes a, b, and c on a path P between two nodes x and y. b is a colliding node iff the nodes are configured: $a \rightarrow b \leftarrow c$

Definition 2.34. Blocked node Consider again any three consecutive nodes a, b, and c on a path P between two nodes x and y. Also consider the set of given nodes $\mathcal{Z}$. The node b is said to be blocked iff

1. $(b \cup descendant(b)) \cap \mathcal{Z} = \varnothing$ and b is a collider on P. Stated in words: that neither b nor any descendant of b in $\mathcal{Z}$ and b is a collider on the path between x and y
2. $b \in \mathcal{Z}$ and b is not a collider on P

where $descendant(b)$ is a function that returns the descendants of b.

Definition 2.35. Blocked path Consider a path P between two nodes x and y. P is said to be blocked iff it contains any blocked nodes.

Definition 2.36. d-separation $\mathcal{Z}$ d-separates $\mathcal{X}$ and $\mathcal{Y}$ iff every path from every node in $\mathcal{X}$ to every node in $\mathcal{Y}$ is blocked by any node in $\mathcal{Z}$.

Now that a firm foundation has been laid about Bayesian networks, it is possible to introduce dynamic Bayseian networks. Dynamic Bayesian networks relate a set of stochastic processes.

Example 2.2. Consider the dynamic Bayesian network seen in Figure 2.3. Notice that this network shows three links of the relation between the three stochastic processes $\{A_t\}_{t\in\mathbb{N}}$, $\{B_t\}_{t\in\mathbb{N}_0}$ and $\{C_t\}_{t\in\mathbb{N}_0}$.

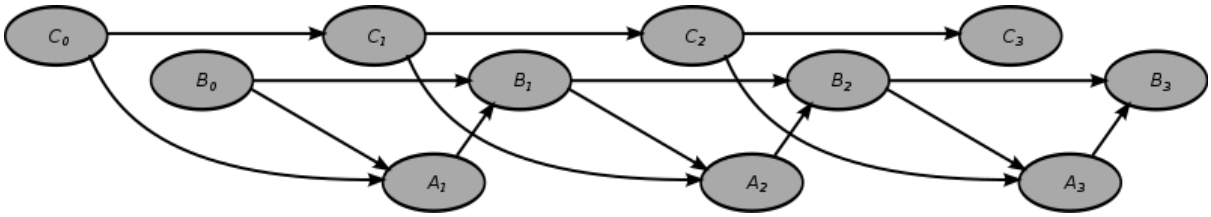

Figure 2.3: Three links in a dynamic Bayesian network (Lozenguez, 2016).

It is possible to simplify this diagram down to its most essential parts as seen in Figure 2.4. This diagram contains all the required dependence relationships, without cluttering up the diagram.

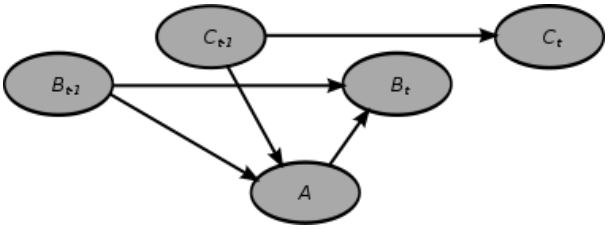

Figure 2.4: Simplified dynamic Bayesian network (Lozenguez, 2016).

Definition 2.37. Dynamic Bayesian Networks are defined by the tuple $(G_S, G_{\rightarrow})$, where $G_S = \{\{X_{(t,s)}\}_{(t,s)\in T\times S}\}$ is the set of n stochastic processes indexed over S and $G_{\rightarrow}$ is the set of conditional probability relations of the form $p(x_{(t+1,i)}|x_{(1:t,1:n)})$. If the network obeys the Markov principle, then the probability relations are of the form: $p(x_{(t+1,i)}|x_{(t,1:n)})$.

This text considers dynamic Bayesian network representations of an HMM that has been adapted to have a deterministic input into the hidden states. The simplified dynamic Bayesian network is shown in Figure 2.5.

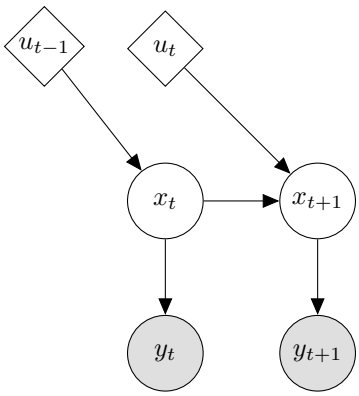

Figure 2.5: Dynamic Bayesian network of an adapted HMM. The diamond shape, indicates that the variable is deterministic; clear circles indicate the hidden variables; and the coloured circles indicate the observable variables. Adapted from Wilken (2015).

2.5 Model predictive control

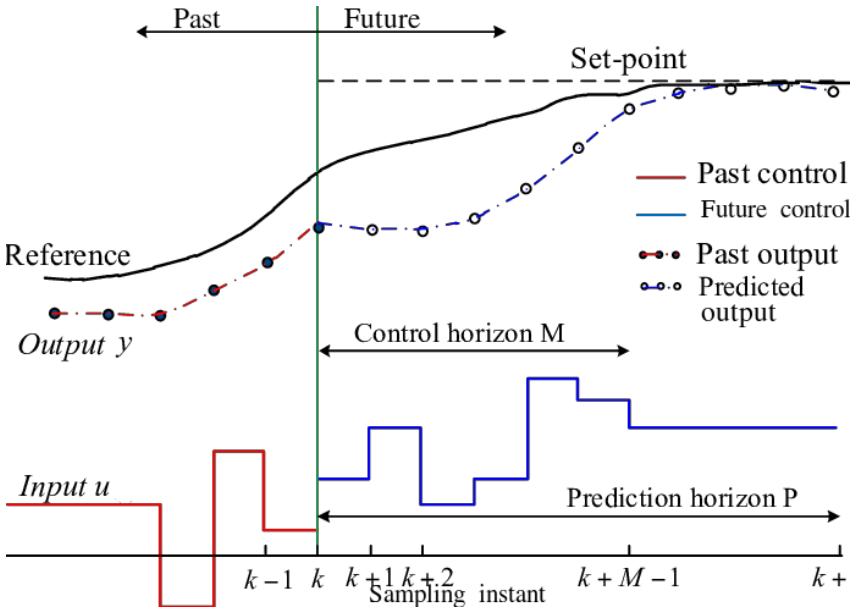

Figure 2.6: Example of MPC receding horizon approach. Adapted with permission from Yang, Liu, *et al.* (2017).

Model predictive control (MPC) is a well-established for advanced process control technique. Kouvaritakis and Cannon (2015) define MPC to use a dynamic model of the process, set point information and measurements from the system in an optimization process to find the optimal future inputs to the system that obey specified constraints on the system. MPC uses a receding horizon approach, so it only implements the first optimal control step and then recomputes another set of optimal future inputs at the next

control step (Seborg and Mellichamp, 2006). This is seen in Figure 2.6. MPC technology has the ability to be used in multivariable systems with constraints, delays, interaction and in cases where feedforward is needed.

ABB (2019) mentions several advantages of MPC controllers such as their ability to reduce variation in the controlled variables. This allows operators to select set points that are closer to the operation constraints, which usually results in a larger profit. It also has the advantage of being able to control several interacting variables that would otherwise have needed complex feedforward or decoupling technology, and it can work in cascade with existing base-layer PID control. Using MPC allows operators to think about the process on a higher level and allows them to interact with the system more intuitively (ABB, 2019). The main advantage of MPC is its receding horizon approach. It optimizes not only for the current time but also future times. MPC control also interacts very well with state estimators in the implementation of Stochastic MPC (SMPC), and it is for this reason — and those mentioned above — that MPC control is used in this project.

A linear MPC is used as the controller. The optimization problem is similar to the one developed by Wilken (2015) and is stated as

$$\begin{aligned} \min_{\mathbf{u}} \quad & \sum_{i=0}^{P-1} \left(e_i^T Q e_i + f_i^T R f_i \right) \\ x_{k+1} &= Ax_k + Bu_k \\ y_k &= Cx_k + Du_k \\ e_k &= r - y_k \\ f_k &= u_{\text{sp}} - u_i \\ Dy_k + e &\geq 0 \quad \forall\, 0 \leq k < P \end{aligned} \tag{2.23}$$

where $\mathbf{u}$ is the vector of all input values u_k; Q and R are diagonal tuning matrices that allow relative weighting between the error vector of the outputs e_k and the "error" vector of the inputs f_k; x_k is the vector of internal system states at the k-th instant; y_k is the vector of system outputs at the k-th instant; A, B, C, and D are linear time-invariant state space matrices for the system model; r and u_{sp} are the output and input reference vectors, respectively; $Dy_k + e \geq 0 \quad \forall\, 0 \leq k < N$ is the set of linear constraints on the output vectors defined by the constant matrix D and the constant vector e.

The structure of the problem in Equation 2.23 allows it to be formulated and solved as a quadratic programming (QP) problem. Edgar *et al.* (2001) define QP problems to have the structure:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \frac{1}{2}u^T Pu + c^T u \\ & Hu + b \leq 0 \end{aligned} \tag{2.24}$$

where P, and H are constant matrices and c, and b are constant vectors. If P is positive definite, then the QP problem becomes convex. It is no accident that Equation 2.23 can be reformulated as a convex QP problem. Since, although convex QP problems are non-linear, they have very efficient solvers because of the convexity. Convexity of Equation 2.23 is guaranteed if Q and R are positive definite, and since these are design variables, they can be chosen as such (Edgar *et al.*, 2001).

2.6 Algorithmic complexity and parallelization

This section by no means gives the reader a complete picture of the field of computer science. It aims to give the reader enough knowledge on the specific aspects that interact with this work. It is assumed that the reader is familiar with programming. Code examples use the Python programming language.

2.6.1 Big $\mathcal{O}$ notation

Cormen *et al.* (2009) describe the commonly used method for defining the upper-limiting behaviour/growth-rate of a function known as Big $\mathcal{O}$ notation. Its name comes from the fact that the growth rate of a function is sometimes called the order of the function. The notation used is: $f(n) = \mathcal{O}(g(n))$, which means that the function f limits the function g for large values of n.

Definition 2.38. Big $\mathcal{O}$ notation $f(n) = \mathcal{O}(g(n))$ iff $\exists M \in \mathbb{R},\ c \in \mathbb{R}$ such that $|f(n)| \leq M(g(n))\ \forall n \geq c$

From this definition, Lundqvist (2013) derives the following identities:

1. *Constant identity*: If $g(n) = c;\ c \in \mathbb{R}$, then $f(n) = 1$
2. *Product identity*: If $f_1 = \mathcal{O}(g_1)$ and $f_2 = \mathcal{O}(g_2)$, then $f_1 f_2 = \mathcal{O}(g_1 g_2)$

When g consists of a sum of different terms, the one that grows the fastest will determine f. The following list from Lundqvist (2013) shows common terms that appear in order from fastest to slowest:

Table 2.1: Common terms, their names and Big $\mathcal{O}$ notation.

Name	$\mathcal{O}(g)$
Constant	$\mathcal{O}(1)$
Logarithmic	$\mathcal{O}(\log(n))$
Linear	$\mathcal{O}(n)$
Quadratic	$\mathcal{O}(n^2)$
Polynomial	$\mathcal{O}(n^c)$
Exponential	$\mathcal{O}(c^n)$
Factorial	$\mathcal{O}(n!)$

Example 2.3. Consider a function

$$g(n) = n + 6 + 3n \log\left(n^7\right)$$

One can firstly use the logarithm law $\log(n^c) = c \log(n)$ to get

$$\mathcal{O}(g(n)) = n + 6 + 21n \log(n)$$

Then one can use the constant identity

$$\mathcal{O}(g(n)) = n + 1 + n \log(n)$$

Using Table 2.1, one can then see that 1 is definitely not the fastest growing term. It is also not possible for n to grow faster than $n \log(n)$, since the latter term is the product of the former term with another expression that grows with n. Thus,

$$\mathcal{O}(g(n)) = n \log(n)$$

2.6.2 Complexity

When dealing with a computational problem, it is common to have different solutions. It is often very helpful to be able to choose which solution/algorithm is best to solve a particular problem. In computer science, algorithms are usually compared by the resources required.

Computational devices usually have two main kinds of resources: processing power and

memory. Processing power affects how much *time* the program will run for, and is thus called the time complexity. The amount of memory the program need to do its calculations determines how much *space* it needs and it thus called the space complexity.

Consider, for example, the task of sorting a list, and two sorting algorithms: bubble sort and merge sort. Imagine using bubble sort on a list of numbers with 10 elements, and using merge sort on a list of numbers with 100 elements. It would obviously be unfair to compare the time or space complexity of the two algorithms in this case. Thus, when comparing algorithms it makes sense to set a standard.

A commonly used standard characterizes the performance by the smallest worst-case big $\mathcal{O}$ time/space the algorithm uses for a problem with size n. This method prevents bias in the form of problem size, and allows the growth rate of required resources to be compared. It does not allow one to trade-off the programming complexity of the algorithms with the algorithms time/space performance. Also, big $\mathcal{O}$ complexity can often give distorted results for small problem sizes. However, for most practical cases and for the cases in this work, big $\mathcal{O}$ complexity is a suitable means of comparison.

Unit operations and unit space

Before a formal definition of complexity can be given, one first needs to define the smallest amounts of computation and space. Unit space is the easier to measure since the problem size n serves as a good metric.

Definition 2.39. Unit space Given an algorithm with input size n, we define unit space as the amount of space that is as large as $\frac{1}{n}$ of the input size

For example, if given a list of integers to sort, then the unit space would be the size of an integer. Unit time is more complicated to define.

Definition 2.40. Unit time is the time taken to do an operation that could not be divided up into smaller operations. For example, an arithmetic operation between two operands, a boolean operation between two operands, an execution branch, or a memory access.

Using Definition 2.39 and Definition 2.40, it becomes possible to formalize the concept of time and space complexity.

Definition 2.41. Time complexity Given a function $g(n)$ that counts the number of unit operations an algorithm A performs in the worst-case run with problem size n, the

time complexity $T_A(n)$ (or just $T(n)$ when A is clear form the context) is defined to be the smallest $\mathcal{O}(g)$.

It is important to note that it is the worst-case that is considered. It is also important that one considers the smallest $\mathcal{O}(g)$, because as seen in Example 2.3, if $g(n) = n + 6 + 21n\log(n)$, then both $f_1(n) = n + 1 + n\log(n)$ and $f_2(n) = n\log(n)$ bound g, but the smallest one is f_2.

Definition 2.42. Space complexity Given a function $g(n)$ that counts the number of additional unit spaces an algorithm A requires in the worst-case run with problem size n, the space complexity $S_A(n)$ (or just $S(n)$ when A is clear form the context) is defined to be the smallest $\mathcal{O}(g)$.

Note that space complexity looks for the smallest worst-case $\mathcal{O}(g)$, and also that it only looks at the additional space required by the algorithm: the initial space taken up by the arguments to the algorithm is not counted.

2.7 Parallel computing

Flynn (1972) designed a taxonomy of computer architectures based on how they handle instructions and data. The taxonomy divides systems into four types: Single-Instruction Single-Data (SISD), Single-Instruction Multiple-Data (SIMD), Multiple-Instruction Single-Data (MISD), and Multiple-Instruction Multiple-Data (MIMD).

SISD architectures are the classic serial design where one instruction is executed on one piece of data at a time. SISD corresponds to the architecture described by Neumann (1945). This kind of architecture is often called a single-core processor.

MISD systems execute multiple instructions on a single piece of data. These architectures make use of a pipeline. It is also used when a signal needs to be passed through multiple frequency filters. MISD finds common use in systems that require high degrees of fault tolerance, where the same instructions are executed on one piece of data so that any computing faults are detected.

MIMD systems are the modern standard of computer hardware and is commonly called multi-core architectures. In this architecture each core can execute different instructions on different data sets. Most personal computers use this architecture. They have the advantage over multiple SISD systems in that MIMD systems can readily share memory space and thus can cooperate more effectively to solve a problem.

SIMD systems perform the same instruction on multiple sets of data. GPGPUs are examples of SIMD systems. SIMD developed naturally from the repetitive linear algebra operations required for graphics processing. Alternate applications of the SIMD architecture have been found in biotechnology, finance and data science, where GPGPU is used to speed-up the performance of computationally expensive calculations. The performance improvement from parallelization is limited by Amdahl's law.

Definition 2.43. Amdahl's law (Amdahl, 1967) Given a program which allows a fraction p to be parallelized and a system with n processors, the maximum speed-up S that can be achieved is given by

$$S = \frac{1}{\frac{p}{n} + 1 - p} \tag{2.25}$$

2.7.1 CUDA

CUDA is a parallel computing API that gives users the ability to write programs that can execute on Nvidia's GPU architectures — which are SIMD systems (Tuomanen, 2018). It is written for the C++ programming language, but has many wrappers that allow it to be used in other programming languages. Numba is a Python library that contains one such wrapping. Understanding CUDA's programming model is important when designing algorithms for this architecture.

Sanders and Kandrot (2010) describes the CUDA architecture in terms of "kernels", which are functions that can be called numerous times (single-instruction) with different parameters (multiple-data) and executed in parallel. Each instance of a running kernel is called a thread. Each thread has its own section of memory. Threads are grouped into "thread blocks" or just "blocks". Thread blocks can be structured to be 1-, 2-, or 3-dimensional. For example, a 2-dimensional thread block with shape $(4, 9)$ will have 36 threads running the same kernel with different data in parallel. Each thread in a block is executed simultaneously on the same streaming multiprocessor. Depending on the specific hardware implementation, the maximum number of threads that can run simultaneously in a block may differ. Each block also has an amount of memory that is shared between all threads in that block.

Sanders and Kandrot (2010) states that one can also have multiple thread blocks, which form a "block grid" or just a "grid". This grid can also be 1-, 2-, or 3-dimensional, but all blocks in the grid must have the same shape as each other. Blocks in the grid are executed on different GPU multiprocessors. All threads in a grid execute the same kernel. There can be multiple grids awaiting execution on the GPU each with a different kernel to be executed.

All threads across all grids have access to a global memory space. However, it is important to note that repeated accessing of this memory by kernels can reduce the performance of the program (Sanders and Kandrot, 2010). There is a section of memory that all threads can read, but not write to. This section of memory can only be written to using the CUDA API outside of a kernel function. Using this memory over global memory can improve the performance of a program. Figure 2.7 provides a useful visualization of the above concepts.

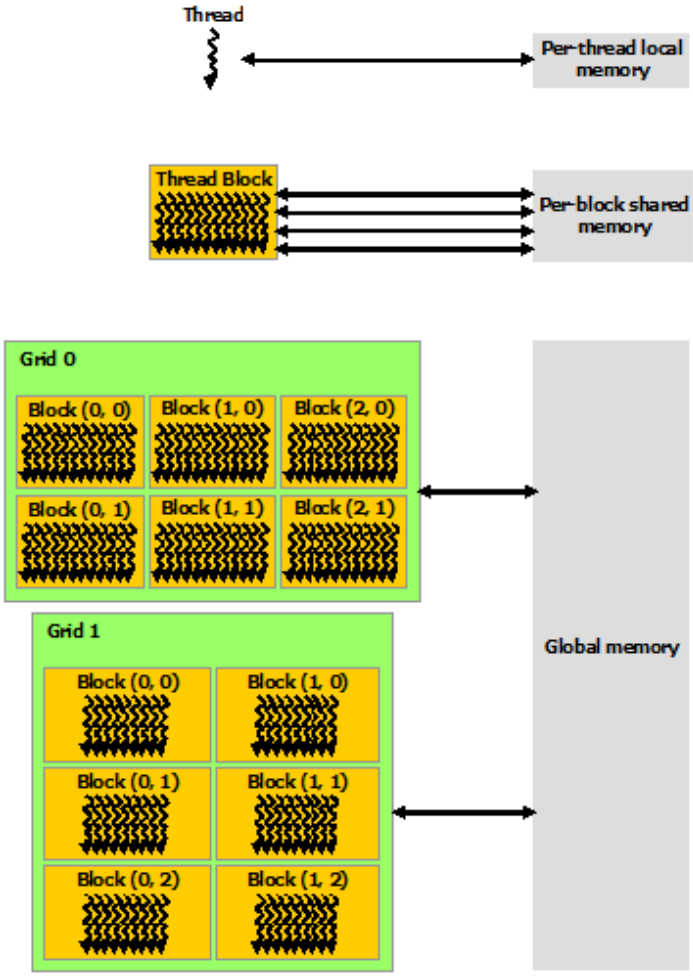

Figure 2.7: Thread, block and grid structure together with the memory hierarchy (Nvidia, 2019).

2.7.2 Algorithmic complexity

Definition 2.39 and Definition 2.40 are defined generally enough that they can be used for both SISD systems and SIMD systems. Kruskal *et al.* (1990) find that it is frequently useful to introduce a second free parameter for the number of processors running in parallel p, giving the problem-independent time/space complexity $T(n, p)/S(n, p)$. The

problem-dependant time/space complexity is derived by finding the function $p(n)$, which frequently has the property

$$\mathcal{O}(p(n)) = \begin{cases} 1 & \text{if} \quad n \leq p_{\max} \\ n & \text{otherwise} \end{cases} \tag{2.26}$$

where $p_{\max}$ is the maximum number of processors that can run in parallel. Speed-up $\mathcal{S}_R$ is defined to be the ratio between the runtime for the serial algorithm and the runtime for the parallel algorithm. The theoretical speed-up $\mathcal{S}_{\mathcal{O}}$ is defined as the ratio between the time complexity of the serial algorithm and the time complexity of the parallel algorithm.

3 State-of-the-art: Parallelized state estimation

This section aims to give an overview of what has been done in this field and to give high-order synthesis of the current research landscape. This process allows the identification of potential research avenues for this work and also allows this work to be contextualized by what has been done.

3.1 Kalman filter

A linear state space model with additive Gaussian state and measurement noise is given by

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ y_k &= Cx_k + Du_k + v_k \end{aligned} \tag{3.1}$$

such that $w_k \sim \mathcal{N}(0, W)$ and $v_k \sim \mathcal{N}(0, V)$, and W and V are the state and measurement covariance respectively. Welch and G Bishop (2001) states that the estimator that minimizes the expected value of the error covariance

$$\begin{aligned} &\mathbb{E}(e_k e_k^T) \\ &e_k = x_k - \hat{x}_k \end{aligned} \tag{3.2}$$

where x_k is the true state and $\hat{x}_k$ is the predicted state, has a closed form solution, known as the Kalman filter (Kalman, 1960). It uses recursive Bayesian prediction and update steps to track the states. It models the estimate as a Gaussian distribution with mean $\mu_k = \hat{x}_k$ and covariance Σ_k. The prediction step moves the state estimate through one discrete time step given no new measurement information:

$$\begin{aligned} \mu_{k+1} &= A\mu_k + Bu_k \\ \Sigma_{k+1} &= A\Sigma_k A^T + W \end{aligned} \tag{3.3}$$

The update step compares the likelihood of current state estimate against new measurement information y_k to correct the estimate:

$$\begin{aligned} K_k &= \Sigma_k C^T (C\Sigma_k C^T + V)^{-1} \\ \mu_{k+1} &= \mu_k + K_k(y_k - C\mu_k) \\ \Sigma_{k+1} &= (T - K_k C)\Sigma_k \end{aligned} \tag{3.4}$$

where K_k is commonly known as the Kalman gain and is intuitively represents how much more the measurement is trusted over the model at a particular time step.

Cisneros-Magana, Medina, and Dinavahi (2013) use a parallel Kalman filter implemented on CUDA and CUDA BLAS to solve a state estimation of harmonics in power generation systems. They test their system against a simulated power system and analysed the accuracy by comparing the discrete Fourier transforms of both the true signal and the estimated one. Their results show that their GPGPU accelerated algorithm speeds up the computational time taken by the state estimation algorithm. They use the GPGPU algorithm developed by Huang *et al.* (2011). The algorithm makes use of the fact that the Kalman filter requires 18 matrix operations to be performed on the prediction and update steps. They found that the matrix inversion and matrix multiplication steps take the most time on CPU architectures and sought of overcome this bottleneck, and use the fact that GPU architectures are designed to perform fast linear algebra calculations to speed up this filter. They report that the computation time of the Kalman filter increases linearly on CPU architectures and increases sub-linearly with GPU acceleration. Zhaofeng Bai and Yuehong Qiu (2015) come to similar conclusions and Xu *et al.* (2016) further optimizes their technique. Xu *et al.* (2016) use GPGPU to speed up Kalman filters for systems with up to five hundred states. They also perform similar analysis using a time domain state estimation filter and find similar results (Cisneros-Magana, Medina, Dinavahi, and Ramos-Paz, 2018).

Several methods exist for reducing the complexity of large-scale dynamic state estimation according to Karimipour and Dinavahi (2015). For example, Jalili-Marandi and Dinavahi (2010), Gomez-Quiles *et al.* (2012), and Xie *et al.* (2012) use dimensionality reduction and/or decoupling methods. However, Karimipour and Dinavahi (2015) finds that these techniques sacrifice accuracy for computational speed and suggest using a GPGPU accelerated extended Kalman filter to combat the issues relating to accuracy and compu-

tational speed. Extended Kalman filters offer improved accuracy over Kalman filters because they make use of a non-linear model of the system which can better describe the dynamics compared to a linear model.

Karimipour and Dinavahi (2015) describe their filtering process as

1. Linearize the nonlinear model

$$\begin{aligned} x_{k+1} &= f(x_k) + w_k \\ y_k &= h(x_k) + v_k \end{aligned} \tag{3.5}$$

   about the initial operating point x_0 to obtain

$$\begin{aligned} x_{k+1} &= F_k x_k + a_k w_k \\ y_k &= H_k x_k + b_k + v_k \end{aligned} \tag{3.6}$$

2. Identify the parameters using Holt's exponential smoothing technique

$$\begin{aligned} F_k &= \alpha(1+\beta)I \\ a_k &= (1+\beta)(1-\alpha)\bar{x}_k - \beta\gamma_{k-1} + (1+\beta)\xi_{k-1} \\ \gamma_k &= \alpha\hat{x}_k + (1-\alpha)\bar{x}_k \\ \xi_k &= \beta(\gamma_k - \gamma_{k-1}) + (1-\beta)\xi_{k-1} \end{aligned} \tag{3.7}$$

   where α and β are smoothing parameters and $\bar{x}_k$ and $\hat{x}_k$ are the predicted and updated state estimates respectively. A similar process is followed for H_k and b_k.

