A Study of Adaptation Mechanisms for Simulation Algorithms

by

Rodolfo G. Esteves Jaramillo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

© Rodolfo G. Esteves Jaramillo 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The performance of a program can sometimes greatly improve if it was known in advance the features of the input the program is supposed to process, the actual operating parameters it is supposed to work with, or the specific environment it is to run on. However, this information is typically not available until too late in the program's operation to take advantage of it. This is especially true for simulation algorithms, which are sensitive to this late-arriving information, and whose role in the solution of decision-making, inference and valuation problems is crucial.

To overcome this limitation we need to provide the flexibility for a program to adapt its behaviour to late-arriving information once it becomes available. In this thesis, I study three adaptation mechanisms: run-time code generation, model-specific (quasi) Monte Carlo sampling and dynamic computation offloading, and evaluate their benefits on Monte Carlo algorithms. First, run-time code generation is studied in the context of Monte Carlo algorithms for time-series filtering in the form of the Input-Adaptive Kalman filter, a dynamically generated state estimator for non-linear, non-Gaussian dynamic systems. The second adaptation mechanism consists of the application of the functional-ANOVA decomposition to generate model-specific QMC-samplers which can then be used to improve Monte Carlo-based integration. The third adaptive mechanism treated here, dynamic computation offloading, is applied to wireless communication management, where network conditions are assessed via option valuation techniques to determine whether a program should offload computations or carry them out locally in order to achieve higher run-time (and correspondingly battery-usage) efficiency. This ability makes the program well suited for operation in mobile environments.

At their core, all these applications carry out or make use of (quasi) Monte Carlo simulations on dynamic Bayesian networks (DBNs). The DBN formalism and its associated simulation-based algorithms are of great value in the solution to problems with a large uncertainty component. This characteristic makes adaptation techniques like those studied here likely to gain relevance in a world where computers are endowed with perception

capabilities and are expected to deal with an ever-increasing stream of sensor and time-series data.

Acknowledgements

It is impossible for me to acknowledge, even in a passing manner, the full set of people who have in one way or another contributed to the work presented in this document. In the long time that it has taken for this work to reach its present shape, a great many people have touched it and it is difficult to find a way in which the work is not the better for it. However, there are a number of influences that were *conditio sine qua non* for this thesis to reach its final form. First and foremost amongst them are my two supervisors, Professors Christiane Lemieux and Michael McCool. I cannot begin to express how much both this work and my life in general owes to their dedication, support, patience and encouragement. Their guidance, both academic and personal has been unfailing, and their support and friendship has been crucial for me during this time. In the academic front, I would also like to thank my Committee, Professors Alan Genz, Pascal Poupart, Hans de Sterck and Paul Fieguth for their helpful comments and suggestions, as the thesis has greatly benefited from them. I would like to give special thanks to my external Committee member, Prof. Alan Genz, for agreeing to be part of it. Similarly, I would be remiss not to thank the many opportunities and much kindness of the staff at the School of Computer Science in the University of Waterloo.

In this thesis, and in life in general, I cannot give enough thanks to my wife, Lijun, whose support, grace, patience and encouragement serves as the engine for everything I do. If I live a million lives, in each and every one of them I will be indebted to her. Similarly unfailing in their love and care are my mother, Elvira, and my sisters, Alejandra and Claudia, to whom I am eternally grateful. I would also like to thank my many friends in Waterloo, but especially Zheng, Will and Janet, on whom I can always count.

Dedication

To my father, Rodolfo Esteves Fuentes, *in memoriam*, who would have liked nothing more than to see this work completed.

To my wife, Lijun Wang, because everything I do makes sense if I think of her when doing it.

To my mother, Elvira, who has been with me every step of the way, never letting me fall or remain down.

Table of Contents

List of Tables **xi**

List of Figures **xiv**

1 Introduction **1**

1.1 Problem statement and motivation 1

1.2 Contributions of this work 2

1.3 Thesis organization and bibliographic notes 4

2 Bayesian models and Monte Carlo methods **7**

2.1 Dynamic Bayesian networks 10

2.2 Modeling with DBNs 15

2.2.1 Solution to Stochastic Differential Equations 15

2.2.2 State space models 18

2.3 Monte Carlo for inference and simulation 19

2.3.1 Simulation of Stochastic Differential Equations 21

2.3.2 Model-based inference on Dynamical Systems 22

2.4 Issues concerning Monte Carlo algorithms 25
2.4.1 Algorithmic complexity and estimator quality 25
2.4.2 Variance reduction 26
2.5 Quasi Monte Carlo methods 27
2.5.1 Discrepancy and low-discrepancy sequences 28
2.5.2 Constructions 30
2.5.3 Randomizations 33

3 Adaptation via Code Generation and its Application to Filtering 35

3.1 Motivations and Central Idea 36
3.2 Code Generation for Domain-Specific Adaptation 39
3.2.1 Traditional Run-time Code Generation 40
3.2.2 Run-time Code Generation Infrastructure 43
3.2.3 Domain-specific User-level Guest Code Manipulation 44
3.3 Non-Linear State Estimation with the Kalman Filter 48
3.3.1 Generalizations of the KF 51
3.3.2 Dealing with non-linearities 52
3.3.3 Dealing with non-Gaussianity 53
3.4 The Input-Adaptive Kalman Filter 54
3.4.1 Clustering and model selection for the IAKF 55
3.4.2 Run-time code generation of the IAKF 58
3.5 Numerical Evaluation 64
3.5.1 Empirical performance analysis of the IAKF 66

3.5.2 Limitations of the IAKF . . . 75
3.6 Other Approaches to KF Code Generation . . . 75
3.7 Summary . . . 76

4 Functional ANOVA decomposition for QMC algorithms on DBNs 77

4.1 Introduction . . . 78
4.2 QMC for Simulation and Inference . . . 80
4.3 Sensitivity Analysis . . . 81
4.3.1 The functional ANOVA decomposition . . . 82
4.3.2 Quasi-regression . . . 84
4.3.3 The effective dimension . . . 86
4.3.4 Choosing direction numbers . . . 87
4.4 Functional ANOVA-Based QMC Algorithms . . . 89
4.4.1 General Design . . . 89
4.4.2 Implementation . . . 90
4.5 Evaluation . . . 92
4.6 Summary and Future Work . . . 100

5 Real options for mobile communication management 102

5.1 Introduction . . . 103
5.2 Real Options Analysis . . . 105
5.2.1 Real options valuation . . . 106
5.2.2 LSM for switching options . . . 108
5.3 Problem Setup . . . 109

5.3.1 Utility function . . . 110
5.3.2 Estimating transmission costs . . . 112
5.3.3 An alternative approach to the offloading problem: MDP . . . 113
5.4 System Architecture and Evaluation . . . 115
5.4.1 System architecture . . . 115
5.4.2 Evaluation . . . 118
5.5 Related Work . . . 123
5.6 Summary and Future Work . . . 124

6 Conclusions and future work **126**
6.1 Conclusions . . . 126
6.2 Future work . . . 128

References **146**

List of Tables

2.1 Posterior distributions of interest 23

3.1 Speedups of loop-unrolled w.r.t. loop-based pure C++ IAKFs 72

3.2 Speedups of loop-unrolled w.r.t. loop-based Python/C++ IAKFs 73

3.3 Comparative performance of “fixed” vs generated-on-the-fly GSF-QMC-KF 74

4.1 Different notions of effective dimension [90] 86

5.1 Correspondence of financial options, real options and applications to offload 106

5.2 Application of ROA to computation offloading 110

5.3 Summary of the experimental results 120

List of Figures

1.1 DBN-based software construction 4

2.1 State space representation of a dynamical system 12

2.2 The family of dynamic Bayesian networks [110] 13

2.3 Two more structured Bayesian Networks 14

2.4 DBN representation of a stock price 16

3.1 Alternative-based adaptation in domain-specific libraries. 37

3.2 The "code factory" approach to adaptation in domain-specific libraries. . 38

3.3 Two-stage DSL processing 46

3.4 Conceptual diagram of the IAKF 57

3.5 Non-linear functions f and h for track fitting 65

3.6 Magnitude of the filtering error of the UKF, MCKF and QMCKF (system under Gaussian noise) 66

3.7 The cross-section of a track generated by a system under mixture-of-Gaussians observation noise. 68

3.8 Magnitude of the filtering error of the UKF, QMC-KF and IAKF (system under MoG Gaussian noise) 69

3.9 Error as a result of mis-specification 70

3.10 Running time of Python implementation of the IAKF for 1024 iterations, at differing sample count 71

4.1 An explicit representation of multivariate state Each node in a “slice” represents a component of an input to a model. Taken as a whole each slice represents a multi-variate state space 82

4.2 Design of the FANOVA-based Simulation framework 89

4.3 FANOVA-based Sobol’ direction numbers construction 93

4.4 Static direction numbers Sobol’ sequences-driven simulation 94

4.5 Asian option $s = 16, K = 45$ 95

4.6 Asian option $s = 32, K = 45$ 96

4.7 Asian option $s = 16, K = 50$ 97

4.8 Sobol’ function g_1 variation (i) with $s = 20$, comparing FANOVA- and randomly-generated Sobol’ sequences. 98

4.9 Sobol’ function g_1 variation (ii) with $s = 20$, comparing FANOVA- and randomly-generated Sobol’ sequences. 99

4.10 Sobol’ function g_1 variation (i) with $s = 32$, comparing FANOVA- and randomly-generated Sobol’ sequences. 99

4.11 Sobol’ function g_1 variation (ii) with $s = 32$, comparing FANOVA- and randomly-generated Sobol’ sequences. 100

5.1 The distributed mobile computing system 109

5.2 ROA-based offloader workflow 116

5.3 Simulated system for the ROA-based offload manager 119

5.4 Distributed task load under ROA and fixed-host policies in the home network environment 121
5.5 Distributed task load under ROA and fixed-host policies in the "hotspot" environment 122
5.6 Distributed task load under ROA and fixed-host policies in the "cloud" environment 123

Chapter 1

Introduction

In this thesis I study adaptation mechanisms for computations based on the on-line analysis of their input data, final specification or operating environment. This information is used to modify the behaviour of the program so as to achieve greater performance. In this chapter I briefly state the problem that motivates this work, the contributions of this thesis towards solving it and give a bird's-eye-view of the thesis organization.

1.1 Problem statement and motivation

A wide range of real-life situations can be mathematically treated as sequential problems with uncertainty components. A very powerful mathematical and computational framework to express these problems is that of dynamic Bayesian networks (DBNs). A DBN model is useful in simulation-based algorithms that result in approximate solutions to a variety of problems. However, these simulation algorithms are highly sensitive to input data, the operating conditions of the implementation and domain-programmer specification. In effect, there is a trade-off between how flexible a simulation algorithm is, and how much computational and statistical performance it can achieve: high-performance simulation implementations have change points "frozen in" the code, so they are very inflexible,

whereas to maintain flexibility, a general simulation implementation must pay a considerable "interpretational tax" at run-time. More importantly, some information that can be exploited to improve the execution behaviour of a simulation is available only after that simulation has started executing, and is therefore too late to make changes. Examples of this late-arriving information are the features of the input a program is to process, the concrete operating parameters of its execution, or the specific environment it is to run on. All of this useful information is left unused at the cost of degraded performance, due to the means for a program to adapt or reconfigure itself to the features of this data.

Since simulation algorithms are an essential part of systems that carry out decision-making, inference and valuation problems, any possibility to improve run-time or statistical efficiency that is left unused is done so at the end-user's cost in accuracy or run-time performance.

1.2 Contributions of this work

In this thesis, I address the exploitation of late-arriving information with two adaptation mechanisms. The first is provided almost by accident by some software components and libraries that facilitate the programming of parallel software, and it is the ability to incrementally construct, while the program is running, pieces of code that can be re-injected into the program thus modifying its behaviour. This ability can be directed to analyze the late-arriving information described above and generate the appropriate customized routines that specifically target features of this information set. I explore these ideas specifically in the context of *model-based time series filtering*, where a number of pre-processing stages are done on the input data so as to configure the general-solution backend to better adapt to its features and improve the overall statistical efficiency. I also provide a second implementation of the same idea making use of more traditional scripting languages, which are suitable for "gluing" components, but are less often used to generate the components they are to glue together on-the-fly.

The second adaptation mechanism is that of dynamic *computation offloading*. Computation offloading is a general strategy that a program can use to delegate a particular portion of its workload to an auxiliary device, e.g., accelerating co-processors or remote servers reachable through the network. For example, a distributed system operating on a mobile, wireless network environment is often in a position where it could delegate some or all of its computation to a more generously endowed, fixed server. However, to do this in a rational way, the decision must be taken according to the specific current circumstances of the network, local and remote computer loads and other considerations. This is therefore a complicated optimization problem. The framework of Real Options analysis, however, provides the concepts and methods to take this decision in a principled way, based on a dynamic system model of the environment and a (fixed) utility function. Armed with this machinery, a program running on a resource-constrained computational node can monitor and analyze its operating environment, and, should the analysis determine it is cost-effective to do so, delegate part of its workload onto available servers. The methodology presented here allows for the user to specify what goals to seek, for example, improved response time, better utilization of scarce resources (for example, battery time), etc. This behaviour makes the program well suited for operation in mobile environments, which are typically subject to many sources of uncertainty. Note that the Real Options Analysis framework is a reformulation of the problem that provides a new point of view under which to assess the offloading decision in an uncertain environment.

In summary, this thesis advances the state of the art in the following respects:

- It illustrates the value of dynamic code generation in a variety of situations and target platforms.
- It explores a real-option-driven adaptation mechanism, which is extensible and well-furnished to deal with complicated operating environments.
- It allows a software construction discipline that relies on the mathematical description of the environment the software is to operate. A schematic of the kind of architecture

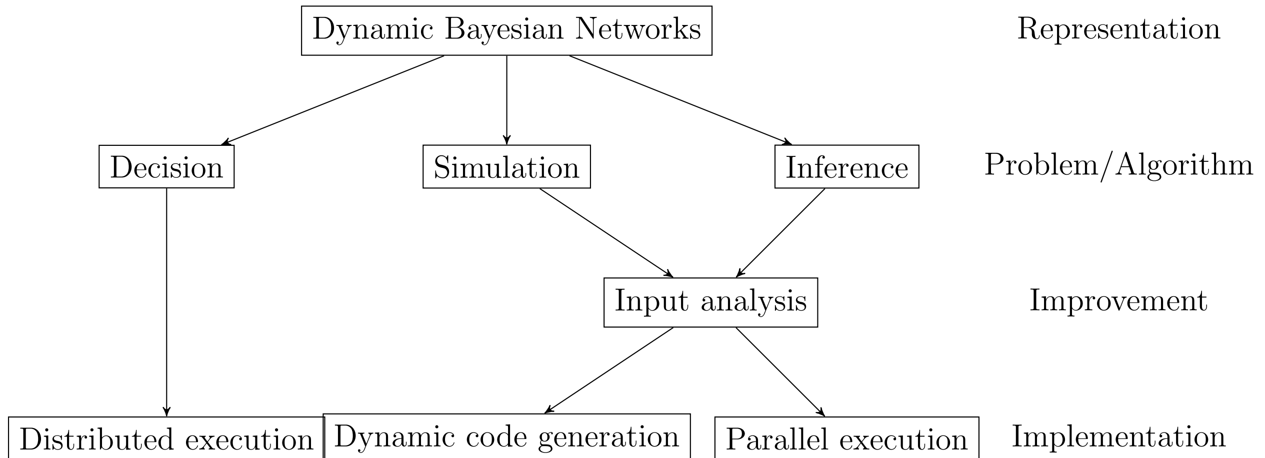


Figure 1.1: DBN-based software construction

I advocate here is presented in Figure 1.1

1.3 Thesis organization and bibliographic notes

The organization of this thesis is as follows:

Chapter 2 presents the dynamic Bayesian network formalism, both in the form of state-space representation of a dynamic system and as a Markov-chain expression of a sequential problem. The former is amenable for inference, specifically state-estimation in the light of an observed trace (i.e., *filtering*); whereas the latter is well-suited for simulation, be it of a natural or artificial system. Later in this thesis I make use of such a representation to conduct valuation of a financial instrument via simulation. In this chapter, I also briefly explore Monte Carlo algorithms for inference (time-series filtering) and simulation.

Chapter 3 introduces the "code factory" approach to adaptation and illustrates this idea by developing the Input-Adaptive Kalman filter, a technique to perform state estimation on a dynamic system given a trace of its behaviour. For the purposes of adaptation, a

clustering task on the data corresponding to the first measurement is carried out to fit a Gaussian mixture. The output of this procedure determines the number of filters a Quasi-Monte Carlo Gaussian-sum filter is to run on the data. These filters are then assembled on the fly and run over the rest of the data, which improves the filtering process for models with multi-mode noise. Preliminary ideas to this work appear in [45], whereas a more detailed exposition and peer-reviewed validation appears in [43].

In Chapter 4, a more sophisticated input analysis is carried out for the purpose of adaptation via run-time code generation. In effect, a functional ANOVA decomposition via quasi-regression, a sensitivity analysis technique, is performed on an end-programmer-specified function. The output of this analysis is then used to generate a Sobol' sequence that is tailored to the behaviour of the user-specified function, instead of relying on compile-time parameters chosen *a priori* by the builder of the sampling library, or needing to undergo the lengthy process of compilation via a traditional tool-chain. By using the resulting sampling scheme, the performance of simulation-based algorithms are improved. This work is in preparation for submission.

Chapter 5 explores another form of adaptation, dynamic computation offloading in the context of a workload that is to operate on a wireless, mobile environment. Mobile computing devices have become pervasive in modern life, but they hardly ever make adaptive use of network-reachable resources even when doing so would result in better performance or service. In general, mobile workloads are designed to either run fully locally or to completely rely on a server, not to employ a mixed-operation model. A cost-benefit analysis in the framework of Real Options is applied to a system that considers the option of offloading. Real Option Analysis provides well-understood optimization methods that are fundamentally extensible, being able to accommodate a great number of restrictions and utility models. The design is validated on a simulated wireless system under changing network conditions and limited battery life. This work has been presented and published [44].

Concluding remarks and potential research directions for future work are presented in

Chapter 6.

Chapter 2

Bayesian models and Monte Carlo methods

Holding a complete and accurate mental picture of the complex interactions that comprise any real system, be it physical, economic or social, is practically impossible. It is inevitable to rely on *models*, simplified representations that try to capture the essential features of a system, at least to the extent that is sufficient to get at the essence of a specific question we are trying to answer. A great number of fields of activity, including business strategy, policy making, and scientific research consist mainly of building, manipulating and evaluating models. Mathematical models, i.e., models that are expressed in a mathematical formalism, have proven greatly effective for their richness of expression, their convenience of manipulation, and their ease of communication. Despite its increasing formalization and abstraction, Mathematics continues to aid in the construction of progressively more detailed and accurate models of reality.

Regardless of the phenomenon under study, two aspects are pervasive: that human beings experience these phenomena unfolded in time (and therefore time and change are an inextricable aspect that need to be considered in any model) [16], and that it is impossible for an observer to have perfect knowledge about all but the simplest systems.

The need to consider the temporal aspect of real-world phenomena was recognized from the beginnings of the Modern scientific enterprise, and even the earliest mathematical models present a certain *sequential structure* to account for the temporal behavior of factors intervening in a process. The notion of *uncertainty*, which reflects the lack of complete information about the phenomenon under observation, took much longer to be explicitly incorporated in the mindset of mathematical modeling practitioners[1]. Starting from the post-World War II work of Wiener [165] and the independent foundational efforts of Kolmogorov [138], from the formalizations in the context of information theory developed by Shannon and Bode [20], and from the feedback control applications of Kalman and others, the notion of stochastic process not only was established, but has become pervasive in the mathematical treatment of real-world phenomena [167].

The uncertainty in a particular mathematical model may not be solely due to lack of information. We may also consider the treatment of uncertainty by randomness as a notational expression of conditions that are inessential, superfluous, and extraneous to the aspects of the phenomenon we wish to consider in our model. Furthermore, uncertainty may be (and often is) an inextricable part of the human perception of the world: the initial conditions of objects under observation may change imperceptibly to our instruments, although these small changes may cause significant alterations in the final results; accurate computation is impossible in practice; even minor disturbances in the phenomena or our means of observation makes noise tightly linked to our measurements, regardless of the sophistication of our instruments. In the social realm, specifically the Economic Sciences, it has become clear in the last couple of decades that economic models capable of addressing real policy questions must be both stochastic and dynamic. There are fundamental aspects of the economy that static models cannot capture. Deterministic models, even chaotically deterministic models, seem unable to explain our observations of the world [11].

Uncertainty is mathematically studied by the field of Probability Theory. In particular,

[1]Even though occasionally a mathematical model would incorporate hints of uncertain inputs, like Laplace's *error function* or Einstein's analysis of Brownian motion – the latter recognized by the Nobel Prize of Physics in 1905.

the Bayesian school of probability emphasizes the modeling of uncertainty and randomness via probability distributions. Although the manner in which probability distributions evolve in time can be described in terms of differential or difference equations, these relations are generally very difficult to solve, either analytically or numerically. Difference/differential equations can be reinterpreted as a (possibly infinite-state) automaton whose state represents the possible configurations of the conserved quantity over time. Moreover, this formulation can relate this posited possible states with a mechanism by which this state can be observed. This approach, known as *state-space formulation* or *time-domain method*, has its origins in systems theory, and has been proven very useful in working with models of phenomena over a wide selection of domains of knowledge.

With the means to model and incorporate uncertainty in our models, we now need methods of analysis to pose and answer questions about them. Regardless of their irregularities and exceptions, observations and experiments on real-world phenomena show some patterns, a regularity that is termed statistical stability [107]. Our models of these phenomena should also show this property, and analysis techniques on them search for the underlying statistical stability. Various forms of statistical stability are formulated as a set of rules collectively known as "laws of large numbers" [107], and it is rules like these that underpin most methods of analysis of stochastic systems.

In this thesis we focus on *probabilistic* models. This type of models encode the relationships among the factors relevant to the phenomenon under consideration using the notion of conditional probability. I use Section 2.1 to describe in some detail the dynamic Bayesian network (DBN) formalism, placing particular emphasis on its relationship with the formalism of Dynamic Systems. Section 2.2 goes into some detail into the uses of DBNs for modeling Stochastic Differential Equations for the purposes of simulation, as well as Dynamical Systems for inference purposes. Then, Section 2.3 treats in detail Monte Carlo (sampling-based) algorithms for simulation, inference and decision-making under uncertainty, methods which we will make use of in later portions of this thesis. Some common issues when using Monte Carlo algorithms for the tasks outlined above are outlined in Section 2.4. Finally, Section 2.5 presents a variation of Monte Carlo algorithms known

as *quasi*-Monte Carlo techniques, which make use of especially constructed sequences of numbers to drive the sampling, towards the general improvement of the performance of Monte Carlo algorithms.

2.1 Dynamic Bayesian networks

The family of probabilistic *graphical* models [125] are a representation that has proven very effective and has given rise to a very active research community. Probabilistic graphical models combine Graph Theory and Probability, thus providing a rich framework in which to model large-scale multivariate systems wherein uncertainty is an intrinsic feature. This representation allows for the formulation of problems and answering of questions that are amenable to solution by simulation. These models include Markov Random fields (MRFs) [77], conditional random fields (CRF) [86], and hidden Markov models (HMMs) [142]. In this work we specifically concentrate on a subset of probabilistic graphical models, variously known as *Bayesian*-, *belief*-, or *influence networks*.

A Bayesian network (BN) is a *directed* acyclic graph $G = (V, E)$ where each node $v_i \in V$ is associated with a random variable V_i and arcs $(v_i, v_j) \in E$ reflect the existence of local interactions among neighbouring variables. Although the term Bayesian *network* emphasizes the graphical description of the *structure* of interactions, the formalism is meant to include the quantitative specification of such relationships, in the form of conditional probability distributions[2]. In particular, the BN formalism emphasizes the representation of conditional independence statements: when presented with a node v_i and its parents $\text{Pa}(v_i)$ (where parents are defined as $\text{Pa}(v_i) \triangleq \{V_j : (v_j, v_i) \in E\} \subset V$), v_i is conditionally independent of the rest of the variables in the graph given its parents, i.e. $p(V_i \mid V) = p(V_i \mid \text{Pa}(v_i))$. In traditional notation for conditional independence (where '$\perp$' denotes that the left-hand side is conditionally independent of the right-hand side given the factors

[2]The 'Bayes' in 'Bayesian networks' is a metonym for 'conditional probability'

after the bar), this is expressed as:

$$v_i \perp \{v_j \in V \backslash \mathrm{Pa}(v_i)\} | \mathrm{Pa}(v_i)$$

The conditional probability densities can be represented as conditional probability tables (CPTs) if the neighbouring variables are discrete. In case of neighbouring continuous nodes, the respective conditional densities are most often described by functional relationships modified by noise, i.e., the distribution $p(v \mid \mathrm{Pa}(v))$ is represented by the functional relationship $v = f(\mathrm{Pa}(v), \eta)$, where η is stochastic. As a whole, a BN encodes the joint density $p(V) = p(v_1, \ldots, v_k, ...), \forall v_i \in V$.

The BN formalism allows a complete, clear and concise understanding of the abstractions and assumptions incorporated into a statistical model. Because BNs directly identify which variables depend on which other variables and which variables are conditionally independent, the BN formalism allows the straightforward connection of statistical reasoning with causal modeling, side-stepping (to some extent) the disclaimer "correlation is not causation" associated with most statistical inferencing techniques. More importantly, the use of BNs fosters disciplined thinking: assumptions underlying equations are made explicit and precise; feedback loops and delays are visualized and formalized; and the causality relationships underlying a model are made explicit. As for extensibility, BNs allows the integration of diverse sources of information.