3. Use the regular Kalman filter method to predict and update the state estimates

They use the GPGPU to perform parameter updates and the regular Kalman prediction and updates in parallel, and to perform many of the required linear algebra operations to further speed-up their algorithm. Their results show an up to 15 times speed-up for a power system with 4992 buses. They find that their method has a 99.9 % accuracy for voltage states and a 85 % accuracy for phase angle states. They do not compare this accuracy with a method that uses a complexity reduction technique.

3.2 Weighted least squares

Weighted least squares estimators are popular in state estimation of power systems. They are different from the other estimators that are considered, because they are static

estimators. This means that they attempt to estimate the state of a system at steady state. The relationship between the Kalman filter and the weighted least squares filter is that in Equation 3.1 (Namrata, 2005):

$$\begin{aligned} A &= I \\ B &= 0 \\ D &= 0 \end{aligned} \tag{3.8}$$

this effectively means that the state does not change over time and that inputs are constant or have no effect on the system.

Xia *et al.* (2017) observe that typical SCADA systems have an update time in the order of milliseconds, while typical state estimation systems have update times in the order of a second. They find that improving the state estimation update time improves performance. Numerous algorithmic techniques have been developed to lower the computational load of state estimation on CPU systems as found by Guo *et al.* (2013); Gol and Abur (2015); and Garcia *et al.* (1979).

Gomez-Exposito *et al.* (2011) and Korres (2011) use a method similar to system decoupling mentioned for Kalman filters. They, along with Xiong and Grijalva (2016) and Minot *et al.* (2016), break the model up into relatively independent parts. Zheng *et al.* (2017) and Kekatos and Giannakis (2013) use a similar method and perform distributed state estimation on a multi-core computing architecture. However, the speed is still limited by the state estimation subsystems and these methods do not improve performance on computational level.

Jalili-Marandi and Dinavahi (2010) and Debnath *et al.* (2011) have had success with using GPGPU computational architectures for problems with similar levels of computational load. While, Xia *et al.* (2017) develop a GPGPU accelerated weighted least squares state estimator that decouples some Jacobian calculations such that they can be parallelized. They also parallelize the large matrix multiplication operations for power systems with as many as 147841 nodes. They also parallelize the back-substitution operations after an LU-decomposition by using a dependency DAG to identify independent parallelizable subtasks. They make use of the CUDA architecture and report that GPU acceleration performance increases as problem size increases. This is due to the extra load of copying data to the GPU having less of an effect.

The use of GPGPU on a weighted least squares state estimator with a dishonest Gauss-Newton evaluation is investigated by Rahman and Venayagamoorthy (2016). Dishonest

Gauss-Newton evaluation keeps the Jacobian matrix constant, which significantly speeds up the processing time at the cost of reduced accuracy. They use CUDA BLAS to perform matrix multiplications for a system with 599 states. They test their state estimator on a 68-bus and 118-bus non-linear model of a power system. Their state estimator runs in the order of 200 ms per iteration and observe that the method is 97 % – 99 % accurate this result is comparable to the result stated earlier for the EKF by Karimipour and Dinavahi (2015) who had a similar accuracy for voltage measurements for a 4992-bus system. However, Rahman and Venayagamoorthy (2016) do not give the model of the power system they use and so it is not possible to determine the level of non-linearity in their model. They also do not give the parameters for the noise.

3.3 Particle filter

Particle filters attempt to track the distribution $p(x_k|y_{0:k})$ by using samples of estimated distribution $p(\hat{x}_k|y_{0:k})$. Particle filters can filter nonlinear multi-modal systems with any kind of noise distributions. The system description is given by:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) + w_k \\ y_k &= g(x_k, u_k) + v_k \end{aligned} \tag{3.9}$$

such that w_k and v_k are drawn from the state $p(w_k)$ and measurement $p(v_k)$ noise distributions respectively. The particle filter relies on knowing the observation distribution $p(y_k|x_k)$ and the initial distribution $p(x_0)$.

Tulsyan *et al.* (2016) explain that in order to maintain the same level of accuracy, particle filters require significantly more particles as the number of state dimensions or the distribution complexity increases. However, increasing the number of particles increases the computational load which makes real-time applications difficult.

Particle filters operate by performing prediction, update and resampling steps as follows (Ristic *et al.*, 2003):

1. Generate N samples $q_{0,i}$ with equal weights $w_{0,i} = \frac{1}{N}$ from the initial distribution $p(x_0)$
2. Proceed to *Prediction* (step 3) if no measurements are available or *Update* (step 6) if there are measurements

3. *Prediction* aims to transform the current estimate $p(\hat{x}_k|y_{0:k})$ to the future estimate $p(\hat{x}_{k+1}|y_{0:k})$

4. Update the samples:

$$q_{k+1,i} = f(q_{k,i}, u_k) \tag{3.10}$$

   Go to step 2

5. *Update* aims to transform the prior $p(\hat{x}_k|y_{0:k-1})$ into the posterior $p(\hat{x}_k|y_{0:k})$

6. Given the measurement $\boldsymbol{y}_k$, compute new unnormalized weights

$$v_{k,i} = w_{k,i}p(\boldsymbol{y}_k|q_{k,i}) \tag{3.11}$$

7. Normalize the weights

$$\begin{aligned} v &= \sum_i v_{k,i} \\ w_{k+1,i} &= \frac{v_{k,i}}{v} \end{aligned} \tag{3.12}$$

8. If the particles are degenerating, go to *Resampling* (step 9) otherwise go to step 2.

9. *Resampling* is important to prevent particle weight degeneration, which is when a few particles have high weights and most of the particles have low weights (Nicely and Wells, 2019).

10. Draw N particles $r_{k,i}$ from the current set of particles $q_{k,i}$ according to their weights $w_{k,i}$

11. Set $q_{k,i} = r_{k,i}$ and $w_{k,i} = \frac{1}{N}$

    Go to step 2

Sowman *et al.* (2016) use GPGPU to accelerate a particle filter that tracks the NOx and NH_3 concentrations inside a selective catalytic reduction reactor of a diesel exhaust system. Hsieh and Wang (2010a), Hsieh and Wang (2010b), and H Zhang *et al.* (2015) have implemented an extended Kalman filter to solve the same problem, but they all assume that ammonia sensors are available, however, according to Sowman *et al.* (2016), ammonia sensors are both very expensive and not production ready. Also, the current C_{NOx} sensors are cross-sensitive to ammonia and this cross-sensitivity is affected by catalyst age/degradation and highly sensitive to catalyst temperature. This means that the ammonia effect cannot be measured nor ignored. Thus, Sowman *et al.* (2016) propose

the use of a particle filter. This requires the filter to be run in real-time applications, which is solved by the use of GPGPU.

Distributed particle filtering techniques for multi-core architectures are developed by Chitchian *et al.* (2013). It is important to note that unlike Sowman *et al.* (2016), they are not parallelizing a particle filter, but instead designing a particle filtering algorithm that is able to run on a distributed system. They do this by having each core run a small particle filter called a sub-filter and then investigate ways for the cores to share particles. They do get up to 64 times speed-up for filters with 4 million particles.

A GPGPU accelerated particle filter that uses measurements from a depth sensor is described by Ikoma (2014) to detect the location the hands and arms of a driver in a car. They require the speed-up for a real-time implementation of the detection algorithm for use in the design of safety support systems. They observe that the GPU based-particle filter can achieve a similar frame rate as its CPU counterpart, but with 65 times more particles. This allows their filter's accuracy to improve dramatically. Their implementation runs on CUDA. They use a particle filter because it has a constant time complexity update step and because many operations in a particle filter are performed on each particle independently. This makes parallelization straightforward and very beneficial. They find that after parallelization the resampling step poses the biggest bottleneck, since it cannot be parallelized easily.

Nicely and Wells (2019) explore the parallelization of the resampling step. The need for particle resampling occurs because during each update step, the covariance of the particle weights increases, leading to situations described above where most particles have very low weights. This problem was first solved by Gordon *et al.* (1993). Resampling removes particles with low weights and replaces them with particles with higher weights. This does however create a problem of sample impoverishment (where most particles are in the same states) when the process and measurement noise covariances are low.

Nicely and Wells (2019) look into parallelizing the stratified and systematic resampling approaches. They prefer these over the more easily parallelizable Metropolis algorithm because they are unbiased. It is important to note that these algorithms are not purely random, but do provide a more uniform distribution of random numbers (Hol *et al.*, 2004).

Algorithm 3.1. Stratified resampling

```
def stratified_resample(weights):
        N = len(weights)
        c = numpy.cumsum(weights)
```

```
        indx = []
        k = 0
        for i in range(N):
                u = (i + numpy.random.rand())/N
                while c[k] < u:
                        k += 1
                indx.append(k)
        return indx
```

Algorithm 3.1 shows the stratified sampling algorithm by Nicely and Wells (2019). It runs in linear time ($\mathcal{O}(n)$) due to the `numpy.cumsum` and for-loop both being $\mathcal{O}(n)$. The systematic resampling technique shown in Algorithm 3.2 is very similar, except only one random number is generated.

Algorithm 3.2. Systematic resampling

```
def systematic_resample(weights):
        N = len(weights)
        c = numpy.cumsum(weights)

        indx = []
        k = 0
        r = numpy.random.rand()
        for i in range(N):
                u = (i + r)/N
                while c[k] < u:
                        k += 1
                indx.append(k)
        return indx
```

Their parallel algorithm is work efficient, and they report up to 32 times speed-up using the parallel algorithm compared with the serial implementation, and at least 12 times faster than naive Metropolis resampling. However, it is at least twice as slow as a well-designed coalesced Metropolis filter with the same accuracy.

3.4 Progressive Gaussian filter

The progressive Gaussian filter relies on preforming recursive Bayesian updates to estimate hidden states in a hidden Markov model. The filter approximates the prior and posterior as Gaussian distributions. From the work of Yang, Zhao, *et al.* (2019), this implies that the filter can only maintain a unimodal approximation of the distribution, unlike particle filters and Gaussian sum based filters which can handle multimodal distributions. The filter uses a particle-based homotopy continuation approach for update step.

The description of the operation of a progressive Gaussian filter is built from Hanebeck (2013), Steinbring and Hanebeck (2015) and Yang, Zhao, *et al.* (2019). Given the prior $p(x_k|y_{1:k-1}) \sim \mathcal{N}(\mu_k, \Sigma_k)$, the current measurement $\boldsymbol{y}_k$ and the likelihood $p(y_k|x_k)$, the posterior $p(x_k|y_{1:k})$ can be determined using Bayes' rule as follows:

$$p(x_k|y_{1:k}) \propto p(\boldsymbol{y}_k|x_k)p(x_k|y_{1:k-1}) \tag{3.13}$$

However, doing this with particles samples from the prior can lead to a posterior approximation with low weights. Thus, a homotopy continuation is used. For this, Equation 3.13 is modified to

$$p(x_k|y_{1:k}, \gamma) \propto p(\boldsymbol{y}_k|x_k)^{\gamma}p(x_k|y_{1:k-1}) \tag{3.14}$$

From this, one can derive a recursive formula for $p(x_k|y_{1:k}, \gamma+\Delta\gamma)$ in terms of $p(x_k|y_{1:k}, \gamma)$ to be

$$p(x_k|y_{1:k}, \gamma + \Delta\gamma) \propto p(\boldsymbol{y}_k|x_k)^{\Delta\gamma}p(x_k|y_{1:k}, \gamma) \tag{3.15}$$

It is important to note that when $\gamma = 0$ that the new measurement is not taken into account at all, but when $\gamma = 1$, then we get Equation 3.13. We then proceed with the homotopy continuation by following steps:

1. Set $\gamma = 0$
2. Sample $p(x_k|y_{1:k}, \gamma) \sim \mathcal{N}(\mu_{k,\gamma}, \Sigma_{k,\gamma})$ with N equally weighted samples x_{k,i_γ} to form a Dirac mixture approximation

$$p(x_k|y_{1:k},\gamma) \approx \frac{1}{N}\sum \delta(x_k - x_{k,i_\gamma}) \tag{3.16}$$

3. Update the weights of the new approximation to $\gamma + \Delta\gamma$ using Equation 3.15 by multiplying them by $p(\boldsymbol{y}_k|x_{k,i_\gamma})^{\Delta\gamma}$, where Δ is a tuning parameter.

4. Calculate the mean and sample covariance of the samples

$$\begin{aligned}\mu_{k,\gamma+\Delta\gamma} &= \frac{1}{N}\sum_i p(\boldsymbol{y}_k|x_{k,i_\gamma})^{\Delta\gamma} x_{k,i_\gamma} \\ \Sigma_{k,\gamma+\Delta\gamma} &= \frac{1}{N-1}\sum_i (x_{k,i_\gamma} - \mu_{k,\gamma+\Delta\gamma})(x_{k,i_\gamma} - \mu_{k,\gamma+\Delta\gamma})^T\end{aligned} \tag{3.17}$$

5. Repeat Step 2 with the new Gaussian parameters $(\mu_{k,\gamma+\Delta\gamma}, \Sigma_{k,\gamma+\Delta\gamma})$ and γ until $\gamma = 1$.

This method prevents the problem of particle weight degradation, by applying the update slowly.

Steinbring and Hanebeck (2015) uses GPGPU to accelerate a progressive Gaussian filter with application to object tracking with multiple sensory devices. They detail the need to use GPGPU due to the fact that modern object tracking uses measurements from multiple depth sensing devices. This creates problems because multiple update steps are done at once. For linear estimators, like Kalman filters, this can be parallelized very easily due to the nature of the estimator. However, several nonlinear estimators have the problem of a collapsing covariance matrix when multiple update steps are performed in parallel. They considered particle filters, but believed that the resampling step as well as the need for large amounts of particles to avoid divergence would be too slow. This result is also found by Hendeby *et al.* (2010), who also stated that particle filters require parallel random number generation which is slow. This has since been alleviated by Nvidia's cuRAND, which allows the generation of random numbers in parallel (Nvidia, 2015). However, results by L Zhang *et al.* (2012) still find that for a state space with 22 dimensions that particle filter is too slow for real time image tracking applications. Thus, Steinbring and Hanebeck (2015) opted to use the progressive Gaussian filter described by Hanebeck (2013), which also uses a particle based approach, but is a bit more robust. They found that the GPGPU accelerated progressive Gaussian filter allowed for real time tracking.

Part II

Methods

4 Bioreactor model

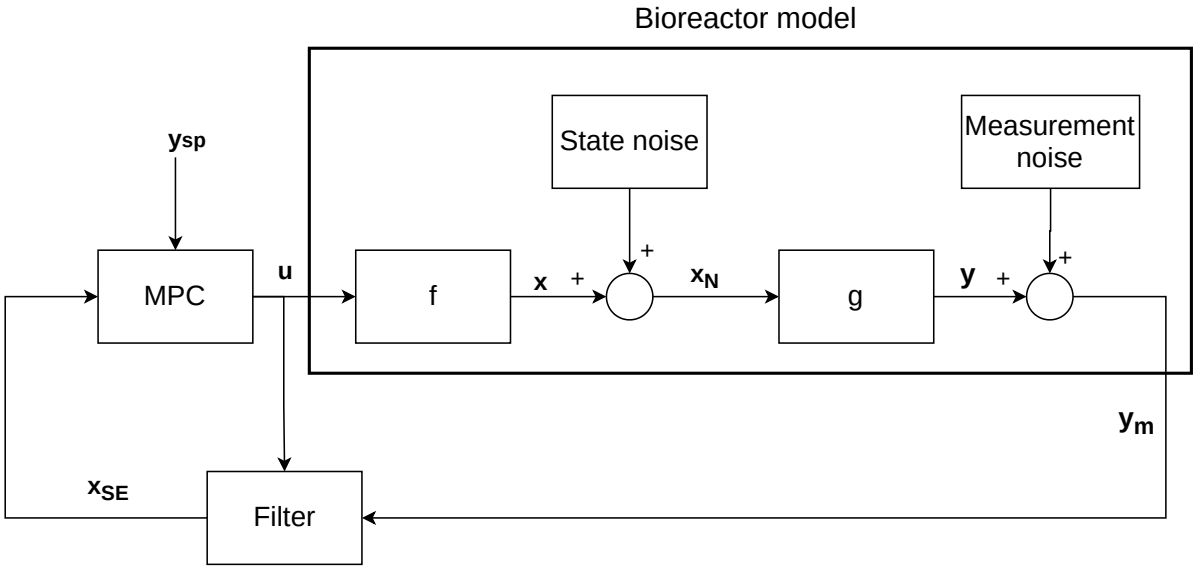

Figure 4.1: Closed loop code simulation: A diagram showing how the various parts of the simulation work together.

This part guides the reader through the method that is followed to develop the parts of the closed loop simulation shown in Figure 4.1. It begins by describing and justifying the bioreactor model that is used in the closed loop simulation, and then the MPC controller is defined. The part concludes by describing the GPGPU accelerated state estimation algorithm.

The bioreactor has two code aspects: the state transition function (f); and the state observation function (g). Noise generation code is used by both the model and the filter. Later sections will detail the code implementations of the state estimators.

Delimitation 4.1. Bioreactor: System models can be quantified according to the system's complexity C and dynamics $\{\tau_i\}$. An all-encompassing investigation would ideally sweep across the entire domains of these parameters. In this text, only a single system is analysed. This makes the results application specific, but allows the investigation to be completed in the limited time.

Delimitation 4.2. Simulated model: While the model is based on real world experimentation, this texts only considers the use of a simulated model. This speeds up the gathering of results and allows for faster testing of the software.

4.1 Model description

The bioreactor is modelled using results from Swart (2019) and the model developed by Iplik (2017). Swart (2019) performed experiments on a lab-scale fermenter that ferments glucose to fumaric acid using immobilized *Rhizopus Oryzae*. The fungus also produces unwanted ethanol if it needs to remove large amounts of glucose. A dynamic model of the system is derived by using first principles modelling, and by selecting kinetics based on the results found by Swart (2019).

The model's kinetics have two different forms to represent the two regimes of operation: a high nitrogen batch environment for the growth of the biomass, and a low nitrogen continuous production phase. For the growth phase, Swart (2019) reports: complete consumption of $3.00\,\mathrm{g\,L^{-1}}$ glucose in $25\,\mathrm{h}$; with a biomass yield of $0.20\,\mathrm{g\,g^{-1}}$; a fumaric acid yield of $0.06\,\mathrm{g\,g^{-1}}$; and an ethanol yield of $0.20\,\mathrm{g\,g^{-1}}$.

Swart (2019) reports that for the production phase: glucose concentration tends to stabilize around $0.280\,\mathrm{g\,L^{-1}}$; fumaric acid production rate remains more or less constant at 0.250 grams of fumaric acid per gram of biomass per hour; maximum ethanol production is 0.025 grams ethanol per gram biomass per hour; the fungus can consume 0.400 grams of glucose per gram of biomass per hour before ethanol overflow occurs; the fungus can consume 0.500 grams of glucose per gram of biomass per hour before glucose overflow occurs.

These results will be used to develop the kinetics for the model, with the change that the times will be sped up by a factor of 60 so that the dynamics are faster. This gives a better system to compare CPU and GPU performance on since the time scale of the control calculations is on the order of seconds.

4.2 Nonlinear model

This section refers to the equations found in Table 4.1. Equation #1 through Equation #5 [1] represent ordinary CSTR concentration balances with the relevant terms set to zero for glucose (G), biomass (X), fumaric acid (FA), ethanol (E), and the homoeostasis enzyme (H), respectively. The symbols C_M and r_M refer to the concentration and rate of formation of compound M, respectively. All concentrations are in $\mathrm{mol\,L^{-1}}$, and all rates are in $\mathrm{mol\,L^{-1}\,min^{-1}}$. $F_{G,in}$, $C_{G,in}$, F_{out} and V refer to the glucose feed rate, glucose feed concentration, flow rate out of the reactor, and volume of the reactor (L), respectively. All feed rates are in $\mathrm{L\,min^{-1}}$. Equation #6 is so defined because the volume of the system is kept constant. F_m is the mineral solution feed rate.

Table 4.1: Model equations.

#	Equations	Inputs	Outputs	Parameters
Mole balances				
1)	$\frac{dC_G}{dt} = (F_{G,in}C_{G,in} - F_{out}C_G + r_G)/V$	$F_{G,in}$	F_{out}, C_G, r_G	$C_{G,in}$, V
2)	$\frac{dC_X}{dt} = r_X/V$		C_X, r_X	
3)	$\frac{dC_{FA}}{dt} = (-F_{out}C_{FA} + r_{FA})/V$		C_{FA}, r_{FA}	
4)	$\frac{dC_E}{dt} = (-F_{out}C_E + r_E)/V$		C_E, r_E	
5)	$\frac{dC_h}{dt} = r_h/V$		C_h, r_h	
6)	$F_{out} = F_m + F_{G,in}$	F_m		

Five bacterial reactions are modelled. These reactions are seen in Equation 4.1 through Equation 4.5. The first two reactions (Equation 4.1 and Equation 4.2) are fermentation reactions and show the incomplete breakdown of glucose to fermentation products. The third reaction (Equation 4.3) condenses the relevant reactants and product of the TCA cycle into a single reaction. The fourth reaction

[1] #n is used to represent equations from Table 4.1

(Equation 4.4) takes place in the mitochondria of the bacteria where cellular respiration occurs. The last reaction (Equation 4.5) shows how the bacteria produces more of itself by combining smaller molecules into larger ones that allow it to undergo mitosis.

$$\text{Glucose} + 2\,CO_2 + 6\ \text{ATP} \rightarrow 2\ \text{FA} + 2\,H_2O \qquad \textit{(Fermenetation)} \tag{4.1}$$

$$\text{Glucose} \rightarrow 2\ \text{Ethanol} + 2\,CO_2 + 2\ \text{ATP} \qquad \textit{(Fermenetation)} \tag{4.2}$$

$$\text{Glucose} \rightarrow 6\,CO_2 + 12\ \text{NADH} + 4\ \text{ATP} \qquad \textit{(TCA cycle)} \tag{4.3}$$

$$\text{NADH} + \frac{1}{2}O_2 \rightarrow \frac{7}{3}\text{ATP} \qquad \textit{(Resperation)} \tag{4.4}$$

$$\text{Glucose} + \gamma\ \text{ATP} \rightarrow 6\ \text{Biomass} + \alpha CO_2 + \beta\ \textit{NADH} \qquad \textit{(Anabolism)} \tag{4.5}$$

Equation #7 through Equation #17 define the rates for these reactions for the batch phase of the reactor. The $\frac{C_G}{k+C_G}$ terms in Equation #7 through Equation #10 are Monod terms for substrate inhibition. Biomass requires energy/ATP to reproduce, and it also requires energy to maintain its current state. This maintenance energy requirement θ is dependent on the concentrations and other factors in the cell. The cell also makes use of NAD^+ and NADH for charge transfer. The cell does not allow the build up of energy or charged molecules which are consumed and produced as seen in Equation 4.1 through Equation 4.5. Equation #11 and Equation #12 are the energy/ATP and redox/NADH balances, respectively.

#	Equations	Inputs	Outputs	Parameters
	Batch rates			
7)	$r_{FA,f} = k_{FA,\max}(\frac{C_G}{k_{FA}+C_G})$		$r_{FA,f}$	$k_{FA,\max}$, k_{FA}
8)	$r_{E,f} = k_{E,\max}(\frac{C_G}{k_E+C_G})$		$r_{E,f}$	$k_{E,\max}$, k_E
9)	$r_{X,f} = k_{X,\max}(\frac{C_G}{k_X+C_G})$		$r_{X,f}$	$k_{X,\max}$, k_X
10)	$r_\theta = k_{\theta,\max}(\frac{C_G}{k_\theta+C_G})$		r_θ	$k_{\theta,\max}$, k_θ
11)	$6r_{FA,f} + 6\gamma r_{X,f} = 4r_t + \frac{7}{3}r_r + 2r_{E,f}$	r_t, r_r		γ
12)	$r_r = 12r_t + 6\beta r_{X,f}$			β
13)	$r_{G,b} = -(r_{FA,f} + r_{E,f} + r_{X,f} + r_t)C_x V$		$r_{G,b}$	
14)	$r_{X,b} = 6r_{X,f}C_x V$		$r_{X,b}$	
15)	$r_{FA,b} = 2r_{FA,f}C_x V$		$r_{FA,b}$	
16)	$r_{E,b} = 2r_{E,f}C_x V$		$r_{E,b}$	
17)	$r_{h,b} = 0$		$r_{h,b}$	

Equation #18 through Equation #24 show the homoeostatic rates in the low nitrogen environment. The biomass production rate in this phase is assumed to be zero as not enough nitrogen is present to allow growth of the biomass. The cell is modelled to attempt to keep the glucose concentration at $0.28\,\mathrm{g\,L^{-1}}$. It operates similar to a PI controller. This idea has been investigated by Mairet (2018) and Veen *et al.* (2019). The concentration of an enzyme h acts as the integral of the error. Equation #19 shows this by defining the rate to be the current error between the current glucose concentration and the desired homoeostatic concentration C_{hs}. Using this, the model has the ability to determine the desired glucose consumption rate to bring the cell into homoeostasis.

The required glucose consumption rate is given by Equation #20. Depending on how large this rate is, the cell uses various mechanisms to consume glucose. The first mechanism is through fumaric acid production. Equation #21 shows the fumaric acid production rate, which is constant at high enough glucose concentrations. The second mechanism is a cellular maintenance mechanism $r_\theta = r_{theta,r} + r_{theta,q}$(the

cell changes how efficient it is at consuming glucose thus changing this rate). Equation #22 shows this mechanism. These two mechanisms are only able to consume 0.4 grams of glucose per gram of biomass per hour, as found by Swart (2019).

Once this limit is reached the cell begins producing ethanol to dissipate the glucose (Equation #23). The rate at which the cell can produce ethanol is limited as well, and once this limit $r_{E,\max}$, the cell further adapts its maintenance dissipation rate to consume glucose as seen in Equation #24. This is limited as well as found by Swart (2019), and all these mechanisms can only consume 0.5 grams of glucose per gram of biomass per hour. Once this limit is reached glucose overflow begins, which means that r_G, h in Equation #25 is smaller than the rate of glucose addition and the cell is no longer able to maintain homoeostasis.

Equation #26 through Equation #30 serve to select the regime (growth or production) and is based on a fictitious input B to the model.

#	Equations	Inputs	Outputs	Parameters
	Homoeostatic rates			
18)	$r_{X,h} = 0$		$r_{X,h}$	
19)	$r_{h,h} = C_{hs} - C_G$		$r_{h,h}$	C_{hs}
20)	$r_{G,r} = \theta_{r,\max} C_x V - (K_p r_{h,h} + K_I C_h)$		$r_{G,r}$	$\theta_{r,\max}$, K_p, K_I
21)	$r_{FA,h} = r_{FA,h,\max} C_x V (\frac{C_G}{k_{FA,h}+C_G})$		$r_{FA,h}$	$r_{FA,h,\max}$, $k_{FA,h}$
22)	$r_{\theta,r} = \text{bounded}\,[r_{G,r};\ 0;\ \theta_{r,\max} C_x V]$		$r_{\theta,r}$	
23)	$r_{E,h} = \text{bounded}\,[r_{\theta,r} - \theta_{r,\max} C_x V;\ 0;\ r_{E,h,\max} C_x V]$		$r_{E,h}$	$r_{E,h,\max}$
24)	$r_{\theta,q} = \text{bounded}\,[r_{\theta,r} - \theta_{r,\max} C_x V - r_{E,h};\ 0;\ \theta_{q,\max} C_x V]$		$r_{\theta,q}$	$\theta_{q,\max}$
25)	$r_{G,h} = -(r_{FA,h} \frac{MM_{FA}}{MM_G} + r_{E,h} \frac{MM_E}{MM_G} + r_{\theta,r} + r_{\theta,q})$		$r_{G,h}$	MM_{FA}, MM_G, MM_E
	where $\text{bounded}\,[e;\ l;\ u]$ takes the value of e provided that $l < e < u$, otherwise, it takes the value of l if $e < l$ or u if $e > u$			
	Rate switching			
26)	$r_G = \begin{cases} r_{G,b} & \text{if } B \\ r_{G,h} & \text{otherwise} \end{cases}$	B		
27)	$r_X = \begin{cases} r_{X,b} & \text{if } B \\ r_{X,h} & \text{otherwise} \end{cases}$			
28)	$r_{FA} = \begin{cases} r_{FA,b} & \text{if } B \\ r_{FA,h} & \text{otherwise} \end{cases}$			
29)	$r_E = \begin{cases} r_{E,b} & \text{if } B \\ r_{E,h} & \text{otherwise} \end{cases}$			
30)	$r_h = \begin{cases} r_{h,b} & \text{if } B \\ r_{h,h} & \text{otherwise} \end{cases}$			

Table 4.2: Values for model parameters.

Parameter			Parameter		
$C_{g,in}$	$\frac{5}{180}$	$\mathrm{mol\,L^{-1}}$	V	1	L
$k_{FA,\max}$	$\frac{1}{230}$	$\mathrm{mol\,L^{-1}\,min^{-1}}$	k_{FA}	1×10^{-3}	$\mathrm{mol\,L^{-1}}$
$k_{E,\max}$	$\frac{1}{12}$	$\mathrm{mol\,L^{-1}\,min^{-1}}$	k_E	1×10^{-3}	$\mathrm{mol\,L^{-1}}$
$k_{X,\max}$	$\frac{1}{21}$	$\mathrm{mol\,L^{-1}\,min^{-1}}$	k_X	1×10^{-3}	$\mathrm{mol\,L^{-1}}$
$k_{\theta,\max}$	1.1	$\mathrm{mol\,L^{-1}\,min^{-1}}$	k_θ	1×10^{-3}	$\mathrm{mol\,L^{-1}}$
γ	1.8		β	0.1	
C_{hs}	$\frac{0.28}{180}$	$\mathrm{mol\,L^{-1}}$	$\theta_{r,\max}$	0.0205	$\mathrm{mol\,L^{-1}\,min^{-1}}$
K_p	$\frac{369}{56}$		K_I	1×10^{-2}	$\mathrm{min^{-1}}$
$r_{FA,h,\max}$	$\frac{123}{2320}$	$\mathrm{mol\,L^{-1}\,min^{-1}}$	$k_{FA,h}$	1×10^{-5}	$\mathrm{mol\,L^{-1}}$
$r_{E,h,\max}$	$\frac{123}{9200}$	$\mathrm{mol\,L^{-1}\,min^{-1}}$	$\theta_{q,\max}$	0.010 25	$\mathrm{mol\,L^{-1}\,min^{-1}}$
MM_{FA}	116	$\mathrm{g\,mol^{-1}}$	MM_G	180	$\mathrm{g\,mol^{-1}}$
MM_E	46	$\mathrm{g\,mol^{-1}}$			

4.3 Noise

In order to take into account modelling inaccuracies and instrumentation noise, state and measurement noise are added to the differentials and outputs, respectively.

Delimitation 4.3. Gaussian sum noise: Many distributions can be chosen to represent the noise distributions. This text uses the Gaussian sum distribution, because any distribution can be approximated with arbitrary accuracy by using a Gaussian sum distribution.

The Gaussian sum noise distributions take the form of ϵ-mixture distributions described by Plataniotis *et al.* (1997):

$$\begin{aligned} w_k, v_k &\sim \mathcal{N}_\Sigma(x|\mu_\epsilon, \mathbf{\Sigma}_\epsilon) \\ &= (1-\epsilon)\mathcal{N}(x|\mu_1, \Sigma_1) + \epsilon\mathcal{N}(x|\mu_2, \Sigma_2) \\ \lambda &= \frac{\bar{\sigma}(\Sigma_2)}{\bar{\sigma}(\Sigma_1)} \end{aligned} \tag{4.6}$$

where $\epsilon \in [0, 0.25]$ and $\lambda \in [10, 10000]$. The state noise for the model given by

$$
\begin{aligned}
\mu_1 &= \mu_2 = \mathbf{0} \\
\Sigma_1 &= \operatorname{diag}\left(\begin{bmatrix} 1.0 \cdot 10^{-4} & 1.0 \cdot 10^{-7} & 1.0 \cdot 10^{-3} & 1.0 \cdot 10^{-3} & 1.0 \cdot 10^{-7} \end{bmatrix}\right) \\
\Sigma_2 &= \lambda \Sigma_1 \\
\epsilon &= 0.25 \\
\lambda &= 100
\end{aligned}
\tag{4.7}
$$

Delimitation 4.4. Noise complexity: Given a measure of complexity for random variables E, for example Kullback—Leibler divergence or differential entropy, one could investigate how distributions with different values for E affects the results. Varying noise complexity serves to demonstrate performance differences of filters with varying numbers of particles. Due to this not being a focus of this investigation, this text only deals with a single noise distribution instance.

The measurement noise for the model is given by

$$
\begin{aligned}
\mu_1 &= \begin{bmatrix} 0.0001 & 0.0 \end{bmatrix} \\
\mu_2 &= \begin{bmatrix} 0.0 & -0.0001 \end{bmatrix} \\
\Sigma_1 &= \operatorname{diag}\left(\begin{bmatrix} 6.0 \cdot 10^{-2} & 8.0 \cdot 10^{-2} \end{bmatrix}\right) \\
\Sigma_2 &= \operatorname{diag}\left(\begin{bmatrix} 500 & 700 \end{bmatrix}\right) \\
\epsilon &= 0.15
\end{aligned}
\tag{4.8}
$$

5 Model predictive control

The MPC will be modelled after a deterministic reformulation of a stochastic MPC developed by Wilken (2015). Given the chance constrained formulation

$$\begin{aligned} \min_{\mathbf{u}} \quad & \mathbb{E}\left[\frac{1}{2}\sum_{k=1}^{P}\left(x_k^T Q x_k + u_k^T R u_k\right)\right] \\ & x_{k+1} = Ax_k + Bu_k \\ & \mathbb{E}\left[Dx_k + e\right] \geq 0 \\ & \mathcal{P}\left[Dx_k + e \geq 0\right] \geq p_l \end{aligned} \tag{5.1}$$

where $\mathcal{P}\left[condition\right]$ is the probability that *condition* evaluates as true. Using the principle of separation, Gaussian identities and the Mahalanobis distance (which measures how many standard deviations a point P is from a distribution D), Wilken (2015) shows that the stochastic MPC can be reformulated into a state estimator and a deterministic MPC:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \frac{1}{2}\sum_{k=1}^{P}\left(\mu_k^T Q \mu_k + u_k^T R u_k\right) \\ & \mu_{k+1} = A\mu_k + Bu_k \\ & \Sigma_{k+1} = A\Sigma_k A^T + W \\ & D\mu_k + e \geq c\sqrt{\operatorname{diag}\left(D\Sigma_k D^T\right)} \end{aligned} \tag{5.2}$$

where μ_0 and Σ_0 are the mean and covariance from the state estimator; W is the covariance of the state noise; $\operatorname{diag}(M)$ pulls a $n \times 1$ vector m from the $n \times n$ matrix M, that consists of the main diagonal elements of M; the square root is performed element-wise; and c is a constant such that $\mathcal{P}\left[\chi^2_{N_o} \leq c^2\right] = p_l$ where $\chi^2_{N_x}$ is the chi squared distribution with N_o degrees of freedom and N_o is the number of outputs.

5.1 Linearisation

Equation 5.1 shows that a linear model of the system is needed. To this end, nonlinear models can be linearized around an operating point $W_{op} = (X_{op}, U_{op})$ made up of a state and an input vector around which the linearisation should take place. A system of the form

$$\begin{aligned} \dot{x} &= f(x,u) \\ y &= g(x,u) \end{aligned} \tag{5.3}$$

can be linearized using a first order Taylor series approximation:

$$\begin{aligned} \dot{x} &= \mathbf{J}_f(x)\Big|_{W_{op}}(x - X_{op}) + \mathbf{J}_f(u)\Big|_{W_{op}}(u - U_{op}) + f(X_{op}, U_{op}) \\ y &= \mathbf{J}_g(x)\Big|_{W_{op}}(x - X_{op}) + \mathbf{J}_g(u)\Big|_{W_{op}}(u - U_{op}) + g(X_{op}, U_{op}) \end{aligned} \tag{5.4}$$

where

$$\mathbf{J}_h(w) = \begin{bmatrix} \frac{\partial h_1}{\partial w_1}\Big|_{W_{op}} & \cdots & \frac{\partial h_1}{\partial w_m}\Big|_{W_{op}} \\ \vdots & & \vdots \\ \frac{\partial h_n}{\partial w_1}\Big|_{W_{op}} & \cdots & \frac{\partial h_n}{\partial w_m}\Big|_{W_{op}} \end{bmatrix} \tag{5.5}$$

In order to obtain a state space description, we choose W_{op} to be at a steady state of the system W_{ss} such that

$$f(X_{op}, U_{op}) = \dot{x}\Big|_{W_{ss}} = 0 \tag{5.6}$$

After noting that $Y_{op} = g(X_{op}, U_{op})$, the above can be rewritten in terms of deviation variables

$$
\begin{aligned}
\dot{x'} &= A_c x' + B_c u' \\
y' &= C_c x' + D_c u' \\
x' &= x - X_{op} \\
u' &= u - U_{op} \\
y' &= y - Y_{op}
\end{aligned}
\tag{5.7}
$$

However, this is a continuous system and a discrete linear approximation is required. To complete the process, a zero order hold is applied using `scipy.signal.cont2discrete`, this gives the state space equations in terms of deviation variables as:

$$
\begin{aligned}
x_{k+1} &= Ax_k + Bu_k \\
y_k &= Cx_k + Du_k
\end{aligned}
\tag{5.8}
$$

5.2 Changes to the Wilken (2015) implementation

This section documents improvements to the reformulated MPC implementation by Wilken (2015). MPC formulations by Rawlings and Mayne (2009) and Seborg and Mellichamp (2006) are used to guide the design.

5.2.1 Outputs

The implementation by Wilken (2015) forces the user to control all the states, which — given that there are often more states than inputs — makes the control problem ill-posed. Introducing the output equation for the state space model allows the outputs to be tracked (Rawlings and Mayne, 2009). This requires the following equality constraints to be implemented

$$
\begin{aligned}
y_k &= C\mu_k + Du_k \\
e_k &= r - y_k
\end{aligned}
\tag{5.9}
$$

where r is the set point.

5.2.2 Prediction and control horizons

It is often useful to have a separate control horizon M and prediction horizon P as it gives better control over the aggressiveness of the controller: the closer M is to P, the more aggressive the controller is (Seborg and Mellichamp, 2006).

5.2.3 Expectation and chance constraints

For the purpose of this investigation, the expectation and chance constraints will be removed to make the implementation simpler. They offer no value to answering any of the research questions of this investigation, and they slow down the simulations due to the calculation of Σ_k.

5.2.4 Disturbances

Disturbances cause steady state offset. These disturbances can be uncontrolled inputs or model error. Similarly, to the method followed by Seborg and Mellichamp (2006), a bias term is added to the outputs to remove steady state offset

$$\begin{aligned} y_k &= C\mu_k + Du_k + b \\ b &= y_0 - y_{\text{predicted}} \end{aligned} \tag{5.10}$$

where y_0 is the latest measurement $y_{\text{predicted}}$ is the output that the controller predicted for this time step in the previous control calculation.

5.2.5 Limiting constraints

It is often required to restrict the movement of the inputs or to constrain the desired output. For this reason, the following inequality constraints are added to the formulation

$$\begin{aligned} y_{\min} \leq & y_k \leq y_{\max} \\ \Delta u_{\min} \leq & \Delta u_k \leq \Delta u_{\max} \\ u_{\min} \leq & u_k \leq u_{\max} \end{aligned} \tag{5.11}$$

5.2.6 Inputs

For this implementation, the state space description is rewritten so that the optimizer has $\Delta u_k = u_k - u_{k-1}$ as variables instead of the inputs u_k themselves. This allows the optimizer to better solve for steady state solutions where $\Delta u_k = 0$. The new state space equations are derived as follows:

$$\begin{aligned}
\Delta x_{k+1} &= x_{k+1} - x_k \\
&= A(x_k - x_{k-1}) + B(u_k - u_{k-1}) \\
&= A\Delta x_k + B\Delta u_k
\end{aligned} \tag{5.12}$$

$$\begin{aligned}
\Delta y_k &= y_k - y_{k-1} \\
&= C\Delta x_k + D\Delta u_k \\
\implies y_k &= y_{k-1} + C\Delta x_k + D\Delta u_k
\end{aligned}$$

Thus, the implemented statement of the optimization problem is

$$\begin{aligned}
\min_{\mathbf{\Delta u}} \quad & \frac{1}{2}\sum_{k=1}^{P}\left(e_k^T Q e_k\right) + \frac{1}{2}\sum_{k=0}^{M-1}\left(\Delta u_k^T R \Delta u_k\right) \\
\Delta\mu_1 &= Ax_0 + B(u_{-1} + \Delta u_0) - x_0 \\
\Delta\mu_{k+1} &= A\Delta\mu_k + B\Delta u_k \\
y_0 &= C\mu_0 + D(u_{-1} + \Delta u_0) + b \\
y_k &= y_{k-1} + C\Delta x_k + D\Delta u_k + b \\
e_k &= r - y_k \\
y_{\min} \leq & y_k \leq y_{\max} \\
\Delta u_{\min} \leq & \Delta u_k \leq \Delta u_{\max} \\
u_{\min} \leq & u_k \leq u_{\max}
\end{aligned} \tag{5.13}$$

5.3 Standard QP formulation

The OSQP solver (Stellato *et al.*, 2017) is used for the MPC implementation. It requires the problem to be in the standard form shown in Equation 2.24. To this end, Equation 5.13 is reformulated into the standard form. The optimization variables $\mathbf{x}$ are chosen to be

$$\begin{bmatrix} \mu_0 & \Delta\mu_1 & \cdots & \Delta\mu_P & y_1 & \cdots & y_P & u_{-1} & \Delta u_0 & \cdots & \Delta u_{M-1} \end{bmatrix} \tag{5.14}$$

where u_{-1} is most recent control input.

5.3.1 Objective function

Looking at the term $\frac{1}{2}e_k^T Q e_k$ the following can be shown [1]

$$\begin{aligned} \frac{1}{2}e_k^T Q e_k &= \frac{1}{2}(r - y_k)^T Q (r - y_k) \\ &= \frac{1}{2}(r^T Q r - r^T Q y_k - y_k^T Q r + y_k^T Q y_k) \\ &= \frac{1}{2}(r^T Q r - 2 r^T Q y_k + y_k^T Q y_k) \end{aligned} \tag{5.15}$$

The $r^T Q r$ term can be dropped because it is a constant. The remaining two terms can be placed easily into the H and q^T matrices:

$$\begin{aligned} H &= \text{diag}\left(\begin{bmatrix} 0_{(P+1)N_x \times (P+1)N_x} & Q_y & 0_{N_i \times N_i} & R_{\Delta u} \end{bmatrix}\right) \\ Q_y &= I_P \otimes Q \\ R_{\Delta u} &= I_{M+1} \otimes R \end{aligned} \tag{5.16}$$

where diag(*list*) forms a matrix consisting of the blocks in list along the diagonal; $0_{m \times n}$ is a $m \times n$ matrix of zeros; N_x is the number of states; N_i is the number of inputs; I_n is the $n \times n$ identity matrix; and $\otimes$ is the Kronecker product. The q^T matrix is given by

$$q^T = \begin{bmatrix} 0_{1 \times (P+1)N_x} & 1_{1 \times P} \otimes (-r^T Q) & 0_{1 \times (M+2)N_i} \end{bmatrix} \tag{5.17}$$

where $1_{m \times n}$ is a $m \times n$ matrix of ones; and N_o is the number of outputs.

5.3.2 Equality constraints

The constraints matrices A, l, and u are constructed in pieces to simplify the reasoning.

[1] $r^T Q y_k$ is a scalar and thus $(r^T Q y_k)^T = r^T Q y_k \implies r^T Q y_k = y_k^T Q r$ because Q is symmetric by definition

Initial input value

The variable u_{-1} must be fixed to the value given to the optimizer [2]. This is done using

$$\begin{aligned} A_{\text{init}} &= \begin{bmatrix} 0_{N_i \times ((P+1)N_x + PN_o)} & I_{N_i} & 0_{N_i \times (M+1)N_i} \end{bmatrix} \\ l_{\text{init}}^T = u_{\text{init}}^T &= \begin{bmatrix} u_{-1} \end{bmatrix} \end{aligned} \tag{5.18}$$

State equations

The equality constraints set out in

$$\begin{aligned} \mu_0 &= \mu_{\text{init}} \\ \Delta\mu_1 &= Ax_0 + B(u_{-1} + \Delta u_0) - x_0 \\ \Delta\mu_{k+1} &= A\Delta\mu_k + B\Delta u_k \end{aligned} \tag{5.19}$$

are transformed into the relevant matrices

$$\begin{aligned} A_{\text{state}} &= \begin{bmatrix} A_{\text{state},x} & 0_{(P+1)N_x \times PN_o} & A_{\text{state},u} \end{bmatrix} \\ A_{\text{state},x} &= \left[\begin{array}{c|c} -I_{N_x} & 0_{N_x \times PN_x} \\ \hline A - I_{N_x} & \\ \hline 0_{(P-1)N_x \times N_x} & (I_P^{(-1)} \otimes A) - I_{PN_x} \end{array}\right] \\ A_{\text{state},u} &= \begin{bmatrix} 0_{N_x \times (M+2)N_i} \\ B \otimes \begin{bmatrix} 0_{M \times 1}\big|_{(1,0,0)} & I_M & 0_{M \times 1} \end{bmatrix} \\ 0_{(P-M)N_x \times (M+2)N_i} \end{bmatrix} \end{aligned} \tag{5.20}$$

where $I_n^{(-k)}$ is an $n \times n$ identity matrix with the diagonal of ones moved down k diagonals, and $0_{x \times y}\big|_{(v,r,c)}$ is a $x \times y$ matrix with the value v in row r and column c and zeros everywhere else. The bounds matrices are given by

$$l_{\text{state}}^T = u_{\text{state}}^T = \begin{bmatrix} -\mu_{\text{init}} & 0_{1 \times PN_x} \end{bmatrix} \tag{5.21}$$

[2]The OSQP library allows the user to update the matrices using the `update` function

Output equations

The output equalities

$$\begin{aligned} y_0 &= C\mu_0 + D(u_{-1} + \Delta u_0) + b \\ y_k &= y_{k-1} + C\Delta x_k + D\Delta u_k + b \end{aligned} \tag{5.22}$$

are transformed into the matrices

$$\begin{aligned} A_{\text{output}} &= \begin{bmatrix} A_{\text{output},x} & A_{\text{output},y} & A_{\text{output},u} \end{bmatrix} \\ A_{\text{output},x} &= C \otimes \begin{bmatrix} 0_{P\times 1}\big|_{(1,0,0)} & I_P \end{bmatrix} \\ A_{\text{output},y} &= -I_{PN_o} + I_{PN_o}^{(-N_o)} \\ A_{\text{output},u} &= \begin{bmatrix} D \otimes \begin{bmatrix} 0_{M\times 2}\big|_{(1,0,0),(1,0,1)} & I_M \end{bmatrix} \\ 0_{(P-M)N_o\times(M+2)N_i} \end{bmatrix} \\ l_{\text{output}}^T = u_{\text{output}}^T &= 1_{1\times P} \otimes -b \end{aligned} \tag{5.23}$$

5.3.3 Inequality constraints

The inequality constraints

$$\begin{aligned} y_{\min} \leq & y_k \leq y_{\max} \\ \Delta u_{\min} \leq & \Delta u_k \leq \Delta u_{\max} \\ u_{\min} \leq & u_k \leq u_{\max} \end{aligned} \tag{5.24}$$

also need to be cast into the required format. A similar strategy of splitting the problem into parts is used.

Output

$$\begin{aligned} A_{\text{output ineq}} &= \begin{bmatrix} 0_{PN_o\times(P+1)N_x} & I_{PN_o} & 0_{PN_o\times(M+2)N_i} \end{bmatrix} \\ l_{\text{output ineq}}^T &= 1_{1\times P} \otimes y_{\min} \\ u_{\text{output ineq}}^T &= 1_{1\times P} \otimes y_{\max} \end{aligned} \tag{5.25}$$

5.3.4 Input steps

$$\begin{aligned} A_{\text{input steps}} &= \begin{bmatrix} 0_{(M+1)N_i \times ((P+1)N_x + PN_o + N_i)} & I_{(M+1)N_i} \end{bmatrix} \\ l^T_{\text{input steps}} &= 1_{1\times(M+1)} \otimes \Delta u_{\min} \\ u^T_{\text{input steps}} &= 1_{1\times(M+1)} \otimes \Delta u_{\max} \end{aligned} \tag{5.26}$$

5.3.5 Inputs

In order to constrain the total inputs, the input steps must be summed, this is done by using the matrix

$$\begin{aligned} A_{\text{input ineq}} &= \begin{bmatrix} 0_{MN_i \times ((P+1)N_x + PN_o)} & 1_{M\times 1} \otimes I_{N_i} & T^{(1,L)}_M \otimes I_{N_i} \end{bmatrix} \\ l^T_{\text{input ineq}} &= 1_{1\times M} \otimes u_{\min} \\ u^T_{\text{input ineq}} &= 1_{1\times M} \otimes u_{\max} \end{aligned} \tag{5.27}$$

where $T^{(m,L)}_n$ is a $n \times n$ lower triangular matrix filled with the number m.

5.3.6 All constraints

Putting all the previous matrices together gives

$$\begin{aligned} A &= \begin{bmatrix} A_{\text{init}} & A_{\text{state}} & A_{\text{output}} & A_{\text{output ineq}} & A_{\text{input steps}} & A_{\text{input ineq}} \end{bmatrix}^T \\ l &= \begin{bmatrix} l_{\text{init}} & l_{\text{state}} & l_{\text{output}} & l_{\text{output ineq}} & l_{\text{input steps}} & l_{\text{input ineq}} \end{bmatrix}^T \\ u &= \begin{bmatrix} u_{\text{init}} & u_{\text{state}} & u_{\text{output}} & u_{\text{output ineq}} & u_{\text{input steps}} & u_{\text{input ineq}} \end{bmatrix}^T \end{aligned} \tag{5.28}$$

5.4 Performance

Performance for the operation is determined using the sum of the ISE (Integral Squared Error) performance measures for the controller variables. The IAE (Integral Absolute Error) could also be used to measure performance without changing the comparative results between different runs drastically. The calculation is given by

$$P_{\text{ISE}} = \sum_{i=0}^{N_o} \int_0^{t_{\text{end}}} (y_i(t) - r_i(t))^2 dt \tag{5.29}$$

This performance is related to energy efficiency by considering that improved performance implies more product that is on specification. This in turn means that less down stream processing is required to correct the off specification outputs and the energy used in delivering the product is used efficiently.

6 Filters

This chapter details GPGPU accelerated nonlinear filters.

6.1 Particle filter

Assume that the system can be represented by a dynamic Bayesian network of the form seen in Figure 2.5 along with the system description given in Equation 3.9. Where the state transition function $f(x_k, u_k)$ is similar to an Euler integration step from the bioreactor model set up in Section 4.2. The state observation function $g(x_k, u_k)$ is taken almost directly from the model set up in Section 4.2. Assume also that

$$\begin{aligned} x_0 &\sim \mathcal{N}_\Sigma(x|\mu_\mathbf{0}, \mathbf{\Sigma_0}) \\ &= \sum_{i=0}^{N_d} w_i \, \mathcal{N}(x|\mu_i, \Sigma_i) \\ w_k &\sim \mathcal{N}_\Sigma(x|\mu_\mathbf{w}, \mathbf{\Sigma_w}) \\ v_k &\sim \mathcal{N}_\Sigma(x|\mu_\mathbf{v}, \mathbf{\Sigma_v}) \end{aligned} \tag{6.1}$$

where $\mathcal{N}_\Sigma(x|\mu, \mathbf{\Sigma})$ is a Gaussian sum distribution. The choice of Gaussian mixtures comes from the computational ease from which they can be sampled. Drawing from the Gaussian sum is done by drawing a sample from the weighted distribution defined by the weights $\{w_i|i = 1, ..., N_d\}$ using `numpy.random.choice` or `cupy.random.choice` for the serial and parallel implementations, respectively. This selects which Gaussian should be drawn from. The Python functions `numpy.random.multivariate_normal` or `cupy.random.multivariate_normal` are used to draw samples from the Gaussian for the serial and parallel implementations, respectively. This method is used to draw samples for x_0, w_k, and v_k.