For incorporation of temporal considerations and provisions for change, BNs have been proven equivalent to the state-space system formulation for time series analysis. The state space formalism is at the core of the study of *dynamical systems*, and as such, it forms the paradigm for modeling and studying phenomena that undergo spatial and temporal evolution. The state-space formalism was originally proposed by Kalman [73], who restated the well-known model for time series, the vector autoregression model:

$$\mathbf{z}_t = \Phi \mathbf{z}_{t-1} + \mathbf{w}_t$$

where $\mathbf{z}_t$ stands for an observed value at time t. The vector autoregression model explains one-step-ahead observations $\mathbf{z}_{t+1}$ in terms of the current observation $\mathbf{z}_t$, a transition matrix

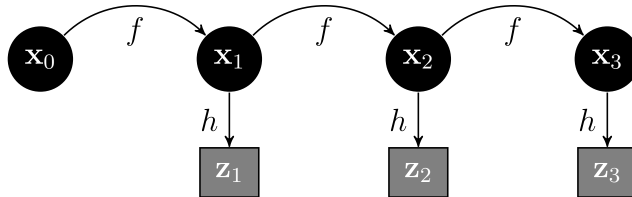


Figure 2.1: State space representation of a dynamical system

Φ and a stochastic term $\mathbf{w}_t$. In this case, if observations are $\mathbf{z}_t, \mathbf{z}_{t+1} \in \mathbb{R}^p$, Φ is a $p \times p$ matrix, and $\mathbf{w}_t$ is a p column vector.

Kalman's alternative formalism posits observations $\mathbf{z}_t$ as noisy projections of unknown *state variables* $\mathbf{x}_t$, whose dynamics ultimately generate the observed time series $\{\mathbf{z}_t\}_{t=1}^k \triangleq \mathbf{z}_{1:k}$. In this case, the DBN encodes the relationships between the state and observation variables $V = \{\mathbf{z}_{1:k}, \mathbf{x}_{0:k}\}$, with a structure like the one illustrated in Figure 2.1 [33]. In this figure, the transition kernel that maps one state to the next is represented by f, and the projection kernel, which maps each state to its corresponding observation is denoted by h. Both mappings f and h are stochastic. I will expand on this kind of DBN in Section 2.2.2 and make use of it in later chapters.

There are a great variety of DBNs, with increasingly baroque structures, as illustrated in Figure 2.2. The best known DBNs models are Hidden Markov Models (HMMs) [130], Linear Dynamic Systems (LDS) [109] and regime-switching (R-S) models [65]. HMMs and LDSs can be considered as the simplest DBNs. In the former, the state space is discrete, whereas the latter features a continuous state-space. Most problems admit modeling by any of these to some degree, so a common strategy is to start with a simple model and refine as necessary by the use of more sophisticated ones. What model to choose initially can in principle be learned from data by non-parametric (unsupervised) Bayesian approaches such as Gaussian or Dirichlet processes. However, the models so constructed may bear little resemblance to a human-scale description of the generating process. For the most part, in this thesis we use models that reflect the physical or social aspects of a system that

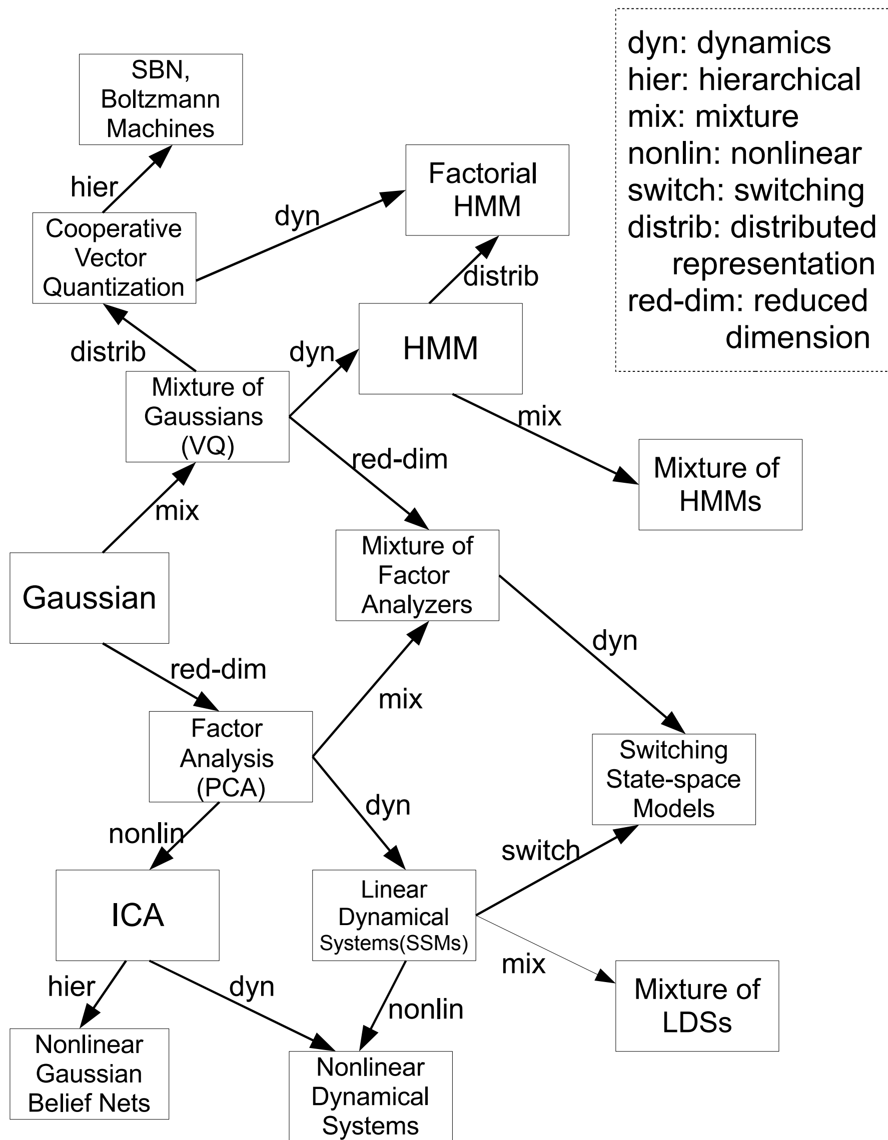
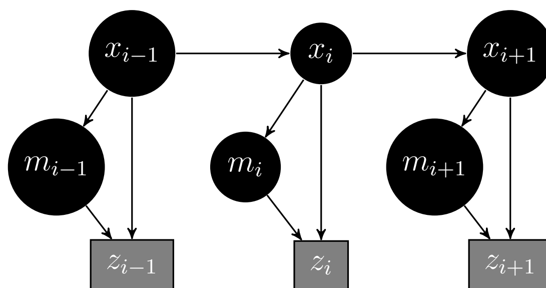
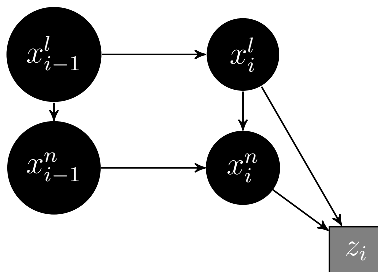


Figure 2.2: The family of dynamic Bayesian networks [110]

we are interested in exploring. These can serve as a basis from which more sophisticated models could be derived, without loss of connection to the physical features. Two examples of such models are shown in Figure 2.3, three time-steps (also called "time slices") of a dynamic model with a mixture of Gaussians output, which can stand for a multi-modal noise description [78], and a single time-step of a multi-variate dynamic system, where part of the dynamics are described analytically and others have to be treated by simulation [75]. Such systems with hybrid descriptions are normally called "Rao-Blackwellized" systems [36].



(a) DBN with a Mixture of Gaussians output



(b) A time slice of a hybrid (Rao-Blackwellized system)

Figure 2.3: Two more structured Bayesian Networks

Beyond their uses in the description of the evolution of sequential systems, the formalism of DBNs can be augmented with the notions of *decisions* and *utility* functions, to form the basis of decision-under-uncertainty frameworks [82]. In effect, the slightly en-

riched DBN formalism of Dynamic Decision Networks (DDNs) [122] plays such a role, by augmenting DBNs with the notion of decision nodes and utilities. In this thesis, I use the slightly different concept of Real Options to use DBNs for decision making in Chapter 5.

In either static or dynamic form, the BN formalism enables the treatment of a wide array of problems by simulation. Of most interest for this thesis is the treatment of the simulation and filtering of DBN models. A contribution of this thesis is the improvement of the statistical efficiency of such algorithms by adapting the sampling regimes to features in the input model or data.

2.2 Modeling with DBNs

DBNs have applications in a great diversity of fields: Physics, Quality assurance, reliability modeling, logistics, communication and computer networks, finance, written- and spoken-language recognition, to name but a few. DBNs allow a richer expression than Markov chains in the sense that they allow for *hidden*, or unobservable variables, and that they are not limited to time-invariant transition kernels. In what follows, we will illustrate this modeling flexibility in two applications: financial option pricing and time-series filtering. In particular, we'll devote this section to describe the mathematical modeling of these problems within the DBN framework. The next section will be focused on the algorithms that, taking advantage of the models, compute the solutions to these problems.

2.2.1 Solution to Stochastic Differential Equations

A stochastic differential equation (SDE) is a differential equation in which one or more of the terms is a stochastic process, thus resulting in a solution which is itself a stochastic process [66]. As such, SDEs describe the evolution of stochastic systems with time, and are used to model phenomena such as stock prices or physical systems subject to stochastic fluctuations (due to thermal conditions, small unpredictable forces, etc.). An ordinary

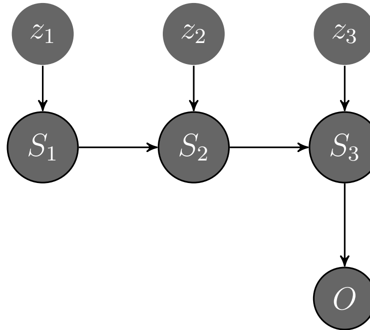


Figure 2.4: DBN representation of a stock price

differential equation (ODE) can be considered as a degenerate form of an SDE where there is no stochastic term.

The inclusion of random fluctuations in models describing physical phenomena into the mainstream of mathematical modeling was hindered by the lack of proper mathematical tools. The field of SDEs, as studied today, has its roots in Einstein's description of the effects of Brownian motion, but had to wait for Langevin and Itô for a solid foundation and inclusion in the canon of differential and integral calculus [52]. In these works, the description of a molecule under Brownian motion needs to include both Newtonian (deterministic) considerations and a stochastic component that transcends the classical Calculus machinery. Such a description takes the form

$$dX_t = a(t, X_t)dt + b(t, X_t)dW_t$$

where X_t is the state of the system, the first term is a deterministic function (or *drift*) term, which is then perturbed by a noisy *diffusive* term represented by the second element in the sum. The diffusion is attributed to a Brownian motion, so it is written in terms of a Wiener process, where dW_t stands for infinitesimal changes in the Brownian motion. The above SDE should be interpreted as describing the change in a stochastic process

$$X_t - X_0 = a(0, X_0)t + \int_0^t b(u, X_u)dW_u$$

where the term $\int_0^t b(u, X_u)dW_u$ is called an *stochastic integral*.

The definition given by Ito of this stochastic integral is expressed in terms of a limiting Riemann sum:

$$\int_a^b X(t)dW_t \equiv \lim_{n\to\infty} \sum_{k=1}^{n} X(t_{k-1})(W_{t_k} - W_{t_{k-1}})$$

Of particular importance in this thesis is the application of SDEs as a model for financial instruments that constitute *underlyings* for options and other financial derivative contracts [170]. At the heart of the option valuation problem lies a description of the evolution in time of its underlying, usually given as an SDE. For example, the dynamics of the price of a stock is commonly abstracted by an SDE of the form

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

where μ is the drift and reflects each investor's risk profiles. The above equation is usually coupled with the stipulation that a riskless asset grows at a continuously compounded rate r and therefore the risk-free measure that characterizes the dynamics is:

$$dS_t = rS_t dt + \sigma S_t dW_t$$

This equation can be solved by applying the log function and using Ito's lemma:

$$d\log S_t = (r - \frac{1}{2}\sigma^2)dt + \sigma dW_t$$

where r and σ are constant, thereby admitting the solution

$$\log S_t = \log S_0 + (r - \frac{1}{2}\sigma^2)t + \sigma W_t$$

which can be discretized to yield [70]

$$S_t = S_{t-1} + (r - \frac{1}{2}\sigma^2)\Delta t + \sigma z_t \sqrt{\Delta t}$$

where $z_t \sim \mathcal{N}(0, 1)$. This structure can be represented as a Markov chain, which is in turn a DBN, as depicted in Figure 2.4.

2.2.2 State space models

As briefly stated in Section 2.1, the state-space formalism relates the dynamics of a *hidden* or *latent* state vector $\mathbf{x}_t$ to an observed noisy projection $\mathbf{z}_t$. Mathematically stated, this model can be described by:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_{0:k}, \mathbf{z}_{1:k}, \eta_k) \tag{2.1}$$

$$\mathbf{z}_k = h(\mathbf{x}_{1:k}, \mathbf{z}_{1:k-1}, \nu_k) \tag{2.2}$$

where equations (2.1) and (2.2) are known as *plant-* and *measurement* equation, respectively [3]. State space models [1] represent both the internal dynamics of the physical process under consideration and the interaction of the process with the outside world. A diagrammatic representation of the above equations and restrictions can be found in Figure 2.1. Here we denote by $\mathbf{x}_{0:k}$ a trajectory over state space that the system follows in the period of time $[0, k]$. Similarly, $\mathbf{z}_{1:k}$ is the trace of the observation process over time $[1, k]$.

From the description above, it can be seen that the mappings f and h are very general rules that transform the history of the state of the system $\mathbf{x}_{0:k}$, and its observations $\mathbf{z}_{1:k}$, along with an stochastic component to the next state $\mathbf{x}_{k+1}$, for f and the observation corresponding to the next step, for h. It is common to impose a Markovian structure on the relations f and h. In the simplest (but still very useful) latent-space case, the unobserved process dynamics is set up to be first-order Markovian, while the observation process is zeroth-order Markovian. This structure can be expressed in terms of conditional independence:

$$\mathbf{x}_{k+1} \perp \mathbf{x}_{1:k-1} | \mathbf{x}_k$$

$$\mathbf{z}_k \perp \mathbf{x}_{1:k-1}, \mathbf{z}_{1:k-1} | \mathbf{x}_k$$

[3] In control theory, a controlling input term is usually considered as argument to function f. In this thesis, I elide this term, as the control problem is not my first concern. It should be noted, though, that a computer system is generally not solely concerned with assessing the state of a system, but it will be most often in charge of reacting to it in some way, and consequently influencing it as well. Such a computer system would require the inclusion of the controlling term for its formalization.

or, in the more familiar notation of (stochastic) functional relationships as:

$$\begin{aligned} \mathbf{x}_{k+1} &= f(\mathbf{x}_k, \eta_k) \\ \mathbf{z}_k &= h(\mathbf{x}_k, \nu_k) \end{aligned}$$

State-space systems can be classified according to the characteristics of the mappings f and h (e.g., whether or not they are linear), the characteristics of the stochastic components η_k and ν_k, or even augmented to accommodate changing dynamics and measurements f_k, h_k, among other considerations. For the purposes of this work, the most relevant considerations are whether the system is linear or nonlinear, and whether the randomness involved can be considered Gaussian or not. Being able to capture the dynamics of the system with linear relationships f and/or h greatly facilitates (and accelerates) the calculation. On the other hand more complicated and highly non-linear links may better reflect effects like trends or seasonal components.

2.3 Monte Carlo for inference and simulation

Traditional mathematical models of realistic systems usually lead to representations that are not mathematically tractable, so their analysis typically proceeds by simplifying or approximating the model until it becomes tractable by "classical" (deterministic) methods for numerical integration and/or optimization. These methods either quickly become too complicated to implement, or do not have the necessary precision or robustness to deal with even slightly complicated data. An alternative, more straightforward technique is to *simulate* the physical or social processes, usually with the help of a computer. Monte Carlo simulation methods for such problems originated at the Los Alamos National Laboratory in the early years after World War II [106], for the solution of neutron scattering problems.

In light of the practical inconvenience and overhead associated with setting up of full-scale physical experimentation, and the ever-growing available computing power, Monte Carlo methods have established themselves, along with conventional experimentation, as

a popular approach to conduct scientific research [166]. Simulation is a powerful approach even when the system under study is mathematically tractable. Simulation studies typically build some representation of the system and observe its behaviour under a variety of circumstances of interest. These representations are often software-based, (in our case, software translations of the appropriate DBN model). To generate suitable inputs, a computer can be programmed to samples the probability distributions associated with the dynamics and the parameters of model. The output is statistically studied, and conclusions about the real system can be drawn from those investigations. Because these statistical studies often ultimately take the form of a multi-dimensional integral:

$$\mu = \int_{[0,1)^s} f(\mathbf{x}) d\mathbf{x}$$

where $f : [0,1]^s \to \mathbb{R}$ and $s \in \mathbb{Z}^+$ denotes the dimensionality of the problem being simulated, they reduce to integration problems. Note that simulation studies very often have as a goal the evaluation of an integral of the form above. Furthermore, using the unit hypercube as the integration domain does not incur in a loss of generality, as this transformation is akin to state the problem in terms of the uniform random numbers driving the simulation [90]. The function f, or *model*, can be interpreted as the mapping that transforms a set of s numbers into an observation of the output quantity of interest, and μ is the expectation of this quantity, i.e. $\mu = \mathbb{E}[f]$. Therefore, we can use Monte Carlo integration techniques, specifically, performing n independent runs to estimate the quantity μ:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}^{(i)}), \mathbf{x}^{(\mathbf{i})} \in P_n, P_n \subset [0,1)^s \approx \mu$$

where P_n is a sequence of iid points uniformly distributed in $[0,1]^s$ [90].

When the mathematical model of a problem exhibits high dimensionality, many coupled degrees of freedom, or significant uncertainty in inputs, Monte Carlo methods are usually the technique of choice. In this chapter we describe the formulation of the problems of simulation and inference in terms that makes them amenable to Monte Carlo integration, the technique that will be used throughout the rest of the thesis.

2.3.1 Simulation of Stochastic Differential Equations

The Monte Carlo method provides a simple and scalable technique for the solution of SDEs. Note that by "solving an SDE" we usually mean characterize a functional over the stochastic process the SDE describes by its distribution, or, most commonly by the moments of the distribution. For example, to compute $\mathbb{E}[f(X_T)]$ where $X_t, t \in [0, T]$ satisfies

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)W_t$$

we can use the solution outlined in section 2.2.1

We can *simulate* a discretized version of the SDE, i.e. generate a discretized stochastic process, $\{X_h, X_{2h}, \ldots, X_{mh}\}$, where m is the number of time steps, h is a constant and $mh = T$. The smaller the value of h, the closer the discretized path will be to the continuous-time path we are approximating. There are a number of discretization schemes available, of which the simplest and most popular is the Euler-Maruyama (E-M) scheme [80]. The E-M scheme is outlined in Algorithm 2.1, where we denote by $\tilde{X}[i]$ the discretization of process X_t at time i.

Algorithm 2.1 The Euler-Maruyama discretization scheme

Require: Horizon T, number of time steps h

Initialization $t \leftarrow 0, \tilde{X}[0] \leftarrow X_0, m \leftarrow T/h$

for $j = 1$ to n **do**

  for $k = 1$ to m **do**

    Generate $z[k] \sim \mathcal{N}(0, 1)$

    Propagate $\tilde{X}[k] \leftarrow \mu(t, \tilde{X}[k-1])h + \sigma(t, \tilde{X}[k-1])\sqrt{h}z[k]$

    Tick $t \leftarrow t + h$

  end for

end for

2.3.2 Model-based inference on Dynamical Systems

A common problem in Dynamical Systems is to estimate the state $\mathbf{x}_k$ given observations up to time k, i.e. $\mathbf{z}_{1:k}$. At time k an observation is performed, resulting in the value $\mathbf{z}_k$. The term $\mathbf{z}_k$ is posited to reflect in a more-or-less precise way the state of the system at time k, $\mathbf{x}_k$. The state of the system can be represented as a vector $\mathbf{x}_k$ within a *state space* $\mathcal{X}$, a mathematical space whose axes are the state variables. The state is considered to be directly unobserved and possibly even unobservable.

The collected results of the observation process form a time series, which is itself a path or *trace* on the observation space $\mathcal{Z}$: $\mathbf{z}_{1:k} \triangleq \{\mathbf{z}_i\}_{i=1}^{k} \triangleq Z_k$, where $\mathbf{z}_i \in \mathcal{Z}$. The time indices where observations of the process were taken form a natural discretization, i.e. $k \in \mathbb{N}$ is mappable to the natural numbers. Most often, a uniform sampling rate is assumed. Since we consider only the discrete-time case, we will use k for the index variable to emphasize the fact that this is a non-negative integer.

Sequential inference/estimation is the solution to sequential problems. Several problems can be identified in this context, most notably, those of *filtering*, *smoothing* and *prediction* [139]. All of these are concerned with the calculation of a specific probability distribution. The probability densities associated with each sequential estimation problem are stated in Table 2.1.

The term "filter" is a remnant from the era of crystal radios and vacuum tubes, when analog circuits filtered out noise from an electronic signal. Both signal and noise were described by their power spectral densities, a characterization that was abstracted independently by Kolmogorov and Wiener into a probability distribution in their mathematical formalization of the problem. In its modern meaning, filtering is an operation that, given a time series $\mathbf{z}_{1:k}$ and a model, extracts from $\mathbf{z}_{1:k}$ information about a quantity of interest at time k by using data measured up to and including k. In its most basic form, a filter estimates the instantaneous state of a dynamic system perturbed by noise using (possibly corrupt, possibly incomplete) measurements $\mathbf{z}_{1:k}$. As such, the distribution that we are

Problem	Distribution
filtering	$p(\mathbf{x}_{1:k}\|\mathbf{z}_{1:k})$
fixed-interval smoothing	$p(\mathbf{x}_{1:L}\|\mathbf{z}_{1:L})$, for fixed L
fixed-lag smoothing	$p(\mathbf{x}_{1:k-L}\|\mathbf{z}_{1:k})$
fixed-point smoothing	$p(\mathbf{x}_{1:k}\|\mathbf{z}_{1:k+L})$
predicting	$p(\mathbf{x}_{1:k+L}\|\mathbf{z}_{1:k})$

Table 2.1: Posterior distributions of interest

.

concerned about is the so-called *filtering distribution*:

$$p(\mathbf{x}_{1:k}|\mathbf{z}_{1:k})$$

The filtering task revives the ages-old distinction between *phenomena* (what one is able to observe), *noumena* (the reality imperfectly perceived by our senses), and the state of knowledge about the noumena that one can deduce from the phenomena. The process of filtering can be described within the mathematical framework of Bayesian inference, specifically sequential Bayesian inference. Sequential inference is carried out by (recursively) updating some estimates about a time-evolving system as observations $\mathbf{z}_k$ are obtained. These estimates are typically statistics of the posterior distribution of interest, but in its most general form, can be expressed as the computation of the full distribution via the iterative application of the Bayes theorem. In the case of filtering, the *recursive Bayesian estimator* [38] can be derived as:

$$\begin{aligned} p(\mathbf{x}_k|\mathbf{z}_{1:k}) &= \frac{p(\mathbf{z}_{1:k}|\mathbf{x}_k)p(\mathbf{x}_k)}{p(\mathbf{z}_{1:k})} \\ &= \frac{p(\mathbf{z}_k, \mathbf{z}_{1:k-1}|\mathbf{x}_k)p(\mathbf{x}_k)}{p(\mathbf{z}_k, \mathbf{z}_{1:k-1})} \\ &= \frac{p(\mathbf{z}_k|\mathbf{x}_k, \mathbf{z}_{1:k-1})p(\mathbf{z}_{1:k-1}|\mathbf{x}_k)p(\mathbf{x}_k)}{p(\mathbf{z}_k, \mathbf{z}_{1:k-1})p(\mathbf{z}_{1:k-1})} \end{aligned} \tag{2.3}$$

The Bayesian formulation of $p(\mathbf{z}_{1:k-1}|\mathbf{x}_k)$

$$p(\mathbf{z}_{1:k-1}|\mathbf{x}_k) = \frac{p(\mathbf{x}_k|\mathbf{z}_{1:k-1})p(\mathbf{z}_{1:k-1})}{p(\mathbf{x}_k)}$$

can be substituted in (2.3) to result in:

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) \quad = \quad \frac{p(\mathbf{z}_k|\mathbf{x}_k, \mathbf{z}_{1:k-1})p(\mathbf{x}_k|\mathbf{z}_{1:k-1})}{p(\mathbf{z}_k|\mathbf{z}_{1:k-1})} \tag{2.4}$$

The term $p(\mathbf{x}_k|\mathbf{z}_{1:k-1})$ can be obtained by marginalization of a transition relationship (an application of the *total probability law*):

$$p(\mathbf{x}_k|\mathbf{z}_{1:k-1}) \quad = \quad \int p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1})\mathrm{d}\mathbf{x}_{k-1} \tag{2.5}$$

which introduces the recursive term $p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1})$. Expressions (2.4) and (2.5) postulate the posterior filtering density as the product of the likelihood and the prior (projected) estimate, modulated by a normalization factor sometimes referred as *evidence* term.

An algorithm that has proven to be greatly effective for carrying out the task of state estimation is variously known as “particle filter” or “Sequential Monte Carlo” [32]. Particle filters are sample-based approximate inference algorithms that encode a Bayesian approach to state estimation or “tracking”, and owe in great part their popularity to their ability to handle nonlinear dynamics and observation functions, as well as transitional and observation distributions that may greatly differ from the Gaussian distribution.

While particle filters can deal with non-linear, non-Gaussian filtering tasks in principle, their performance often varies significantly due to their stochastic nature. Because of this, vast number of variations to the basic algorithm have been proposed (recently surveyed in [37]). The *statistical efficiency* of the particle filter algorithm, i.e., to obtain similar-quality solutions using a fraction of the samples, an objective that can be achieved by applying randomized quasi-Monte Carlo (QMC) sampling techniques.

2.4 Issues concerning Monte Carlo algorithms

As for most numerical methods, it is important to assess the convergence rates of Monte Carlo methods, both computationally and statistically. This section discusses these issues.

2.4.1 Algorithmic complexity and estimator quality

The complexity of Monte Carlo algorithms is linear in the number of samples. If the model f is made more complicated, i.e., more difficult to compute, this clearly affects the running time of the algorithm, but not its asymptotic behaviour. The number of runs of the algorithm, i.e., the sample size depends on the required accuracy of the output distribution.