This above assumption is not limiting because given any arbitrary random variable X :

$\Omega \to \mathbb{R}^n$ with probability distribution $\mu_X : \mathbb{R}^N \to [0,1]$ can be represented with accuracy $c \in [0,1)$ by $\mathcal{N}_\Sigma(x|\mu, \mathbf{\Sigma})$. This can be done by solving the regression

$$\begin{aligned} \arg\min_{\mu,\mathbf{\Sigma}} \quad & \int_{\mathbb{R}^n} \|f(x,\mu,\mathbf{\Sigma}) - f_X(x)\| dx \\ f(x,\mu,\mathbf{\Sigma}) &= \sum_{i=0}^{m} w_i \; g(x,\mu_i,\Sigma_i) \\ g(x,\mu_i,\Sigma_i) &= \frac{\exp\left(-\frac{1}{2}(x-\mu_i)^T\Sigma_i(x-\mu_i)\right)}{\sqrt{(2\pi)^d \det(\Sigma_i)}} \\ f_X(x) &= \frac{d}{dx}\mu_X((-\infty, x)) \end{aligned} \tag{6.2}$$

where $\|\cdot\|$ is some suitable norm and as the accuracy c is controlled by m and $c \to 1$ as $m \to \infty$ as shown by Barcharoglou (2010).

Delimitation 6.1. Known noise: The noise model is assumed to be completely known. That is, for the purposes of the simulation, the bioreactor model draws noise measurements from the same distributions as the state estimator methods.

The particle filter approximates prior and posterior distributions with Dirac mixture approximations. These approximations consist of N_p particles $(q_{k|j,i}, w_{k|j,i})$ which are made up of a point estimate $q_{k|j,i}$ and a weight $w_{k|j,i}$ for each particle i at the k'th time step given the measurement at the j'th time step.

$$\delta(x|\mathbf{q_{k|j}}, \mathbf{w_{k|j}}) = \sum_{i=1}^{N_p} w_{k|j,i}\delta(x - q_{k|j,i}) \tag{6.3}$$

where $\delta(x)$ is the Dirac delta function at x. Initially, samples $q_{0|0,i}$ are drawn from x_0 are all assigned equal weights $w_{0|0,i} = \frac{1}{N_p}$. These form a Dirac mixture approximation

$$\begin{aligned} x_0 &\sim \mathcal{N}_\Sigma(x|\mu, \mathbf{\Sigma}) \\ &\approx \delta(x|\mathbf{q_0}, \mathbf{w_0}) \\ &= \sum_{i=1}^{N_p} w_{0|0,i}\delta(x - q_{0|0,i}) \end{aligned} \tag{6.4}$$

From the Law of large numbers, we know that $\delta(x|\mathbf{q}, \mathbf{w}) \to x_0$ as $N_p \to \infty$. Thus, the particles approximate x_0.

6.1.1 Code diagram

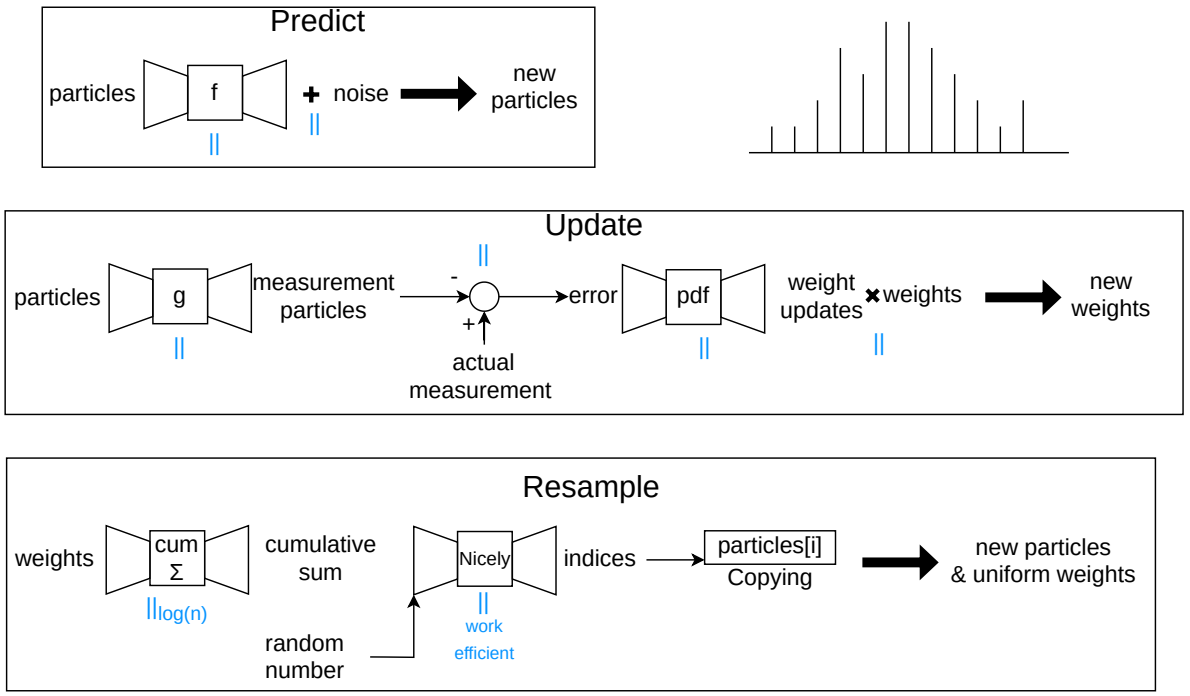

Figure 6.1: Particle filter code diagram. The top right show an example particle distribution diagram.

Figure 6.1 shows the algorithm for the particle filter. Once again, the blue ‖ symbols indicate areas where GPGPU parallelization is implemented.

Delimitation 6.2. Floating point precision: All computations (for the CPU and GPGPU implementations) use single precision for particles. This is chosen because single precision offers better performance over double precision and because based on the values required for the model (see Table 4.2) double precision is not required.

However, all weights are calculated using double precision to prevent a loss of accuracy if the weights become small.

6.1.2 Prediction

Prediction involves finding the prior distribution $p(x_{k+1}|y_{0:k})$ (and in the case of the particle filter the approximation $\delta(x|\mathbf{q_{k+1|k}}, \mathbf{w_{k+1|k}})$), where, by convention, $p(x_0|y_0) = p(x_0)$. The a priori is found as

$$
\begin{aligned}
p(x_{k+1}|y_{0:k}) &= \int_{x_k} p(x_{k+1}, x_k|y_{0:k})dx_k \\
&= \int_{x_k} p(x_{k+1}|x_k, y_{0:k})p(x_k|y_{0:k})dx_k \\
&= \int_{x_k} p(x_{k+1}|x_k)p(x_k|y_{0:k})dx_k
\end{aligned} \tag{6.5}
$$

In particle filters, Equation 6.3 gives an approximation for the posterior $p(x_k|y_{0:k})$. Substituting this in gives

$$
\begin{aligned}
p(x_{k+1}|y_{0:k}) &= \int_{x_k} p(x_{k+1}|x_k) \sum_{i=0}^{N_p} w_{k|k,i}\delta(x_k - q_{k|k,i})dx_k \\
&= \int_{x_k} \sum_{i=1}^{N_p} p(x_{k+1}|x_k) w_{k|k,i}\delta(x_k - q_{k|k,i})dx_k
\end{aligned} \tag{6.6}
$$

Ideally, the new particles would be drawn from the distribution shown in Equation 6.6, however, this is too computationally expensive. Rather, one sample $q_{k+1|k,i}$ is drawn from each of the N_p distributions

$$
c_{k|k,i} \int_{x_k} p(x_{k+1}|x_k) w_{k|k,i}\delta(x_k - q_{k|k,i})dx_k \tag{6.7}
$$

for $i \in 1, ..., N_p$, where $c_{k|k,i}$ is a normalization constant needed to ensure that

$$
\int_{x_k} p(x_{k+1}|x_k) w_{k|k,i}\delta(x_k - q_{k|k,i})dx_k = 1 \tag{6.8}
$$

Closer examination shows that $c_{k|k,i} = \frac{1}{w_{k|k,i}}$. Also, considering the nature of $\delta(x_k - q_{k|k,i})$, this means that a particle is drawn from

$$
\begin{aligned}
\int_{x_k} p(x_{k+1}|x_k)\delta(x_k - q_{k|k,i})dx_k &= \int_{x_k} p(x_{k+1}|q_{k|k,i})dx_k \\
&= p(x_{k+1}|q_{k|k,i})
\end{aligned} \tag{6.9}
$$

As Figure 6.1 shows, this is implemented by passing each $q_{k|k,i}$ to $f(x_k, u_k)$ along with the current input u_k from the MPC. From there N_p samples from w_k are drawn and added to each result from the state transition function to get $q_{k+1|k,i}$. The weights remain the same $w_{k+1|k,i} = w_{k|k,i}$. This gives the prior distribution $\delta(x|\mathbf{q_{k+1|k}}, \mathbf{w_{k+1|k}})$

For the serial case, this is done sequentially in a `for` loop. While for the parallel case, the function `f(particle, u, dt)` is vectorized using the `numba.guvectorize` decorator with the parameter `target='cuda'`. This creates a generalized vector function that runs on the GPU.

6.1.3 Update

When a new measurement $\boldsymbol{y}_{k+1}$ is available, the particle filter distribution is updated to the posterior estimate $\delta(x|\mathbf{q_{k+1|k+1}}, \mathbf{w_{k+1|k+1}})$ according to

$$\begin{aligned} p(x_{k+1}|y_{0:k+1}) &= p(x_{k+1}|y_{0:k}, \boldsymbol{y}_{k+1}) \\ &= \frac{p(\boldsymbol{y}_{k+1}|x_{k+1}, y_{0:k})p(x_{k+1}|y_{0:k})}{p(y_{k+1})} \\ &= \frac{p(\boldsymbol{y}_{k+1}|x_{k+1})p(x_{k+1}|y_{0:k})}{p(y_{k+1})} \end{aligned} \tag{6.10}$$

Again substituting in Equation 6.3 gives

$$p(x_{k+1}|y_{0:k+1}) = \frac{p(y_{k+1}|x_{k+1}) \sum_{i=0}^{N_p} w_{k+1|k,i}\delta(x_{k+1} - q_{k+1|k,i})}{p(y_{k+1})} \tag{6.11}$$

Similar to the prediction step, the new particles would ideally be drawn from the distribution shown in Equation 6.11, however, this is too computationally expensive. [1] Rather, the weights $w_{k+1|k,i}$ are updated from each of the N_p distributions

$$\begin{aligned} \frac{c_{k+1,i}p(y_{k+1}|x_{k+1})w_{k+1|k,i}\delta(x_{k+1} - q_{k+1|k,i})}{p(y_{k+1})} &= \frac{p(y_{k+1}|x_{k+1})\delta(x_{k+1} - q_{k+1|k,i})}{p(y_{k+1})} \\ &= \frac{p(y_{k+1}|q_{k+1|k,i})}{p(y_{k+1})} \\ &\propto p(y_{k+1}|q_{k+1|k,i}) \end{aligned} \tag{6.12}$$

for $i \in 1, ..., N_p$, where $c_{k+1,i}$ the normalization constant and the method followed is similar to that followed in Equation 6.9.

[1] Readers familiar with particle filters will note that sampling from Equation 6.6 and Equation 6.11 resembles a simultaneous normal particle filter prediction/update and resampling.

Figure 6.1 depicts this implementation. Each $q_{k+1|k,i}$ to $g(x_k, u_k)$ along with the current input u_{k+1} from the MPC to get $y_{k+1,i}$. This is then subtracted from the measurement

$$d_{k+1,i} = \boldsymbol{y}_{k+1} - y_{k+1,i} \tag{6.13}$$

The measurement noise probability density function $f_{v_k}(x)$ is then evaluated at $d_{k+1,i}$, the particle weights are updated with this value

$$w_{k+1|k+1,i} = w_{k+1|k,i} f_{v_k}(d_{k+1,i}) \tag{6.14}$$

and the point estimates remain the same $q_{k+1|k+1,i} = q_{k+1|k,i}$ to get $\delta(x|\mathbf{q_{k+1|k+1}}, \mathbf{w_{k+1|k+1}})$.

The serial version does each particle sequentially. The parallel version vectorizes the function `g(particle, u)` using the `numba.guvectorize` decorator with the parameter `target='cuda'`. It also used `cupy.ndarray` for the data so that the mathematical operations can be executed in parallel on the GPU as well.

6.1.4 Resampling

Figure 6.1 depicts the entire resampling process. At the heart of the process is the systematic resampling algorithm. For the parallel version, the algorithm designed by Nicely and Wells (2019) for systematic resampling is used. It performs very badly on very non-uniform weight distributions, but it performs better for more uniform weight distributions. The algorithm has a worst case time complexity of $T(n, p) = \frac{n^2}{p}$ and an average time complexity of $T(n, p) = \frac{n}{p}$. The algorithm is found to be work efficient.

Algorithm 6.1 details the procedure. The premise is to generate the uniform random sampling sequence and then guess the particle that should be sampled by assuming a uniform distribution. The algorithm then increments and decrements its guesses until it has found the correct particle. It uses the `cuda.jit` decorator to compile it to code that is executable on the GPU. The cumulative cum parameter `c` is also calculated on the GPU using `torch.cumsum`.

Algorithm 6.1. Nicely and Wells (2019) systematic resampling

```
@cuda.jit
def parallel_resample(c, sample_index, r, N):
```

```
tx = cuda.threadIdx.x
bx = cuda.blockIdx.x
bw = cuda.blockDim.x
i = bw * bx + tx

if i >= N:
        return

u = (i + r) / N
k = i
while c[k] < u:
        k += 1

cuda.syncthreads()

while c[k] > u and k >= 0:
        k -= 1

cuda.syncthreads()

sample_index[i] = k + 1
```

Once the sample indices have been generated, all that remains is to copy the desired particles to form the new distribution.

6.2 Gaussian sum filter

This section describes the algorithm for the Gaussian Sum Unscented Kalman Filter (GS-UKF or GSF) adapted from Kottakki *et al.* (2014). This implementation corrects errors in the sigma point weight calculations as well as to the local update step. The GSF draws N_p particles similar to the particle filter. However, where the particle filter uses a Dirac mixture approximation for the prior and posteriors, the GSF uses a Gaussian sum distribution:

$$\begin{aligned} \mathcal{N}_\Sigma(x|\mu_{\mathbf{k|j}}, \mathbf{\Sigma_{k|j}}) &= \sum_{i=0}^{N_p} w_{k|j,i} D_{k|j,i} \\ D_{k|j,i} &= \mathcal{N}(q_{k|j,i}, \Sigma_{k|j,i}) \end{aligned} \tag{6.15}$$

$(D_{k|j,i}, w_{k|j,i})$ denotes the i'th particle at the $k|j$'th time step. The initial distribution is formed as follows:

$$\begin{aligned} q_{0|0,i} &\leftarrow x_0 \sim \mathcal{N}_\Sigma(x|\mu_\mathbf{0}, \mathbf{\Sigma_0}) \\ \Sigma_{0|0,i} &= \text{average}(\mathbf{\Sigma_0}) \\ w_{0|0,i} &= \frac{1}{N_p} \end{aligned} \tag{6.16}$$

where $x \leftarrow y$ indicated that x is drawn from distribution y; and average$(\mathbf{\Sigma_0})$ is formed by multiplying the covariances from the Gaussian sum distribution element-wise and then dividing by the number of distributions. The multivariate Gaussian sum distribution $\mathcal{N}_\Sigma(x|\mathbf{q_{0|0,i}}, \mathbf{\Sigma_{0|0,i}})$ formed by the particles $(D_{0|0,i}, w_{0|0,i})$ then approximates $p(x_0) = p(x_0|y_0)$.

6.2.1 Code diagram

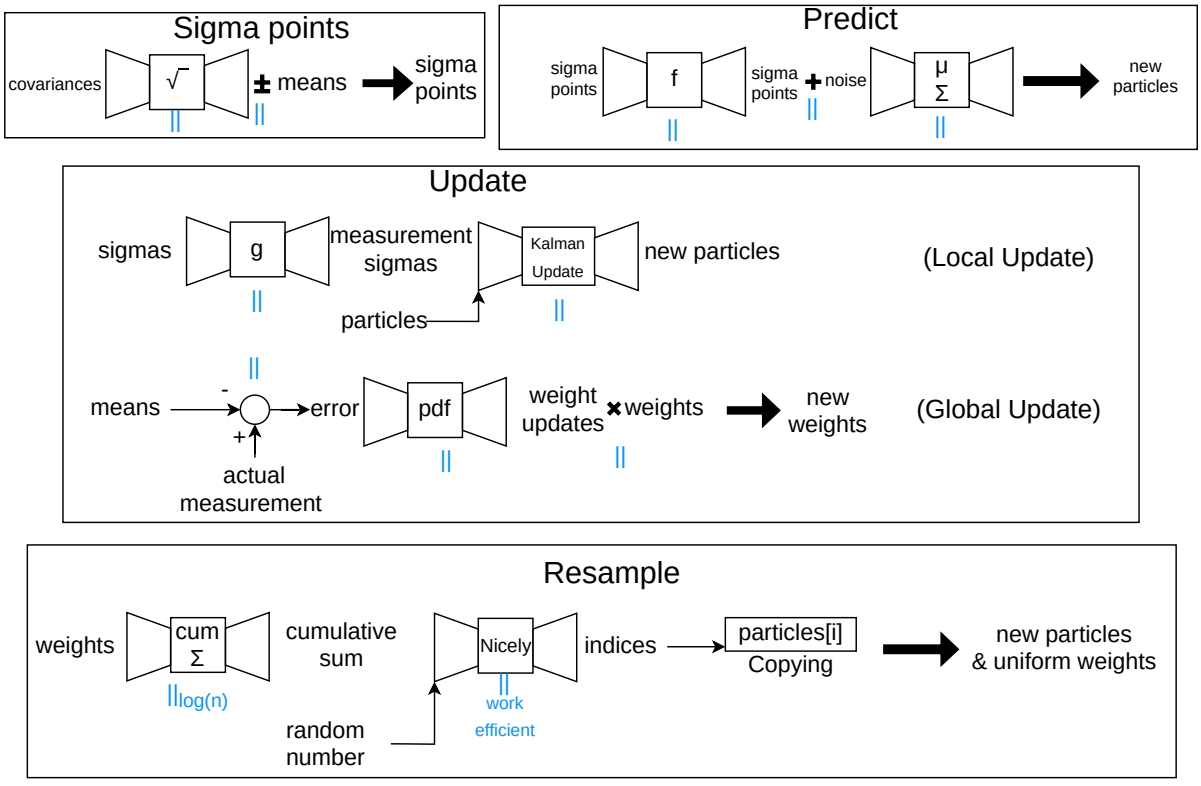

Figure 6.2: Gaussian sum filter code diagram.

Figure 6.2 shows the algorithm for the Gaussian sum filter. The blue ‖ symbols indicate areas where GPGPU parallelization is implemented. The sections below describe the derivation of this algorithm.

6.2.2 σ-points

An important subtask of the GSF is obtaining sigma points from a multivariable Gaussian distribution. These points serve to aid in representing the Gaussian distribution so that it can be transformed through the nonlinear state observation and state transition functions. This section will outline the algorithm used to obtain them. Figure 6.2 depicts how the $N_\sigma = (2N_x + 1)$ sigma points for the distribution $D_{k|j,i}$ are generated. Mathematically it is stated as:

$$\begin{aligned}
\sigma^{(0)}_{k|j,i} &= q_{k|j,i} \\
\sigma^{(l)}_{k|j,i} &= q_{k|j,i} + [\sqrt{\Sigma_{k|j,i}}]_l && \forall\, l \in [1, N_x] \\
\sigma^{(l)}_{k|j,i} &= q_{k|j,i} - [\sqrt{\Sigma_{k|j,i}}]_{l-N_x} && \forall\, l \in [N_x + 1, N_\sigma]
\end{aligned} \tag{6.17}$$

where $[M]_l$ selects the l'th column of matrix M; and $\sqrt{M}$ is the Cholesky decomposition of matrix M. It can happen that, during the running of the filter, the Cholesky decomposition fails. This is a common occurrence and occurs due to numerical inaccuracies. The solution implemented is to then compute $\sqrt{\Sigma_{k|j,i} + \epsilon I}$, where ϵ is a small positive number. For the CPU implementation, the `numpy` library is used for the decomposition and matrix additions. The GPU implementation uses `cupy`.

Each sigma point has an associated weight $w^\sigma_{k|j,i}$, defined as:

$$\begin{aligned}
w^\mu_{k|j,i} &= w^{(0)}_{k|j,0} \\
w^\eta_{k|j,i} &= w^{(l)}_{k|j,i} \forall\, l \in [1, N_\sigma] \\
w^\mu_{k|j,i} + 2N_x w^\eta_{k|j,i} &= 1 \\
\frac{w^\mu_{k|j,i}}{w^\eta_{k|j,i}} &= \frac{8}{5} \\
&\approx \frac{f_\mathcal{N}(0|0,1)}{f_\mathcal{N}(1|0,1)}
\end{aligned} \tag{6.18}$$

where $f_\mathcal{N}(x|0,1)$ is the value of the normal distribution's probability density function at x. This simplifies to:

$$
\begin{aligned}
w_{k|j,i}^{(0)} &= \frac{8}{8+10N_x} \\
w_{k|j,i}^{(l)} &= \frac{5}{8+10N_x} \forall\ l \in [1, N_\sigma]
\end{aligned}
\tag{6.19}
$$

The sigma points can be packed into a $N_\sigma \times N_x$ matrix $\sigma_{\mathbf{k|j,i}}$. The weights can be packed into a $N_\sigma \times 1$ matrix $\mathbf{w_{k|j,i}}$. The sigma point weight vector is a constant throughout the filter's runtime and is only generated once.

6.2.3 Prediction

Similar to the particle filter, the prediction algorithm for the particle filter, the Gaussian sum filter substitutes Equation 6.15 into Equation 6.5.

$$
\begin{aligned}
p(x_{k+1}|y_{0:k}) &= \int_{x_k} p(x_{k+1}|x_k) p(x_k|y_{0:k}) dx_k \\
&= \int_{x_k} p(x_{k+1}|x_k) \sum_{i=0}^{N_p} w_{k|K,i} D_{k|k,i} dx_k \\
&= \int_{x_k} \sum_{i=1}^{N_p} p(x_{k+1}|x_k) w_{k|k,i} \mathcal{N}(x_k|q_{k|k,i}, \Sigma_{k|k,i}) dx_k
\end{aligned}
\tag{6.20}
$$

As with the particle filter, a Gaussian sum distribution would ideally be fit to this, but due to the complexity, a simplified approximation is used. A single Gaussian is fit to each of the N_p distributions:

$$
c_{k|k,i} \int_{x_k} p(x_{k+1}|x_k) w_{k|k,i} \mathcal{N}(x_k|q_{k|k,i}, \Sigma_{k|k,i}) dx_k \tag{6.21}
$$

for $i \in 1, ..., N_p$, where $c_{k|k,i}$ is a normalization constant. Drawing a Gaussian from Equation 6.21 is done using the unscented Kalman filter prediction algorithm. The algorithm generates sigma points $\sigma_{k|k,i}$ with weights $w_{k|k,i}^{(l)}$ and then passes them through the state transition function (see Figure 6.2) to get the transformed sigma points $\sigma_{k+1|k,i}$:

$$
\begin{aligned}
\sigma_{k+1|k,i} &= f(\sigma_{k|k,i}, u_k) + w_k^* \\
w_k^* &\leftarrow w_k
\end{aligned}
\tag{6.22}
$$

The CPU implementation does each transformation sequentially, while the GPU implementation does them in parallel. The new Gaussian $D_{k+1|k,i} = \mathcal{N}(x|q_{k+1|k,i}, \Sigma_{k+1|k,i})$ is then given by:

$$\begin{aligned} q_{k+1|k,i} &= {\mathbf{w}^{\sigma}_{\mathbf{k}|\mathbf{k},\mathbf{i}}}^{T} \sigma_{\mathbf{k+1}|\mathbf{k},\mathbf{i}} \\ \Sigma_{k+1|k,i} &= (\sigma_{\mathbf{k+1}|\mathbf{k},\mathbf{i}} - q_{k+1|k,i})^{T} (\mathbf{w}^{\sigma}_{\mathbf{k}|\mathbf{k},\mathbf{i}} \odot (\sigma_{\mathbf{k+1}|\mathbf{k},\mathbf{i}} - q_{k+1|k,i})) \end{aligned} \tag{6.23}$$

where $\odot$ performs an element-wise multiplication. The weights remain the same $w_{k+1|k,i} = w_{k|k,i}$. This gives the prior distribution $p(x_{k+1}|y_{0:k}) \approx \mathcal{N}_{\Sigma}(x|\mathbf{q_{k+1|k}}, \mathbf{\Sigma_{k+1|k}})$ as desired.

Both the CPU implementation and GPU implementation perform Equation 6.23 using array broadcasting and vectorization. However, the GPU implementation makes use of `cupy` arrays which allow the operations to be done on the GPU.

6.2.4 Update

Given a new measurement y_{k+1}, the GSF distribution is updated to the posterior estimate $p(x_{k+1}|y_{0:k+1}) \approx \mathcal{N}_{\Sigma}(x|\mathbf{q_{k+1|k+1}}, \mathbf{\Sigma_{k+1|k+1}})$ according to Equation 6.10. Substituting in Equation 6.15 gives

$$p(x_{k+1}|y_{0:k+1}) = \frac{p(y_{k+1}|x_{k+1}) \sum_{i=0}^{N_p} w_{k+1|k,i} \mathcal{N}_{\Sigma}(x|\mu_{\mathbf{k+1|k}}, \mathbf{\Sigma_{k+1|k}})}{p(y_{k+1})} \tag{6.24}$$

Similar to the prediction step, the new prior would ideally be fit to Equation 6.24. However, this is too computationally expensive and so it is approximated by performing local unscented Kalman filter updates on each particle, and performing a global update on the collection.

Local Update

The local update is an update unscented Kalman filter performed on each particle. As seen in Figure 6.2, the update begins by drawing sigma points $\sigma_{k+1|k,i}$ with weights $w^{(l)}_{k+1|k,i}$ and then passing these through the state observation function $g(x_k, u_k)$ to get the transformed points $\eta_{k+1|k,i}$. From these, the Kalman gain is determined as:

$$
\begin{aligned}
\hat{\eta}_{k+1|k,i} &= \mathbf{w_{k+1|k,i}}^T \eta_{\mathbf{k+1|k,i}} \\
\hat{\sigma}_{k+1|k,i} &= \mathbf{w_{k+1|k,i}}^T \sigma_{\mathbf{k+1|k,i}} \\
\Sigma^{\eta\eta}_{k+1|k,i} &= (\eta_{\mathbf{k+1|k,i}} - \hat{\eta}_{k+1|k,i})^T (\mathbf{w_{k|k,i}} \odot (\eta_{\mathbf{k+1|k,i}} - \hat{\eta}_{k+1|k,i})) \\
\Sigma^{\sigma\eta}_{k+1|k,i} &= (\sigma_{\mathbf{k+1|k,i}} - \hat{\sigma}_{k+1|k,i})^T (\mathbf{w_{k|k,i}} \odot (\eta_{\mathbf{k+1|k,i}} - \hat{\eta}_{k+1|k,i})) \\
K_{k+1|k,i} &= \Sigma^{\sigma\eta}_{k+1|k,i} (\Sigma^{\eta\eta}_{k+1|k,i})^{-1}
\end{aligned}
\tag{6.25}
$$

Using the Kalman gain, the distribution $D_{k+1|k,i}$ is updated:

$$
\begin{aligned}
D_{k+1|k+1,i} &= \mathcal{N}(x | q_{k+1|k+1,i}, \Sigma_{k+1|k+1,i}) \\
q_{k+1|k+1,i} &= q_{k+1|k,i} + K_{k+1|k,i}(\boldsymbol{y}_{k+1} - \hat{\eta}_{k+1|k}) \\
\Sigma_{k+1|k+1,i} &= \Sigma_{k+1|k,i} - K_{k+1|k,i} \Sigma^{\eta\eta}_{k+1|k,i} K^T_{k+1|k,i}
\end{aligned}
\tag{6.26}
$$

Both the CPU and GPU implementations use vectorized array operations, but the GPU's operations benefit from being run on the GPU and thus receive some parallelization. The CPU implementation must transform each sigma point through the state observation function sequentially, while the GPU implementation can parallelize the operation.

Global Update

The local update saw the updating of the distribution of each particle. The global update uses the current measurement $\boldsymbol{y}_{k+1}$ to update the weights of each particle (see Figure 6.2). The weights are updated as follows:

$$
\begin{aligned}
w_{k+1|k+1,i} &= w_{k+1|k,i} \left[f_{v_k}(\boldsymbol{y}_{k+1} - y_{k+1,i}) \right] \\
y_{k+1,i} &= g(q_{k+1|k+1,i}, u_{k+1})
\end{aligned}
\tag{6.27}
$$

6.2.5 Resampling

As seen in Figure 6.2, the resampling method for the GSF is exactly the same as for the particle filter. The parallel algorithm uses the method designed by Nicely and Wells (2019) for systematic resampling.

Part III

Results and discussion

7 Model

This chapter shows important open and closed loop results for bioreactor and MPC combination. These results are used to confirm proper implementation of their methods as well as to lay the required foundation on which the filtering results are built.

7.1 Open loop

This section shows the results of the open loop bioreactor model. Three important results are shown, the first is seen in Figure 7.1. The step tests show the dynamics of the system around the relevant operating point. The system's dynamics give direction to the scale of the fast and slow elements of the system. The slowest dynamics take around 300 min to settle out completely, and the fastest dynamics happen on the order of 5 min. Based on this, the following parameters are chosen for future closed loop filtered simulations: integration time step — suitable at 0.1 min; MPC prediction horizon — 300 min; MPC control horizon — 200 min; smallest required control horizon — 0.1 min. The prediction and control horizons allow the controller to anticipate future dynamics sufficiently, and the low control, integration and prediction times occur fast enough that the controller and filters can react and the simulation is accurate enough.

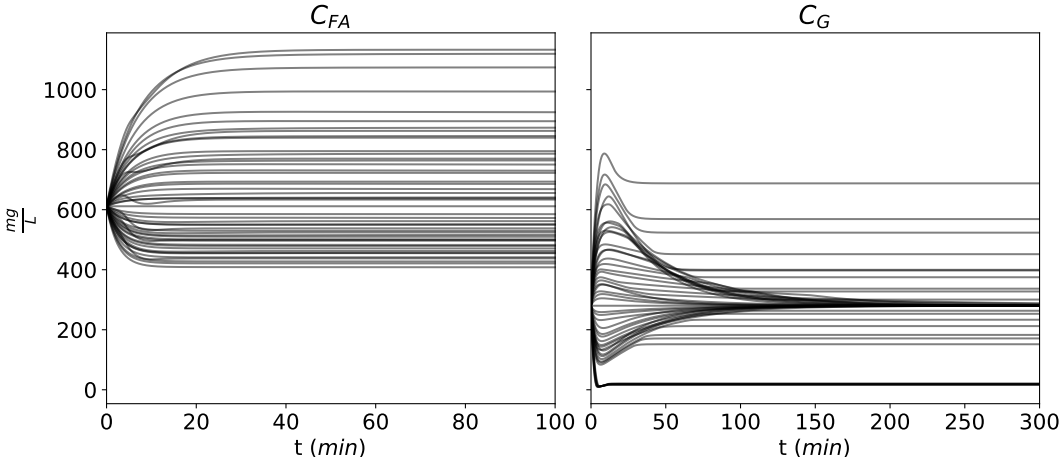

Figure 7.1: Open loop step tests of the system around the point (C_{FA} =640 mg L^{-1}, C_G =280 mg L^{-1}) with inputs ($F_{m,in}$ =0.200 L min^{-1}, $F_{G,in}$ =0.060 L min^{-1}). The tests are found by varying the inputs to 70 %, 50 %, 80 %, 120 %, 130 % and 150 % of their steady state values. The simulation's integration time step is 0.1 min. It is important to note that only the general shapes of the curves are relevant, and thus, labels of individual lines have been omitted.

The next result is seen in Figure 7.2. This figure shows the following results from Swart (2019) are emulated by the model: the 300 mg L^{-1} glucose is completely consumed in the first 25 min which is the batch run; before glucose overflow, the cell maintains homoeostasis (280 mg L^{-1} Glucose), but cannot maintain homoeostasis after glucose overflow; ethanol overflow occurs after the ethanol overflow point as predicted. The steady state C_{FA} changes due to the increased dilution rate of the reactor caused by increasing the glucose feed rate.

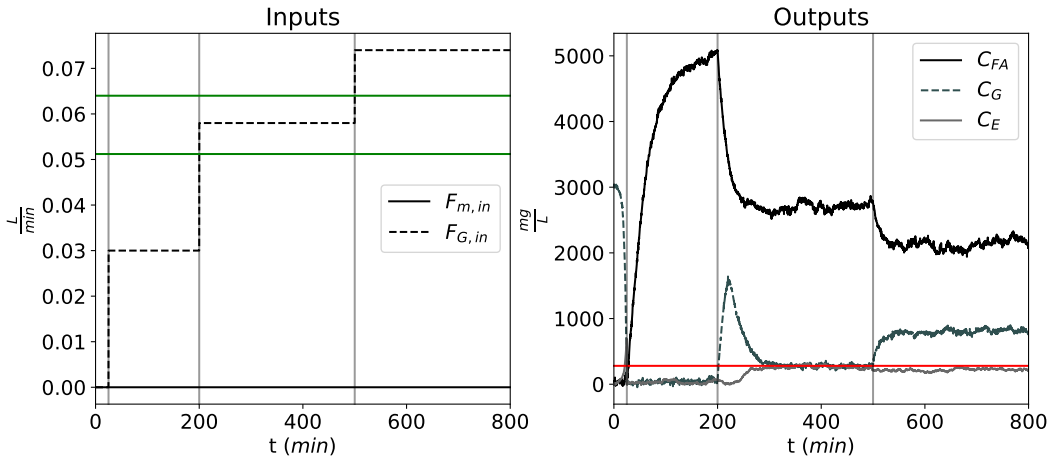

Figure 7.2: Open loop bioreactor showing both the growth and production phases. The grey vertical lines show the times when the glucose feed rate is changed. The lower and upper green horizontal lines mark the glucose feed rates at which ethanol and glucose overflow occur, respectively.

The final result seen in Figure 7.3 shows how the reactor moves from a steady state operating point at (C_{FA} =640 mg L^{-1}, C_G =280 mg L^{-1}) with inputs ($F_{m,in}$ =0.2 L min^{-1}, $F_{G,in}$ =0.06 L min^{-1}) to a new steady state operating point at (C_{FA} =1150 mg L^{-1}, C_G =280 mg L^{-1}) with inputs ($F_{m,in}$ =0.1 L min^{-1}, $F_{G,in}$ =0.04 L min^{-1}).

This is an important result, because the closed loop tests for the remainder of this discussion will move between these two operating points. The model is linearized around the second operating point to give the linear model for the MPC.

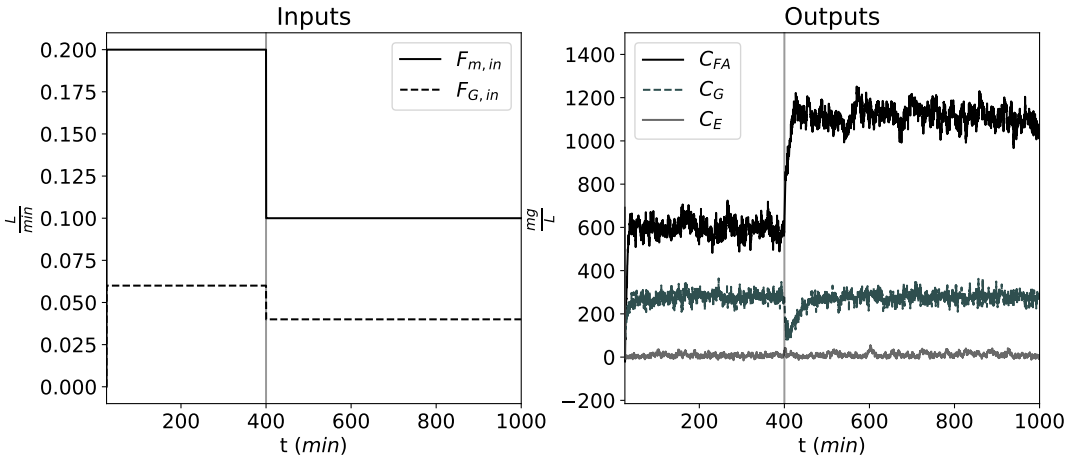

Figure 7.3: Open loop bioreactor transition between steady states. The grey vertical lines show the times when the glucose and mineral feed rates are changed.

7.2 Closed loop

This section shows the results of using the MPC to move the system between the two operating points. The result shown in Figure 7.4 shows that the MPC can move an ideal noiseless system to the desired operating point and maintain it. The result comes from using tuning parameters

$$\begin{aligned} Q &= \begin{bmatrix} 100 & 0 \\ 0 & 1000 \end{bmatrix} \\ R &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ P &= 200 \\ M &= 160 \\ 0 &\leq F_{m,in} \\ 0 &\leq F_{G,in} \end{aligned} \tag{7.1}$$

and the control period is 1 min. The controller's outputs ($F_{m,in}$ and $F_{G,in}$) show the classic MPC shapes, with large moves in the beginning that overshoot the final output value, followed by correction moves and smaller moves to take the output to its final steady value. The bias reaches a steady value which also shows good controller operation.

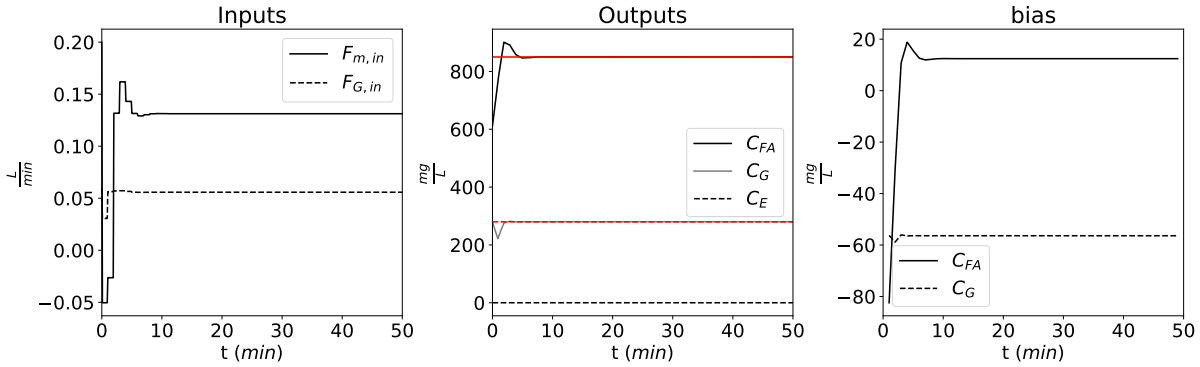

Figure 7.4: Closed loop bioreactor simulation transitioning between steady states (no noise). The red lines are the set points for the fumaric acid (solid red line) and glucose (dashed red line) concentrations.

Figure 7.5 shows that the controller can also maintain set point for a system with noise. The noise causes the outputs to no longer show the ideal MPC outputs as before. The run below exhibits worse performance than the noiseless case. This is to be expected, since the controller needs to reject disturbances caused by plant–model mismatch and measurement noise.

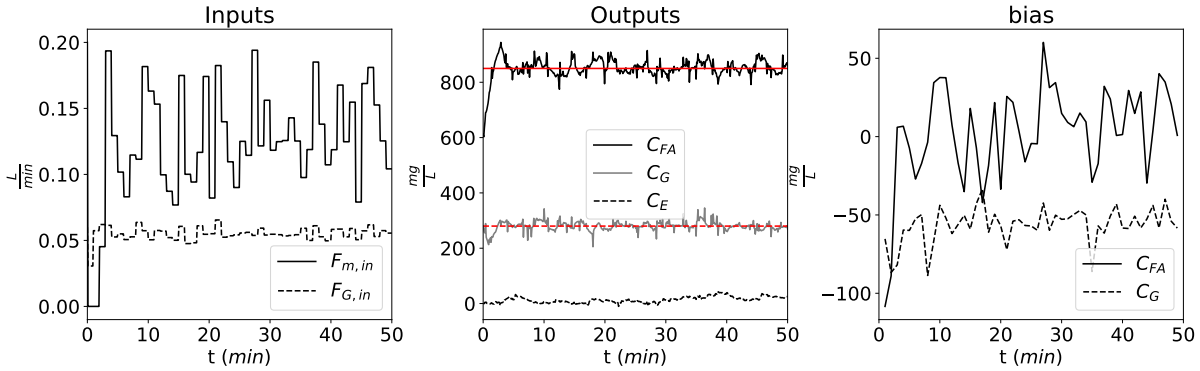

Figure 7.5: Closed loop bioreactor transition between steady states. The red lines are the set points for the fumaric acid (solid red line) and glucose (dashed red line) concentrations.

Figure 7.6 shows a Monte Carlo simulation indicating the effect of control period on control performance. Each point in the figure shows the P_{ISE} value for some control period. The figure shows how increasing control period decreases performance. However, it also requires the use of better and faster hardware which is often unnecessary depending

on the system's dynamics. Thus, taking into consideration the result found in Section 7.1 for the system's time constant, the lowest control period considered for this work is chosen to be 0.1 min.

The figure also shows that as the control period increases the variance in performance increases as well. This is due to the increased effect of state noise between control calculations.

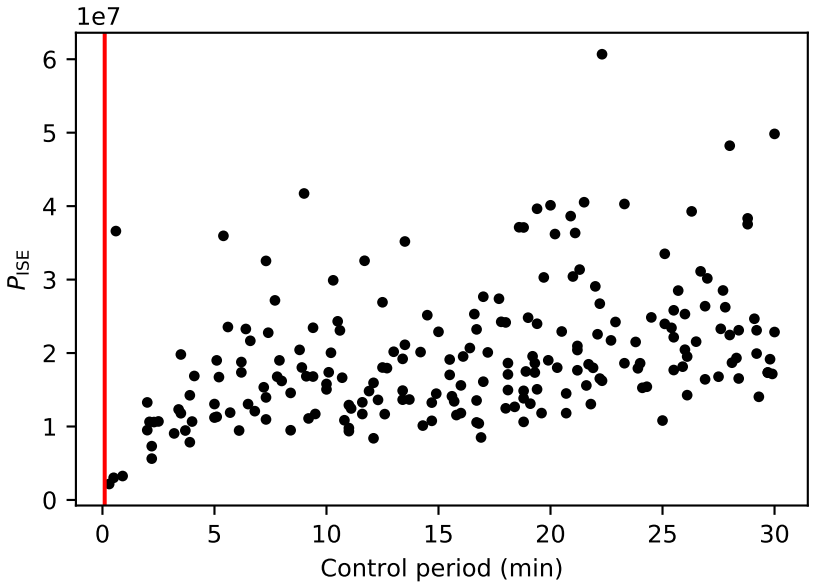

Figure 7.6: Effect of control period on ISE performance (P_{ISE}).

The run time of the MPC is benchmarked. The benchmarking process involves running the MPC code multiple times to get a run sequence. An example run sequence for the no-op and timing method are shown in the left plots of Figure 7.7. The no-op function (top row) is a method that does nothing and the timing function (bottom row) makes a call to the function that does the timing for the other run sequences. The first graphs show the run sequence plots for 100 runs.

Further analysis is done on the run sequence plots to determine if there is any statistical correlation between the times of successive runs. The second plots show the -1 lag plot. It is constructed by lagging consecutive run times by -1. The last plot is known as an autocorrelation plot. It measures the autocorrelation for different lag plots. This is the most useful in determining patterns amongst the times. An autocorrelation of less than 0.2 is chosen to mean that there is no statistical significance between run times.

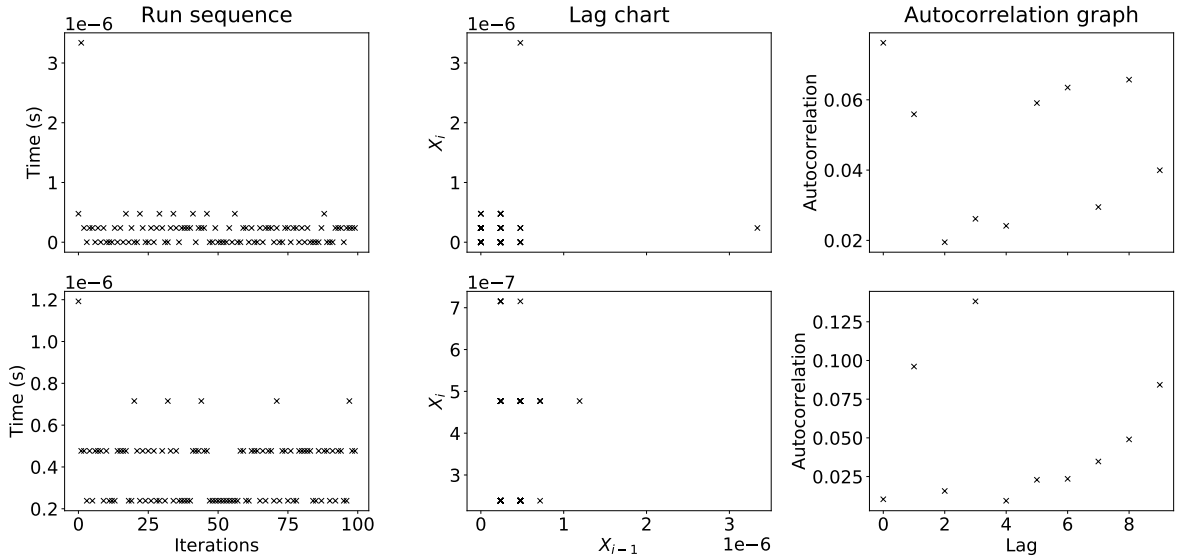

Figure 7.7: Benchmarking for no-op and `time.time()`.

The MPC benchmarking is shown in Figure 7.8. The figure shows little statistically significant correlation between points for most lag configurations. For this reason the median run time of 0.03 s is an accurate representation of the runtime of the MPC code.

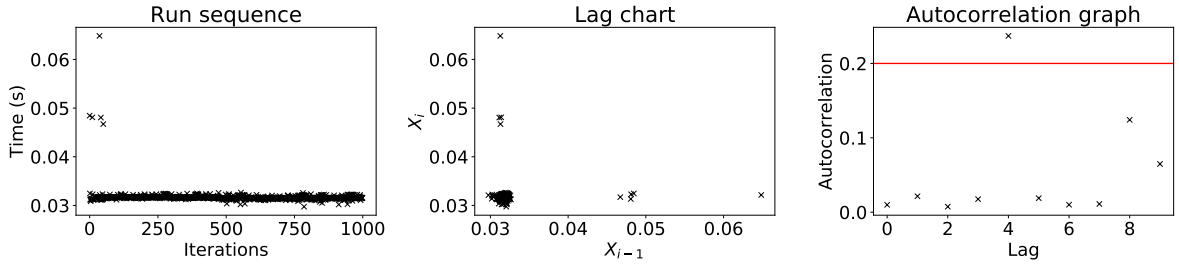

Figure 7.8: Benchmarking for MPC step function.

8 Filters

Open and closed loop results for the particle and Gaussian sum filters.

8.1 Open loop

Analysis, using Section 2.6, of the algorithms for the parallel prediction and update routines (for both filters) gives

$$\begin{aligned} T_{\parallel\text{predict}}(n,p) &= T_{\parallel\text{update}}(n,p) \\ &= \frac{n}{p} \\ \mathcal{O}(p(n)) &= \begin{cases} n & \text{if} \quad n \leq p_{\max} \\ 1 & \text{otherwise} \end{cases} \\ \implies T_{\parallel\text{predict}}(n) &= T_{\parallel\text{update}}(n) \\ &= \begin{cases} 1 & \text{if} \quad n \leq p_{\max} \\ n & \text{otherwise} \end{cases} \end{aligned} \tag{8.1}$$

and for the serial algorithms for prediction and updates:

$$T_{\perp\text{predict}} = T_{\perp\text{update}} = n \tag{8.2}$$

In order to test this if these results agree with the implementations, the prediction, update and resampling methods are benchmarked. A run sequence is done for each method for both the CPU and GPU implementations. Maximum autocorrelation values for each filter size is shown in Figure 8.1 for the particle filter and Figure 8.2 for the GSF. The figures show that the autocorrelation between runs is statistically significant. The cause

of this is beyond the scope of this work. It is conjectured that the cause is internal to Python (e.g. garbage collection). This significance is investigated in later results.

Due to memory constraints, the maximum number of particles for the particle filter is $2^{19.5}$, and is $2^{18.5}$ for the Gaussian sum filter.

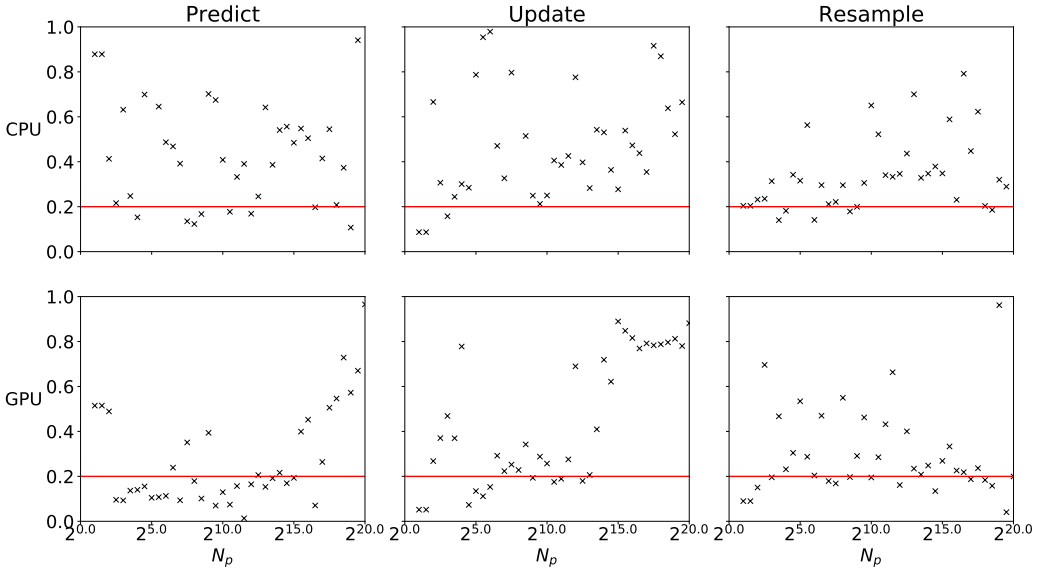

Figure 8.1: Maximum autocorrelation plot for each method for the particle filter. Done for both the CPU and GPU implementations.