A major concern of any simulation-driven estimation task is to determine how efficiently an estimator of a given quality can be computed. This assessment is computed for Monte Carlo methods via the Law of Large Numbers (LLN) and the Central Limit Theorem (CLT) [40]. To obtain a Monte Carlo estimator $\hat{\mu}_n$ of the model $f : [0,1)^s \to \mathbb{R}$:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \approx \mathbb{E}[f(\mathbf{x})] = \mu \tag{2.6}$$

the standard Monte Carlo algorithm proceeds as follows:

1. Generate $\{\mathbf{x}^{(i)}\}_{i=1}^{n}$ as iid uniform numbers over $[0, 1^s]$
2. Estimate $\mathbb{E}[f(\mathbf{x})]$ by $\hat{\mu}_n$

The LLN states that, as long as the expectation $\mathbb{E}[f(\mathbf{x})]$ exists and f behaves in a way that complies with minimal conditions (e.g., that f is integrable), the Monte Carlo estimator $\hat{\mu}_n$ converges to the desired expectation as n increases, i.e.,

$$\lim_{n\to\infty} \hat{\mu}_n = \mathbb{E}[f(\mathbf{x})]$$

in other words, the Monte Carlo estimator is *consistent* [69].

Whereas the LLN guarantees the convergence of a Monte Carlo estimator to the correct answer (as the sample size n becomes infinitely large), the CLT provides the means to determine how that estimator is distributed for large but finite sample sizes n. The CLT states, in a nutshell, that the sum of a large number of random variables is normally distributed, regardless the distribution of each individual summand. This result can be used to establish that the asymptotic distribution of $\hat{\mu}_n$ is a Gaussian, information that can be used to state that

$$\frac{\sqrt{n}}{\sigma}(\hat{\mu}_n - \mu) \Rightarrow \mathcal{N}(0, 1)$$

as $n \to \infty$, from where it can be seen that the estimated expected approximation error is proportional to $1/\sqrt{n}$.

2.4.2 Variance reduction

It is clear that reducing the variance of the estimator $\hat{\mu}_n$ would automatically translate into better estimates. Given two estimators W and Y where $\mathbb{E}[W] = \mathbb{E}[Y] = \mu$, so that, to estimate μ the researcher is free to choose whether to use W or Y for simulation, it is clearly beneficial to choose the simulation for which the estimator has less variance. In fact, a central goal and common theme of research in Monte Carlo methods is design of estimators of decreasing variance. There are several general patterns for the reduction of the variance of an estimator, and techniques that make use of those patterns [156]:

Analytically integrate a function that is similar to the model f. Following the general principle that, if at any point during an estimation procedure, it is possible to replace an estimation with an exact value, it is useful to do so, this technique substitutes the original model f with a similar one that may be better suited for analytic integration, or whose expected value is known. Techniques that follow this idea include conditional Monte Carlo, importance sampling and the use of control variates. In particular, the importance sampling method consists in proposing a change of

probability measure, under which the original system is then simulated, and the bias introduced by the different measure is systematically corrected. Importance sampling is a very useful and popular approach. Since we make limited use of it in this thesis, the interested reader is referred to [133] for a mathematically rigorous treatment of the subject.

Uniformly placing sample points across the integration domain. The use of sample regimes that more uniformly sample the integration domain can result in reduced estimator variance. This is the idea used by the technique of quasi-Monte Carlo sequences, which is central to the work in this thesis.

Adaptive sampling. Adaptive sampling consists of adaptively controlling the sample density based on information gathered during sampling. Techniques that exploit this idea include adaptive sampling and "Russian roulette" methods.

Combine samples of two or more estimators whose values are correlated. This technique is very popular because of its simplicity. Its use is commonly illustrated by the method of antithetic variates.

2.5 Quasi Monte Carlo methods

Random point sets generated by Monte Carlo sampling often show clusters of points and tend to take wasteful samples because of gaps in the sample space [90]. This observation led to proposing error-reduction methods by means of deterministic point sets, such as low-discrepancy sequences [90]. Low-discrepancy sequences try to utilize more uniformly-distributed points. Application of low-discrepancy point sets instead of the pure random sampling of vanilla Monte Carlo in the context of multivariate integration is usually referred to as *quasi-Monte Carlo (QMC)* approaches. QMC methods have been successfully applied to Computer Graphics [74], Computational Physics [88] and Financial Engineering [124], among other fields.

Section 2.5.1 discusses several theoretical aspects of deterministic QMC sequences, which we will then proceed to explore in more detail by looking at individual construction methods in section 2.5.2. The important issue of randomization of these constructions is treated in section 2.5.3.

2.5.1 Discrepancy and low-discrepancy sequences

A low-discrepancy point set is one whose sample points are distributed so as to mimic as closely as possible the uniform distribution. It is clear, therefore, that any attempt to construct low-discrepancy point sets requires a way to measure their uniformity. Several such metrics exist, for example the spectral test used to assess the equidistribution in some random-number generators (and of which I will talk in more detail in chapter 4). An alternative approach, which is not specific to any particular construction is to measure the distance between the empirical distribution induced by the point set and the uniform distribution, via the Kolmogorov-Smirnov (K-S) statistic [46]:

$$D_n(P_n) = \sup_{x\in[0,1)} |F(x) - \hat{F}_n(x)|$$

where $\hat{F}_n(x)$ is the *empirical cumulative distribution function induced by point set* P_n:

$$\hat{F}_n(x) = \frac{1}{n} \sum_{x^{(i)}\in P_n} \mathbf{1}_{x^{(i)}\leq x}$$

this is, the proportion of $x^{(i)} \in P_n$ that are smaller or equal to x. The quantity $F(x)$ is the cumulative distribution function of $\mathcal{U}(0,1)$, the uniform distribution function over $[0,1)$, i.e. $F(x) = x$.

The K-S statistic measures the non-uniformity of a sequence of points placed in a unary interval $[0,1]$. The K-S statistic notion can be extended to the multi-variate case, where it is known as the *star-discrepancy* of point set P_n, $D_n^\star(P_n)$ [42]:

$$D_n^\star(\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(n)}) = \sup_{E(\mathbf{v})} |\prod_{j=1}^{s} v_j - \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\mathbf{x}^{(i)}\in E(\mathbf{v})}|,$$

this is, for every hyperbox $E(\mathbf{v}) \subset [0,1)^s$ with a corner at the origin and other corners defined by $\mathbf{v} = (v_1, \dots, v_s)$, i.e., a region of the form $[0, v_1) \times [0, v_2) \dots \times [0, v_s)$, we count the number of points $\mathbf{x}$ in P_n that fall inside $E(\mathbf{v})$, divide by n and take the absolute difference between this quotient and the volume of $E(\mathbf{v})$. The maximum difference is the star discrepancy $D_n^\star$. The "star" qualifier emphasizes the peculiarity that the hyperboxes $E(\mathbf{v})$ are all anchored at the origin.

The lower the star discrepancy, the more uniformly the point set is distributed. A sequence $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}$ of points in $[0,1)^s$ is a *low-discrepancy sequence* if, for any $n > 1$

$$D_n^\star(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}) < c(s)\frac{(\log n)^s}{n}$$

where $c(s)$ is a constant specific to the problem dimension s.

In the context of Monte Carlo-based integration, the integration accuracy relates to star discrepancy via the Koksma-Hlawka inequality [81]:

$$|\mu - \hat{\mu}_n| \leq V(f) D_n^\star(\mathbf{x}^{(1)}, \dots \mathbf{x}^{(n)})$$

where μ and $\hat{\mu}_n$ are as in 2.6 and $V(f) < \infty$ is the variation of f in the sense of Hardy and Krause [90]. With an increase in n QMC methods may offer better convergence rates than straight-up Monte Carlo.

Theoretical analyses as well as empirical studies [123] suggest that the error $|\mu - \hat{\mu}_n|$ can be significantly lower for QMC methods than under pseudo random sampling. The error bounds above, $O((\log n)^s/n)$, are, for fixed dimension s, better asymptotic rates than Monte Carlo's $O(1/\sqrt{n})$. Hence, we can expect that QMC methods can approximate the integral with a smaller error than traditional Monte Carlo if the point set size n is sufficiently large.

Having determined the theoretical superiority of QMC methods over Monte Carlo's sampling strategy, let us proceed to the description of how to construct suitable low-discrepancy point sets P_n.

2.5.2 Constructions

Low-discrepancy point sets are constructed based on the notion of sampling a sub-volume of the volume of interest with the same uniformity as they do the whole volume itself. Informally, low-discrepancy point sets can be thought of as a stream of self-avoiding points, where the next point in the sequence is as far away from the earlier members of the sequence as possible. Note that the term *low-discrepancy* refers to points sets P_n designed to be *more uniform* than a random point set *for some measure of uniformity*, which does not need to be the star-discrepancy. Due to their deterministic nature, it seems intuitive that QMC point sets should be constructed in such a way so as to exhibit the best-possible spread in the sample space, avoiding the gaps and clusters that arise from (pseudo) random sampling. There are two main families of low-discrepancy point sets are the *digital nets* and the *lattice points* [90].

Lattice rules

The idea behind lattice rules comes from the observation that quadrature-based point sets, when projected onto an axis, frequently result in point sets with much fewer points that the original point set. This is because several points on the higher dimension map to the same point on the projection. When the quadrature point set is intended to serve as the driver for numerical integration, this feature of the low-dimensional projections results in increased integration errors. In an effort to side-step this difficulty, low-discrepancy point sets P_n are constructed so that their lower-dimensional projections do not degenerate, i.e., still have n points. In fact, it is greatly desired that lower-dimensional projections of a low-discrepancy point set are low discrepancy point sets themselves.

If the quadrature rule is constructed via a generator of the form

$$\sum_{i=1}^{s} z_i \mathbf{v}_i, |\mathbf{v}_i| \leq 1, z_i \in \mathbb{Z}$$

where the vectors $\mathbf{v}_i$ are parallel to the axes, it will suffer from poor lower-dimensional

projections. If instead we construct P_n by taking all the multiples $z\mathbf{v} \bmod 1$ for $z = 0, \dots, n$, where the modulo 1 operation [4] is applied component-wise and the vector $\mathbf{v}$ is carefully-chosen and not parallel to any axis, we can assure that the low dimensional projections of P_n will have n points. This is the idea behind *lattice rules*, which in general have the form $P_n = L \cap [0,1)^s$, with an *integration lattice* L defined by:

$$L = \{\mathbf{x} = \sum_{j=1}^{s} z_j \mathbf{v}_j : \mathbf{z} = (z_1, z_2, \dots, z_s) \in \mathbb{Z}^s\}$$

where the linearly-independent vectors $\mathbf{v}_1, \dots, \mathbf{v}_s \in \mathbb{R}^s$ are the *basis* of L, i.e., a lattice is obtained by taking all integer linear combinations of the vectors in its basis. We assume that L contains $\mathbb{Z}^s$ [89]. A lattice point set P_n has a *rank*, defined as the smallest integer r such that P_n can be obtained by taking all integer combinations, modulo 1 of r vectors $\mathbf{v}_1, \dots \mathbf{v}_r$, independent over $\mathbb{R}^s$.

A commonly-used example family of a lattice rules is that of Korobov point sets [83]. Korobov point sets were introduced in the 1960s, and consist of an instantiation of a lattice rule with basis of the form $\mathbf{v}_1 = (1, a, a^2, \dots, a^{s-1})/n \bmod 1$, and $\mathbf{v}_j = \hat{\mathbf{e}}_j$ for $j = 2, \dots, s$, where $\hat{\mathbf{e}}_j$ is the s-vector of zeros in all components except the jth, which is 1. It can therefore be seen that Korobov rules have rank 1.

In practice, most lattice rules are based on lattices of rank 1, where the lattice point set can be written as

$$P_n = \{\frac{i}{n}(z_1, \dots, z_s) \bmod 1 \mid i \in [0, n)\}$$

and $(z_1, \dots, z_s) \in \mathbb{Z}^s$ is the *generator* vector, with the parameters $z_1, \dots, z_s$ to be decided by the user. Rank-1 lattice rules have a number of advantages, for example, they are *fully projection regular*, i.e., each projection $P_n(I)$ contains n distinct points, while lattices of higher rank do not, in general, feature point sets with this desirable property. Here, we denote by $P_n(I)$ the projection of the point set P_n over the dimensions specified by the positive-integer set I,

[4] The $\bmod 1$ operation consists of keeping only the part after the decimal point of a number.

A widely-used class of lattice rules, Korobov rules, mentioned above, are rank-1 rules. For Korobov rules, the generator vector $(z_1, \ldots, z_s) = (1, a, a^2, \ldots a^{s-1})/n \bmod 1$, where a is a positive integer. This means that only one parameter, the generator a, an integer between 1 and $n-1$, of P_n, needs to be specified. The parameter a is chosen to minimize a discrepancy measure. Tables of good generators a for various values of n are available [85].

Digital nets

The foundation of digital nets is based on a different rationale than lattice rules. For digital nets we define the point set $P_n = \{\mathbf{u}_i\}_{i=1}^n$ by looking at the expansion of the index i in a given base $b \geq 2$. More precisely, for a non-negative integer i, we first write the base b-expansion of i:

$$i = \sum_{l=0}^{\infty} a_l(i) b^l$$

where $a_l(i)$ denotes the l-th digit in the base b-expansion of i, and infinitely many coefficients $a_l(i)$ are zero. Then, we use the *radical inverse function* in base b, $\phi_b(i)$:

$$\phi_b(i) = \sum_{l=0}^{\infty} a_l(i) b^{-l-1}$$

where, by definition $\phi_b(i) \in [0, 1)$. The simplest construction, the van der Corput sequence [], is defined in terms of the radical inverse function:

$$P_n^{vdC} = \{\phi_b(i-1)\}_{i=1}^n$$

The van der Corput sequence dates back to 1935, and serves as a basic building block for other digital net constructions. At an intuitive level, the points in the van der Corput sequence are placed in an order that in some sense attempts to never leave wide intervals in $[0, 1)$ containing no points.

Most low-discrepancy sequences consist of more sophisticated uses of the van der Corput sequence. The Halton s-dimensional sequence [63], for example is defined as:

$$P_n^{Halton} = \{\mathbf{u}_i\}_{i=1}^n \text{where } \mathbf{u}_i = (\phi_{p_1}(i), \ldots \phi_{p_s}(i))$$

where $p_1, p_2, \ldots p_s$ are the first s prime numbers. The star discrepancy corresponding to the Halton sequence is $D_n^\star(P_n^{Halton}) = O(\frac{1}{n}(\log n)^s)$.

Other examples of digital nets include the Hammersley [90], Faure [47], Sobol' [22] and Niederreiter [90] sequences. These sequences are based in an alternative interpretation of the construction of the van der Corput sequence [101] in terms of *direction numbers*. I will go into much deeper detail in the case of the Sobol' sequence in Chapter 4. Several low-discrepancy sequences following this pattern were proposed in the literature early on, (e.g. the constructions by Halton, Sobol' and Faure), and were later generalized by Niederreiter, into a general framework, the so-called (t, d)-sequences [34].

2.5.3 Randomizations

The vanilla Monte Carlo method provides probabilistic assurances on the estimation error, as well as on the bias characteristics of estimators (typically unbiased estimators are preferred). One of the main practical concerns to the use of deterministic QMC sequences in a simulation context is that those guarantees and the techniques that were used to derive them are not applicable. However, these seemingly insurmountable objections can be avoided if, instead of using purely-deterministic QMC sequences, one uses *randomized* QMC sequences (RQMC). The idea is to randomized a highly-uniform point set P_n so that each of its points follows the uniform distribution over $[0, 1)^s$, while preserving the high uniformity of P_n. Many randomization techniques that achieve this have been proposed and used in practice.

A general formulation for the randomization of low-discrepancy sequences is presented below. Given the low-discrepancy point set $P_n = \{\mathbf{u}_i\}_{i=1}^n, \mathbf{u}_i \in [0, 1)^s$ and a uniform random

vector $\mathbf{v} \in \Omega$, we are interested in finding a randomization function $r : \Omega \times [0,1)^s \rightarrow [0,1)^s$ such that we can construct the *randomized low-discrepancy* point set $\tilde{P}_n = \{\tilde{\mathbf{u}}_i\}_{i=1}^n, \tilde{\mathbf{u}}_i = r(\mathbf{v}, \mathbf{u}_i)$. The function r should provide the following two properties:

- Every element of the sequence $\tilde{P}_n$ has a uniform distribution over the unit cube and
- the low-discrepancy properties of the sequence P_n are preserved by the randomized sequence $\tilde{P}_n$

Several randomization techniques exist, of which, the most popular are:

Shift modulo 1 aka Cranley-Patterson method [30]. Sets $\Omega \equiv [0,1)^s$ and $r(\mathbf{v}, \mathbf{u}) = (\mathbf{v}, \mathbf{u}) \bmod 1$.

Digital shift [90] is the counterpart of the above method for digital nets base b: $r(\mathbf{v}, \mathbf{u}) = \mathbf{v} \oplus_b \mathbf{u}$, again with $\Omega \equiv [0,1)^s$. $\oplus_b$ is the coordinate-wise addition of the base b expansions of v_j and u_j.

Scrambling is actually a family of randomization techniques. One of the first, and a representative member of this family is the *nested uniform scrambling* [121], which applies uniform random permutations to the digits of each coordinate $u_{i,j}$ in its base b expansion.

Note that all of these techniques apply to any low-discrepancy construction, but not to the same level of effectiveness. For example, digital shift and scrambling are better suited for (t, m, s)-net constructions than other methods. In the remainder of this thesis, we make extensive use of randomized QMC sequences for a variety of simulation and inference problems.

Chapter 3

Adaptation via Code Generation and its Application to Filtering

In this chapter I propose an adaptive mechanism for domain-specific specialization of a class of programs dealing with time-series data. This proposal allows a general algorithmic solution to be customized to the specific input the implementation is to work on. The mechanism proposed here relies on run-time code generation, since, by generating code on-the-fly, the customization strategy becomes more flexible than if it, for example, relied on "canned" alternative strategies already present in the system. Furthermore, a synthesized algorithm may result in an implementation that, since it is tailored to the input, avoids costly and repetitive run-time choices. To illustrate the proposed mechanism, I present a program that implements a non-linear time series filter. For this task, a representation of a program as a Abstract Syntax Tree (AST) is manipulated to incorporate known filtering algorithmic building blocks. The output of this manipulation is a filter that is adapted to the specific input. Furthermore, the generated filter can be re-used for input of similar characteristics. I find that generated filters exhibit improved statistical performance when compared to fixed-filter solutions. Moreover, upon changes in the computing environment, run-time code generation enables the creation of filters with better running times than

their similarly-adaptive, but "fixed code" (no intervening code generation) counterparts.

3.1 Motivations and Central Idea

The task of programming is to a large extent a balancing act of two concerns constantly at odds with each other: on one hand, a program needs to model a problem to the point that this problem is amenable to algorithmic treatment, whereas on the other hand this algorithmic solution is to make best use of available computer resources. The achievement of the latter goal is often facilitated by information obtained at the modeling level. The incorporation of this information into the construction of the end program to improve its execution by some measure is known as *domain-specific program optimization* [95].

Domain-specific program optimization is most often carried out by the use of domain-specific libraries, which are often tightly-coupled to a specific computing platform[1]. This approach allows for the efficient execution of an application at the building-block level, but does not make use of the context in which these building blocks are used. This is explained by the fact that context information comes late in the process of program execution. In this chapter, I explore run-time code generation as an alternative technique that can make use of this late-coming information to drive the construction of a program that is better adapted to the underlying platform as well as to the input. To illustrate the adaptation technique in detail, I apply it to the task of state estimation in time series, otherwise known as the *filtering* problem.

In the field of non-linear filtering, most algorithms follow the same pattern: a prediction step is followed by a correction step, a process that continues until the input is consumed completely. Despite this commonalities, a great number of variations exist to deal with different features of the input. Different approximations are used to account for non-linearities in the model, for example, or for different characteristics of the probability

[1]Examples of such libraries for the field of Computer Vision is OpenCV [8], for the field of Quantitative Finance is Quantlib [150] and for the field of Signal-Processing and Data Analysis is IPP [29].

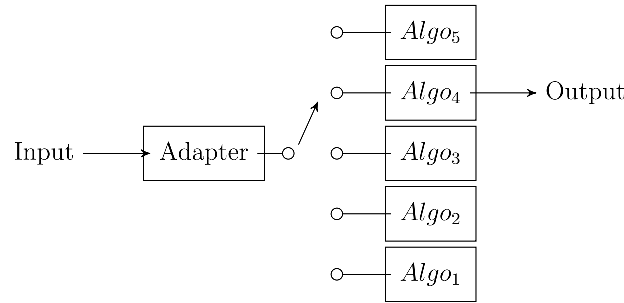


Figure 3.1: Alternative-based adaptation in domain-specific libraries.

distribution of the noise. Likewise, different evaluation strategies may be employed on the system representation. These differences create multiple points of variation in the basic predictor-corrector algorithm, each of which is suited to a greater or lesser extent to a particular input. Most software libraries that cater to the task of non-linear filtering offer some of the available variations of the algorithm, and leave to the end-user the decision of which to use in a given situation. This decision is usually taken with limited information, as the actual features of the system are not usually available until right before the filtering task starts, i.e., at run-time. In an effort to attain some flexibility, an implementation may provide implementations of variants of a basic algorithm, more-or-less suited to a particular kind of input. Which specific variant to use at any given time can be left for the end-user to choose via configuration switches, environment variables or even run-time adaptation modules. A graphical depiction of this software architecture is shown in Figure 3.1. In this design, adaptation decisions must be taken repeatedly with every new input. If, as it is often the case, the characteristics of the input are similar for the same batch, this process is unnecessary, as it results in the same choice for all inputs in the batch.

To overcome the shortcomings mentioned above, I present in this chapter an alternative design in the form of a meta-program, that analyzes the input to be processed, and assembles a filter appropriate to that particular input from a collection of building blocks. This strategy is depicted graphically in Figure 3.2, where the block labeled "Factory" ma-

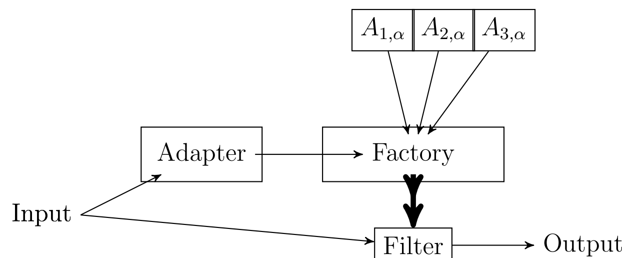


Figure 3.2: The "code factory" approach to adaptation in domain-specific libraries.

nipulates variant algorithms $A_{i,\alpha}$ and generates the adapted filter. In the design proposed in this chapter, the role of the factory is played by a code generator subsystem. The α configuration directive denotes whatever parameters the variant requires for a concrete instantiation. The actual value of these parameters are obtained from the "Adapter" block. This stands in contrast with Figure 3.1, where each variant is fully instantiated, and the adapter module only determines *which* variant to use. By using such an approach, the end-user programmer is relieved of the burden to decide which specific variation of non-linear filter to use, and the choice more closely matches the filtering task, with the associated gains in statistical performance. This meta-filter makes use of run-time code generation facilities becoming available in mainstream programming languages and tool chains [31].

The remainder of this chapter is organized as follows: The technique and necessary support for dynamic code generation is briefly presented in Section 3.2. The filtering problem, on which the proposed technique is later applied, is introduced in Section 3.3, where both the basic Kalman filter (KF) algorithm and its variants designed to handle non-linear dynamics in the presence of non-Gaussian noise are explained. Then, in Section 3.4, we illustrate our proposed system, the "Input-Adaptive" Kalman filter (IAKF), going into some detail in its operation as well as two possible implementations, both of which make use of the dynamic code generation facilities, one from a very-high-level language (Python) and the other from a recent parallelizing framework in C++, Intel's ArBB. In Section 3.5, we proceed to evaluate the performance of the IAKF against traditional filters

both in terms of statistical performance (i.e., estimation error) and running time, carrying out a number of experiments. I concisely explore other approaches that use code generation for the creation of filtering (specifically Kalman filtering) solutions in Section 3.6, followed by concluding remarks in Section 3.7.

3.2 Code Generation for Domain-Specific Adaptation

Dynamic code generation is an umbrella term for any kind of modification to the instruction stream of a program once that program has started its execution. Moreover, this modification is often initiated and driven by the program in question. This technique is one of the ideas in Computer Science that, almost as old as digital computers themselves [19], comes in and out of fashion and has to be rediscovered continuously. Correspondingly, it emerges in a multitude of forms and driven by a variety of requirements. In particular, as the level of abstractions of programming languages has risen, direct dynamic code generation has been discouraged, insofar as end-user programmers are concerned [126].

In this section, we first present this "traditional" form of code generation, and a variant that allows more end-user control in subsection 3.2.1. Then, in subsection 3.2.2, we move on to describe what run-time facilities a programming environment must offer to enable any kind of run-time code generation. This delineates the applicability of our technique, as any environment that lacks the functionality described in this part is probably a poor implementation choice for the methods we here propose. In subsection 3.2.3 we describe the "user interface" of a run-time code generation and manipulation system. Although strictly unnecessary as our technique may be used with the infrastructure of subsection 3.2.2 via the facilities described in subsection 3.2.1, the API described in this section allows much more readable and maintainable code, and is thus the level that we implement our application.

3.2.1 Traditional Run-time Code Generation

For programmers of systems software, dynamic code generation, (or similar techniques, usually going by names like automatic program specialization, binary translation, monkey patching, etc.) represents a legitimate approach to efficiently implement programs, often starting from generic components. Used in this way, dynamic code generation bridges the gap between the often-opposing objectives of generality and efficiency. Even though the driving motivation of resorting to run-time code generation is efficiency improvement, this technique has also been used to address other concerns, like energy-usage awareness [153], safe binary information exchange in distributed systems [9] or user input validation [162].

When the main pursuit is performance improvement, dynamic code generation often appears in the form of just-in-time (JIT) compilation [15], where the translation or execution of portions of a program is delayed until enough information to adequately carry out the translation is deemed collected. JIT compilation can significantly reduce the performance impact of high-level abstraction constructs and modularity, as well as improve portability. Through JIT, some of the following code characteristics can be achieved, resulting in improved performance:

- Elimination of cost for late binding of functions
- Freezing of control flow once parameters are known
- Freezing the size of dynamically-sized data structures

In higher-level languages, dynamic code generation JIT facilities allows for code that exhibits both flexibility and acceptable performance. Flexible, portable, readable software, in particular *scientific computing* software, is often built from abstract components which have been independently verified and optimized. Unfortunately, there are performance costs associated with such modularity: components are deployed outside the context in which they have been optimized, or even within that context, but in combination with other, similarly optimized, components that may have conflicting requirements. In this

situation, JIT can help, as contextual information (for example, the circumstances of the function call) is available for better-targeted translation to take place [135]. JIT subsystems have become essential for carrying out cross-component optimization [17]. As such, most managed programming environments (Java, scripting languages, Matlab) incorporate a JIT compiler in their runtime. In this incarnation, the dynamic-code generation subsystem is most often offered as part of the run-time system and kept "under the hood", away from the end-user. In contrast, for this investigation, I make use of an alternative design, where the end-user or library programmer can explicitly manipulate the code that is to be generated at run-time. The latter approach allows the end-user to take control of the code generation module and direct code synthesis that may be too domain-specific to be included in the hidden JIT subsystem.

The circumstances in which this alternative form of run-time code generation most often appears in mainstream programming practice is when two vastly different programming models need to coexist in a single application. Here, one language acts as a *host*, and provides the facilities for execution of the whole application. Embedded in this host environment, fragments of source code of the second programming language (the *guest*) are encoded in the host's native string representation, and suitably annotated. This form of code generation, closer to the end-programmer than a hidden run-time JIT system, has been used for domain-specific libraries to offer specializations of programs that are only possible at a time *after* the library has been deployed. The guest representation in this case takes the form of data structures specific to the problem domain, e.g., regular expressions [151], *plans* for Fast Fourier Transforms [53, 54], and sparse [115, 159] and dense [163] matrices for linear algebra routines. In those investigations, a domain-level representation D, specified by the end-programmer, undergoes a process of (offline) code generation, to finally produce code in a host language that is optimized for the case where the input to the program matches D. By specializing to D, the program can take advantage not only of the characteristics of the description D, but of its interaction with the underlying computing environment. The generated code is then linked with programs written in the host language by a traditional tool chain, thereby constituting an off-line, two-stage process. In

this thesis, domain representations are used, but the code generation stage is done on-line.

The host/guest hybrid program source is very commonly seen in database programming, and is quickly becoming prevalent in the realm of 3D graphics programming, where a general-purpose language hosts *shader* program fragments. At suitable times during the applications lifetime, these shaders will be translated and downloaded onto graphics processing units (GPUs) for execution. Below we present an example of this sort of application.

```
program = compile_program("
varying vec3 pos;
void main() {
     pos = gl_Vertex.xyz;
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
", "
varying vec3 pos;
void main() {
     gl_FragColor.rgb = pos.xyz;
}
")
# ...
glUseProgram(program)

def compile_program(vertex_text, fragment_text):
    program = glCreateProgram()
    vshader = compile_shader(vertex_text, GL_VERTEX_SHADER)
    fshader = compile_shader(shader_text, GL_FRAGMENT_SHADER)
    glAttachShader(program, vshader)
    glAttachShader(program, fshader)
    glLinkProgram(program)
    return program

def compile_shader(shader_text, shader_type):
    shader = glCreateShader(shader_type)
```

```
    glShaderSource(shader, 1, shader_text)
    glCompileShader(shader)
    return shader
```

In the code above, Python provides the host programming environment and the guest is a graphics driver's implementation of OpenGL. Also shown is the host language application programming interface (API) for the end-programmer to direct when the guest's source code is to be prepared, compiled and executed, as well as the means of communication with the host program. Other examples of popular APIs for this kind of specification in the realm of high-performance computing graphics programming are Nvidia's Compute Unified Device Architecture (CUDA) language [117] and the Khronos group standard OpenCL [76].

3.2.2 Run-time Code Generation Infrastructure

Regardless of the form of the guest source code fragments, for a host programming environment to support run-time code generation it must provide, at the very least, the following features:

a dynamic loader, which loads objects into the address space of an application, along with any dependencies the plugin may have. It can make use of system's facilities like `dlopen`.

a compilation manager, which determines when a particular code fragment needs to be compiled into a form suitable for loading, and, if it has been compiled already, whether to cache that code or recompile.

an evaluator mechanism, which constitutes the end-programmer interface to the loader and compilation manager. It bridges the data-to-code gap by constructing or identifying a data structure as to be suitable for interpretation within the running program.

This may be done in text form, by using a string as program representation (for example, in hosted SQL queries in database interaction or GLSL shader code for computer graphics applications).

Embedding guest program fragments as string values in the host, despite its immediacy and simplicity, is not ideal, as the contents of a string are effectively out of the control of the host programming language environment. In effect, if the fragment is malformed, the host language is oblivious to it. Moreover, errors are reported in unhelpful ways, and debugging communication points between host and guest is complex. Clearly, a more controlled interface for guest specification is needed.

3.2.3 Domain-specific User-level Guest Code Manipulation

Obviously, the syntax idiosyncrasies of the guest are a determining factor in the actual complexity of the embedding task. Lisp is highly favoured for code generation tasks because its simple, regular syntax blurs the difference between code and data structures[2]. With such a feature, an embedded guest language takes the form of regular host language data structures, and statements in it can be both stated and manipulated uniformly. The code generation/injection subsystem is invoked by the use of a special function, (usually called `eval`) on the guest program representation. `eval` is also used when the guest is represented as host's string literals:

```
function_source = "f(3) + 40"
# ...
x = eval(function_source)
```

As it can be seen from the example, the variable `function_source` holds (as text) a source code fragment, which is later executed via `eval`. The function `eval` was included from

[2]In fact, Lisp is the prototypical *homoiconic* language, where code and data are syntactically indistinguishable.

very early versions of Lisp, and it has made its way to more mainstream programming languages, mostly due to its success in scripting (e.g. Python, Ruby, R and Matlab) and managed-runtime languages (like Java).

Not all guest languages enjoy Lisp's seamless integration with the host, and, as mentioned above, incrementally building strings encoding correct and complete programs is difficult enough. Furthermore, having to deal with dependencies, variables and other bookkeeping that allows communication between host and created subprograms constitutes a cost that tends to out-weight the possible benefits of multi-stage programming. A solution that is both more general and safer is to restrict the guest language to a "safe" subset of the language, or to a completely different language whose safety is known. "Safe" subsets, or specialized languages are known as domain-specific languages (DSL). This approach would typically imply the development of a full language processing infrastructure for the DSL source, translating it to the general-purpose language of the host; this results in two program fragments in the same language that are linked together by the usual means. The best-known example of this form of DSL is the pair `lex/yacc` for lexer/parser generation, which generates code from the DSL specification at compile time. The results of this code generation step and the specification of the host program can then be linked together in using the full general-purpose programming language tool chain, therefore enabling the host to execute the functionality of the DSL specification via the means provided by the host language. This process is illustrated in Figure 3.3

The idea of making use of a full programming language infrastructure to support the creation and execution of guest programs is sometimes known as *language embedding* [27]. With the inclusion of facilities like operator overloading and generic types, string representations are not the only possible way to embed a guest language into a host. Now, languages like C++, Ruby, Python or Java include the necessary flexibility to allow the construction of combinator libraries [71], as well as of *active libraries* [157]. Operator overloading and templates (generic types) allows the use of constructs that differ in semantics from those provided by the host language, and that can be viewed as a way to embed another language. Generic types allow the expression of the rules of assembling

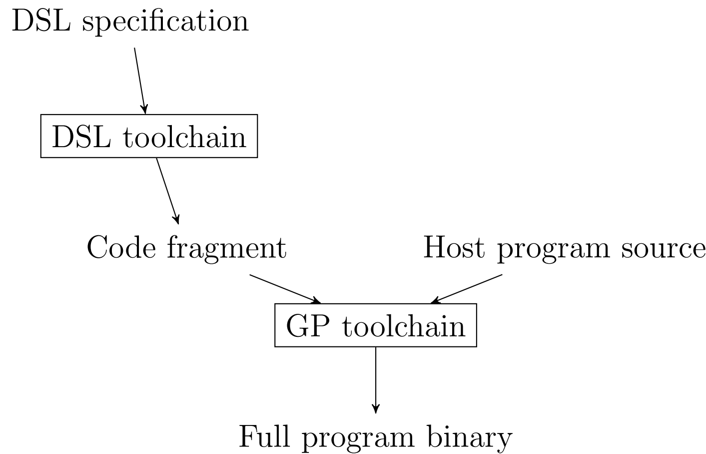


Figure 3.3: Two-stage DSL processing

a code fragment, whereas operator overloading provides a natural way to compose fragments through a non-intrusive notation. All these features can be used to "wrap" the guest language into a host language API, thus blurring the difference between building a guest language fragment and incrementally assembling any other kind of data structure. The following is an example of the use of one such API, constructed in Python around the Low-level Virtual Machine instruction set [149]. The use of the LLVM is relevant to what follows since it can be considered as a more general version of the virtual machine the Intel ArBB library makes use of.

```
my_mod = build_function()
ee = ExecutionEngine.new(my_mod)

# feed the input arguments to 'sum'
ty_int = Type.int()
arg1 = GenericValue.int(ty_int, 100)
arg2 = GenericValue.int(ty_int, 42)

# compile and run
f_sum = my_mod.get_function_named("sum")
```

```
retval = ee.run_function(f_sum, [arg1, arg2])

def build_function():
    my_module = Module.new('my_module')

    # all the types involved here are 'int', which is represented by
    # an instance of llvm.core.IntegerType.
    # To create, use the llvm.core.Type factory class
    ty_int = Type.int() # by default 32 bits

    # Construct function signature
    # The class of functions that accept two integers and return an integer
        is
    # represented by an instance of  llvm.core.FunctionType
    #                        return type, list of input args
    ty_func = Type.function(ty_int, [ty_int, ty_int])

    # Now we need a function named 'sum' of this type.  in llvm-py,
    # functions are not free-standing, but need to be contained in a module.
    f_sum = my_module.add_function(ty_func, "sum")

    f_sum.args[0].name = "a"
    f_sum.args[1].name = "b"

    # We need a /basic block/ for function sum.  A basic block is a set
    # of instructions that end with a terminator (return, branch, &c), i.e.
    # a set of straight-line code.  By convention, the first basic block
    # is called "entry"
    bb = f_sum.append_basic_block("entry")

    # To add instructions into basic blocks we need a 'builder' (an instance
        of
    # llvm.core.Builder) associated to the basic block.
    builder = Builder.new(bb)
```

```
    # The instructions
    # we need to create an 'add' instruction. 'add' returns the sum as a
        value,
    # which we will use as return value
    tmp = builder.add(f_sum.args[0], f_sum.args[1], "tmp")
    builder.ret(tmp)

    # We've completed the definition
    return my_module
\caption{Construct the function ``f(a,b) = { a + b}'' as a guest program
   fragment. }
```

It can be seen that this form of language embedding, despite possibly looking slightly more cumbersome than simple encoding of guest source language in strings, allows for the safer manipulation of guest program fragments, opening the door to user-level manipulation and domain-specific fragment combination. In this chapter, we explore exactly such a use for the generation of state estimators.

3.3 Non-Linear State Estimation with the Kalman Filter

As explained in Section 2.3.2, the task of filtering consists of estimating the hidden state of a dynamical system from a noisy time series. Filtering is a central algorithm in Quantitative Finance, Signal Processing, Time Series Analysis and a multitude of other fields, but in this chapter we concentrate on the filtering algorithms most relevant to the analysis of results of High-Energy Physics (HEP) experiments. HEP studies the fundamental components of matter and radiation, as well as their interactions, thereby addressing questions crucial for the understanding of the Universe. HEP experiments pose unique challenges in design, implementation and data analysis. For one, HEP experiments often have to

sort through the massive data streams produced by particle accelerators. Particle accelerators use electromagnetic fields to induce high momenta on sub-atomic particles, whose collisions among each other generate data that is invaluable to understand matter under extreme conditions. This data often takes the form of *traces*, measurements of various physical aspects of the particle taken along their collision paths at specific detection points (the "stations"). A considerable portion of the data mining that takes place in a HEP experiment is spent in the *track reconstruction* task, which consists of taking the traces and reconstructing the underlying physical process.

On-line track reconstruction is one of the bottlenecks of the pattern recognition task in HEP. This problem has been traditionally broken up in the complementary sub-tasks of *track finding* and *track fitting*. Track finding involves associating a set of readings with the likely trajectory of a specific particle. The "likely trajectory" track finding takes as input is computed by the track fitting task. Specifically, the goal of track fitting is to estimate the state of a particle inside a detector moving under the influence of a magnetic field. For the last twenty or so years the most popular solution to the track fitting problem has been the Kalman filter (KF). It is well known that the KF is only guaranteed to compute the optimal estimator if the dynamics of the system are linear and subject to Gaussian noise. However, these conditions are not met in the track fitting problem, in particular, the dynamics are strongly non-Gaussian due to effects such as multiple Coulomb scattering and energy loss. A proposed solution is the "Gaussian sum filter" (GSF) which runs a bank of KFs to estimate each of the modes of the noise distributions, modeled here as a mixture of Gaussians. But this solution is limited by the fact that the usual implementation of the GSF uses the same distribution for every input dataset. To address this issue, we present in this chapter the Input-adaptive KF (IAKF), which makes use of the dynamic code generation features of modern software frameworks to create a GSF that matches the given (observation) noise distribution. The IAKF further deals with non-linearity by having the generated GSF drive, instead of (linear) KFs, a non-linear filter. Without loss of generality, for this work we use the recently proposed Quasi-Monte Carlo KF (QMC-KF). The generated code is not only tailored to the data, but takes advantage of several levels

of parallelism in multi-core processors.

In a way, the goal of a track fitting solution is to determine the values that best conciliate the experimental readings and the mathematical description of the trajectory. These values, or *state* are usually denoted by $\mathbf{x}_k \in \mathbb{R}^5$, and consist of the exact intersection point of the particle with each of the detectors, the track direction and the curvature. A *measurement* $\mathbf{z}_k$ is taken at each station k, and a collection of stations represents a model of the whole detector. Note that the measurements taken at each station constitute *noisy* projections of the real parameters, which are what the filter is trying to recover. The mathematical setting of this problem is to consider the particle accelerator as a nonlinear dynamic system [72]:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k) + \omega_k \tag{3.1}$$

of which the measurement $\mathbf{z}_k$ is a function, also polluted by noise to form the stochastic "observation" process:

$$\mathbf{z}_k = h(\mathbf{x}_k) + \nu_k \tag{3.2}$$

It can be seen from the notation that the system has been discretized in time with indices k. In the transition between measurement points $k-1$ and k, the state is considered to be subject to *process noise*, which is denoted here by ω_k. Moreover, precise observations are not possible because of the limitations in the measuring model and instruments, a circumstance which is considered by introducing the *measurement noise* ν_k. The precise probability distributions of ω_k and ν_k heavily depend on the application, but they are commonly taken to be mutually independent.

To fit the track we need then to *filter* out the noise, i.e., to calculate the posterior distribution $p(\mathbf{x}_k \mid \mathbf{z}_{1:k})$ at all measurement indices k, where $\mathbf{z}_{1:k}$ denotes a sequence of observations $\{\mathbf{z}_i\}_{i=1}^{k}$. The optimal solution to this problem is given by the recursive Bayesian estimation algorithm [62], which recursively updates the posterior density of the system state as new observations arrive. However, it is worth noting that this recursive solution is only tractable for linear, Gaussian systems, in which case the closed-form recursive solution

to the Bayesian integral equations is the well-known KF. Because in HEP experiments process noise arises from interactions between charged particles and detector materials, the linearity and Gaussianity assumptions the KF relies on are hardly ever met. The measurement noise ν_k is also rarely Gaussian, since in real detectors there is an ever-present possibility of outlying or ambiguous observations. Therefore, approximate solutions need to be applied to this problem.

Approximate filtering for general dynamic state space systems can be roughly categorized into two approaches: deterministic and Sequential Monte Carlo-based. The first approach, which is the focus of this chapter, generalizes the KF, trying to keep its simplicity and well-understood theory [13, 62, 72]. However, these generalizations suffer from the requirement that filters must be manually constructed from the input data, which imposes inordinate effort to the user. Furthermore, typically the same filter is applied to every input dataset, which may degrade the quality of the estimator. To overcome these disadvantages, we present the Input-Adaptive KF (IAKF), an approximate solution to the filtering problem based on the KF where an automatically constructed Gaussian mixture models the measurement noise and a set of non-linear KFs are used for the actual filtering.

3.3.1 Generalizations of the KF

The KF is a prediction/correction scheme with the predicted state and observation being calculated from the estimate at the previous measurement point (or from the prior distribution $p(x_0)$):

$$\begin{aligned} \mathbf{x}_{k|k-1} &= f(\mathbf{x}_{k-1|k-1}, \omega_k) \\ \mathbf{z}_{k|k-1} &= h(\mathbf{x}_{k|k-1}, \nu_k) \end{aligned}$$

where ω_k and ν_k are independent and normally-distributed, with zero mean and covariance matrices Q and R respectively. The mapping f is a stochastic transition kernel from state $\mathbf{x}_{k-1}$ to state $\mathbf{x}_k$ and h is a stochastic non-linear mapping from the current state to the

observation. The state estimate given the observations $\mathbf{z}_{1:k-1}$ is represented by $\mathbf{x}_{k-1|k-1}$, and the *predicted* estimate for the next state given the same trace is denoted by $\mathbf{x}_{k|k-1}$.

These predictions are then updated with the *innovation*, the difference between predicted and actually observed measurement, modulated with a correcting factor K_k, the *Kalman gain*, to render the next state estimate $\mathbf{x}_{k|k}$ and corresponding covariance $\mathbf{P}_{k|k}$:

$$\begin{aligned} \mathbf{x}_{k|k} &= \mathbf{x}_{k|k-1} - K_k(\mathbf{z}_k - \mathbf{z}_{k|k-1}) \\ \mathbf{P}_{k|k} &= \mathbf{P}_{k|k-1} - K_k(P_z)_{k-1}K_k^T \end{aligned}$$

The KF is *optimal* (in the least-squares sense) provided process and measurement mappings are linear, and the associated noise is Gaussian. However, as mentioned above, this is not the case in HEP experiments. Below, we present two general approaches of dealing with non-linearities and non-Gaussianity that are particularly relevant to track fitting.

3.3.2 Dealing with non-linearities

When considering non-linearities, a common simplifying assumption is that the noise distribution for transition and observation models are Gaussian, i.e.,

$$\begin{aligned} p(\mathbf{x}_{k-1} \mid \mathbf{z}_{1:k-1}) &= \mathcal{N}(\mathbf{x}_{k-1|k-1}, \mathbf{P}_{k-1|k-1}) \\ p(\mathbf{x}_k \mid \mathbf{z}_{1:k-1}) &= \mathcal{N}(\mathbf{x}_{k|k-1}, \mathbf{P}_{k|k-1}) \end{aligned}$$

This simplification makes the resulting filtering distribution $p(\mathbf{x}_k \mid \mathbf{z}_{1:k})$ normally distributed as well.

A great variety of non-linear filters make use of this simplification. Among them, the Extended Kalman filter (EKF) is by far the most popular. Despite the advantages in terms of ease of understanding and implementation of the EKF, there is a considerable risk of estimation degradation by its use. This risk can be attributed to the EKF's strategy of linearizing process and measurement equations around the previous estimate. This approach

does not take into account the statistical properties of the noise, and may ultimately cause the divergence of the filter [72].

Alternative approaches go back to the definition of first- and second-moments of the filtering distribution [62]:

$$\begin{aligned}
\mathbf{x}_{k|k-1} &= \int \mathbf{x}_k p(\mathbf{x}_k \mid \mathbf{z}_{1:k-1}) \mathrm{d}\mathbf{x}_k \\
&= \int f(\mathbf{x}_{k-1}) p(\mathbf{x}_{k-1} \mid \mathbf{z}_{1:k-1}) \mathrm{d}\mathbf{x}_{k-1} \\
\mathbf{P}_{k|k-1} &= \int (\mathbf{x}_k - \mathbf{x}_{k|k-1})(\mathbf{x}_k - \mathbf{x}_{k|k-1})^T p(\mathbf{x}_k \mid \mathbf{z}_{1:k-1}) \mathrm{d}\mathbf{x}_k
\end{aligned} \tag{3.3}$$

A commonly-used family of non-linear filters that is based on this problem re-stating is the so-called "sigma-point" family, of which the most prominent member is the Unscented Kalman Filter (UKF) [72]. In Section 3.4 we present the Input-adaptive KF, which makes use of another member of the same family, the QMC-KF algorithm, to construct filters that are tailored to the input data, thereby improving the robustness of a track-fitting system as shown in Section 3.5. The QMC-KF numerically approximates (3.3) using Monte Carlo or quasi-Monte Carlo integration. QMC-KF [62] relies on the approximation

$$\mathbf{x}_{k|k-1} = \sum_{i=1}^{n} f(\mathbf{x}_k^{(i)}) \tag{3.4}$$

where $[\mathbf{x}_k^{(i)}]_{i=1}^n$ is a low-discrepancy point set of the appropriate dimensionality (in this work, as in [62], we use randomized Halton point sets) under some transformation that maps to a Gaussian distribution.

3.3.3 Dealing with non-Gaussianity

Gaussian distributions are not appropriate for the phenomena studied in HEP experiments. Measurements include outliers and ambiguity that introduce tails in the measurement error distribution ν_k, whereas the biggest contributors to process noise, energy loss and multiple

scattering, are highly non-Gaussian. Forcing a Gaussian distribution to describe these effects greatly reduces the amount of information that can be recovered from the true densities, especially in the case of multimodal densities.

A common approach to avoid information loss while remaining within the KF framework consists of modeling the non-Gaussian distributions by *Gaussian mixtures*. For example, measurement outliers can be handled by a Gaussian mixture with a "core" component describing the "regular" measurements, and one or more components describing the outlier-induced tails [55] (e.g., by a mixture of Gaussians sharing the mean but with different covariances). Likewise, ambiguous measurements can be modeled by a mixture of Gaussians with one component per possible value, i.e., with the mean set to the possible value and identical variances, thus concurrently using all possible meanings of the ambiguous measurement. As for process noise, we can model the tails of the multiple scattering for low-energy particles, or the highly asymmetric energy loss of electrons by suitable Gaussian sums [10]. In principle, every distribution involved in the filtering process (state priors, measurement and process noise) can be modeled as a Gaussian mixture. Taking this notion as a guideline, Alspach and Sorenson [13] proposed the Gaussian-sum filter (GSF), where every component of the mixture is propagated and updated by a standard KF. This is to say that the GSF consists of a bank of Kalman filters running in parallel. A detailed specification of a prediction/correction step of the GSF (made up of a combination of Extended Kalman Filters) is presented in Algorithm 3.1. The quantities F_l and H_l used in that description are the Jacobians with respect to the state of the process and observation functions of the system, respectively, where the Jacobians are evaluated at the l-th component of the estimate.

3.4 The Input-Adaptive Kalman Filter

All the filtering techniques described in the above section assume that the same filter will be applied to every input, since parameters chosen for the filter are fixed at program con-

Algorithm 3.1 The Gaussian-sum filter

Require: Estimate $p(x_{k-1} \mid z_{1:k-1}) \approx \gamma^l_{k-1} \sum_{l=1}^{G} \mathcal{N}(x^l_{k-1|k-1}, P^l_{k-1|k-1})$

A-priori state estimate $x^l_{k|k-1} = f(x^l_{k|k})$

A-priori state covariance estimate $F_l P^l_{k|k-1} F_l^T + Q_k$

Innovation covariance $L_l = H_l P^l_{k|k-1} H_l^T + R$

Kalman gain $K_l = P^l_{k|k-1} H_l^T (L)^{-1}$

A-posteriori state estimate $x^l_{k|k} = x^l_{k|k-1} + K_l(z_k - h(x^l_{k|k-1}))$

A-posteriori state covariance estimate $P_{k|k} = P^l_{k|k-1} - K_l H_l P^l_{k|k-1}$

Mixture weight $\gamma^l_k = \frac{\gamma^l_{k-1}\beta^l_k}{\sum_{g=1}^{G} \gamma^g_{k-l}\beta^g_k}$, where $\beta^l_k \sim \mathcal{N}((z_k - h(x^l_{k|k-1})), L_l)$

struction time. This strategy does not take into account the actual system being analyzed, and may therefore result in inaccurate or divergent filters [160]. Here this issue is addressed by dynamically building the filter based on the input data. In this section I present the Input-Adaptive KF (IAKF), a GSF driving QMC-KFs with a preprocessing step of data clustering, followed by a code-generation step where the needed filter is constructed and then run. Interposing a learning preprocessing stage to the filtering task is a standard adaptation mechanism, the novelty of the IAKF is to use the results of such a stage to direct the subsequent construction of a specialized software module. Figure 3.4 shows the conceptual diagram of the IAKF, an Algorithm 3.2 presents the same in pseudo-code form. A detailed description of the components of the system is presented in the following sections. Specifically, the initial clustering and model selection stages are detailed in Section 3.4.1 and the subsequent code generation tasks are described in Section 3.4.2.

3.4.1 Clustering and model selection for the IAKF

The initial step for the construction of an adapted filter is to determine the observation noise distribution as a Gaussian mixture. For this, a sample of the tracks at the first station $z_{1,1:N}$ is read as a training set and assumed to have been generated by a Gaussian

Algorithm 3.2 Input-Adaptive Kalman Filter

Require: Parameter `MAX_MODES`, Cross-section $z_{1,1:N}$

Require: Parameter N is the number of traces, each of length T

for $m = 2$ to `MAX_MODES` **do**

BIC[m] ← Cluster($m, z_{1,1:N}$) {*Refer to Section 3.4.1*}

end for

G ← BICModelSelection(BIC[m]) {*Best-fit number of modes for the observ. noise MoG*}

GSF ← GenerateCode(G) {*Refer to Section 3.4.2*}

for $t = 1$ to N **do**

for $k = 1$ to T **do**

$x^{l'}_{k|k,t}, P^{l'}_{k|k,t}, \gamma^{l'}_{k,t} \leftarrow \text{GSF}(x^{l}_{k-1|k-1,t}, P^{l}_{k-1|k-1,t}, \gamma^{l}_{k-1,t})$, for $l = 2, \ldots G$ {*Algorithm 3.1*}

$x^{l}_{k|k,t}, P^{l}_{k|k,t}, \gamma^{l}_{k,t} \leftarrow \text{Consolidate}(x^{l'}_{k|k,t}, P^{l'}_{k|k,t}, \gamma^{l'}_{k,t})$ {*Algorithm 3.3*}

end for

end for

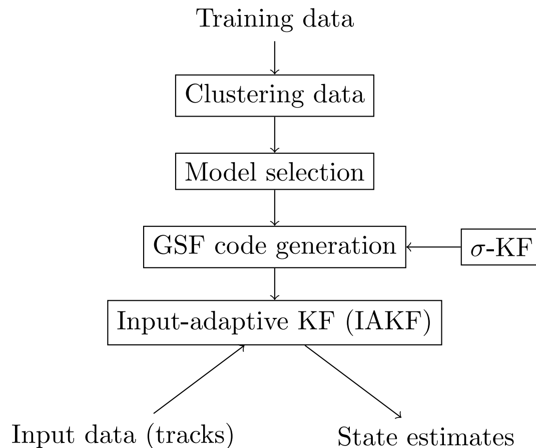


Figure 3.4: Conceptual diagram of the IAKF

mixture density

$$p(x_0) = \sum_{l=1}^{G} \gamma_l \mathcal{N}(\mu_l, \Sigma_l) \tag{3.5}$$

The Gaussian mixture models the notion that the measurement noise is multimodal due to outliers. It is premature to decide *a priori* how many terms G the mixture is to have because of the lack of information on the outlier distribution. Therefore we run a parameter-estimation task on several candidate component counts (in Algorithm 3.2 denoted by `MAX_MODES`) and use a model selection criterion to choose the best fit among them. The parameter estimation procedure determines the values for mixture parameters and proportions. Having determined the appropriate number of components, we generate the corresponding QMC-KF instances, which will be "baked into" the final filter. The resulting filter is run on the input to perform the actual state estimation. Note that the mixing proportion of the Gaussian sum needs to be re-weighted at every iteration to maintain an accurate estimate. This re-weighting is specified in Algorithm 3.1.