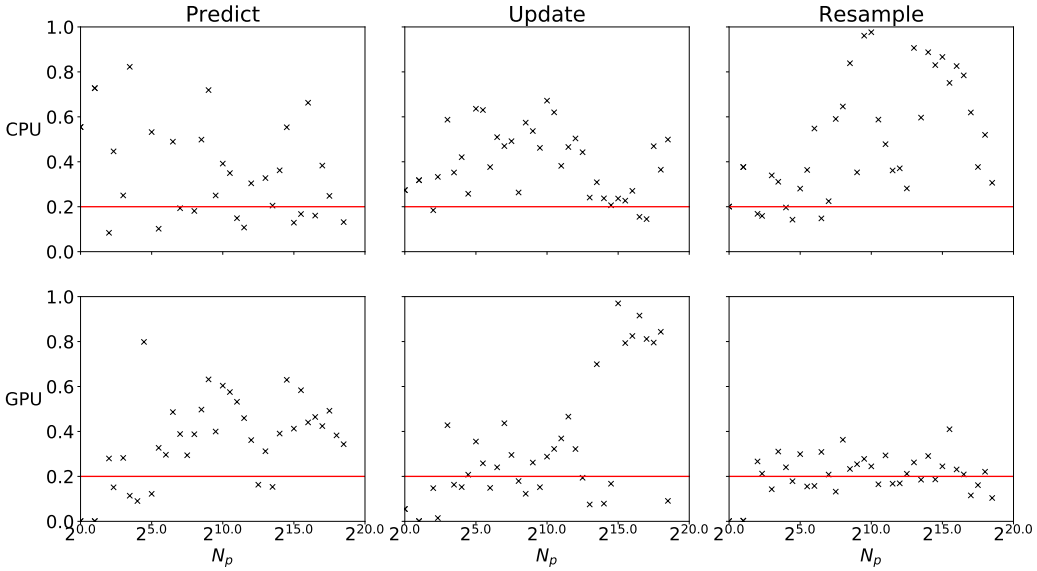

Figure 8.2: Maximum autocorrelation plot for each method for the Gaussian sum filter. Done for both the CPU and GPU implementations.

The run time results in Figure 8.3 and Figure 8.4 are in line with the complexity results shown earlier. [1]. The difference in $p_{\max}$ between the different methods is accounted for by internal effects on the GPU like multi-threading and memory caching. The GPGPU particle filter with $2^{19.5}$ particles has prediction, update and resampling run times of 0.011 s, 0.065 s, 0.0013 s, respectively. These times imply that the filter can easily be used in real time applications on systems with dynamic time constants on the order of 0.1 s (assuming that it is desired for the control period to be around ten times faster that the fastest dynamics). This is compared to the prediction, update and resampling step times of 176 s, 32 s, 0.55 s, respectively, for the CPU particle filter.

The GPGPU Gaussian sum filter with $2^{18.5}$ particles has prediction, update and resampling step times of 0.087 s, 0.1 s, 0.0024 s, respectively. These times would allow the filter to be used on systems with dynamic time constants on the order of 1 s. This is compared to the prediction, update and resampling step times of 240 s, 38 s, 0.27 s, respectively, for the CPU Gaussian sum filter.

Figure 8.3 and Figure 8.4 use error bars to show the range of data between the 10th and 90th percentiles. Outliers are plotted separately in blue. For most points, most of the data lies very closely grouped between the 10th and 90th percentiles. For this reason, the median value for run sequences is assumed to be a good representative for the data.

[1] If $\log(\text{Time}) = \log_2(N_{\text{particles}}) + c$, then $\text{Time} = kN_{\text{particles}} \implies T(n) = n$. Similarly, if $\log(\text{Time}) = c$, then $\text{Time} = kN_{\text{particles}} \implies T(n) = 1$

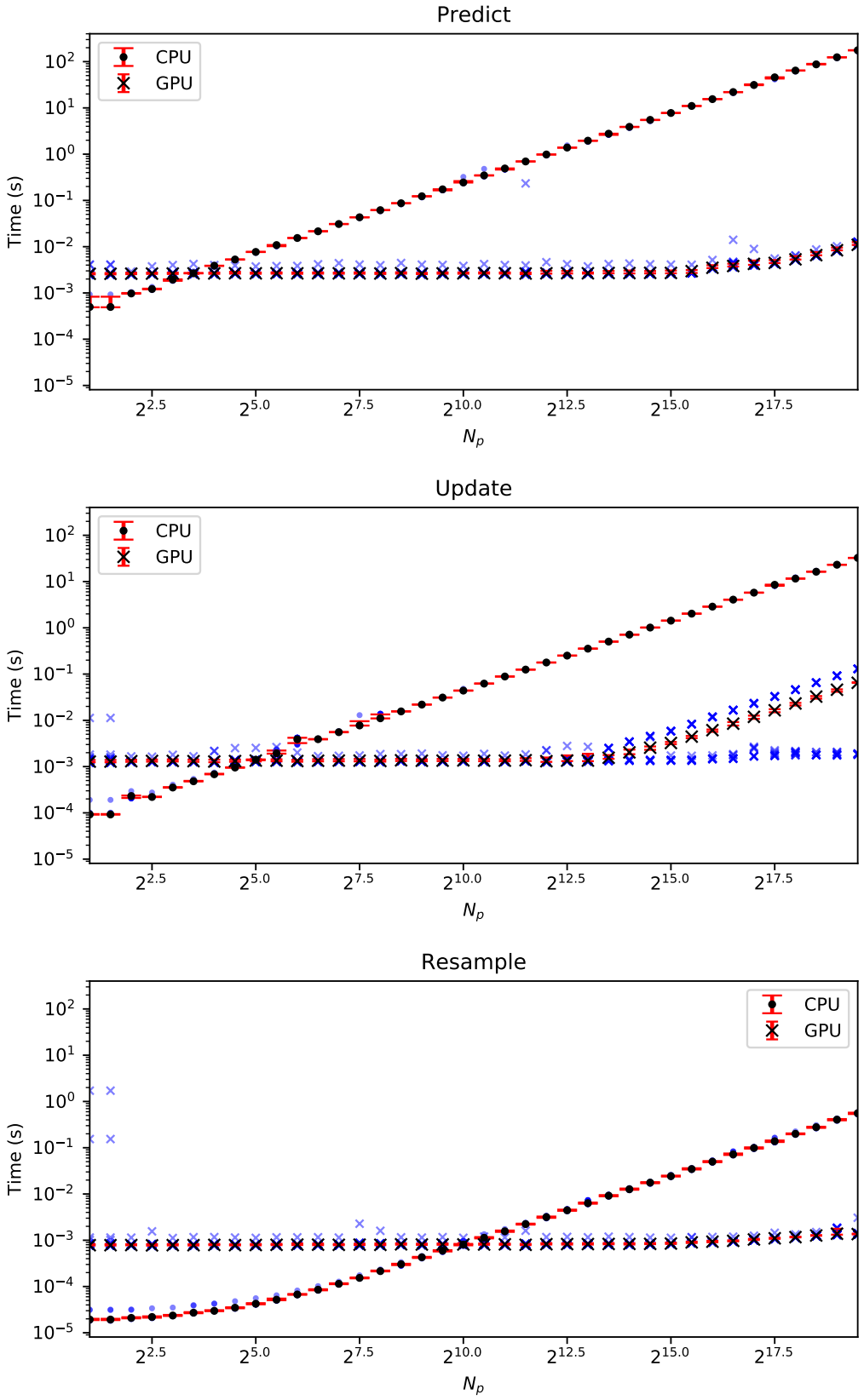

Figure 8.3: Run times of CPU and GPU particle filters. The red error bars show the 10th to 90th percentile. The blue points show the outliers.

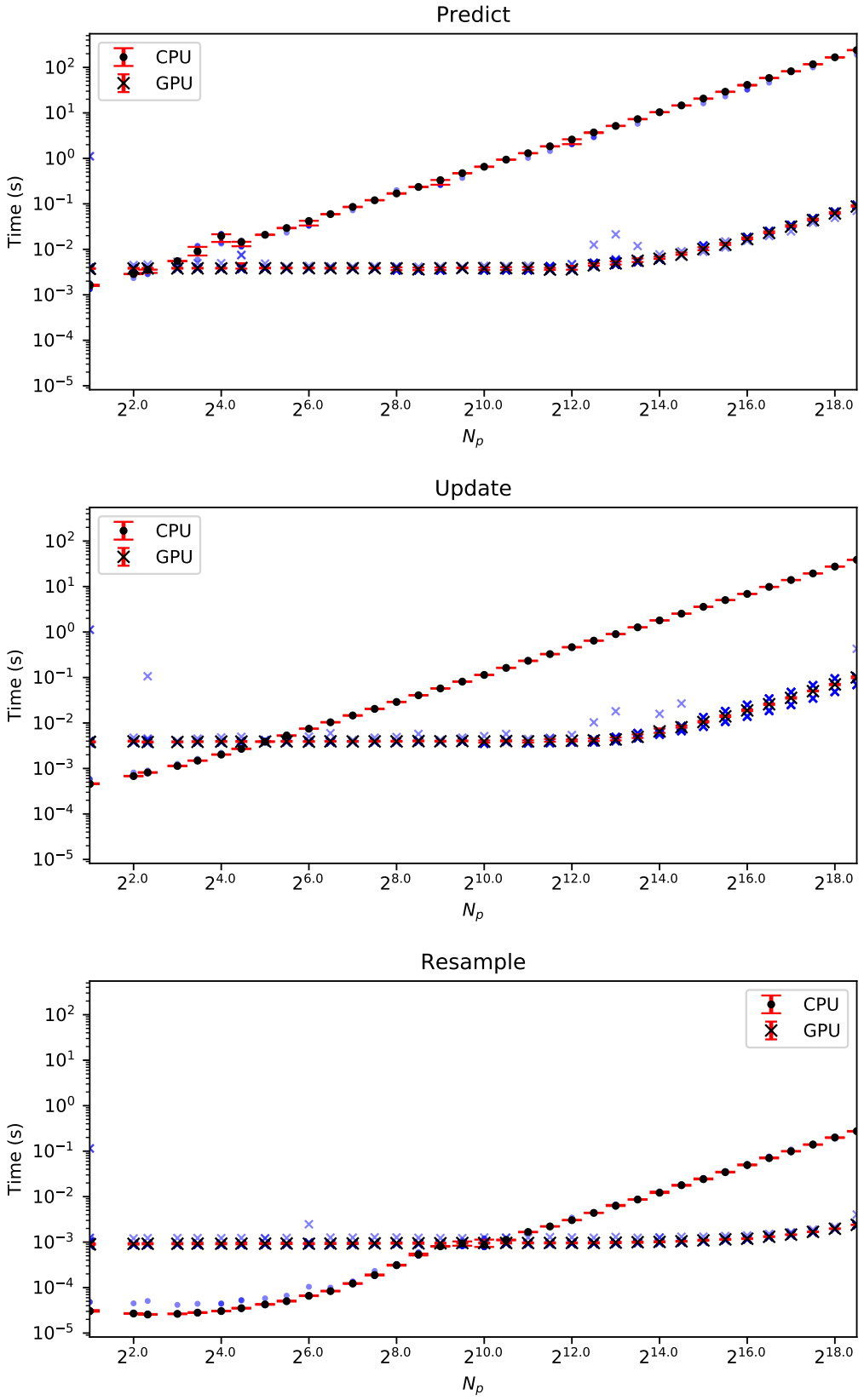

Figure 8.4: Run times of CPU and GPU Gaussian sum filters. The red error bars show the 10th to 90th percentile. The blue points show the outliers.

Figure 8.5 and Figure 8.6 allows better comparison between the GPU and CPU implementations. They show the median speed-up for each method for various numbers of particles. Figure 8.5 shows that for smaller numbers of particles (up to $2^{4.5}$ for update, 2^3 for prediction, and 2^{10} for resampling) the CPU implementation is between 5 and 40 times faster. This is to be expected because GPU code has more overhead than CPU code [2]. This apparent advantage of the CPU code over the GPU code is further diminished when one considers that for most practical implementations one requires high numbers of particles for even the most basic systems to get an accurate representation of the distribution.

For larger particle numbers ($2^{19.5}$), the GPU implementation is up to $10^{4.2}$ times faster for prediction, $10^{2.7}$ times faster for updating, and up to $10^{2.6}$ times faster for resampling. The resampling result is in line with the results found by Nicely and Wells (2019) in their implementation of their resampling algorithm.

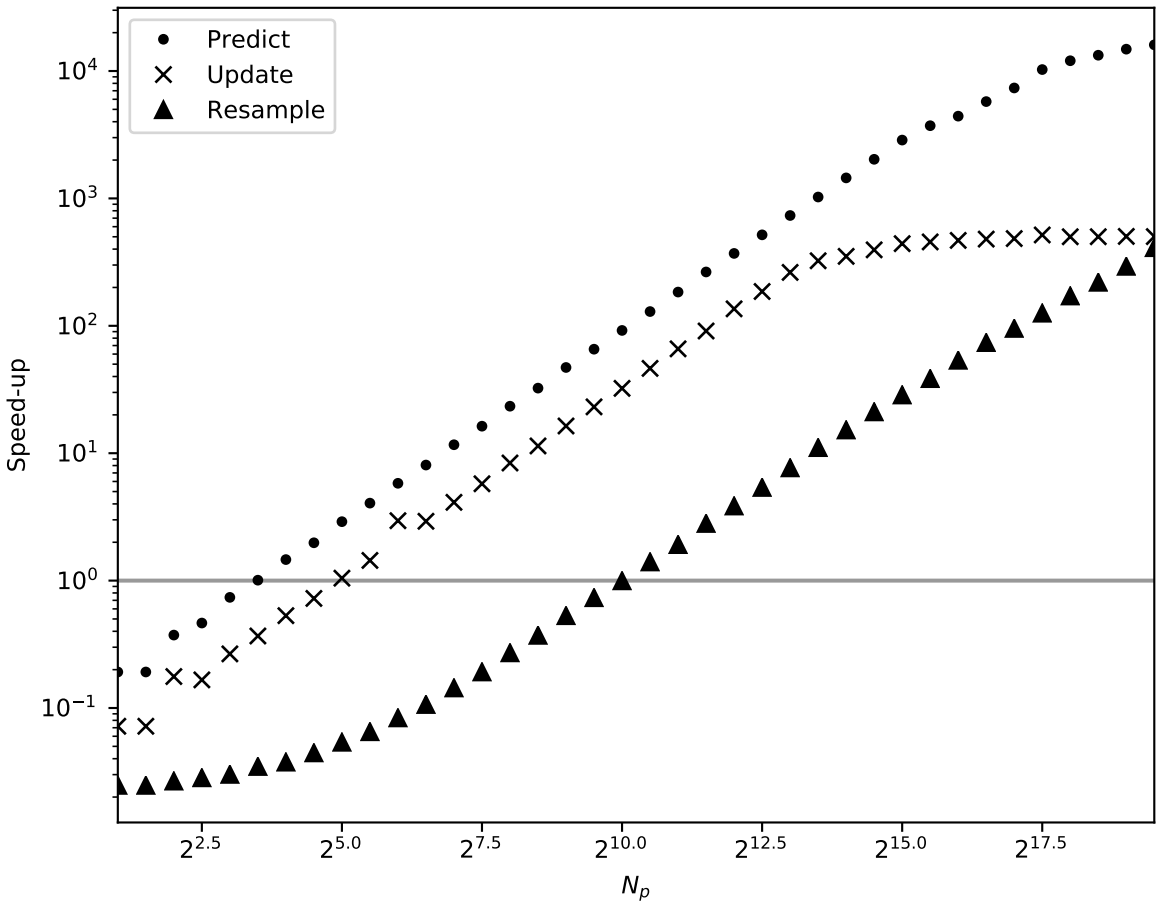

Figure 8.5: Median GPU speed-up of particle filter. Calculated as $\frac{\text{Time}_{\text{CPU}}}{\text{Time}_{\text{GPU}}}$ from the times in Figure 8.3. The horizontal line at 1 marks where the filters have equal performance.

[2]Overhead is the amount of time taken to set-up the memory and threads before the program can execute.

For the GSF, Figure 8.6 shows that for smaller numbers of particles (up to 2^5 for update, $2^{2.5}$ for prediction, and 2^{10} for resampling) the CPU implementation is between 4 and 45 times faster. For larger particle numbers ($2^{18.5}$), the GPU implementation is up to $10^{3.44}$ times faster for prediction, $10^{2.6}$ times faster for updating and up to $10^{2.06}$ times faster for resampling.

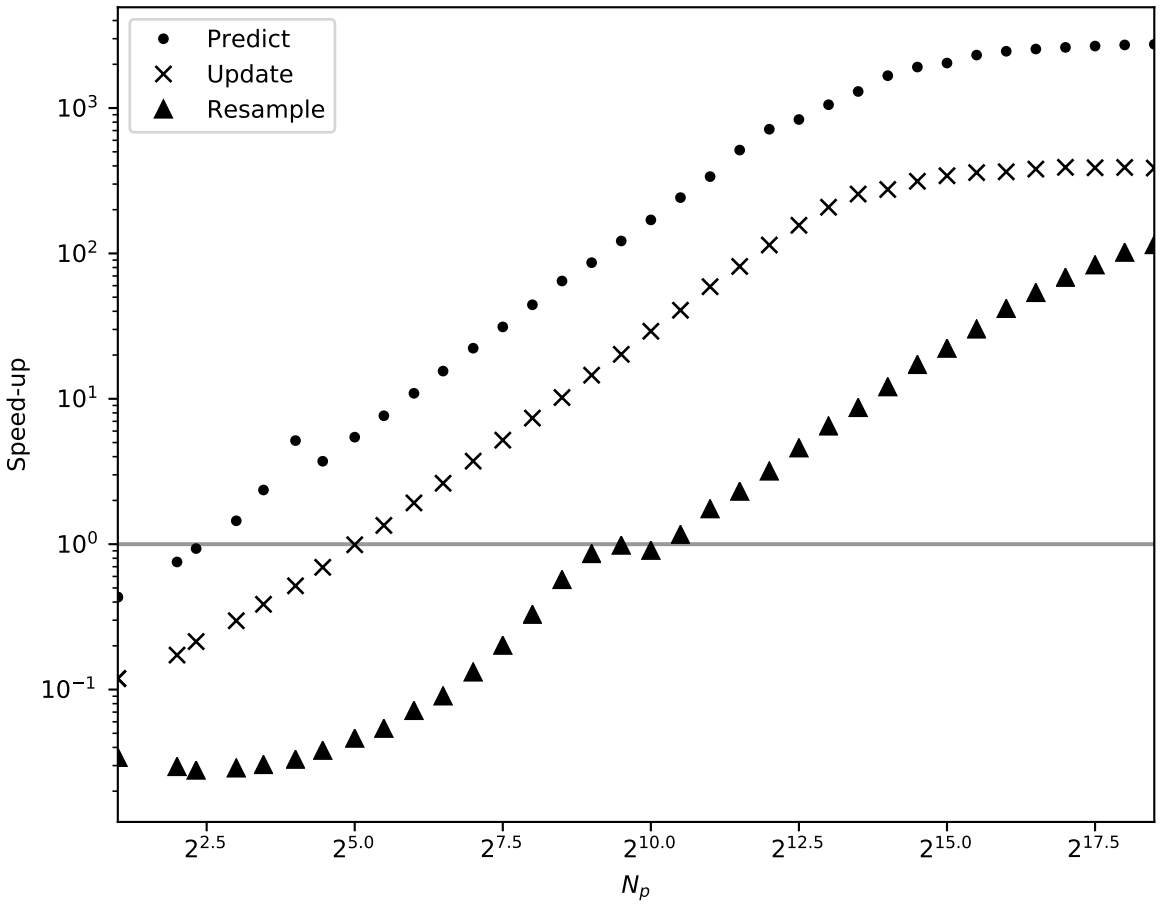

Figure 8.6: Median GPU speed-up of Gaussian sum filter. Calculated as $\frac{\text{Time}_{\text{CPU}}}{\text{Time}_{\text{GPU}}}$ from the times in Figure 8.3. The horizontal line at 1 marks where the filters have equal performance.

As stated in Section 1.5, the GPGPU has 1920 cores while the CPU only uses a single core. However the GPGPU clock speed is 53 % of the CPU clock speed. If one assumes that the computation time of the algorithm is directly proportional to clock speed, then the differences between the devices would give a speed-up of approximately $10^{3.01}$ for the GPGPU over the CPU. However, as seen in Figure 8.5 and Figure 8.6, this speed-up is only obtained for the prediction step with higher numbers of particles. The cause for this is potentially attributed to factors like not all of the GPGPU cores being used and GPGPU overheads.

In order to evaluate the effectiveness of the GPU implementation, each of the three methods were subdivided and the subroutines were timed. Figure 8.7 shows the results for the particle filter. It can be seen that memory copying between the CPU and GPU takes negligible time and does not form a bottleneck. For the prediction and update steps, the noise function dominates the runtime for larger numbers of particles. This indicates that future work could focus on improving the performance of the noise distribution code. For resampling, the Nicely algorithm dominates the runtime for lower numbers of particles, but cumsum method grows faster once $p_{\max}$ is reached.

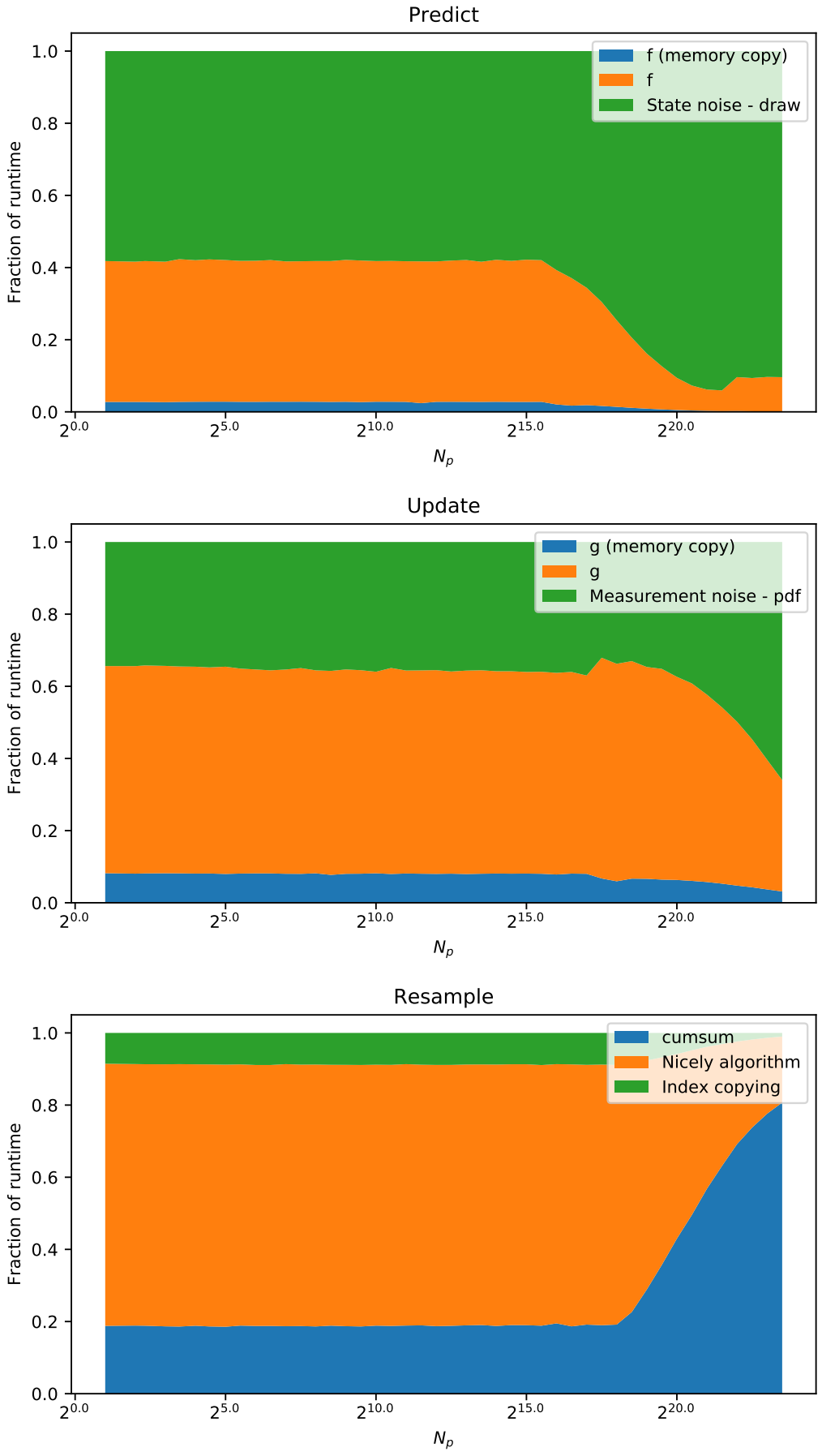

Figure 8.7: Subroutine breakdown for the GPU particle filter implementation.

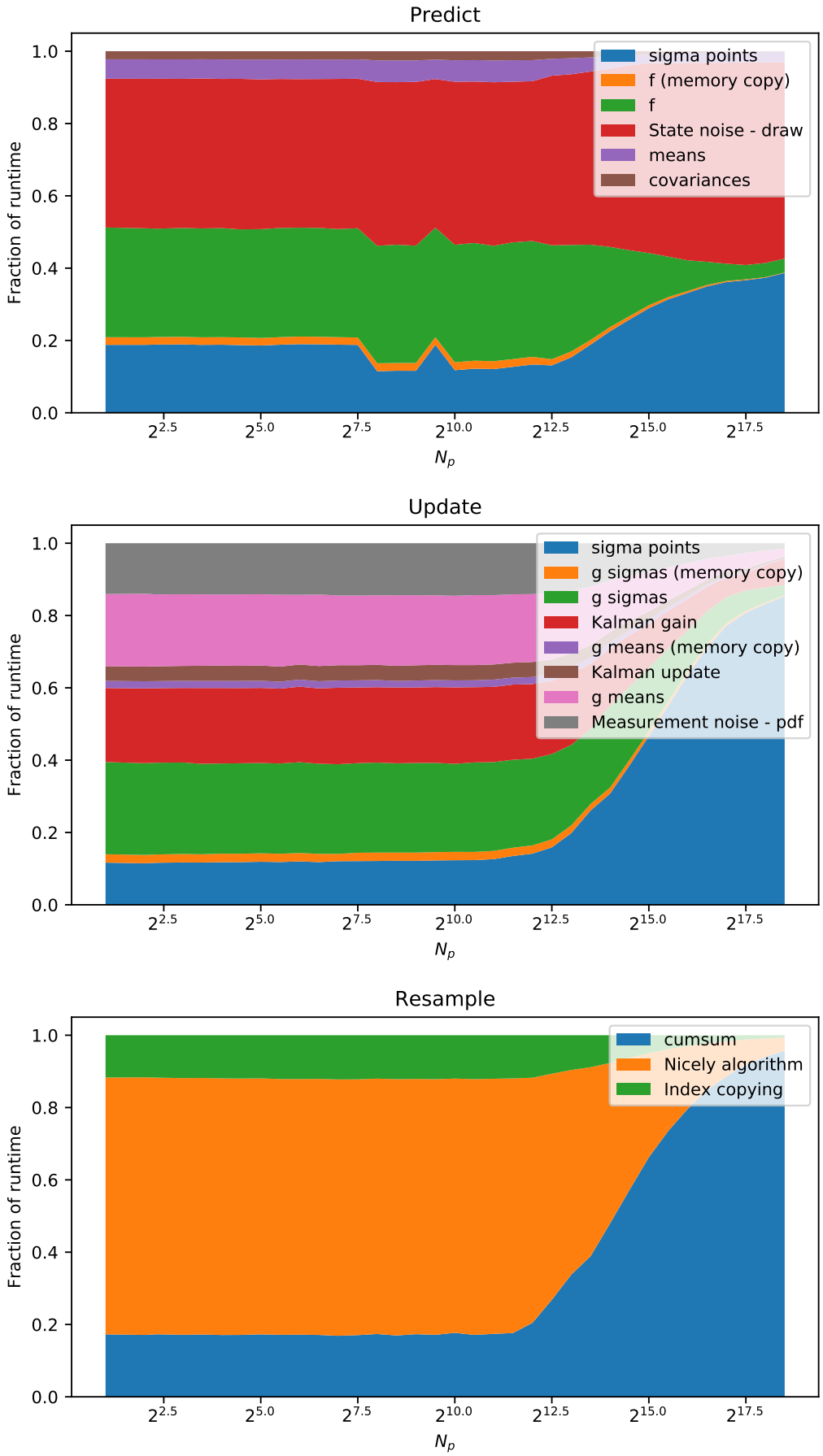

Figure 8.8: Subroutine breakdown for the GPU Gaussian sum filter implementation.