To estimate the parameters of the initial noise density the expectation maximization (EM) algorithm is used [100]. EM is a maximum-likelihood technique that can estimate

the parameters Θ of a distribution of a given form that best corresponds to the data. In the case of a Gaussian mixtures, the parameter vector Θ corresponds to

$$\Theta = (\theta_1, ...\theta_G, \gamma_1, ...\gamma_{G-1}) \tag{3.6}$$

and $\theta_l = (\mu_l, \Sigma_l)$ for each of the G components of the mixture. The G mixture weights γ_l are subject to the additional constraint that $\sum_{l=1}^{G} \gamma_l = 1$, so only $G - 1$ weights need to be estimated. As the computational load of the GSF is directly related to the number of components in the mixture, we run EM for a low component count (2 to 5). The EM algorithm can estimate the parameters of a Gaussian mixture of any (a-priori) given component count, as long as the algorithm is provided with enough samples for each component. The model selection criteria we use to determine how many components the Gaussian mixture should have is the Bayesian information criterion (BIC) [100].

In the GSF, care must be taken to control the possible combinatorial explosion of Gaussian terms in the posteriors. As the IAKF has a fixed number of components, when any of the component estimates becomes multi-modal, the smaller components are combined in order to maintain a constant component count. The combination algorithm is taken from the RAVE track fitting toolkit [129], and is described in Algorithm 3.3. In that description "closest neighbour" can be chosen according to a user-defined distance metric. A common choice is the Kullback-Leibler distance.

3.4.2 Run-time code generation of the IAKF

I have built both Python and ArBB implementations of the IAKF, where the number of components `NUM_KF` that constitute the mixture-of-Gaussians approximation of the non-Gaussian noise is determined by the clustering procedure described in [100] This information is subsequently used to generate as many QMC-GSF instances as necessary, in a manner sketched by:

```
for k = 0; k < NUM_KF; k++) {
  _for( i = N - 1, i >= 0, i-- ){
```

Algorithm 3.3 Combining components in a Gaussian mixture

Require: Input MoG $\sum_{i=1}^{G'} \gamma_i \mathcal{N}(\mu_i, \Sigma_i)$

Ensure: Output MoG $\sum_{i=1}^{G} \gamma_i \mathcal{N}(\mu'_i, \Sigma'_i)$, where $G \leq G'$

  `new_comps` $\leftarrow$ Sort $\gamma_i \mathcal{N}(\mu_i, \Sigma_i)$ by γ_i

  `cur_count` $\leftarrow G'$

  while `cur_count` $> G$ **do**

    $g_1, g_2 \leftarrow$ Choose mode with smallest weight γ and its closest neighbour

    Remove g_1, g_2 from `new_comps`

    $g_{new}.(\gamma, \mu, \Sigma) \leftarrow g_1.\gamma + g_2.\gamma, \text{WeightedMean}(g_1, g_2), \text{WeightedCov}(g_1, g_2)$

    `new_comps`, `cur_count` $\leftarrow$ `new_comps` + g_{new}, `cur_count` -1

  end while

  Construct MoG from `new_comps`

```
    // ... magnetic field setup ...
    filter(ts,ss,xInfo, ts.hitsX2.row(i), w,T,C);
    filter(ts,ss,yInfo, ts.hitsY2.row(i), w,T,C);
    for( int j = 0; j < 3; j++ ){
      H2[j] = H1[j];
      H1[j] = H0[j];
    }
    z2 = z1; z1 = z0;
  } _end_for;
}
```

The `filter` kernels are wrapper functions over multiple parallel function applications, which result in a parallelization strategy both over tracks and over concurrent QMC-KFs. Below we present the specific implementation for two host/guest language combinations, and in Section 3.5 we present the results of some numerical experiments designed to determine the effect of the IAKF strategy on the overall performance of the track fitting task.

Code generation in Python

For the purposes of this chapter, we make use of the `codepy` [79] code generation framework, designed specifically for generation of OpenCL and CUDA-based modules that can be automatically connected to the Python runtime. Note that this is not the only code-generation framework available for Python (a very similar one is Pyinline [140]), but it is the one that best adapts itself to the task of dynamic code generation. A simple example of the use of `codepy` is shown below, where, for the purposes of illustration the generated module is the Monte Carlo calculation of π.

```
from cgen import *
from codepy.bpl import BoostPythonModule

foo_decl = FunctionDeclaration(Value("double", "pi_montecarlo")
                               [ Value("unsigned int", "N") ])

body = [ Initializer(Value("unsigned int", "cnt"), 0) ]

loopbdy = []
loopbdy.append(Initializer(Value("double", "x"), "(double)rand()/RAND_MAX"))
loopbdy.append(Initializer(Value("double", "y"), "(double)rand()/RAND_MAX"))
loopbdy.append(Initializer(Value("double", "rad"), "sqrt(x*x + y * y)"))
loopbdy.append(If("rad <= 1.", Block([Statement("cnt++")])))
loop = For("unsigned int i = 0", "i < N", "i++", Block( loop_body ))

body.append(loop)
body.append( Statement("return 4 * ((double)cnt / N)") )

foo = FunctionBody(foo_decl, Block(body))

mod = BoostPythonModule()
mod.add_function(foo)
from codepy.jit import guess_toolchain
```

```
cmod = mod.compile(guess_toolchain(), wait_on_error=True)

print cmod.pi_montecarlo(int(1e7))
```

The details of the code generation process (i.e., the box marked "GSF code generation" in the conceptual diagram in Figure 3.4) are hidden from the end-user by a combinator-like interface, that allows the specification of, for example, a Gaussian-Sum Filter composed of two QMC-Kalman filters as:

```
x0s = [ zeros(2), array([0.4, 0.5]) ]
P0s = [ eye(2) * 0.1, eye(2) * 20.2]
gsf_spec = alpha_1 * QMCKF(x0s[0], P0s[0]) + alpha_2 * QMCKF(x0s[1], P0s[1])
```

Once the filter has been fully specified, it is necessary for the user to specifically generate the backend code, which can then be used in a traditional-looking filtering loop (which corresponds to the bottom "IAKF" flow in Figure 3.4):

```
gsf = gsf_spec.generate()

s = MoGNoiseMixin(NonLinearSystem(x0, deltat), means, covs, mix)

for i in range(n_iter):
    obs = s.get_observation()
    gsf.step(obs)
    s.step()
```

The uses of `NonLinearSystem` and `MoGNoiseMixin` instances in the code refer to a state-based simulator of the system to be estimated and a 'mixin' class that injects noise of certain characteristics to that system, respectively.

Code generation in ArBB

At a lower level of abstraction, we apply the same principles of run-time code generation in a C++ implementation of the IAKF. To this end, we make use of Intel's Array Building Blocks (ArBB), formerly Ct, library [7]. ArBB is a retargetable dynamic compilation framework, whose main purpose is to facilitate the exploitation of modern multi- and many-core architectures. It provides a set of implicitly data-parallel collection data structures and computational patterns (including *map*, *reduce* and *prefix sum*). A programmer can make use of this abstract notation and target several levels of parallelism present in multi-core homogeneous systems, as well as potentially heterogeneous accelerator-based many-core architectures (for example, GPUs) and computing clusters.

A more complete description of ArBB is best found elsewhere [103]. Here, we focus on the code-generating aspect. ArBB is a two-stage system, where the programmer works within the usual confines of the C++ language (the *uncaptured* environment), but specifies the functions that are to be run in parallel using library-provided data types, functions and control flow (the *captured* environment). The clearest way to describe this architecture is through a simple example:

```
// kernel definition (captured)
void plusone_kernel(arbb::f32 x, arbb::f32& y)
{
  y = x + arbb::f32(1);
}

const unsigned int vec_length = 128;
float x[vec_length];
arbb::dense<f32> vec_x;
arbb::dense<f32> vec_y;

// ... fill in array a ...

arbb::bind(vec_x, x, vec_length); // bind to captured environment
```

```
// now it is possible to operate in captured environment
arbb::map(plusone_kernel)(vec_x, vec_y);

// vec_y holds the result of vec_x + 1, which can now be brought back to the
// uncaptured environment
```

In the above program fragment, `float` is the usual C++ type and `f32` is the corresponding embedded-language version. Functions (aka *kernels*) that have been written in terms of ArBB types will be JIT-compiled and executed on demand. In other words, ArBB types and functions are not executed in the same way as the surrounding C++ code is. Uses of ArBB types and functions are *retained*, i.e., stored by the compilation manager, and compiled and executed should the host program later require the result.

The rationale for this two-level architecture is that captured code can be compiled to make maximum use of the system it is running on: vectorized ALUs, multiple processors, accelerators and other system configurations.

Besides analogs of C++ types, ArBB also provides mirrors of C++ control structures, in the form of the 'keywords' `_if`, `_for`, `_while` and `_do`. These allow serial constructs to be used in a parallel environment. For example, the kernel:

```
void nr_sqrt_kernel(arbb::f32 a, arbb::f32& s)
{
    _if (a < 0) { return -1; } _end_if;
    arbb::f32 sprev = arbb::f32(0);
    s = a;

    _while (arbb::abs(xprev -x) > tol) {
        sprev = s
        s = 0.5 * (sprev + a / sprev)
    } _end_while;
}
```

can be the argument of a `map` operator to find the square root of each (positive) element in an ArBB array. Each invocation of `nr_sqrt_kernel` will execute concurrently.

An interesting side effect of this two-level execution approach can be seen when C++ control flow is used in *captured* mode. The following code

```
unsigned int unroll_factor = 4;

void fill_vec_kernel(arbb::dense<arbb::f32>& v, arbb::f32 val)
{
    for (size_t i = 0; i < unroll_factor; i++) {
      v[i] = val;
    }
}
```

has the same effect as unrolling the loop:

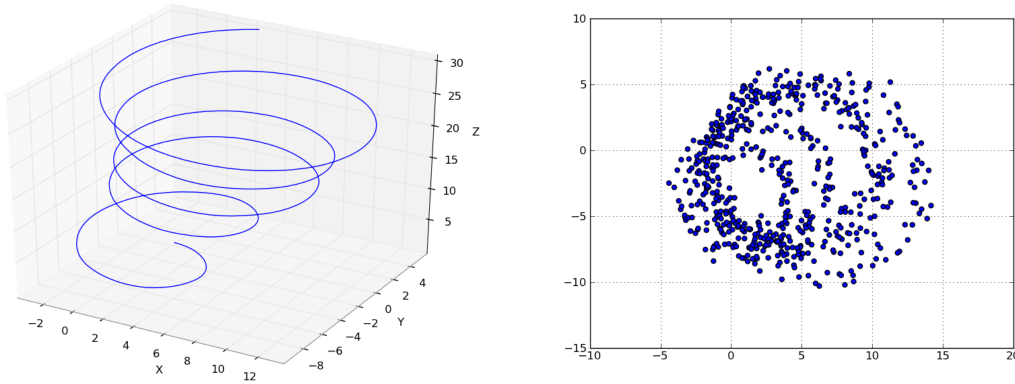
```
void fill_vec_kernel(arbb::dense<arbb::f32>& v, arbb::f32 val)
{
 v[0] = val; v[1] = val; v[2] = val; v[3] = val;
}
```

3.5 Numerical Evaluation

To validate the computational performance of the IAKF implementation, the system is tested on simulated data. I use the methodology proposed in [60] to simulate the transit of a charged particle in a magnetic field. In this approach, the function f takes the form of a 4th-order Runge-Kutta solution (with fixed-step size) to the equation

$$d\mathbf{p} = \kappa q(\mathbf{v} \times \mathbf{B})ds/|\mathbf{v}|$$

where $\mathbf{p}$ is the momentum of the particle, q its charge, $\mathbf{v}$ its velocity, $dt = ds/|\mathbf{v}|$ the trajectory length, $\mathbf{B}$ the magnetic field, and κ is a constant. The state is the 5-tuple $(x, y, t_x, t_y, q/|\mathbf{p}|)$ where $(x, y)_z$ denote the intersection of the trajectory with detecting surface z, $t_x(z) = dx/dz$ and $t_y(z) = dy/dz$ indicate the particle's direction at that point. The observation function h mimics the way silicon micro-strip detectors carry out measurements, projecting the x, y-coordinates at the intersections with the stations z. An illustration of a simplified version of the above system is shown in Figure 3.5. The simplifications consist of using an homogeneous magnetic field and a (uni-modal) Gaussian noise for system noise. The dynamics of the system are encoded in the Runge-Kutta solver described above.



(a) Dynamics process f (motion of charged particle in a magnetic field)

(b) Measurement process h

Figure 3.5: Non-linear functions f and h for track fitting

The remainder of this section includes a battery of empirical tests to assess the performance, benefits and overheads of the IAKF. Following that I discuss the limitations of the method.

3.5.1 Empirical performance analysis of the IAKF

In this section I conduct a series of experiments to explore the behaviour of different filtering strategies under a variety of circumstances with the goal of determining the benefits and overheads of the IAKF relative to popularly-used filtering solutions.

Filtering performance on uni-modal, non-linear system

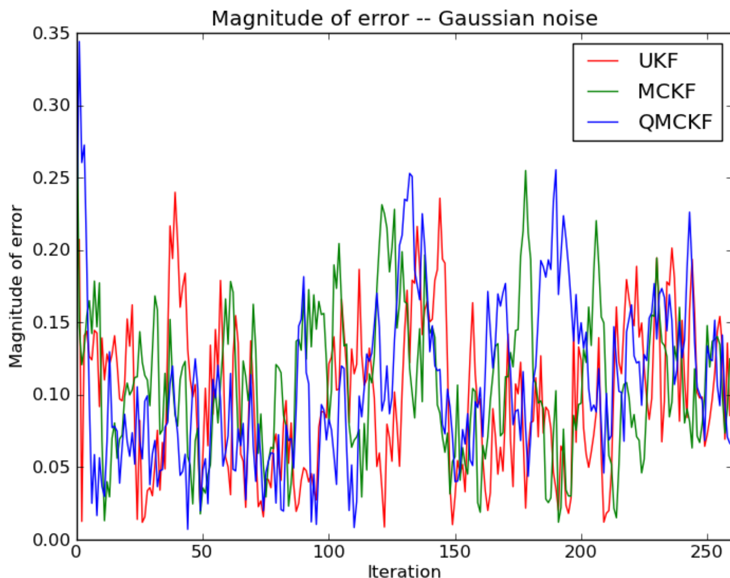


Figure 3.6: Magnitude of the filtering error of the UKF, MCKF and QMCKF (system under Gaussian noise)

Figure 3.6 shows the difference of performance of uni-modal (i.e, not Gaussian-sum) filters in a state estimation task of a system under Gaussian noise (specifically, under a covariance of 0.1). Three filters are tested: the Unscented Kalman filter (UKF) and the IAKF driving both QMC-KFs and traditional Monte Carlo filters (i.e., a non-linear filter making use of Monte Carlo integration of the Kalman filter recurrences). Our QMC-KF

filters are fed by a 1024-randomized Halton sequence of state dimension 5. The particular randomization method is to use randomized linear scrambling [90]. Note that in this and the following experiments we measure statistical performance by calculating the L_2-distance between estimated and true states. As all the data arises from a physical system simulation, we have access to the "true" state, as it comes out of the mathematical description of the system without noise. Performance is reported in the "y" axis of the figures. The difference of performance between the Unscented Kalman Filter (UKF) and the QMC-KF (using 1024 samples) is not statistically significant (the p-value corresponding to the null hypothesis is 0.5191). This undifferentiated performance is expected, as the noise the system under consideration is subjected to is Gaussian. For this case, these filters are largely equivalent. This test is carried out by taking the average estimation error over all iterations of a particular run. The IAKF proposed in this work makes use of the QMC-KF as a building block, but it is not limited to this choice, as any of the above non-linear filters could serve as a plug-in replacement.

Impact of multi-modal noise

When the system is not under single-Gaussian observation noise, but under the influence of a mixture-of-Gaussians (MoG) observation noise, the performance of uni-modal filters suffers in comparison to that of filters that contemplate multiple-mode noise distributions. This effect is tested on a system under MoG observation noise with the following characteristics:

$$\begin{aligned} x_0^l &= [(0,0),(5.5,5.5),(15.5,15.5)] \\ \sigma_0^l &= (0.1,20.5,10.5) \\ \gamma^l &= (0.5,0.4,0.1) \end{aligned}$$

The system in question is depicted in Figure 3.7, where histograms of the MoG-noise distribution are plotted besides (to the right) a simulated trajectory (on the left). Outliers

are simulated by incorporating wide-Gaussians at some distance from the mean of the real state, thereby forming clusters that make the noise distribution multi-modal.

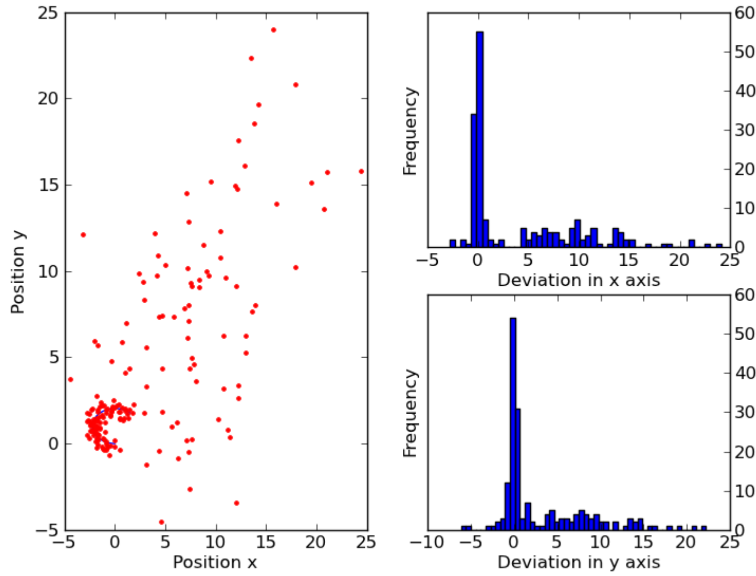


Figure 3.7: The cross-section of a track generated by a system under mixture-of-Gaussians observation noise.

The comparative performance of the UKF, QMC-KF (both uni-modal) and IAKF (multi-modal) is illustrated in Figure 3.8. The IAKF (driving QMC-KFs) shows on average an about 1.5 times estimation error improvement over a QMC-KF, to a high level of statistical significance (the null hypothesis is rejected by a p-value of 0.000226 on a two-sample t-test). In this way, the notion that the filtering task on systems under MoG noise is best carried out by specialized filters is verified.

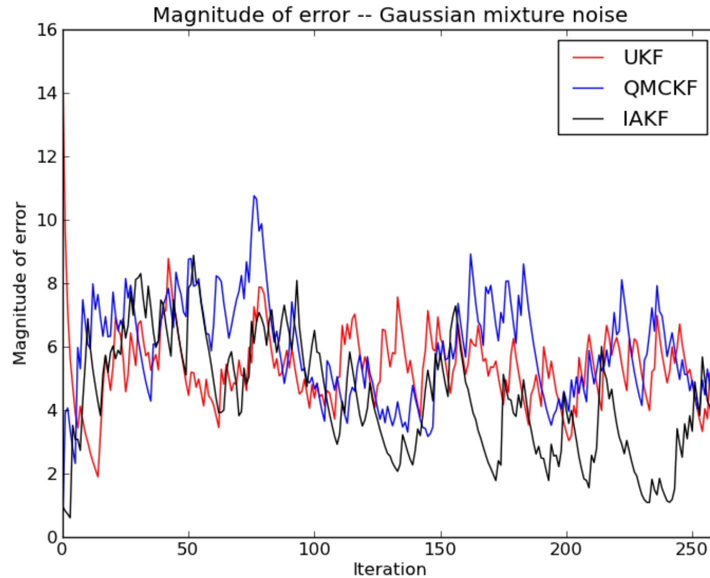


Figure 3.8: Magnitude of the filtering error of the UKF, QMC-KF and IAKF (system under MoG Gaussian noise)

Adapting vs. not adapting to the mode count

I now turn to the study of the effect of a filter mis-specification, i.e., running a GSF with the wrong number of components. For this experiment, I use a 3-component MoG to inject in the system, and test the performance of Gaussian-Sum Filters fixed to look for 2-, 3- and 4-components respectively. As is shown in Figure 3.9, a filter that is specified correctly, which is, using an accurate number of components, shows on average a 2.43 times performance improvement over one that "misses low", i.e., that has less components than it should (in the figure, the 2-component GSF filter), and a 2.31 times average improvement over one that "misses high" (uses more components that actual noise does). The filter that best matches the system is shown in blue in the figure. One of the advantages of the IAKF is that this kind of mis-specification is avoided. In the IAKF approach, several candidate

components are tested, and the best one (in this case, the 3-component GSF) is selected and automatically generated at run-time.

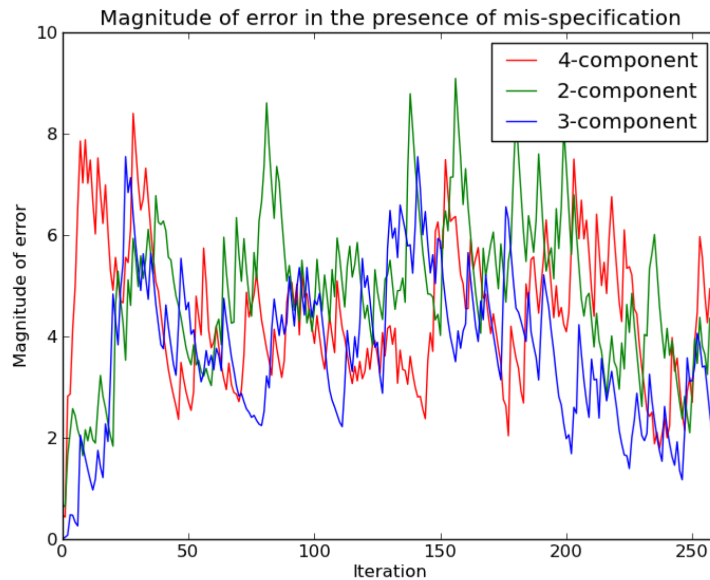


Figure 3.9: Error as a result of mis-specification

Running times of adapted vs. fixed code

Thus far I have been exploring the statistical performance benefits of the adaptation phase to the filtering process. Now I study the benefits that this adaptation can have in the running-time of the resulting filter. In the case of the Python-based implementation the comparison of running times of the adaptive vs. the fixed-filter variations is shown in Figure 3.10. Unsurprisingly, the running time is much shorter on the variation that uses run-time code generation (the speedup factor is about 18 for the presented run), as the generated code in question is compiled to a much more efficient language. The improved running time is then largely attributable to the difference between execution models (the compiled

C++ vs. the interpreted Python). This gain is obtained despite the costs involved in copying data across the Python Virtual Machine to the C++ runtime, and the fact that no compiler optimizations have been specified. Also, no optimized linear algebra routines have been used on the C++ side. Note that the time incurred in compilation/linking is not considered in the report as `codepy` includes a caching feature that avoids unnecessary recompilation. The measurement was taken on a"warm" run, where the involved code generation/linking phase had already been performed and did not need to be repeated.

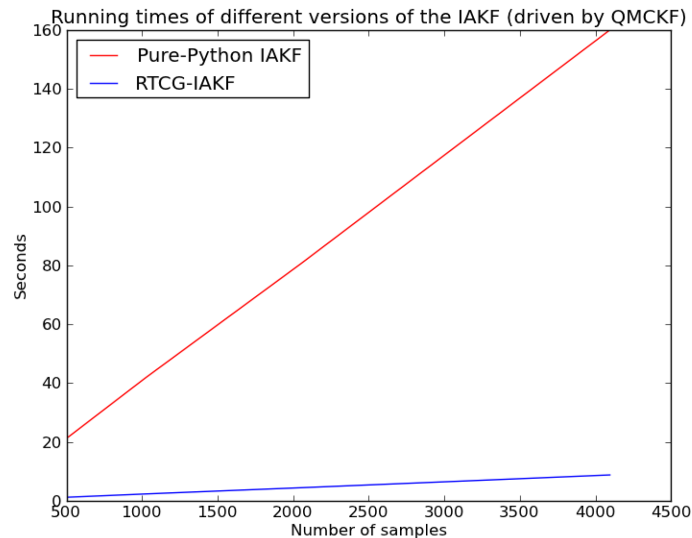


Figure 3.10: Running time of Python implementation of the IAKF for 1024 iterations, at differing sample count

Comparative performance of generated code on "static" computational environments

As stated above, run-time code generation is but one approach to adaptability. The IAKF algorithm can be implemented without using this technique. In this case, the number

Component count	3	5	10
Speedup	1.0285	1.0349	1.022
p-value	0.0189	0.0023	0.039

Table 3.1: Speedups of loop-unrolled w.r.t. loop-based pure C++ IAKFs

of modes of the observation noised distribution is calculated as in Section 3.4.1, but the resulting GSF is computed via a "for" loop. On the other hand, a solution that makes use of run-time code generation essentially expands ("unrolls") this loop into a straight-line code block. In this section, I compare both implementation options in the context of the IAKF. For this I study two implementations: a pure C++ IAKF and a hybrid Python/C++ variation.

For this experiment, I set the trace number to be 100, each trace consisting of 500 iterations. I compare the relative performance of both IAKF variations when the number of modes in the noise observation distribution is 3, 5 and 10. Table 3.1 shows the behaviour of the pure C++ implementations. The code-generation approach shows some speedup in this scenario relative to the loop-based technique. These gains are, however, slight. Since the performance differential between both methods is small, I also report the statistical significance in the form of the p-value of a two-sample t-test to verify the effect of the change. The fact that the codegen strategy does not result in significant benefits in this case is not surprising. On a "static" computational environment, i.e., when the filtering library has been developed in the same environment as the filters to be executed, an interposed codegen stage is unlikely to bring big benefits to the table, as the quality of the code in both implementations is similar. Moreover, loop unrolling is a standard compiler optimization and is likely to have been carried out anyways.

Note that a static computational environment does not preclude a codegen solution to achieve clear performance gains over its loop-based counterparts. These benefits, however, do not derive alone from the loop unrolling, but from avoiding any additional call overhead the GSF may be subject to. To quantify the achievable gains in this situation, I carried

Component count	3	5	10
Speedup	3.110	4.8771	8.46883

Table 3.2: Speedups of loop-unrolled w.r.t. loop-based Python/C++ IAKFs

out an experiment with the same settings as the one above, but in the context of a hybrid Python/C++ IAKF. In this case, the loop-based filter consists of a Python driver looping over as many C++-based KFs as number of components the mixture of Gaussians has been estimated to have. Each of these calls carries some overhead. In contrast, a run-time code-generated solution unrolls this loop in the C++-side, and therefore presents a specialized GSF filter that only requires one call per iteration from the Python side. Table 3.2 summarizes the relative performance, and shows significant benefits of using a codegen-based approach. Note that, for the Python/C++ hybrid implementation studied here, the code generated is serial; further gains could be achieved by having the codegen module create parallelized code, an interesting topic for future investigation.

The overhead of run-time code generation

To investigate the code construction time cost I compare the performance of a pure-Python implementation of the GSF-QMC-KF with 3 modes on a 180-iteration run with the equivalent run-time generated configuration that makes use of `codepy`. A summary of an average of five trials is provided in Table 3.3. It is worth mentioning that the construction time is a one-time cost, so running multiple traces through the generated IAKF will not need to incur in re-compilation overhead. For the problem of track fitting in particular, and time-series filtering in general, the most common scenario is to run a filter over multiple (oftentimes many) traces, which amortizes out the construction cost of the customized program. In mathematical terms, the total run-time savings by using run-time code generation is:

$$N \times (T_{FC} - R_{CG}) - C_{CG} \tag{3.7}$$

Version \ Running time (sec)	Construction	Running	Total
Codegen via `codepy`	1.453	8.72	10.173
Pure Python	0	159.97	159.97

Table 3.3: Comparative performance of "fixed" vs generated-on-the-fly GSF-QMC-KF

where N denotes the number of work items in general (for track fitting in particular, this would be the number of traces to process), T_{FC} is the total time per work item for the "fixed code" version, R_{CG} is the running time (not counting construction time) for the run-time code generated version, and C_{CG} is the construction time for that same version.

From the expression (3.7) above, it is easy to see that the run time savings come mainly from the difference between running times of the different implementations. However, it is important to realize the crucial facilitating role of run-time code generation. This mechanism enables the system to build code that is better performant than that of the host language. Without this enabler, the only way to make best use of computational resources is to resort to a platform-specific module if such exists. Run-time code generation sidesteps this very stringent requirement by building such a module on the fly.

It is relevant to note that the variant of run-time code generation that I use in this chapter and other places in the thesis is a template-based metaprogramming system. However, it is pertinent to clarify that the term *metaprogramming* has of late taken to mean *template-based metaprogramming* in the C++ community. In this latter sense the term "metaprogramming" takes the more restricted meaning of the kind of code generation that is made possible by *C++ templates*. While powerful and interesting in its own right, C++ template metaprogramming is an altogether different mechanism than the one described here. In this chapter and those that follow, I have only made very restricted use of C++ template facilities, and concentrated on other forms of code generation.

3.5.2 Limitations of the IAKF

The IAKF belongs to the family of deterministic, approximate, non-linear filters. As such, it is built upon the framework of the Kalman Filter. This framework is a very efficient tool for dealing with linear systems under Gaussian noise. With the appropriate extensions and corrections (e.g. the GSF), this framework can also adequately handle non-linear systems subject to non-Gaussian noise, as long as this non-Gaussianity can be well approximated by a mixture of Gaussians. However, when noise distributions are beyond the MoG representation, more general filters are better suited to the filtering task than the GSF. The IAKF generates a specialized GSF implementation, and therefore inherits the limited ability to handle strongly non-Gaussian noise. When noise distributions are not suitable for MoG form, it is better to use filters like the particle filter [35]. The adapter module of the IAKF can be extended to warn the user if none of the candidate component counts of a MoG fits the input noise within a user-defined threshold, giving the user the option to carry out the filtering or resort to other methods.

3.6 Other Approaches to KF Code Generation

The automatic generation of data analysis programs has been explored for at least a decade. For example, the AutoBayes program synthesis system [49] generates C++ code from a declarative specification of the statistical model via deductive synthesis directed by code templates. As such, it most resembles the `lex/yacc`-approach of DSLs described above, where a separate programming language infrastructure is required.

Another example of a system that synthesizes KF and variants code is AutoFILTER [132], which similarly to AutoBayes, outputs C++ code from a textual specification specialized to the description of noise distributions and differential equations. An interesting variation that AutoFILTER includes is that it links against the libraries from the Octave linear algebra system, within which its output is supposed to be used. This illustrates another aspect of code generation within C++, its interoperability.

In contrast to the above-mentioned systems, our work only requires a standard C++ compiler. Furthermore, the generated code under ArBB takes full advantage of the parallel features of the architecture the program is running on. On the other hand, significant work has gone into the automated verification and certification of the code generated by AutoBayes, an aspect of considerable importance, which is not covered here. The interoperability aspect of AutoFILTER is also a straightforward addition to our system.

3.7 Summary

In this chapter I propose the "code factory" approach to adaptation, an idea that uses run-time code generation for domain-specific specialization of programs. To give concrete form to this technique, I introduced the Input-Adaptive Kalman filter (IAKF), a member of the deterministic, approximate, non-linear filter family. In contrast with traditional methods, the IAKF adapts to the input, running as many filters as necessary to best fit the input data. This feature, validated by numerical results, makes its estimates more accurate. Furthermore, the IAKF is more robust to different input data than its non-adaptive counterparts. To implement the IAKF, we make use of the run-time code generation and compilation afforded us by modern parallelism libraries and scripting languages. It is my contention that furnishing the end-programmer with the ability to tailor the program in data-driven tasks, such as inferencing, allows for simple and straightforward implementations that are also able to better cope with realistic scenarios. Having a general framework that facilitates improved running times can enable the use of computationally-intensive algorithms in realistic models and data sizes whose complexity may be too great for fixed-code solutions.

Chapter 4

Functional ANOVA decomposition for QMC algorithms on DBNs

The study of numerical integration is central to the transaction of modern day Science and Engineering, and therefore motivates the effort into techniques that may improve its performance. These studies vary in focus from general algorithmic improvements to the use of newly-available hardware platforms, to the study of classes of integrands that may be subject to treatment by numerical integration. An interesting development in this latter endeavor, especially when dealing with high-dimensional integrands, is the study of *sensitivity*. Sensitivity analysis is the study of how changes in subsets of inputs to the model relate to changes in its output. The information of what input subsets contribute the most to output changes can be exploited by numerical integration techniques. Since the integrand is one of the last pieces of information into a numerical integration framework, having such a framework adapt to it can result in improved performance.

In this chapter I apply the functional ANOVA decomposition [136], a global sensitivity analysis technique, to determine the set of input variables that contribute the most to the variance of a given statistical model. Once these "important" input variables have been identified, I construct customized sampling sequences that improve the efficiency of

simulation-based algorithms. This task is carried out with the help of the technique of *quasi-regression*. Furthermore, I take advantage of run-time code generation facilities of modern software libraries to construct these enhanced-sampling driven routines on the fly, to great improvement in statistical performance as measured by variance reduction. This method is validated both in application problems, specifically from the field of Financial Engineering, and on synthetic functions.

4.1 Introduction

It is well-known that the performance of quasi-Monte Carlo techniques are sensitive to the features of the integrand and to the effect of "important" inputs in the integrand's domain, i.e., those input variables which exert greater influence in the model output. This stands in contrast to "traditional" (pseudo-random number driven) Monte Carlo methods, which are more resilient to these features. As an illustration of this fact, studies on some financial applications have reported improvements of QMC over MC inconsistent with their relative asymptotic errors [124]. These results are now commonly accepted as the effects of positive interaction between the low-discrepancy sampling sequence and the integrand. In particular, QMC sampling sequences often have especially well-distributed projections over the important lower-dimensional sub-space of the integrand. In this chapter, I apply the functional ANOVA decomposition, a global sensitivity analysis (GSA) technique, to identify this important set of input variables. I then make use of the technique of quasi-regression [14] to approximate the function and construct QMC sequences that work well in the function approximation, which I then use in the original integrand.

The custom-made sampling sequences results in QMC estimators with lower variance and general improved efficiency of simulation-based algorithms. To the author's best knowledge, no similar work was been done to date. This novel idea can serve as a foundation for methods to exploit late-arriving pieces of system specification in a more general setting.

For software libraries that support the implementation of simulation algorithms, the

strategy proposed above is difficult to accommodate, since the characteristics of the integrand are part of the information that arrives long after the sampling sequences have been designed and implemented. However, a very common use case is that the integrand, most likely a statistical model to be studied under simulation is specified by a domain expert, an "end user" programmer. End user programmers have only limited access to the inner workings of the simulation library, and such libraries are often unaware of the purpose of end programmers, but in the most general of terms. To address this issue, I take advantage of run-time code generation facilities of modern software libraries to construct enhanced-sampling driven routines on the fly. This strategy results in increased flexibility and shorter running times. I test this method in two kinds of problems: the valuation of a path-dependent financial instruments, and synthetic test functions.

This chapter is structured as follows. In Section 4.2 I briefly review quasi-Monte Carlo algorithms, emphasizing their application in the solution of the option valuation. Also, I go into some detail into the construction of the Sobol' sequence, which is the sampling scheme that I use for demonstration purposes. Section 4.3 delves into the topic of *sensitivity analysis*, a technique that identifies those input variables that contribute the most to the total variance of the problem. Sensitivity analysis can be combined with QMC for greater statistical efficiency, and I describe this combination to some detail. In this section I also introduce the technique of quasi-regression [14], an approximation method that represents high-dimensional integrands through a linear combination over a basis of orthogonal functions, usually the tensor product of low- or single-dimensional functions. The coefficients in the approximation are related to the ANOVA decomposition, and efficient estimators of the components can inform the construction of efficient estimators of the original integrand. Section 4.4 describes the design and implementation of the functional-ANOVA-driven QMC constructions. Section 4.5 studies its performance in simulated experiments, to finally conclude in Section 4.6.

4.2 QMC for Simulation and Inference

As explained in Section 2.5, the use of *low-discrepancy* point sets to drive simulation algorithms instead of the traditional pseudo-random number stream has led to the field of quasi-Monte Carlo (QMC) methods [90]. QMC methods have been successfully applied in a large number of fields, like Finance [123, 124] and Computer Graphics [74]. In particular, QMC methods were introduced in the academic literature for the numerical valuation of options in the early 1990s [68], and have since become a mainstay for both theoreticians and practitioners alike, and are usually included in undergrad textbooks that treat financial Monte Carlo calculations in any extent. Furthermore, it was their application in Finance that first illustrated the unexpected effectiveness of *randomized* QMC in high-dimensional problems [124].

Many low-discrepancy constructions have been used for numerical integration. Among these, the better known are those by Hammersley, Halton, Faure, Sobol' or Niederreiter [90]. Each of these sequences has a different set of construction parameters that can be "tuned" according to different concerns, to result in better simulation performance. In the remainder of this chapter, I will concentrate on the Sobol' sequence, although the techniques outlined here apply in a more or less direct way to other constructions as well.

A sequence of n points P_n based on the Sobol' sequence of dimensionality s, i.e. $P_n = [\mathbf{u}^{(i)}]_{i=1}^{n}$ where $\mathbf{u}^{(i)} = (u_{i,1}, u_{i,2}, \ldots u_{i,s}) \in \mathbb{R}^s$ is built by setting the jth coordinate ($j = 1, \ldots s$) of the ith member of the sequence to

$$u_{i,j} = i_0 v_{j,1} \oplus \ldots \oplus i_{k-1} v_{j,k}$$

where i_l is the lth digit of the binary expansion of i (typically the elements $u_{i,j} \in \mathbb{Z}[1, 2^b - 1]$ will be mapped to $[0, 1)$ by a transform like $x_{i,j} = 2^{-b} u_{i,j}$). The numbers $v_{j.l} \in \mathbb{R}$ are the so-called *direction numbers* of the sequence. I will detail the role and influence of direction numbers later in this chapter.

As stated in Chapter 2, the idea of QMC methods is to use the sequence $P'_n = \{\mathbf{x}^{(i)}\}$

to drive the approximation to the integral:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \approx \mu = \int_{[0,1)^s} f(\mathbf{x})d\mathbf{x}$$

One important principle in the application and research on QMC methods is that these methods, in contrast to traditional Monte Carlo, benefit from some notion of the importance of domain components and subsets of components of the model $f : \mathbb{R}^s \to \mathbb{R}$ to its output. I make use of the techniques of global sensitivity analysis (GSA) to incorporate the idea of component importance into our implementation. In particular, I employ the GSA method of functional ANOVA decomposition, which I describe in the following section.

4.3 Sensitivity Analysis

Most practical problems involve multi-variate state spaces. Ideally, the task of modeling includes in the state only those variables that will advance the explanatory power of the model for a specific application. However, this ideal is usually not achieved in practice. In Figure 4.1, I illustrate a dynamic Bayesian network where a multivariate state node has been broken up into components. It is likely that not all components of the state $\mathbf{x}_k$ are equally important to the observation $\mathbf{z}_k$, as quantified by the likelihood function. Similarly, for the pricing of derivative instruments, not all innovations on interest rates are equally relevant to the final value of the asset. In this work, I aim to enable a simulation implementation to detect those subspaces of the state that are most relevant, and to construct a Sobol' sequence that better explores these subspaces.

Sensitivity analysis (SA) techniques are by no means a foreign notion to QMC methods. In fact, the unexpected fact that these methods outperform traditional Monte Carlo to an extent that goes beyond the asymptotic bound for approximation error, especially for high dimensions, is generally accepted as a consequence of a SA-motivated concept, the *effective dimension*. The effective dimension [25] encodes the notion that for some problems, the models in question owe their variance mostly to a small number of variables in their domain,

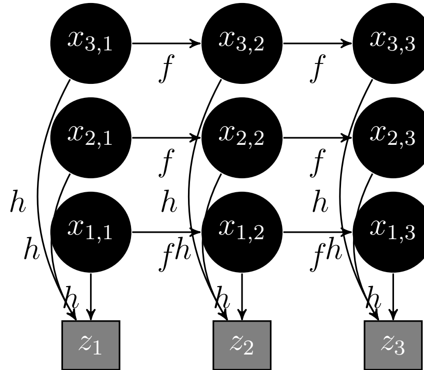


Figure 4.1: An explicit representation of multivariate state Each node in a "slice" represents a component of an input to a model. Taken as a whole each slice represents a multi-variate state space

and that the highly-uniform point sets used for their integration happen to have good projections on the subspaces spanned by those variables. The technique of *functional ANOVA decomposition* [136] quantifies the contribution of each subset of input variables to the variance of the model, and is useful to estimate the effective dimension. I will turn to this technique now, to return to a more detailed explanation of the effective dimension, and finally to explore a method to incorporate the results of the sensitivity analysis into the construction of Sobol' sequences.

4.3.1 The functional ANOVA decomposition

The goal of SA is, given a function

$$f(\mathbf{u}), \text{where}\, \mathbf{u} = (u_1, \ldots u_s) \in \mathbb{R}^s$$

to decompose the *variance* of f into contributions arising from the components u_i and to assess the magnitude and significance of each component or group of components. Note that SA assumes that f is square-integrable, i.e., $\int_{[0,1)^s} f^2(\mathbf{u}) d\mathbf{u} < \infty$.

A commonly used way to carry out a SA is to apply Hoeffding's *functional analysis of variance (ANOVA) decomposition* [41], which describes the contribution to the variation of a square-integrable function $f : [0,1]^s \to \mathbb{R}$ of each subset of input variables $I \subset \{1, \ldots s\}$. The general idea is to express f as a sum of 2^s components, each of which depends only on a subset of input variables:

$$f(\mathbf{u}) = \sum_{I \subseteq \{1,\ldots,s\}} f_I(\mathbf{u}_I) \quad \text{where } \mathbf{u}_I \in \mathbb{R}^{|I|}$$

The ANOVA decomposition can be used to determine which components u_i of f are significant contributors to the total variation. The terms f_I are defined by

$$f_I(\mathbf{u}) = \int_{[0,1)^{s-d}} f(\mathbf{u}) d\mathbf{u}_{-I} - \sum_{J \subset I} f_J(\mathbf{u})$$

where

$$\begin{aligned} -I &= \{1, \ldots, s\} \backslash I \\ d &= |-I| \\ f_\emptyset &= \int_{[0,1)^s} f(\mathbf{u}) d\mathbf{u} \end{aligned}$$

We know that the *total variance* of function f is

$$\sigma^2 = \int_{[0,1)^s} f^2(\mathbf{u}) d\mathbf{u} - f_\emptyset^2$$

and that the ANOVA decomposition is orthogonal, i.e., for two sets $I \neq J, I, J \subset \{1, \ldots s\}$

$$\int_{[0,1)^s} f_I(\mathbf{u}) f_J(\mathbf{u}) d\mathbf{u} = 0$$

If we define the variance of each f_I as

$$\sigma_I^2 = \int_{[0,1)^s} f_I^2(\mathbf{u}_I) d\mathbf{u}_I$$

for $I \neq \emptyset$ and $\sigma_\emptyset^2 = 0$, the total variance can be expressed as the sum

$$\sigma^2 = \sum_{I \subseteq \{1,\ldots,s\}} \sigma_I^2 \tag{4.1}$$

The contribution of each f_I can be summarized by normalized ratios σ_I^2/σ^2, which are called the *Sobol' sensitivity indices* (SI) [145]. The quantity σ_I^2/σ^2 represents the fraction of the variance of f that is due to f_I, and therefore, attributable to the input set with indices I. This can be taken as a measure of the relative importance of f_I. Several methods have been proposed to estimate Sobol' indices. In here, I follow that based on quasi-regression [94], which is described in the next section.

4.3.2 Quasi-regression

The technique of quasi-regression is based on the approximation of a high-dimensional function f in terms of an orthonormal basis which is created by taking tensor products of a complete orthonormal univariate basis. For concreteness, consider the complete orthonormal univariate basis that is formed by the Legendre (shifted) polynomials [119]:

$$\phi_m(x) = \frac{\sqrt{2m+1}}{m}\left[\sqrt{2m-1}(2x-1)\phi_{m-1}(x) - \frac{(m-1)}{\sqrt{2m-3}}\phi_{m-2}(x)\right] \tag{4.2}$$

where $\phi_1(x) = \sqrt{3}(2x-1)$ and $\phi_0(x) = 1$ [1]. Notice that $\phi_m(x) : [0,1] \to \mathbb{R}$. From the basis $\{\phi_m : m = 0, 1, 2\ldots\}$ we construct the multivariate (s-dimensional) basis functions by applying tensor products. This means that every element of such a basis has the form

$$\phi_{\mathbf{r}}(\mathbf{u}) = \prod_{j=1}^{s} \phi_{r_j}(u_j)$$

where $\mathbf{r} = (r_1, \ldots, r_s)$ and $r_i \in \mathcal{Z}, r_i \geq 0$.

[1] A convenient computer-readable table of shifted Legendre polynomials (albeit not normalized), is provided by the R package `orthopolynom` [116]

In this way, any s-dimensional function $f(\mathbf{u})$ can be expressed as a linear combination over the infinite basis above:

$$f(\mathbf{u}) = \sum_{r_1=0}^{\infty} \sum_{r_2=0}^{\infty} \dots \sum_{r_s=0}^{\infty} \beta_{\mathbf{r}} \phi_{\mathbf{r}}(\mathbf{u}) \tag{4.3}$$

The coefficients $\beta_{\mathbf{r}}$ can be obtained by computing

$$\beta_{\mathbf{r}} = \int_{[0,1)^s} f(\mathbf{u}) \phi_{\mathbf{r}}(\mathbf{u}) d\mathbf{u}$$

An important result [94] links the calculation of coefficients $\beta_{\mathbf{r}}$ and the contribution of components of subspaces with indices $I \subset \{1, \dots, s\}$ to the variation of function f:

$$\sigma_I^2 = \sum_{\mathbf{r} \in R(I)} \beta_{\mathbf{r}}^2 \tag{4.4}$$

where $R(I)$ is the set of s-dimensional tuples where every component is 0 but for the ones in I, i.e.,

$$R(I) = \{(r_1, \dots r_s) : r_i = 0 \text{ if } i \in I; r_i > 0 \text{ if } i \notin I\}$$

Clearly, the exact value for formula (4.3) is impossible to calculate in practice and some truncation needs to take place. To this effect, the restricted set $R(I, d, m)$ is introduced [94]:

$$R(I, d, m) = \{(r_1, \dots r_s) \in R(I) : \sum_{i=1}^{s} r_i \leq d; r_j \leq m \text{ for } j \in I\}$$

where the parameters d and m are called *degree* and *order* respectively [14].

Furthermore, we are very interested in the fraction of the variance of f that can be attributed to the input subspaces of a particular dimensionality. To this end, we define

$$\gamma(p) := \frac{1}{\sigma^2} \sum_{I:|I| \leq p} \sigma_I^2$$

Sense	Mathematical expression
Truncation	$\sum_{I\subseteq\{1,\ldots,t\}} \sigma_I^2 \geq \alpha\sigma^2$
Successive coordinates	$\sum_{I\subseteq\{i,i+1,\ldots,i+t-1\}} \sigma_I^2 \geq \alpha\sigma^2$
Superposition	$\sum_{I:\lvert I\rvert\leq t} \sigma_I^2 \geq \alpha\sigma^2$

Table 4.1: Different notions of effective dimension [90]

which quantifies this notion in terms of the user-defined parameter p. By using $\gamma(p)$ we can query the variance portion due to components of dimensionality p and lower. This turns out to be important information about model f, as integrands with large portions of variance attributable to low p values are better tractable by QMC methods. This relationship is formalized in the concept of *effective dimension*, which I treat in the following section.

4.3.3 The effective dimension

We can determine, based on Equation (4.1), a positive integer t for which $\sum_{|I|<t} f_I(\cdot)$ provides a good enough/acceptable approximation of $f(\cdot)$. I use the SIs to determine the dimensionality of the components that contribute to a user-defined fraction α of the total variance. If l-dimensional components with $l \leq t$ contribute to more than an α fraction of the variance of f, i.e.,

$$\sum_{I:|I|\leq t} \sigma_I^2 \geq \alpha\sigma^2$$

we say that f has an *effective dimension in the superposition sense* of at most t in proportion α [90].

There are alternative criteria to define the effective dimension, illustrated in Table 4.1. The truncation sense captures the notion of f being "almost t-dimensional", whereas the superposition and successive-coordinate criteria express that f is "almost" a sum of t-dimensional functions.

In general, the effective dimension of a function is a means for assessing the difficulty of its multidimensional integration. The effective dimension has played an important role explaining the effectiveness of QMC integration techniques in high-dimensionality problems, specifically financial problems [123]. When an integrand has low-effective dimension, even if it has high nominal dimension, a QMC method based on a point set P_n that has good low-dimensional projections (i.e., for a small $|I|$, the projection $P_n(I)$ over the subspace of $[0,1)^s$ indexed by I is well-distributed) can provide accurate estimators for the integral. I now discuss how to use this idea to construct a low-discrepancy point set, specifically, a Sobol' sequence.

4.3.4 Choosing direction numbers

As outlined in section 4.2, the Sobol' low-discrepancy sequence (not to be confused with the Sobol' indices for sensitivity measurement) is constructed based on a set of numbers known as *direction numbers*. Each set of direction numbers generates a different Sobol' sequence.

The canonical implementation of the Sobol' construction sequence, by Bratley and Fox [22], provides initial values for the $v_{j,k}$ for values of $j \in [1, 40]$. For dimensions beyond 40, "good" initial values can be found according to a number of criteria for P_n. Several measures of uniformity by which to judge the quality of 2D projections of a particular sequence, and by extension the initial values of their constructors have been proposed. Of these, a popular one is the *resolution* [91], the so-called uniformity properties A and A' [144] and more complicated criteria [5]. The resolution technique is proposed and applied in the `randqmc`[91] library. Initial values for $v_{j,k}$ can be chosen by studying specific 2D projections of the generated point set $P_n(j-i, j), i = 1, \ldots 8$, i.e., those values that interact with previously-constructed dimensions. The strategy I follow for our search is illustrated in detail in Algorithm 4.1.

To generate direction numbers, I use Algorithm 4.1, which works component-by-component. If the dimension under consideration at the time has been identified by the GSA stage as

Algorithm 4.1 DirectionNumberSearch

Require: configuration array $R(I, d, m) = (s, d, m, p)$

Require: array $\beta_{\mathbf{r}}$ where $\mathbf{r} \in R(I, d, m)$

for all $i \in \{1, \ldots s\}$ **do**

indices $\leftarrow$ `rs` where `rs` are rows in array $R(I, d, m)$ where right-most non-zero element is i

Determine β, the maximum of $\beta_{\mathbf{r}}$ where $\mathbf{r} \in indices$

Determine $\mathbf{r}$ corresponding to β

Search among $2^{\frac{g(g-1)}{2}}$ possibilities for best direction numbers for component $\beta_{\mathbf{r}} f(\cdot)\phi_{\mathbf{r}}(\cdot)$, where g is the degree of the primitive polynomial corresponding to dimension i. Since this search is extensive, only examine a fraction of candidates in proportion to β.