For the GSF, Figure 8.8 shows the results. It can be seen that memory copying also does not bottleneck the process. For the prediction and update steps, the noise function dominates the runtime for larger numbers of particles. This again indicates that improvement of the multivariate Gaussian distribution code should be investigated. For resampling, the Nicely algorithm dominates the runtime for lower numbers of particles, but cumsum method grows faster once $p_{\max}$ is reached. Overall, the graphs show that memory operations are not the limiting factor.

Another important aspect related to efficiency is the energy usage of the filters. The median energy used per run for each method is shown in Figure 8.9 and Figure 8.10. For the particle filter, the CPU implementation starts out using between 135 and 1240 times less energy than the GPU implementation. However, for larger numbers of particles, the GPU implementation uses between 20 and 135 times less energy. The trends on this graph are similar to the trends in Figure 8.3. This is to be expected because energy usage is calculated as $\int_0^{t_{\text{end}}} \text{power}(t)dt$, and thus a smaller t_{end} would contribute to a lower energy cost.

The results for the GSF take a similar form as seen in Figure 8.10. For each method, the CPU implementation starts out using between 61 and 885 times less power than the GPU implementation. However, for larger numbers of particle, the GPU implementation uses between 8 and 85 times less energy.

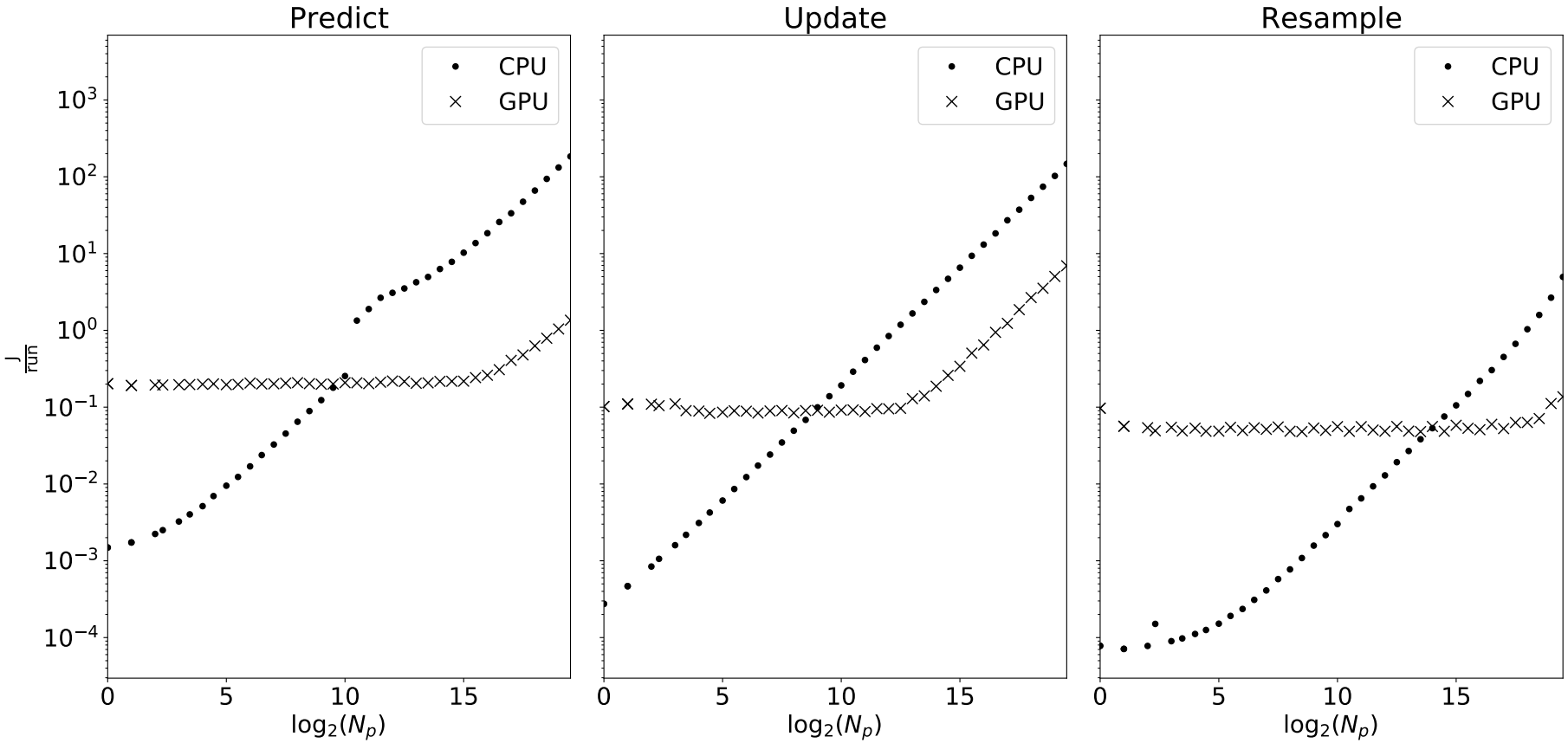

Figure 8.9: Median energy per run for particle filter methods.

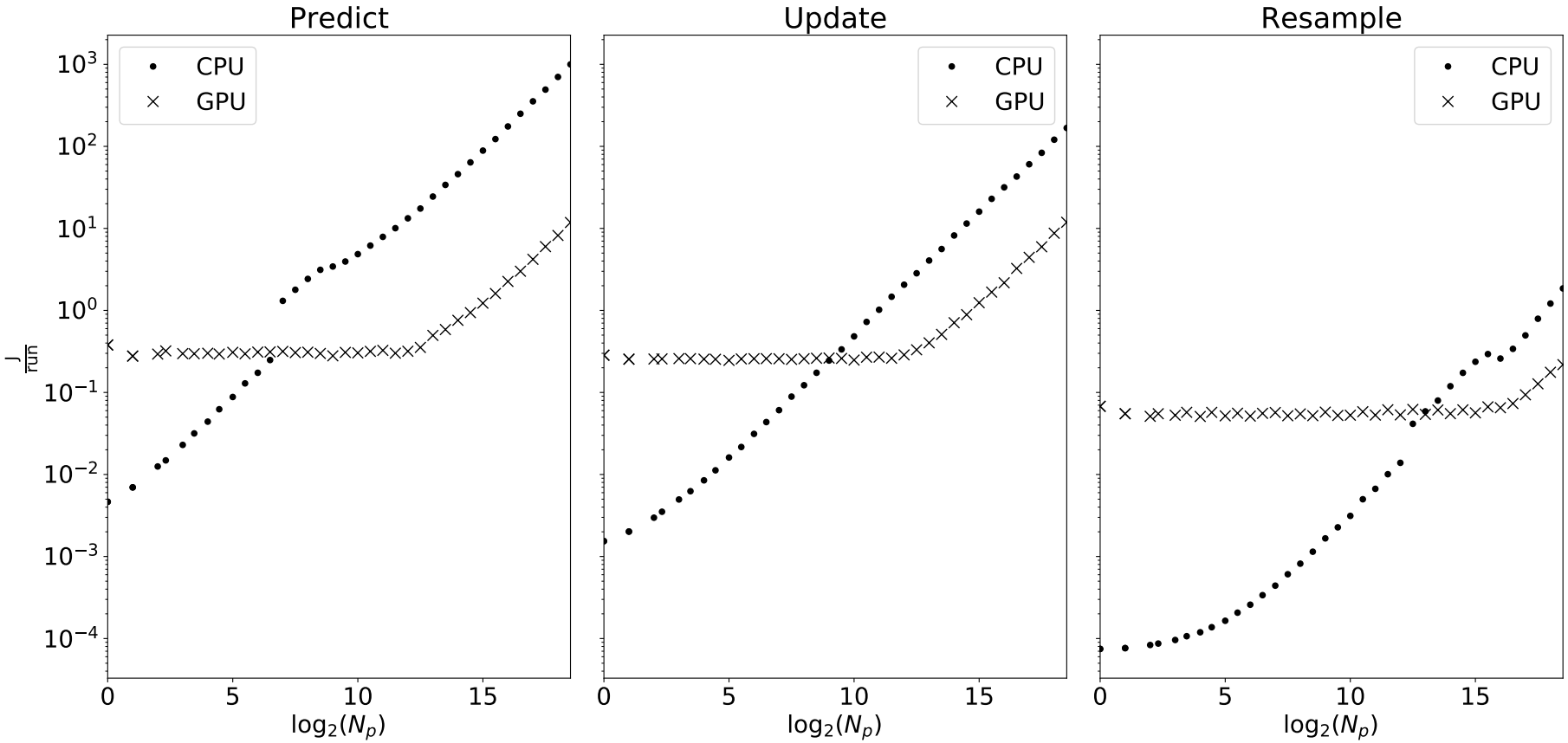

Figure 8.10: Median energy per run for Gaussian sum filter methods.

8.2 Closed loop

Closed loop simulations are performed in the set-up shown in Figure 4.1. For each closed loop simulation, the variable is the number of particles and whether the CPU or GPU filter is used. The value for the control period is taken from the time scales shown in Figure 7.1 to be 0.1 min. The utilization fraction is defined to be the ratio between the control period and the filters' run times:

$$\begin{aligned} t_{\text{sum}} &= t_{\text{predict}} + t_{\text{update}} + t_{\text{resample}} + t_{\text{resample}} \\ u &= \frac{t_{\text{sum}}}{t_{\text{control}}} \end{aligned} \tag{8.3}$$

For runs with $u > 1$, are physically unrealizable, but are shown for purposes of completeness. Figure 8.11 and Figure 8.12 plot the closed loop error versus the utilization. The red lines separate physically unrealizable runs. Note that all GPGPU accelerated runs fall within the physically realizable zone. Lowering the control period would shift the points to the right. The figures show that for the GPGPU accelerated runs, the sampling rate of a physically realizable system can be increased by at least a factor of 100. For both the particle filter and the Gaussian sum filter, the physically realizable run with the lowest error uses the GPGPU accelerated filter. Although, the CPU filters do provide similar performance.

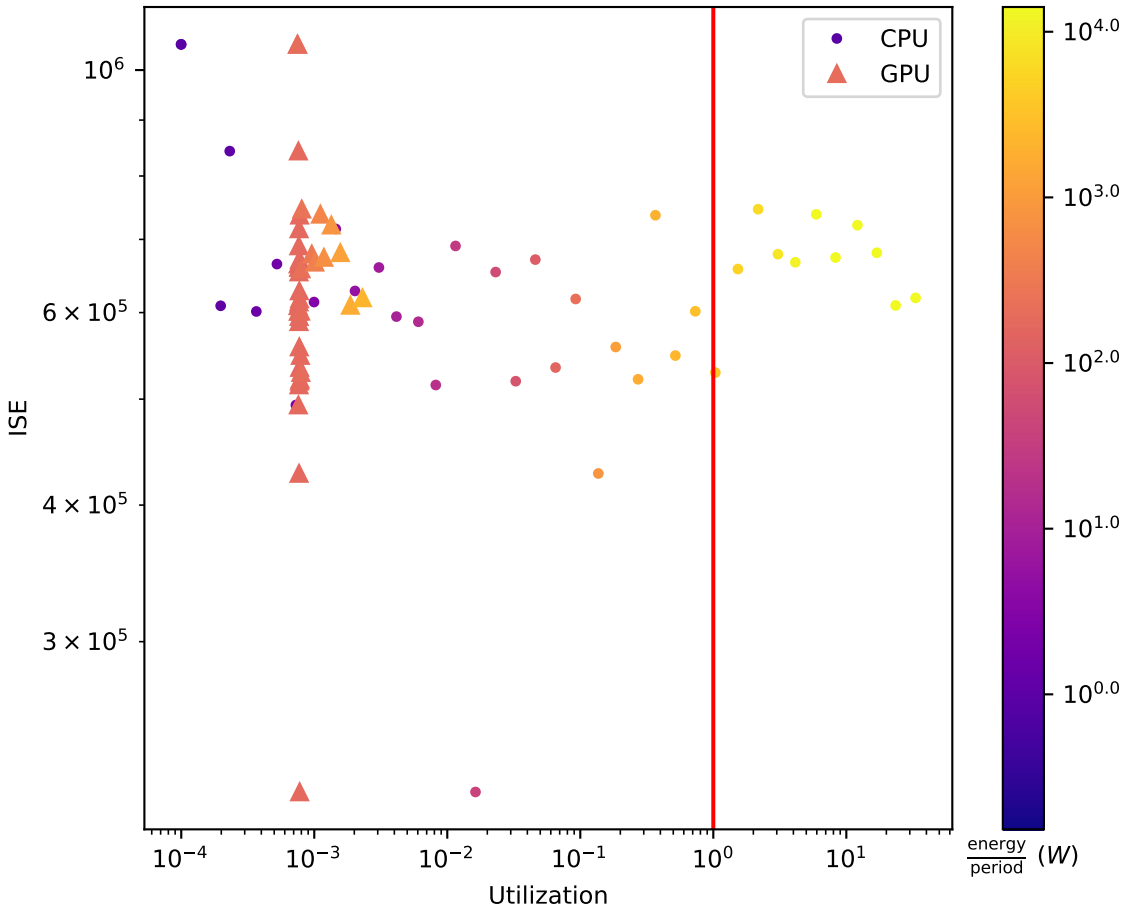

Figure 8.11: Particle filter performance versus utilization.

The colour bar adds the energy consumption axis to the figures. Both figures show a trend that increasing the energy reduces the error. Figure 8.9 and Figure 8.10 give more insight into the energy aspect of the analysis.

The clustering of many GPU points around a single utilization value is caused by the constant run times of the GPU filter methods seen in Figure 8.3 and Figure 8.4.

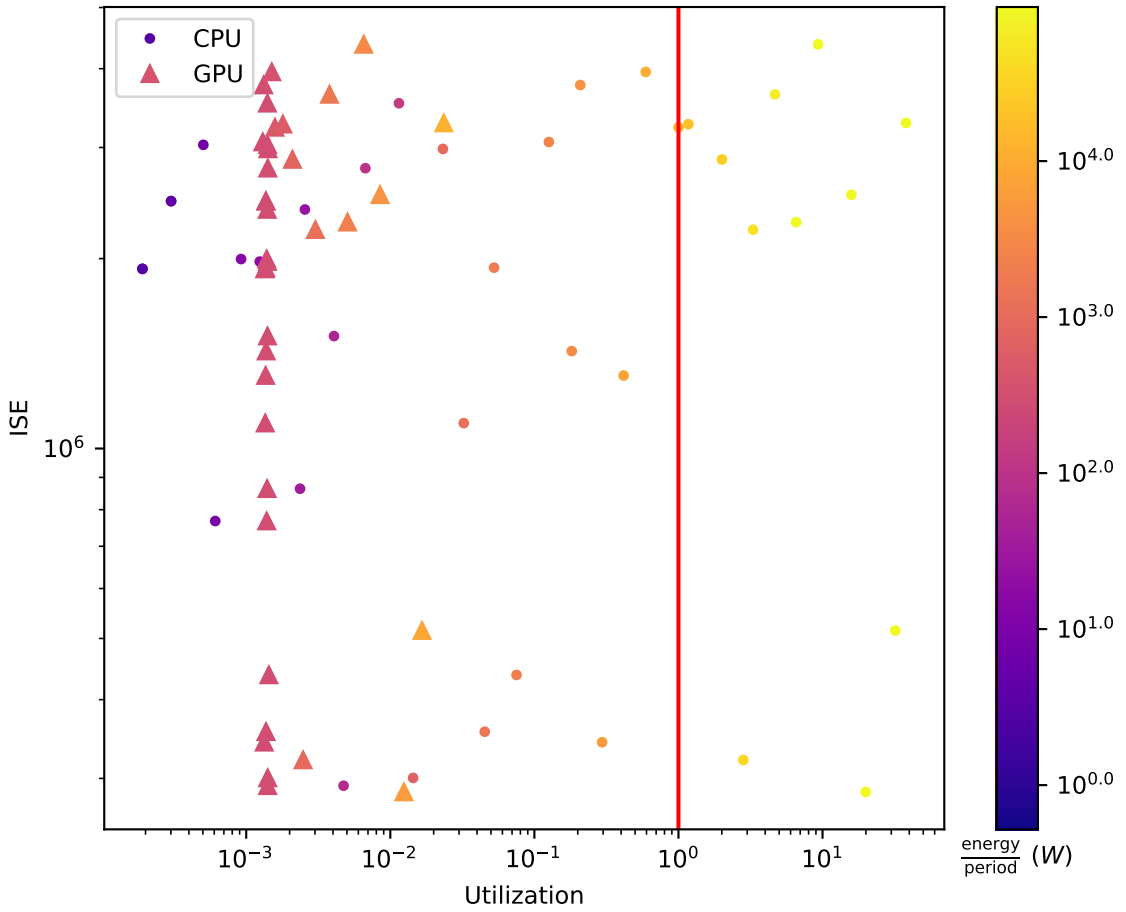

Figure 8.12: Gaussian sum filter performance versus utilization.

Error per watt plots are shown in Figure 8.13 and Figure 8.14 for the closed loop simulations.

The low energy cost of the CPU implementation for low numbers of particles contributes to the low power values as well as the moderate performance seen in the figure. The performance increases as the number of particles increases, as expected.

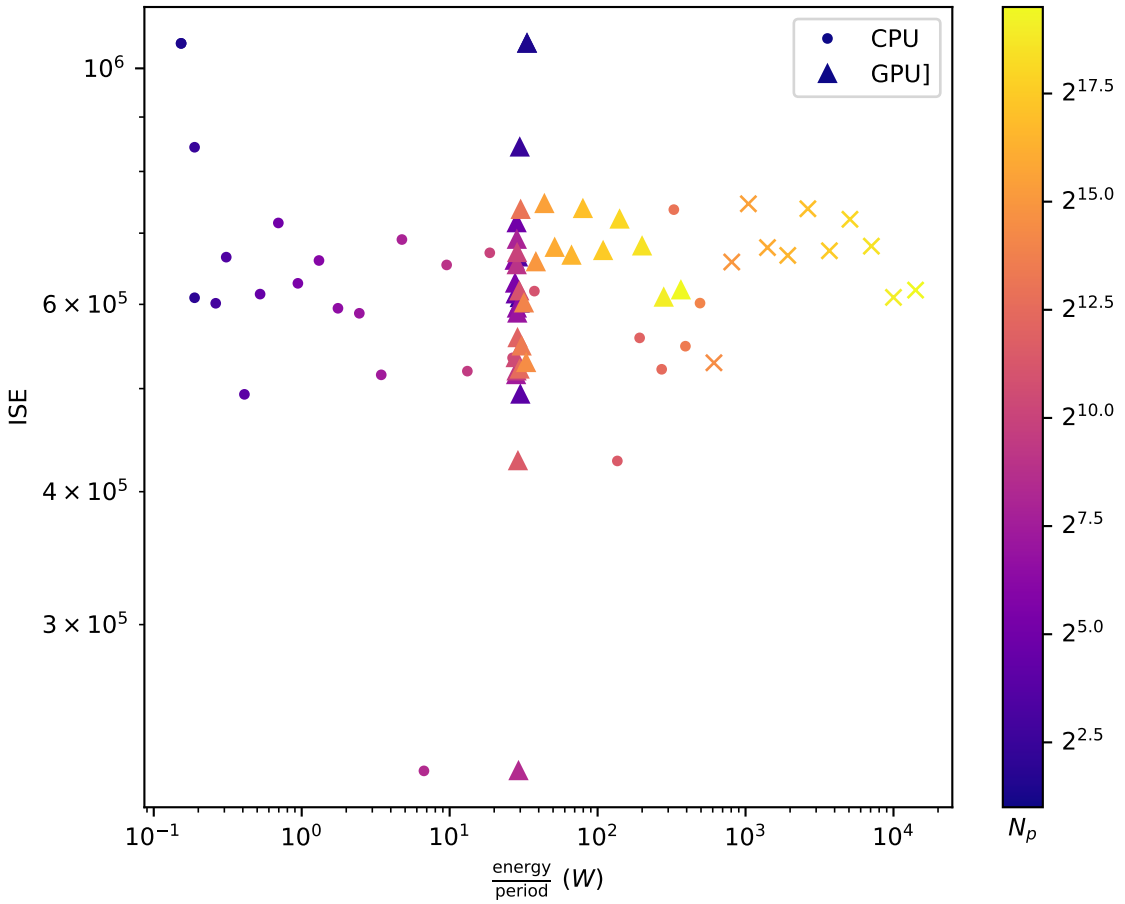

Figure 8.13: Particle filter closed loop performance versus power. Crosses indicate that $u > 1$ for a CPU run.

For the GPU implementations, the constant energy costs for low particle values, results in the clustering of power values. For higher numbers of particles, the increased performance gained by capturing more of the noise complexity, results in decreased error values.

From these results, it can be seen that the GPU implementation is more beneficial when a high sampling rate is needed. It would also find use in cases when high numbers of particles are needed for the system.

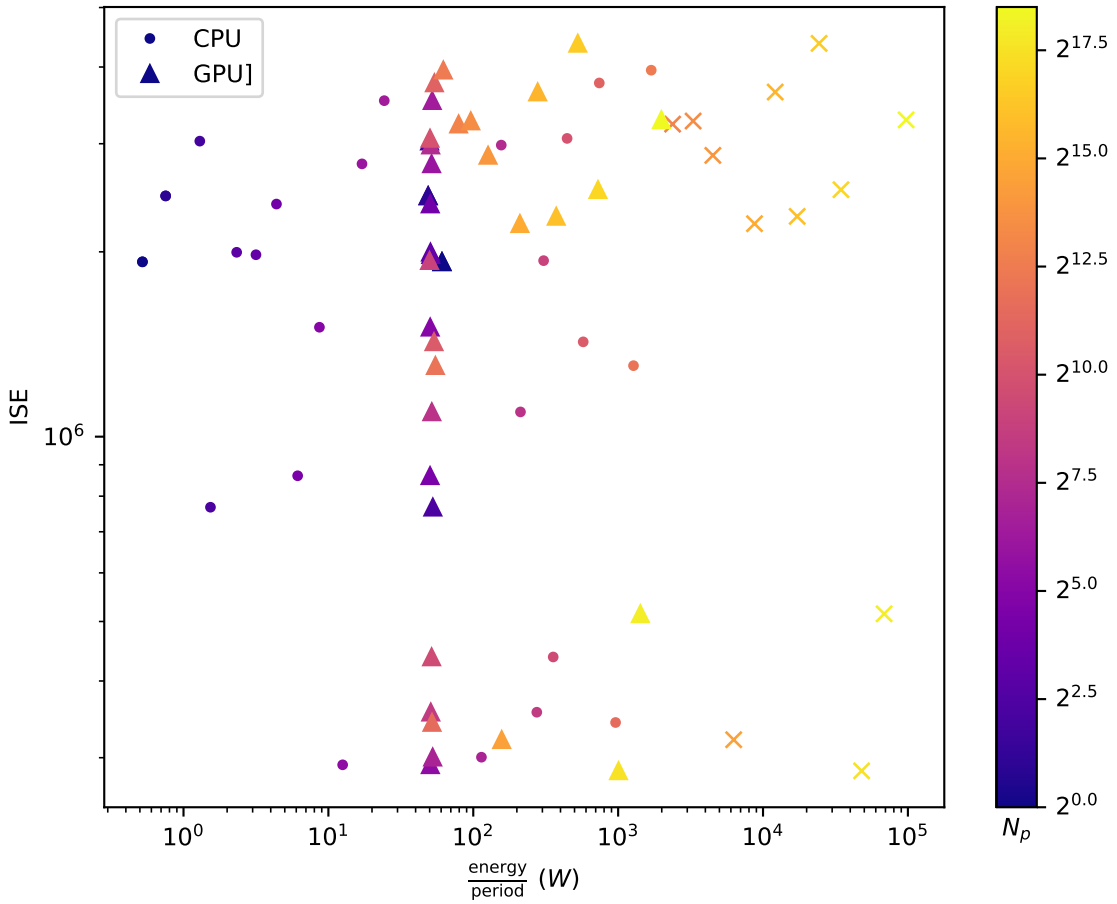

Figure 8.14: GSF closed loop performance versus power. Crosses indicate that $u > 1$ for a CPU run.

It is also important to ensure that the filter is not diverging. For this reason, the maximum singular value of the particle filter's covariance is plotted in Figure 8.15. The colour of the line indicates how many particles are in the filter.