$DN[i] \leftarrow$ chosenDNs

end for

return DN

important, and the dimensionality allows it (where the dimension under consideration $i < 8$) an exhaustive search of all the possible initialization numbers m is conducted. For dimensions above 8, a random search is conducted over the space of initialization numbers in proportion to the value of the coefficient $\beta_{\mathbf{r}}$ of the configuration $\mathbf{r}$ corresponding to the maximum coefficient β among configurations $\mathbf{r}$ whose rightmost non-zero value is dimension i. Notice that the quasi-regression approximation is configured to only explore components $\phi_{\mathbf{r}}$ of dimensionality of at most 2 (i.e., $p = 2$) What set of initialization numbers to choose from those among the examined is determined by trying the Sobol' sequence corresponding to each initialization number set to calculate an estimator of the $\mathbf{r}$ component of the approximation, $\int_{[0,1)^s} \beta_{\mathbf{r}}\phi_{\mathbf{r}}(\mathbf{u})d\mathbf{u}$. The initialization numbers whose estimator exhibit the minimum variance is kept and the procedure continues until dimensionality $i = s$ is reached.

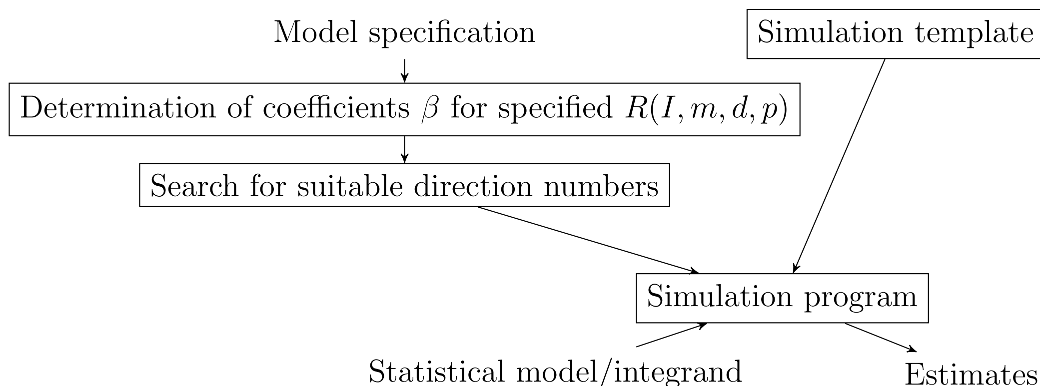


Figure 4.2: Design of the FANOVA-based Simulation framework

4.4 Functional ANOVA-Based QMC Algorithms

In this section, I make use of the ideas presented above to integrate them into a full simulation-based application. I will tailor the sampling scheme based on an analysis of the (multi-dimensional) system under study, i.e, the system to be simulated or carry out inference on. To follow this approach, two questions need to be answered:

- What dimensions to examine
- How to incorporate the knowledge of "important" dimensions to tailor the QMC scheme.

Our solution makes use of the algorithms presented above to answer these questions.

4.4.1 General Design

The general design of the numerical integrator based on the ANOVA decomposition is illustrated in Figure 4.2. The steps in the implementation are as follows:

1. The user defines the model via an implementation callback or a symbolic expression.
2. Quasi-regression is carried out on the model are calculated to assess the contribution of each subset of input variables.
3. Algorithm 4.1 is run to create of a set of direction numbers that, in turn, determine a Sobol' sequence.
4. The simulation kernel is run. If the problem is an inference query, this requires the provision by the user of an input time series. For the particular set of experiments below, I use the (biased) Monte Carlo estimator for $\beta_{\mathbf{r}}$, but more precise estimators have been proposed [94] and can be used.

This algorithm can be implemented on any general-purpose programming language. Our particular implementation uses a combination of Python and C++ code, exhibiting instances of code generation on both sides of the system.

4.4.2 Implementation

For the purposes of this investigation, I have opted for a hybrid Python/C++ implementation. The Python side of the system is better suited for symbolic manipulation and computation over combinatorial structures, as well as driving the compilation of generated code. Quasi-regression and Monte Carlo simulation is computationally intensive, and therefore it benefits from native (compiled) code for those phases of the process. Below, a brief description of the implementation of each algorithmic stage is given

Generation of $\{\mathbf{r} : \mathbf{r} \in R(I, d, m)\}$ for a given p : From the specification of (the dimensionality s) of a model f, and user-defined parameters for degree, order and maximum dimensionality for the components of the quasi-regression p, the set of tuples $\mathbf{r}$ is generated. Note that this process is in many ways largely independent of the actual model f, so sets $\mathbf{r}$ for different values of s, d, m and p can be pre-computed and stored to be thawed into the system when needed, to improved running time.

Generation of $\phi_{\mathbf{r}}$: The creation of the tensor-product base proceeds from a user-defined "base" uni-dimensional basis. For this chapter, and following other works in the literature [94], the system uses shifted Legendre polynomials, but it is equipped to deal with other choices. For this the polynomials are extracted in textual form from the representation of the `orthonompoly` R package. The computational mathematics module for Python `sympy` [28] is then used to parse this representation into actionable form. specifically, a list of symbolic expressions, whose index can be mapped to the corresponding components of tuple $\mathbf{r}$. For this purpose, the single-dimensional variable of the expression needs to be transformed into an indexed member of a multi-dimensional vector which is the actual member of the tensor product basis. It is for this task that the rewriting capabilities of `sympy` shine. The result of carrying out this task, still a symbolic expression, must be cast into a format that can be used for the quasi-regression stage. Therefore, a code generation module is run over the symbolic expressions, and a C++ library is generated, compiled, and dynamically-linked into the interpreter.

Quasi-regression : With the library of basis function $\phi_{\mathbf{r}}$ code available, the process of quasi-regression is now feasible. The only remaining obstacle is the difficulty of passing executable code (like the model) across address-space boundaries (the C++ library and the Python interpreter). To overcome this, when generating the basis $\phi_{\mathbf{r}}$ library, a hash table relating each $\mathbf{r}$ to the code for $\phi_{\mathbf{r}}$ and a system-generated function identifier. The Python driver needs only pass in the corresponding index to calculate the coefficient $\beta_{\mathbf{r}}$, as the model f and the Monte Carlo estimator calculation routine are also implemented in C.

Direction number search : Algorithm 4.1, implemented in C but driven by Python generates the set of direction numbers (on the C side of the system). The output is a C library that can be used on-line or off-line (the system has the ability to memoize the direction number structures to disk).

Simulation execution using the generated Sobol' sampler : The final stage of the

estimation system consists of running the created Sobol' sequence on the model f. As stated earlier, this takes place on the C side of the system. but Python interface functions are provided to retrieve the results on that environment as well.

Figure 4.3 presents a graphical depiction of the above process. Each of the items in the algorithm is represented by a block in that diagram. Contrast this workflow with the one associated with a typical simulation-supporting library, `librqmc`. The `librqmc` library has ample coverage for low-discrepancy sequences and associated randomizations. Its use requires the use of a special input file with the characteristics of the sampling regime. More importantly for our case, the direction numbers are provided by the library, and the only way to change those numbers is through the recompilation of the library from scratch. This workflow is illustrated in Figure 4.4.

4.5 Evaluation

For the purposes of evaluating our technique, I make use of the problem of pricing an Asian option under the Black-Scholes model. An Asian option is a financial contract that has an expiration date T, a *strike price* K and it depends on the price of an underlying asset whose value at time t is denoted by S_t, for $0 \leq t \leq T$. Assuming the Black-Scholes model implies that S_t/S_0 has a log-normal distribution with parameters $(\mu t, \sigma\sqrt{t})$, where μ is the mean return on the asset and σ is its volatility. The Asian option depends on the average value taken by the underlying asset over a predetermined period of time; for the purposes of experimentation, I consider these periods to consist of s equally-spaced times $t_1, \dots t_s$ where $t_1 = T/s$ and $t_s = T$. In this case, the final value of the option at the expiration date is given by

$$C(T) = \max\{0, \frac{1}{s}\sum_{j=1}^{s} S_{t_j} - K)$$

The quantity of interest is $\mu = \mathbb{E}[e^{-rT}C(T)]$, where the expectation is taken under the risk-neutral measure, and r represents the risk-free rate in the economy. In this case the

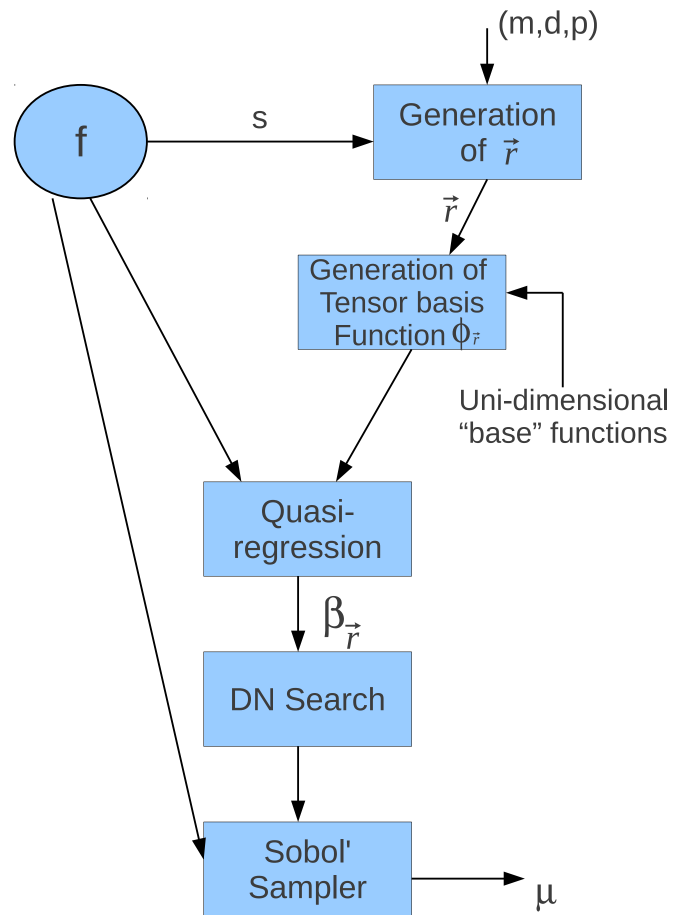


Figure 4.3: FANOVA-based Sobol' direction numbers construction

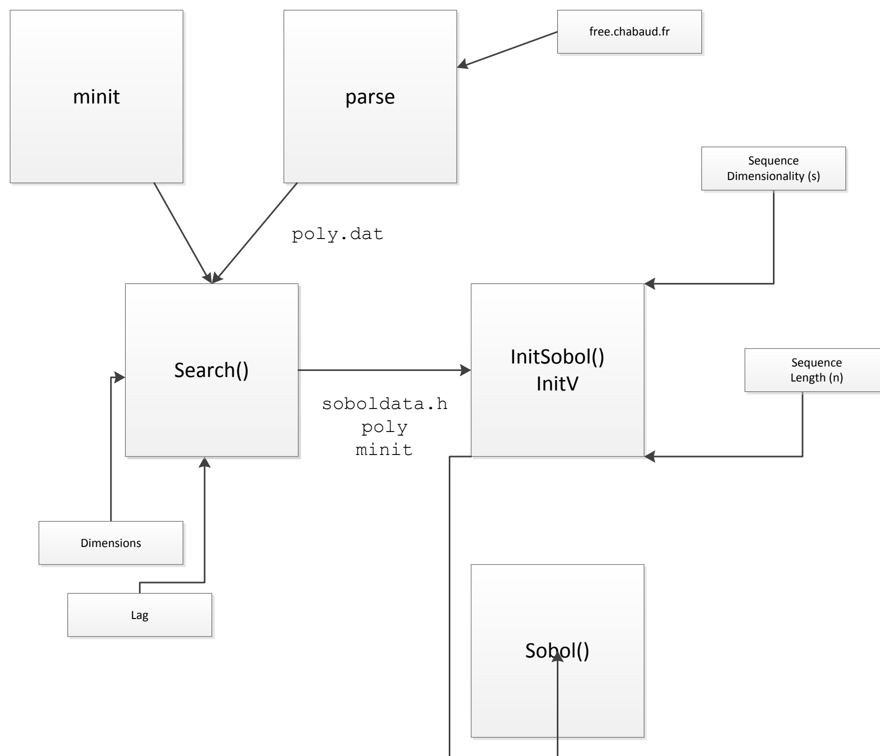


Figure 4.4: Static direction numbers Sobol' sequences-driven simulation

function that is integrated is:

$$\mu = \int_{[0,1]^s} e^{-rT} \max(0, \frac{1}{s}\sum_{i=1}^{s} S(0)exp\Big[(r-\sigma^2/2)t_i + \sigma\sqrt{T/t}\sum_{j=1}^{i}\Phi^{-1}(u_j)\Big] - K) du_1 du_2 \dots du_s$$

where Φ is the standard normal distribution.

The results of applying our algorithm to the problem of Asian option pricing are illustrated in Figures 4.5 and 4.6. The parameters of the model are set to be $S_0 = 50, T = 1, r = 0.05$ and $\sigma = 0.3$. To observe the effect of 2nd-order interactions, the strike price K is first set to 45, a setting that is known to exhibit this feature [94]. As for s, I use the values 16 and 32. It can be seen that the FANOVA-based option pricer, which picks which two-dimensional projections to fully examine offers a consistently lower-variance estimator than a set of randomly-chosen direction numbers. Specially in the case of $s = 16$, it can be seen that the variance reduction effect is significant, in particular as the number of samples increases beyond 2^{12}. Note that the y-axis of the figures in this section quantify the **log** of the variance.

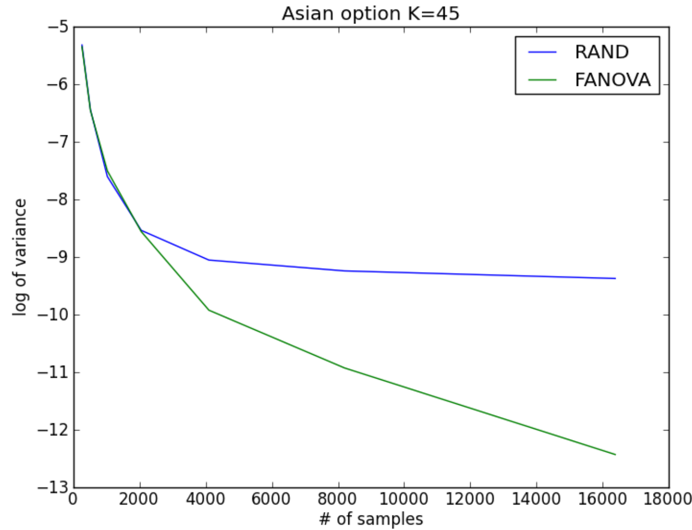


Figure 4.5: Asian option $s = 16, K = 45$

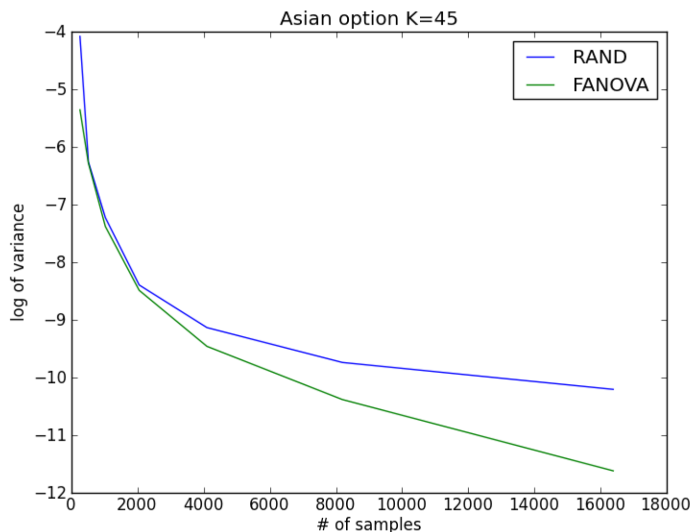


Figure 4.6: Asian option $s = 32, K = 45$

As a second problem configuration, I consider the at-the money option situation, where the strike price is set to $K = 50$, for a problem with dimensionality $s = 16$. As it can be seen in Figure 4.7, the reduction of estimator variance is also significant, and this effect can be appreciated at a lower sample count (about 2^{11}). By comparing Figures 4.5 and 4.7 it shows that the relative performance of both sampling schemes are very similar, which suggests that the effect of changing the strike price is minimal. The baseline for this problem, as for the previous experiments, is the Sobol' sequence with randomly-generated direction numbers.

It has been pointed out (for example, in [48]), that there are two main approaches of empirical testing of low-discrepancy sequences. The first one we have illustrated above, resorting to well-known application problems (in this case, from the field of Financial Mathematics). Now explore the second method, which consists of making use of synthetic multi-dimensional test functions with well-known properties and integrals. In this case, I make use of the class of multiplicative functions described by various authors under the name of g functions, in particular, the one commonly known as $g_1 : \mathbb{R}^s \to \mathbb{R}$, first proposed

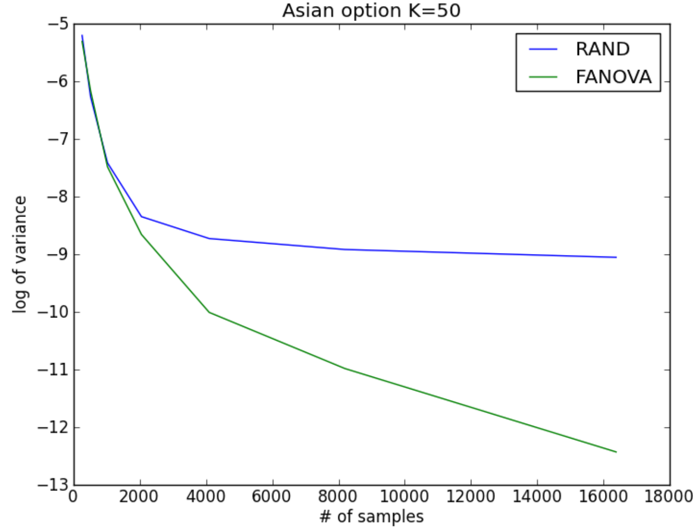


Figure 4.7: Asian option $s = 16, K = 50$

by Sobol' [145]:

$$g_1(\mathbf{x}) = \prod_{j=1}^{s} \frac{\mid 4x_j - 2 \mid + \alpha_j}{1 + \alpha_j}$$

The integral of g_1, $\int_{[0,1)^s} g_1(\mathbf{x}) d\mathbf{x}$ can be analytically calculated to be 1 and component-by-component variance (and correspondingly, the Sobol' index) is $\sigma^2_{\{j\}}(g_1) = \frac{1}{(3(1+a_j)^2)}$ [120]. Function g_1 can be configured in a number of ways, five of which are shown below:

(i) $\alpha_j = 0.01$

(ii) $\alpha_j = 1$

(iii) $\alpha_j = j$

(iv) $\alpha_j = j^2$

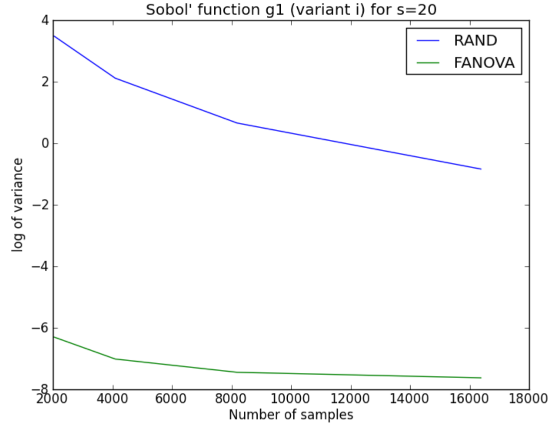


Figure 4.8: Sobol' function g_1 variation (i) with $s = 20$, comparing FANOVA- and randomly-generated Sobol' sequences.

Similarly to the evaluation of the application-specific data points, I measure the performance of the FANOVA-informed Sobol' DN algorithms by their variance, as calculated from a 64-randomization policy (using digital shift). For each of the above configurations, I test over dimensionalities $s = 20$ and $s = 32$.

The results are presented in Figures 4.8 to 4.11. Note that the horizontal axis of these plots starts at a higher number of samples than the ones presented above (2^{11} for $s = 20$, and 2^{12} for $s = 32$) to better show the differences in behavior of both sampling strategies. As shown in those graphs, the improvements due to the FANOVA-directed sampling construction are very significant. Note also that estimators of variant (ii) of the g_1 Sobol' function constructed via FANOVA reach better performance sooner than those for variant (i). For variant (i), however, the performance is flatter (more consistent) along sample counts than for variant (ii). Variants (iii) and (iv) of the g_1 function are not plotted, as the FANOVA-directed sampling construction very quickly reaches a variance of 0 (whose log cannot be plotted). This reflects the fact that these variations are highly sensitive to two-variable interaction, and that even moderate searches over possible projections results in significant performance gains.

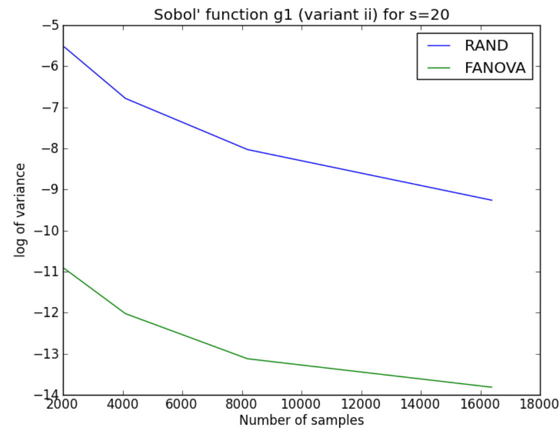


Figure 4.9: Sobol' function g_1 variation (ii) with $s = 20$, comparing FANOVA- and randomly-generated Sobol' sequences.

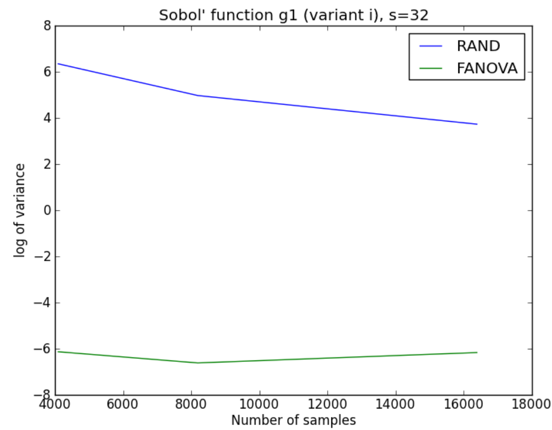


Figure 4.10: Sobol' function g_1 variation (i) with $s = 32$, comparing FANOVA- and randomly-generated Sobol' sequences.

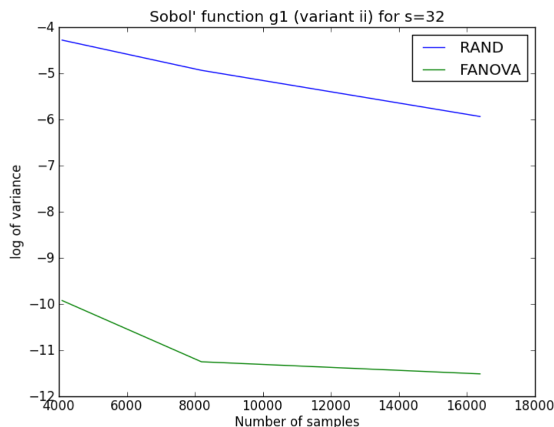


Figure 4.11: Sobol' function g_1 variation (ii) with $s = 32$, comparing FANOVA- and randomly-generated Sobol' sequences.

4.6 Summary and Future Work

In this chapter I propose a technique to improve simulation algorithms by interposing a sensitivity analysis stage on the model. This analysis takes the form of a functional ANOVA decomposition. Based on this analysis and prescribed selection criteria for the construction parameters of low-discrepancy sequences, the QMC sampling schemes that are created are tailored specifically to the model under consideration. The incorporation of these schemes into a simulation-based valuation algorithm results in estimators with lower variance than similar algorithms with a "blind" selection of QMC-sampling parameters.

The ideas described in this chapter constitute ongoing research, and more extensive studies are in progress. I have identified three areas of interesting future research. First, the selection criteria described here are not the only kind of guidance which can be used to choose parameters for generators based on heavy variance contributors. An alternative approach defines "weighted spaces" of functions where input components u_j are assumed to have less and less importance as j increases [4]. I plan to use these alternative formulations to compare with the ones presented here. The second possible line of inquiry is the study

of ANOVA decompositions of transition models for dynamic systems, an area of recent interest for ARMA models [64], but less studied when it comes to non-linear state-space representations. I expect comparable or better statistical improvements from filters that are aware of the important subspaces of the state variables for both transition and measurement functions than for the measurement function alone, as presented here. The third, and probably most general inquiry, corresponds to the study of the computational cost of the pre-processing stage of the algorithm presented here. Similarly to the considerations of the run-time code generation aspect of the IAKF of Chapter 3, the time invested in searching for good direction numbers is amortized later in the integration task. This means that the up-front cost of searching (run once) is paid off in later executions of the generated sampler. The higher the number of integration computations, the bigger the payoff. This feature makes the algorithm suitable for sensitivity analyses of simulation-based models, such as those in Finance and High-energy Physics. A more careful study of this aspect of the algorithm is underway, as is the application of techniques that could accelerate the search. Such techniques include parallelization and the use of *symbolic* specification and manipulation of the model.

As a point of interest, note that the separation between programmers that provide algorithms, and their consumers, is a growing trend. Much has been written of late of the "domain programmer," experts in a particular field that program regularly in the process of conducting their research. This trend makes the techniques presented and used here, together with any feature that makes generic software better adapted to end users, more relevant to the current and future computational environment.

Chapter 5

Real options for mobile communication management

Mobile computing devices have become pervasive in modern life. They provide applications that greatly enhance user activities, especially when several devices are used co-operatively. This communication, however, must be carefully orchestrated to make effective use of available network resources. In this chapter, we present a principled way for a mobile node to determine if and when it is cost-effective to offload computation to a remote server. This approach is based on the framework of Real Options [111], which provides well-understood decision-guiding methods that are fundamentally extensible, being able to accommodate a great number of constraints, sources of uncertainty and utility models. The Real Options approach constitutes a novel formulation of the problem that serves as an alternative to more traditional decision-making frameworks, such as Markov-decision process settings. To validate the Real Options design I use a variety of simulated wireless systems under changing network conditions and limited battery life.