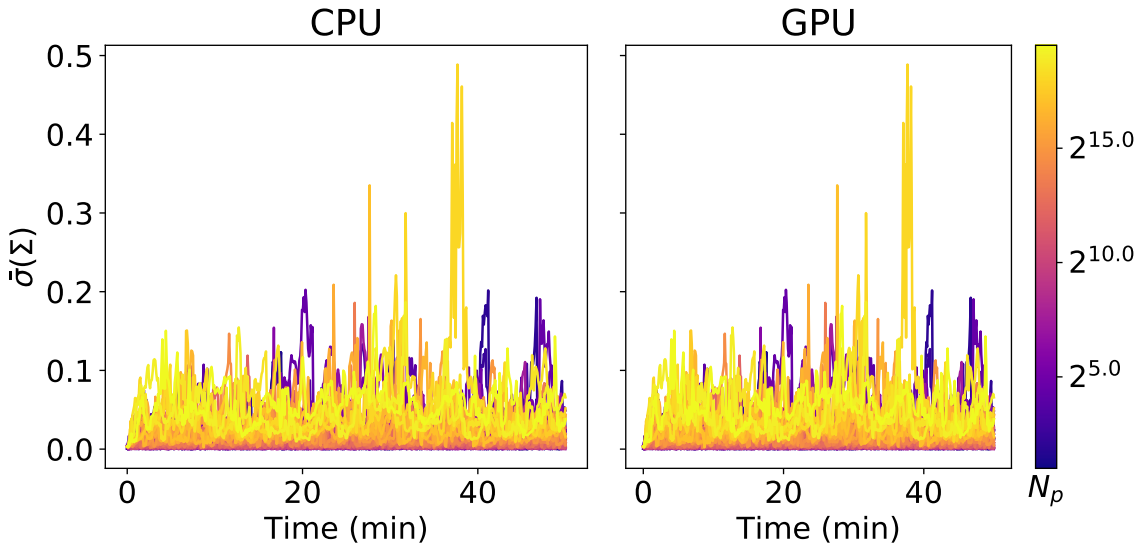

Figure 8.15: Particle filter covariance convergence.

It can be seen that for all particle filters, the covariance appears to be bounded. This indicates that the filter is not diverging. The convergences for the GSF are plotted in Figure 8.16. The covariance of a few of the filters appear to increase initially, but then quickly decrease. The reason for this is unclear and warrants further investigation. However, none of the filters appear to diverge over time, which is the desired result.

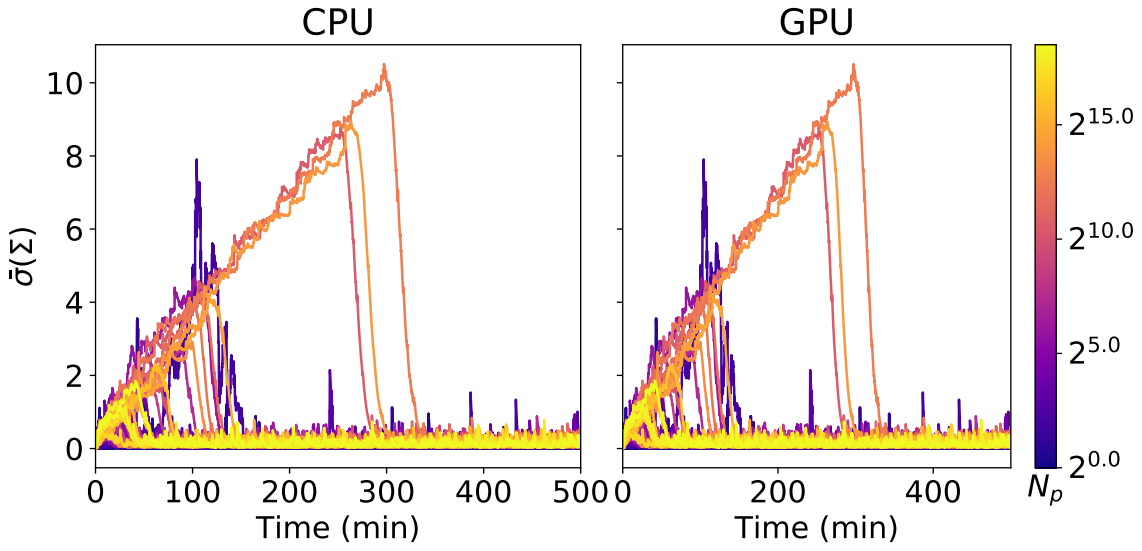

Figure 8.16: GSF covariance convergence.

9 Conclusion and future work

This section uses the results and critical discussion of the previous sections to formulate conclusive answers to the research questions posed in Section 1.3. It also critically analyses these conclusions and gives suggestions for future work.

9.1 Conclusion

The first research questions asks where GPGPU could be used to improve performance of filtering algorithms. It is found that there are several places where the particle and Gaussian sum filters are ideal candidates for such improvements due to the parallel nature of the individual particles. This is detailed in Section 3.3 and Chapter 6. Section 6.1 and Section 6.2 shows how the filters' predict and update steps are embarrassingly parallelizable, and that the resample step can be made work efficient by using a resampling algorithm from Nicely and Wells (2019). Figure 6.1 and Figure 6.2 show a summary of where GPGPU can be used to improve performance.

The second research question seeks to determine how much performance/efficiency is improved by using GPGPU on the filters. Section 8.1 shows the GPGPU particle filter is faster in the prediction, update and resampling steps for filters with more than 2^3, $2^{4.5}$, and 2^{10} particles, respectively. Similarly, the GPGPU Gaussian sum filter is faster in the prediction, update and resampling steps for filters with more than $2^{2.5}$, 2^5, and 2^{10} particles, respectively.

The run times for both filters also indicate that the GPGPU filters can be used for real time systems with fast dynamics. It is also found that, for higher numbers of particles, the GPGPU filters are more energy efficient. These results for the particle and Gaussian sum filters show an improvement to the efficiency and a performance increase for the accelerated filters.

The third research question aims to find out what effect the aforementioned efficiency

improvement has on a modelled bioreactor system. It is seen that it can improve the performance of the bioreactor control. This is seen in Figure 8.11 and Figure 8.12 where the GPGPU filters offer the lowest realizable error. Figure 8.13 and Figure 8.14 indicate a trade off between using the CPU implementation with a lower energy cost to get lower performance, or using the GPGPU implementation with a higher energy cost to get better performance.

These results lead to the conclusion that the GPGPU implementation should be used for applications that have fast dynamics and require more particles (i.e. more complex noise or dynamics).

9.2 Future work

Future work should be aimed at strengthening and furthering the results found in this work, as well as investigating situations where the delimitations used here are removed.

Investigation into systems with faster dynamics would reveal the extent of the performance improvements the speed-up of the GPGPU accelerated filters offers. A formalized investigation into the effects of noise complexity on that system would allow better understanding of the types and complexity that can be handled in real time.

Different non-linear filters can be investigated for possible GPGPU improvement. The results from different filters should be compared in terms of which gives better performance with respect to computational resources.

Similarly, a nonlinear MPC implementation might be useful to investigate. The use of NMPC with linear and nonlinear filters might yield insights into the best way to optimize computational resources. For instance, it might be found that NMPC offers similar performance as MPC with a filter, but it might use less or more energy to do so.

References

ABB (2019). *Model predictive control technology demystified*. URL: https://new.abb.com/control-systems/features/model-predictive-control-mpc.

Amdahl, GM (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

Arasarathnam, I, S Haykin, and R Elliott (2007). *Discrete-time nonlinear filtering algorithms using Gauss–Hermite quadrature*.

Barcharoglou, A (2010). In: *Approximation of probability distributions by convex mixtures of Gaussian measures*. Vol. 138. 7, pp. 2619–2628.

Bishop (2013). *Pattern Recognition and Machine Learning*. Information science and statistics. Springer (India) Private Limited. ISBN: 9788132209065. URL: https://books.google.co.za/books?id=HL4HrgEACAAJ.

Blitzstein, J and J Hwang (2014). *Introduction to Probability*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press. ISBN: 9781466575592. URL: https://books.google.co.za/books?id=z2POBQAAQBAJ.

Bogachev, V (2007). *Measure Theory*. Measure Theory v. 1. Springer Berlin Heidelberg. ISBN: 9783540345145. URL: https://books.google.co.za/books?id=CoSIe7h5mTsC.

Chartrand, G and P Zhang (2012). *A First Course in Graph Theory*. Dover books on mathematics. Dover Publications. ISBN: 9780486483689. URL: https://books.google.co.za/books?id=ocIr0RHyI8oC.

Chitchian, M, AS van Amesfoort, A Simonetto, T Keviczky, and HJ Sips (May 2013). "Adapting Particle Filter Algorithms to Many-Core Architectures". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 427–438. DOI: 10.1109/IPDPS.2013.88.

Cisneros-Magana, R, A Medina, and V Dinavahi (Sept. 2013). "Parallel Kalman filter based time-domain harmonic state estimation". In: *2013 North American Power Symposium (NAPS)*, pp. 1–6. DOI: 10.1109/NAPS.2013.6666831.

Cisneros-Magana, R, A Medina, V Dinavahi, and A Ramos-Paz (2018). "Time-Domain Power Quality State Estimation Based on Kalman Filter Using Parallel Computing on Graphics Processing Units". In: *IEEE Access* 6, pp. 21152–21163. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2823721.

Cormen, T, C Leiserson, R Rivest, and C Stein (2009). *Introduction to Algorithms*. Computer science. MIT Press. ISBN: 9780262033848. URL: https://books.google.co.za/books?id=i-bUBQAAQBAJ.

Debnath, JK, W Fung, AM Gole, and S Filizadeh (Oct. 2011). "Simulation of large-scale electrical power networks on graphics processing units". In: *2011 IEEE Electrical Power and Energy Conference*, pp. 199–204. DOI: 10.1109/EPEC.2011.6070195.

Durrett, R (1996). *Probability: Theory and Examples*. The Wadsworth & Brooks/Cole Statistics/Probability Series. Duxbury Press. ISBN: 9780534243180. URL: https://books.google.co.za/books?id=kkc%5C_AQAAIAAJ.

Durrett, R (2012). *Essentials of Stochastic Processes*. Springer Texts in Statistics. Springer New York. ISBN: 9781461436157. URL: https://books.google.co.za/books?id=i%5C_Ovy6OvI54C.

Edgar, T, D Himmelblau, and L Lasdon (2001). *Optimization of chemical processes*. McGraw-Hill chemical engineering series. McGraw-Hill. ISBN: 9780070393592. URL: https://books.google.co.za/books?id=PqBTAAAAMAAJ.

Flemming, W and M Sonner (2006). "Controlled Markov Processes and Viscosity Solutions". In: *Springer*.

Flynn, M (1972). "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers*.

Folland, G (1999). *Real Analysis: Modern Techniques and Their Applications*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. Wiley. ISBN: 9780471317166. URL: https://books.google.co.za/books?id=N8jVDwAAQBAJ.

Gallager, R (2013). *Stochastic Processes: Theory for Applications*. Stochastic Processes: Theory for Applications. Cambridge University Press. ISBN: 9781107039759. URL: https://books.google.co.za/books?id=ERLrAQAAQBAJ.

Garcia, A, A Monticelli, and P Abreu (Sept. 1979). "Fast Decoupled State Estimation and Bad Data Processing". In: *IEEE Transactions on Power Apparatus and Systems* PAS-98.5, pp. 1645–1652. ISSN: 0018-9510. DOI: 10.1109/TPAS.1979.319482.

Gee, LM, S Schmidt, and G Smith (1962). *Application of statistical filter theory to the optimal estimation of position and velocity on board a circumlunar vehicle*. Tech. rep. NASA.

Gol, M and A Abur (Sept. 2015). "A Fast Decoupled State Estimator for Systems Measured by PMUs". In: *IEEE Transactions on Power Systems* 30.5, pp. 2766–2771. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2014.2365759.

Gomez-Exposito, A, A Abur, A de la Villa Jaen, and C Gomez-Quiles (June 2011). "A Multilevel State Estimation Paradigm for Smart Grids". In: *Proceedings of the IEEE* 99.6, pp. 952–976. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2011.2107490. URL: http://ieeexplore.ieee.org/document/5756670/ (visited on 02/15/2020).

Gomez-Quiles, C, A Gomez-Exposito, and A de la Villa Jaen (June 2012). "State Estimation for Smart Distribution Substations". In: *IEEE Transactions on Smart Grid* 3.2, pp. 986–995. ISSN: 1949-3053, 1949-3061. DOI: 10.1109/TSG.2012.2189140. URL: http://ieeexplore.ieee.org/document/6184355/ (visited on 04/02/2020).

Gordon, NJ, DJ Salmond, and AFM Smith (Apr. 1993). "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". In: *IEE Proceedings F - Radar and Signal Processing* 140.2, pp. 107–113. ISSN: 0956-375X. DOI: 10.1049/ip-f-2.1993.0015.

Guo, Y, W Wu, B Zhang, and H Sun (Aug. 2013). "An Efficient State Estimation Algorithm Considering Zero Injection Constraints". In: *IEEE Transactions on Power Systems* 28.3, pp. 2651–2659. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2012.2232316.

Hanebeck, UD (2013). "Progressive Gaussian Filtering with a Twist". In: *Proceedings of the 16th International Conference on Information Fusion (Fusion 2013)*.

Hendeby, G, R Karlsson, and F Gustafsson (2010). "Particle Filtering: The Need for Speed". In: *Journal on Advances in Signal Processing vol. 2010 Feb.*

Hol, J, T Schön, and F Gustafsson (2004). "On Resampling Algorithms for Particle Filters". In: DOI: 10.1109/NSSPW.2006.4378824.

Hsieh, MF and J Wang (June 2010a). "An extended Kalman filter for ammonia coverage ratio and capacity estimations in the application of Diesel engine SCR control and onboard diagnosis". In: *Proceedings of the 2010 American Control Conference*, pp. 5874–5879. DOI: 10.1109/ACC.2010.5530516.

Hsieh, MF and J Wang (June 2010b). "An extended Kalman filter for NOx sensor ammonia cross-sensitivity elimination in selective catalytic reduction applications". In: *Proceedings of the 2010 American Control Conference*, pp. 3033–3038. DOI: 10.1109/ACC.2010.5531217.

Huang, M, S Wei, B Huang, and Y Chang (Dec. 2011). "Accelerating the Kalman Filter on a GPU". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 1016–1020. DOI: 10.1109/ICPADS.2011.153.

Ikoma, N (Aug. 2014). "On GPGPU parallel implementation of hands and arms motion estimation of a car driver with depth image sensor by particle filter". In: *2014 World Automation Congress (WAC)*, pp. 246–251. DOI: 10.1109/WAC.2014.6935867.

Iplik, E (2017). "Modelling and Control the Production Process of Saccharomyces Cerevisiae". University of Dortmund.

Jalili-Marandi, V and V Dinavahi (Aug. 2010). "SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit". In: *IEEE Transactions on Power Systems* 25.3, pp. 1589–1599. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2010.2042084. URL: http://ieeexplore.ieee.org/document/5422731/ (visited on 02/15/2020).

Jaynes, E, E Jaynes, G Bretthorst, and CU Press (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. ISBN: 9780521592710. URL: https://books.google.co.za/books?id=tTN4HuUNXjgC.

Julier, S and J Uhlmann (2004). "Unscented filtering and nonlinear estimation". In: *Proceedings of the IEEE*.

Kalman, R (1960). "A new approach to linear filtering and prediction problems". In: *Journal of Basic Engineering*.

Karimipour, H and V Dinavahi (2015). "Extended Kalman Filter-Based Parallel Dynamic State Estimation". In: *IEEE Transactions on Smart Grid* 6.3, pp. 1539–1549.

Kekatos, V and GB Giannakis (May 2013). "Distributed Robust Power System State Estimation". In: *IEEE Transactions on Power Systems* 28.2, pp. 1617–1626. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2012.2219629. URL: http://ieeexplore.ieee.org/document/6340375/ (visited on 02/15/2020).

Korres, GN (Feb. 2011). "A Distributed Multiarea State Estimation". In: *IEEE Transactions on Power Systems* 26.1, pp. 73–84. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2010.2047030. URL: http://ieeexplore.ieee.org/document/5454425/ (visited on 02/15/2020).

Koski, T and J Noble (2011). *Bayesian Networks: An Introduction*. Wiley Series in Probability and Statistics. Wiley. ISBN: 9781119964957. URL: https://books.google.co.za/books?id=aoBzNiDPfQsC.

Kottakki, KK, M Bhushan, and S Bhartiya (2014). "An Improved Gaussian Sum Unscented Kalman Filter". In: *IFAC Proceedings Volumes* 47.1. 3rd International Conference on Advances in Control and Optimization of Dynamical Systems (2014), pp. 355–362. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20140313-3-IN-3024.00056. URL: http://www.sciencedirect.com/science/article/pii/S1474667016326805.

Kouvaritakis, B and M Cannon (2015). *Model Predictive Control: Classical, Robust and Stochastic*. Advanced Textbooks in Control and Signal Processing. Springer International Publishing. ISBN: 9783319248530. URL: https://books.google.co.za/books?id=DmoiCwAAQBAJ.

Kruskal, C, L Rudolph, and M Snir (1990). "A complexity theory of efficient parallel algorithms". In: *Theoretical computer science* 71, pp. 95–132.

Lozenguez, G (2016). *Dynamic Bayesian network*. URL: https://en.wikipedia.org/wiki/Dynamic_Bayesian_network.

Lundqvist, K (2013). *Big O Notation*. URL: https://web.mit.edu/16.070/www/lecture/big_o.pdf.

Mairet, F (Feb. 2018). "A biomolecular proportional integral controller based on feedback regulations of protein level and activity". eng. In: *Royal Society open science* 5.2. rsos171966[PII], pp. 171966–171966. ISSN: 2054-5703. DOI: 10.1098/rsos.171966. URL: https://doi.org/10.1098/rsos.171966.

Minot, A, Yue Lu, and Na Li (July 2016). "A distributed Gauss-Newton method for power system state estimation". In: *2016 IEEE Power and Energy Society General Meeting (PESGM)*. Boston, MA, USA: IEEE, pp. 1–1. ISBN: 9781509041688. DOI: 10.1109/PESGM.2016.7741288. URL: http://ieeexplore.ieee.org/document/7741288/ (visited on 02/15/2020).

Namrata, V (2005). "Least Squares and Kalman Filtering". Iowa State University (unpublished lecture slides).

Neumann, J von (Oct. 1945). "First Draft of a Report on the EDVAC". In: *IEEE Ann. Hist. Comput.* 15.4, pp. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389. URL: https://doi.org/10.1109/85.238389.

Nicely, MA and BE Wells (2019). "Improved Parallel Resampling Methods for Particle Filtering". In: *IEEE Access* 7, pp. 47593–47604. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2910163.

Nvidia (2015). *CUDA toolkit documentation: cuRAND*. URL: http://docs.nvidia.com/cuda/curand/.

Nvidia (2019). *CUDA toolkit documentation: Programming model*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

Papoulis, A (1965). *Probability, Random Variables, and Stochastic Processes*. International student edition. McGraw-Hill. URL: https://books.google.co.za/books?id=ldRQAAAAMAAJ.

Plataniotis, K, D Androutsos, and A Venetsanopoulos (June 1997). "Nonlinear Filtering of Non-Gaussian Noise". In: *Journal of Intelligent and Robotic Systems* 19, pp. 207–231. DOI: 10.1023/A:1007974400149.

Rahman, MA and GK Venayagamoorthy (Mar. 2016). "Dishonest Gauss Newton method based power system state estimation on a GPU". In: *2016 Clemson University Power Systems Conference (PSC)*, pp. 1–6. DOI: 10.1109/PSC.2016.7462826.

Rawlings, JB and DQ Mayne (2009). *Model predictive control: Theory and design*. Nob Hill Pub.

Ristic, B, S Arulampalam, and N Gordon (2003). *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House. ISBN: 9781580538510. URL: https://books.google.se/books?id=zABIY--qk2AC.

Sanders, J and E Kandrot (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education. ISBN: 9780132180139. URL: https://books.google.co.za/books?id=49OmnOmTEtQC.

Scutari, M and J Denis (2014). *Bayesian Networks: With Examples in R*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis. ISBN: 9781482225587. URL: https://books.google.co.za/books?id=v-XMAwAAQBAJ.

Seborg, D and T Mellichamp (2006). *Process Dynamics & Control, 2nd ed*. Wiley India Pvt. Limited. ISBN: 9788126508341. URL: https://books.google.co.za/books?id=7CWQ24TKANwC.

Skogestad, S and I Postlethwaite (2005). *Multivariable Feedback Control: Analysis and Design*. Wiley. ISBN: 9780470011676. URL: https://books.google.co.za/books?id=3dxSAAAAMAAJ.

Sorenson, H and D Alspach (1971). "Recursive bayesian estimation using gaussian sums". In: *Automatica* 7.4, pp. 465–479. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(71)90097-5. URL: http://www.sciencedirect.com/science/article/pii/0005109871900975.

Sowman, J, D Laila, A Truscott, P Fussey, and A Cruden (June 2016). "Real-time rejection of ammonia cross sensitivity in sensors for diesel aftertreatment systems by parallel particle filtering". In: *2016 European Control Conference (ECC)*, pp. 1242–1247. DOI: 10.1109/ECC.2016.7810459.

Steinbring, J and UD Hanebeck (July 2015). "GPU-accelerated progressive Gaussian filtering with applications to extended object tracking". In: *2015 18th International Conference on Information Fusion (Fusion)*, pp. 1038–1045.

Stellato, B, G Banjac, P Goulart, A Bemporad, and S Boyd (Nov. 2017). "OSQP: An Operator Splitting Solver for Quadratic Programs". In: *ArXiv e-prints*. arXiv: 1711.08013 [math.OC].

Swart, A (2019). "Fumarate production with Rhizopus oryzae: utilising the Crabtree effect to minimise ethanol by-product formation". University of Pretoria.

Tao, T (2011). *An Introduction to Measure Theory*. Graduate studies in mathematics. American Mathematical Society. ISBN: 9780821869192. URL: https://books.google.co.za/books?id=HoGDAwAAQBAJ.

Trudeau, R (1993). *Introduction to Graph Theory*. Dover Books on Mathematics Series. Dover Pub. ISBN: 9780486678702. URL: https://books.google.co.za/books?id=NunuAAAAMAAJ.

Tulsyan, A, R[Gopaluni], and SR Khare (2016). "Particle filtering without tears: A primer for beginners". In: *Computers and Chemical Engineering* 95, pp. 130–145. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2016.08.015. URL: http://www.sciencedirect.com/science/article/pii/S0098135416302769.

Tuomanen, B (2018). *Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA*. Packt Publishing. ISBN: 9781788995221. URL: https://books.google.co.za/books?id=SH18DwAAQBAJ.

Veen, L van, J Morra, A Palanica, and Y Fossat (2019). *I'm a doctor, not a mathematician! Homeostasis as a proportional-integral control system*. arXiv: 1912.13148 [physics.med-ph].

Welch, G and G Bishop (2001). *An Introduction to the Kalman Filter*. ACM SIGGRAPH 2001 Course 8.

West, D (1996). *Introduction to Graph Theory*. Prentice Hall. ISBN: 9780132278287. URL: https://books.google.co.za/books?id=HuvuAAAAMAAJ.

Wilken, S (2015). “Computationally efficient formulation of stochastic dynamic control within the context of switching probabilistic graphical models”. University of Pretoria.

Xia, Y, Y Chen, Z Ren, S Huang, M Wang, and M Lin (Nov. 2017). “State estimation for large-scale power systems based on hybrid CPU-GPU platform”. In: *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pp. 1–6. DOI: 10.1109/EI2.2017.8245566.

Xie, L, DH Choi, S Kar, and HV Poor (Sept. 2012). “Fully Distributed State Estimation for Wide-Area Monitoring Systems”. In: *IEEE Transactions on Smart Grid* 3.3, pp. 1154–1169. ISSN: 1949-3053, 1949-3061. DOI: 10.1109/TSG.2012.2197764. URL: http://ieeexplore.ieee.org/document/6205641/ (visited on 04/02/2020).

Xiong, L and S Grijalva (July 2016). “Fast decomposition-based state estimation using automatic graph partitioning and ADMM”. In: *2016 IEEE Power and Energy Society General Meeting (PESGM)*. Boston, MA, USA: IEEE, pp. 1–5. ISBN: 9781509041688. DOI: 10.1109/PESGM.2016.7741370. URL: http://ieeexplore.ieee.org/document/7741370/ (visited on 02/15/2020).

Xu, D, Z Xiao, D Li, and F Wu (Aug. 2016). “Optimization of parallel algorithm for Kalman filter on CPU-GPU heterogeneous system”. In: *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pp. 2165–2172. DOI: 10.1109/FSKD.2016.7603516.

Yang, X, G Liu, A Li, and L Van Dai (July 2017). “A Predictive Power Control Strategy for DFIGs Based on a Wind Energy Converter System”. In: *Energies* 10, p. 1098. DOI: 10.3390/en10081098.

Yang, X, C Zhao, and B Chen (2019). “Progressive Gaussian approximation filter with adaptive measurement update”. In: *Measurement* 148, p. 106898. ISSN: 0263-2241. DOI: https://doi.org/10.1016/j.measurement.2019.106898. URL: http://www.sciencedirect.com/science/article/pii/S0263224119307559.

Zhang, H, J Wang, and YY Wang (2015). “Removal of NOx sensor ammonia cross sensitivity from contaminated measurements in Diesel-engine selective catalytic reduction systems”. In: *Fuel* 150, pp. 448–456. ISSN: 0016-2361. DOI: https://doi.org/10.1016/j.fuel.2015.02.053. URL: http://www.sciencedirect.com/science/article/pii/S001623611500201X.

Zhang, L, J Sturm, D Cremers, and D Lee (Oct. 2012). "Real-time human motion tracking using multiple depth cameras". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2389–2395. DOI: 10.1109/IROS.2012.6385968.

Zhaofeng Bai and Yuehong Qiu (Oct. 2015). "A GPU-based implementation of PKF for INS/CNS integrated Navigation System". In: *2015 IEEE 6th International Symposium on Microwave, Antenna, Propagation, and EMC Technologies (MAPE)*, pp. 737–742. DOI: 10.1109/MAPE.2015.7510423.

Zheng, W, W Wu, A Gomez-Exposito, B Zhang, and Y Guo (Jan. 2017). "Distributed Robust Bilinear State Estimation for Power Systems with Nonlinear Measurements". In: *IEEE Transactions on Power Systems* 32.1, pp. 499–509. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2016.2555793. URL: http://ieeexplore.ieee.org/document/7458119/ (visited on 02/15/2020).