5.1 Introduction

Modern advances in large-scale transistor integration and cellular communications technology have revolutionized the computer landscape over the past few years. Computers are now of a size that makes a person not only capable, but likely, to carry several with them. Most small form-factor and embedded computing devices limited in their resources when compared to their desktop or laptop counterparts. The effects of the limitations of mobile computing nodes can be ameliorated by taking advantage of the fact that these devices are continuously connected both to each other and to the Internet. This constant connectivity allows them to enlist the cooperation of connected mobile- or fixed-position computing nodes in carrying out their tasks. Applications that, while designed to run on mobile nodes, make routine use of remote resources are sometimes called "elastic" [168]. Applications that are likely to benefit from the use of elasticity include Recognition, Mining and Synthesis (RMS) workloads [26], such as voice and activity recognition, Augmented Reality (AR) workloads, specifically Computer Vision applications, real-time language translation, route planning and, in general, workloads that consolidate input from the on-board sensors in a way that is usable at a human-scale level.

While remote servers may be outfitted with more and better computational resources to carry out most tasks, communicating with them comes at a price (e.g., the battery drain associated with the use of wireless devices), and runs the risk of network service interruption. The nature of the radio channel and the access to the shared resource cause high variance in the available bandwidth and variable packet delay and loss rate. Moreover, the CPU load on the remote servers, battery charge on the mobile host, and a great number of other factors are continuously changing.

The most common approach to offloading is delegating some or all of the workload to a server accessible from the Internet, using the fixed-functional decomposition. This approach is clearly insufficient, as it does not take into account any of the aforementioned factors. Mobile nodes require a framework to weigh the advantages of offloading to remote servers, including how, when and to which remote server to offload.

The two most common software architectures to deal with unpredictable variations in the ratio of workload to available resources, i.e., to solve the *resource allocation problem* in the face of imperfect information, are (a) a centralized, omniscient job scheduler, or (b) have programs somehow re-configure themselves to better match the demand they are under [24]. Approach (a) is ill-suited for mobile wireless networks, as most environments do not lend themselves to centralized solutions. Approach (b) allocates some of the workload to geographically close computing nodes, mobile or otherwise. This mode has several advantages over traditional strategies: since cooperating computing nodes can communicate with each other directly, less infrastructure and bandwidth utilization are required; furthermore, since the number of network hops is minimized by assigning work to neighboring nodes, the need for the more expensive transports like the cellular network, as well as power consumption, is reduced. The use of strategy (b) requires programs with the ability to measure current operating conditions against present or forecasted resource demand, and to adapt themselves to those conditions. In this chapter, we present a framework and system design that provide these facilities.

The resource allocation problem can be approached by considering programs as agents in a market economy, since they need to formulate strategies towards goals while acting under imperfect knowledge, much like people in a real market. The use of market metaphors to model distributed systems has been found effective in the analysis of infrastructure systems [24]. In the present work, we apply this analogy to find the most cost-effective choice a mobile computing node can make when it comes to offload computation to advance its assigned task. By doing this, I introduce a novel application of the Real Options Analysis (ROA) methodology for capital budgeting decisions to the analysis of cooperating mobile systems.

The organization of this chapter is as follows. Section 5.2 introduces the technique of ROA and relevant valuation algorithms. I develop the mathematical model to frame the resource allocation problem in terms of ROA in Section 5.3. I also spend some time in this section to present an alternative, more mainstream formulation of the problem, the Markov Decision Process (MDP) framework, which I will later use for comparison purposes.

Section 5.4 describes the architecture of an implementation of my system and illustrates its effectiveness by empirical evaluation. Related work in this active area of research is briefly explored in Section 5.5. We identify possible future lines of research and conclude in Section 5.6.

5.2 Real Options Analysis

Real Options Analysis (ROA) is a capital-budgeting technique that is informed by the field of *financial options valuation* [39]. In Finance, an *option* is the right, but not the obligation, to buy or sell an asset (the *underlying*) up until a certain point in time. The advantages (or otherwise) of taking such action are measured by the contract's *payoff*. The *value* of the option is defined as the expected payoff of the contract discounted to the present.

"Real" option analysis extends the notion of option valuation beyond the realm of Finance to be general tools for decision-making under uncertainty ("real" is meant to indicate that the options are on commodities and other assets that are "more real" than financial or potential assets). The central notion of ROA is that any capital budgeting decision can be considered an option. The underlying of such an option is any information that influences the decision of carrying out the capital investment, i.e., all the sources of uncertainty that put the project under consideration at risk. Table 5.1 presents the input terms to a financial option pricer and their correspondence in a real options analysis setting.

ROA has found great use in capital budgeting and other resource-allocation decisions [84], where the opportunity to invest or commit capital to a given project is equated to holding an option on that project. For example, a real option can be placed on the decision to undertake or abandon a project. Of particular interest for the purposes of this work is the extension of ROA that is used to analyze operating flexibility. Flexibility is of great value to manufacturing firms, as they find themselves in continuous need to

Input	Financial Option	Real Option	Offload to server
K	exercise price	cost to complete the project	communication cost
S	price of underlying	sources of uncertainty	current available bandwidth
T	time to expiration	period when the option is viable	decision period
σ	variance in underlying	risk of the asset	bandwidth variance
r	risk-free rate of return	discount factor	discount factor

Table 5.1: Correspondence of financial options, real options and applications to offload

adapt to changing circumstances. A firm that is flexible can compete more effectively in a world of short product life cycles, rapid product development, and demand and/or price uncertainty. Similarly, in a mobile distributed system, flexibility is of value as it allows the system to take the configuration that best suits the current operating environment. The use of ROA for modeling the flexibility surrounding manufacturing operations is well-established [84], as this form of analysis can appraise the flexibility of managers to adapt and revise later decisions in response to unexpected market developments. In this chapter, we apply the ROA framework to improve the operation of a workload running on a mobile computing node by having it delegate some of that workload to whatever server the ROA has determined most effective.

5.2.1 Real options valuation

Early approaches to real option valuation were taken essentially unchanged from their native field of financial option pricing. Later studies (notably Kulatilaka and Trigeorgis [84] henceforth KT) demonstrated that such unprincipled transplantation are insufficient, and proposed a framework general enough to describe a wide variety of decisions that occur often in project management. In this work, I adapt KT's model to the resource allocation problem in mobile networks, specifically the task-offloading problem.

Real options, whose values depend on multiple sources of uncertainty, rarely have closed-

form solutions so numerical approximation procedures must be used. In particular, the Longstaff & Schwartz Least-Squares Monte Carlo (LSM) algorithm for American options [99] has been found effective to implement the KT framework of Real Option valuation.

As an American-style option pricing algorithm, LSM's main goal is to determine the optimal exercise time τ, the moment in the lifetime of the option when exercising it yields the maximum payoff. The details of the LSM algorithm are beyond the scope of this thesis, but we briefly introduce its fundamentals. The optimal stopping time τ allows the calculation of the value $F(\tau, S_\tau)$ of the option. The first stage in LSM is to approximate the American option under consideration by a Bermudan-style contract of the same horizon $[0, T]$. This is done by dividing the lifetime of the option in N periods $\{t_1, t_2, \ldots, t_N\}$, where $t_{i+1} - t_i = \Delta t = T/N, i = 0, \ldots, N-1$, and every t_i denotes an admissible exercise date [134]. Then, M trajectories of the underlying S_t in state space are generated, i.e., the set $\{s_{i,0:T}\}_{i=1,\ldots M}$ is created. A generic path in that set is denoted by ω, the optimal stopping time in path ω is denoted by $\tau(\omega)$, and the associated price of the underlying $s_\tau(\omega)$. The search of the optimal stopping time is pursued by the usual dynamic programming strategy of backward induction, i.e., starting at maturity T, the algorithm traverses the list of exercise dates in reverse chronological order, carrying summarized information from the future that allows it to determine the optimal exercise time between the dates already traversed and the one currently under consideration. This mechanism continues until the starting time $t = 0$ is reached, at which point the currently-optimal exercise time becomes the global exercise time, and the result of the algorithm. At each possible exercise date, the algorithm determines whether it is more profitable to exercise the option or to hold on to it, i.e., compares the payoff with the *continuation value*. The great innovation of the LSM algorithm is the way that the continuation value is approximated, namely, by a least-squares regression on the cross-section of the in-the-money paths at the date currently under consideration.

5.2.2 LSM for switching options

A project that consists of several alternative (mutually exclusive) courses of action can be modeled within the Real Options framework as H mutually exclusive real options. This scenario applies to a network where there are several servers, only one of which will be chosen to take on some of the workload currently running on a mobile device. Once the decision to offload to a selected server has been made, I assume here that it cannot be canceled, and so the decision can be modeled as an irreversible investment.

We denote the payoffs of the options in the original set of alternatives as Π_h, and their maturities as T_h, where $h = 1, 2, \dots H$, assuming that $T_1 \leq T_2 \leq \dots \leq T_H$. The value of real option h is then $F_h(t, S_t)$. Since the decision of which real option among this set to choose needs to be made within some time horizon T_H and is irreversible, it is clear that there is an additional *timing* option involved. Let $G(t, S_t)$ be the value of this timing option, i.e., the opportunity to choose the best out of the H alternatives. Assume further that at least one of the options is American-style (which is adequate to the offloading problem, as we allow the device to make its decision at any point within some time range). To calculate the value of the switching option we need to find the control couple (τ, ζ), where τ is a stopping time in $\mathcal{T}(t, T_H)$ and ζ takes value in the set $\{1, 2, \dots H\}$ that satisfies

$$G(t^*, S_{t^*}) = \max_{(\tau,\zeta)} \{e^{-r(\tau-t)} \mathbb{E}[F_\zeta(\tau, S_\tau)]\} \tag{5.1}$$

Although the opportunity to select the best option seems to depend on the values of the options, $F_h, h = 1, \dots, H$, the choice is not made until the time to exercise the most favorable option has come.

For the decision to offload in a mobile environment, t^* can be interpreted as the best offloading time and ζ is the index of the best server to offload to. The expressions F_h are user-defined utility functions, which will be discussed in Section 5.3.

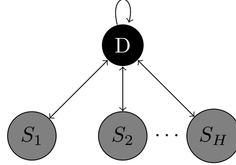


Figure 5.1: The distributed mobile computing system

5.3 Problem Setup

In this section I present a model for the operation of a mobile wireless system with the option to perform a computation on one of H servers $S_i, i = 1, \ldots H$. This system is illustrated in Figure 5.1. The node marked D denotes the mobile device, which can choose to offload computation to any of the servers in S. The task that D is to carry out can be divided in stages, each consisting of N *work units*. The cost of performing the computation per work unit is denoted by p_i where i indicates the node the computation is to take place. Note that the notion of cost establishes a measure on resources (memory, CPU time, battery life, network bandwidth), which maps the use of different resources to a common abstract cost. This quantity allows the characterization of the use of different resources under a single cost metric.

The goal of this work is to improve the operation of an application running on a mobile device by offloading computation to one in a set of servers. The challenge is to decide when and which server to use in the face of constantly changing network dynamics. To facilitate this decision, first the network conditions are estimated from readily available metrics, and then use ROA to select the most cost-effective server to offload (if any), reducing the problem to one of switching option valuation. The summary of this effort is contained in Table 5.2. In the remainder of this section I introduce the mathematical underpinnings of my method, and a concrete design in the subsequent section.

	Real Option	Distributed Resource
Goal	Determine if an investment is worth making	Determine if offload is worth performing
Input	market conditions initial investment payoff function	network conditions, battery status, CPU load offloading cost utility function
Output	Time to invest, if any	Time to offload, if any

Table 5.2: Application of ROA to computation offloading

5.3.1 Utility function

A utility function, with general form $U(s,t)$, represents a numerical rating assigned to every possible outcome a decision maker may be faced with. In this case, the utility function expresses a ranking of remote servers. As it can be seen from the expression, the utility is a function of the state of the system (in this case the computational context, including network conditions, CPU load, etc.) and the time when the decision is to be made. Therefore, the server computation should be offloaded to is the one that maximizes $U(s,t)$, i.e.,

$$(\tau, \zeta) = \underset{(t,h)}{\operatorname{argmax}}\, U((s_t, h), t)$$

where τ is the optimal offloading time, ζ is the index of the best server to offload to, s_t is the remainder of the computing context (i.e., the various state variables that define the computation environment other than the offloading server) and h is the possible choices of servers. Note that U is the right-hand side of equation (5.1). The utility function encodes the users's preferences, i.e., what the user wants to optimize for. In the following, I present some possible utility functions, optimizing for running time and battery life, respectively.

In real option pricing, the *payoff* is the difference between the expected returns from an investment vs. the involved costs. Note that the payoff function is a restricted version of an utility function, which quantifies the savings (if any) from performing a computation locally and offloading it. Even though utility functions are general enough to accommodate

many performance evaluation strategies, in this work I concentrate on utility functions that are proportional to payoffs. The payoff takes the form:

$$\{N_s p_l - (N_r p_r + s_r + b_r)\}^+ \tag{5.2}$$

where the function $\{x\}^+$ denotes the maximum between 0 and x.

p_l: cost per unit of work on the local device.

p_r: cost per unit of work on remote server.

N_s: required number of workload units for a fixed quality of the result [1].

N_r: number of work items calculated remotely at the previous offload engagement.

s_r: transmission costs to/from selected remote server.

b_r: power costs incurred to support the transfer operation

As an example, I optimize for running time. First, I have the payoff measure be the time that is saved by running the computation remotely. To do this, I apply cost functions that map the usage of different resources to commensurate quantities. We then set the utility function to a linear transform of this payoff:

$$U(s,t) = K\big(N_s p_l - (N_r p_r + s_r + b_r)\big). \tag{5.3}$$

In this case, p_l and p_r reflect how long it takes for a CPU to process a unit of work on the local and remote devices, respectively. The quantity s_r indicates the combination of latency and transmission time of the inputs to work units, and b_r reflects the cost of battery usage (e.g., by expressing usage in terms of battery lifetime drain). Finally, K is

[1] The term N_s, denoting the number of work units required to achieve a specified quality of computation implies that the task being carried out has an associated quality that is a linear function of the number of work items. An example of this is a numerical algorithm that renders an increasingly better precision the more samples it considers.

a scaling constant. For the purposes of the numerical evaluation illustrated in section 5.4, the particular form that Equation (5.1) takes is that of (5.3).

Another metric for which it may be useful to optimize is battery consumption. The utility function for this purpose can be a nonlinear decaying mapping because the smaller the battery charge, the more valuable it is. Hence, the utility function can be expressed as:

$$U(s,t) = e^{-s_b}\big(N_s p_l - (N_r p_r + s_r + b_r)\big) \tag{5.4}$$

where s_b is the remaining battery charge at the beginning of the transmission. The decaying exponential operates as a increasing penalty the closer the mobile device gets to battery depletion.

Strictly, the quantities p_l, p_r, s_r and b_r are stochastic processes. A form for these processes needs to be chosen and their parameters estimated so as to use them as input for the option pricer. Below I present such an estimator for the network delay, as this is the most common cause of concern. How this estimation fits in the overall workflow is detailed in section 5.4 and illustrated in Figure 5.2.

5.3.2 Estimating transmission costs

The term s_r in Section 5.3.1 is a random variable whose behaviour needs to be estimated. In this subsection I present an approach for estimation of transmission costs based on round-trip time (RTT). Following [59], I will fit a linear model to describe the network delays by running a Kalman filter/smoother over suitable measurements to calculate the parameters Θ of the model. This technique is briefly described in what follows.

Our parameter estimation task uses RTT as the basis of inference. RTT is defined as the length of time between the moment a package is sent to a node and the moment that node's acknowledgment of package reception is detected by the sender. RTT depends on network equipment, throughput and congestion conditions in the network during packet

transmission. I use an ICMP (Internet Control Message Protocol)-based tool to measure punctual RTT.

To actually fit a diffusion model for s_r to the RTT readings, I make use of the Kalman filter (and associated smoother). The Kalman filter is a popular technique for joint state/-parameter estimation of dynamic systems of the form:

$$s_r(t+1) = As_r(t) + \omega_t \tag{5.5}$$

$$\text{RTT}(t) = h(s_r(t)) + \omega_t^{\text{RTT}} \tag{5.6}$$

where $s_r(t)$ stands for network delay between the client and a specific server at time t, and process and measurement noises $\omega_t \sim \mathcal{N}(0, Q)$ and $\omega_t^{\text{RTT}} \sim \mathcal{N}(0, R)$ are iid distributed. As stated above, in this case the observations consist of RTT readings, and the latent states of network delays. The combination of Kalman filter and smoother form an EM-like algorithm that can fit the parameters $\Theta = (A, R, Q, \mu_0, \Sigma_0)$ (where $s_t(0) \sim \mathcal{N}(\mu_0, \Sigma_0)$) of the linear model based on the observed RTT readings [57]. The model estimated thus is suitable for use in simulations, as required by the LSM algorithm in Section 5.2.2. A recent article [59] has shown that the combined Kalman filter/smoother is an adequate estimator for network delays.

In the next section, I present an architecture that implements the components I have described above, and integrates them into a complete system that is amenable to evaluation by simulation. Before proceeding to that stage, I will present a Markov Decision Process-based solution to the optimal decision problem.

5.3.3 An alternative approach to the offloading problem: MDP

A popular mathematical framework for modeling sequential decision problems under uncertainty is known as Markov Decision Process (MDP) [100]. MDPs formalize the process a decision-making *agent* goes through when faced with the choice of what action to take at a given time. Specifically, the "life" of the agent is a stream of such decisions, each of

which changes the circumstances of the agent, and may become part of the considerations for the subsequent decisions. The notion of "circumstances" is formalized in the concept of state. We can then visualize a series of decisions taken by an agent as a trajectory on that state space. Every move from state to state of an agent via the performance of an action may have an associated reward (or cost), sometimes called the "short-term" reward. The ultimate goal of an MDP formulation is to find a rule that directs an agent to carry out an action given the current state in such a way that the overall ("long term") reward is maximized (respectively, the cost minimized). Any rule that maps actions to states is called a *policy*, which are usually denoted by π. A policy that maximizes the long-term reward is called *optimal*, and denoted by π^*.

In more formal terms, an MDP is a five-tuple (S, A, P, R, γ), where S is the set of possible states, A is the set of actions, P is the transition model that informs the dynamics of the agent in state space, R is the short-term reward model and γ is a real number $\gamma \in [0, 1]$ called the *discount factor*. Each policy π is then a mapping $\pi : S \to A$.

Algorithms that find optimal policies typically associate a *value function* to each state, action pair (s, a) where $s \in S, a \in A$, and concentrate on the manipulation of such functions to guide their operation. A value function $Q^\pi(s, a)$ represents the expected, discounted total reward starting from state (s, a) if the agent is to follow policy π. In formal terms, this is expressed as

$$Q^\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a]$$

where the expectation is taken with respect to the policy π and r_t is the expression of the short-term reward model [100]. An optimal policy maximizes the value function Q^π over all state-action pairs. Most MDP-solving algorithms then work on value functions in the search for optimal policies.

MDP-solving algorithms are very sensitive to the size of the state space and the cardinality of the action set, and typically become impractical as the state space grows. An approach that has been found effective to address this concern is the use of approximation architectures, which consist of using alternative representations of policies, value functions

and state spaces at the cost of some precision. A very popular and highly effective approximation architecture is the linear architecture, where value functions are represented by a linear combination of basis functions $\phi(s, a)$, modified by suitable weights w [87]:

$$\tilde{Q}^{\pi}(s, a; w) = \sum_{i=1}^{k} \phi_i(s, a) w_i$$

Clearly, to use an approximation framework the basis functions need to be chosen, as well as a method needs to be agreed upon to fit the weights w_i over the chosen basis.

A successful algorithm that works on linear approximations of MDPs is the so-called Least-Squares Policy Iteration (LSPI) [87]. Starting from a random policy represented as a set of weights over a chosen basis, LSPI efficiently searches for optimal policies over the policy space, and can accommodate continuous state- and action spaces [23] by using sample trajectories in state space. Recent work has applied the LSPI algorithm to value American Options [96], where the action space is confined to the two singular actions of exercise immediately or defer the exercise (i.e., opt for the continuation path). Similarly, the state for this particular application needs to take into account the time at which each state is reached. In the numerical evaluations, I use this variation of the LSPI algorithm (sometimes called LSPI for continuous-state, fixed action, LSPI-CSFA) as a point of comparison with the Longstaff-Schwartz LSM.

5.4 System Architecture and Evaluation

In this section I describe a system architecture for making offloading decisions using ROA, which I will then evaluate using simulation. The overall architecture of the system is graphically illustrated in Figure 5.2, and described in pseudo-code in Algorithm 5.1.

5.4.1 System architecture

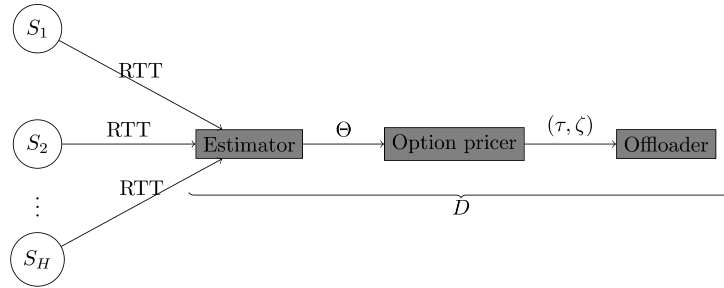


Figure 5.2: ROA-based offloader workflow

Algorithm 5.1 ROA-based offloader algorithm

Require: Parameter list `available_servers`
Require: Workload `WL`
  for s in `available_servers` **do**
    TransmissionParams[s] ← KalmanFilterSmoother(RTT[s]) {*Refer to Section 5.3.2*}
  end for
  for i in Iterations(`WL`) **do**
    for s in `available_servers` **do**
      ROA[s] ← LSM (TransmissionParams[s]) {*Refer to Section 5.2.1*}
    end for
    s^* ← SwitchingOption (ROA[s]) {*Refer to Section 5.2.2*}
    Offload `WL` iteration i to server s^*
  end for

Our system is based on grid architectures, which use a layered design to implement the services necessary for an application to operate transparently in a distributed-resource environment. The most notable divergence between the system below and a traditional grid is that the functionality of the *broker*, the software agent that maintains a loosely-coupled system, has been migrated into the application itself. The application running on the mobile node implements the following services:

- **Resource discovery** Registers and maintains a list of accessible servers.
- **Network traffic estimator** Given a list of servers, tracks the transmission delay to each of them.
- **Decision module** Given a network behaviour model and the necessary utility and cost functions, evaluate the cost-effectiveness of performing the offload.
- **Offload manager** Given an offload decision, transfer the necessary input data to the remote server, and provide the means to wait and recover it asynchronously. Note that for the purposes of this thesis, the notion of "offload" consists of transporting some suitably serialized version of the input data, and does not imply code migration.

The workflow of the system proceeds as follows:

1. As servers come on-line, they broadcast their availability over a suitable broadcast channel. This message may include the current load of the respective server.
2. The system maintains a list of accessible servers and estimates the network delay to each of them using the default routing.
3. The modified LSM algorithm for option pricing is run with the network behaviour model calculated in previous steps to determine the cost/benefit of offloading.
4. Should offloading be deemed the best choice, the offloader module downloads the necessary input data to the specified remote server.

5. On the next step, an updated network traffic model is used and the process is repeated from step 3.
6. As servers prepare go offline they broadcast their intention. This information is used to keep an updated list of downloading hosts. Moreover, the network conditions estimation algorithm also serves as a heartbeat monitor to detect when a server has become inaccessible and should no longer be considered.

This workflow does not consider the eventuality that a server becomes inaccessible during a computation, in which case this computation would need to be restarted from the beginning. This scenario is subject to future investigations. Note, however, that this is not a shortcoming of the ROA framework, but an under-specification of the model it uses as an input.

5.4.2 Evaluation

To evaluate the system I choose three typical scenarios that involves a wireless client and multiple choices for computation offload, with varying degrees of reachability. These scenarios are:

(I) A home environment, where a single wireless access point connects the mobile node with a LAN of desktop/laptop nodes on a home LAN. Bigger servers are available as part of the "Cloud", Internet-accessible data centers, which is reachable through a DSL line. A graphical depiction of this system is shown in Figure 5.3.

(II) A public wifi "Hotspot", where a single wireless access point supports a number of wireless and mobile computing nodes. The Internet is also reachable through a dedicated DSL connection.

(III) A fully-mobile environment, where the Internet ("the Cloud") is reachable via a LTE (Long Term Evolution) base station.

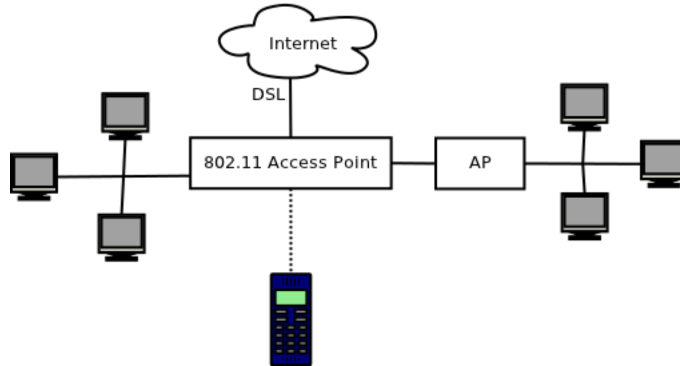


Figure 5.3: Simulated system for the ROA-based offload manager

We use the `ns-3` [2] simulation environment to implement the test systems. Without loss of generality the delays on the wireless 802.11 links and in the Ethernet connections are set to follow Gaussian distributions. This is a simplification that has been studied in the literature [59]. The load on each server p_i is drawn from a uniform distribution, but the servers on the wired networks are consistently faster than the mobile node (mobile nodes are 1.5 GHz, local wired nodes are considered to be 2.6 GHz, and remote servers in the cloud are set to be 3.0 GHz). More importantly, the variance in different kinds of nodes are different. For mobile nodes the variance is set according to measurements under regular loads at 200 ms, whereas the wired nodes, under normal conditions, were measured to have a variance of 0.2 ms.

The task that the mobile client is to carry out is divided in iterations. At each of this iteration, the (prospective) client decides whether to carry out the computation corresponding to that iteration locally or remotely. This decision is made by running the modified LSM algorithm (5.1) using the diffusion model of network delay estimated via the Kalman filter and the payoff function (5.3). Once the best server has been identified, the offload takes place. The computation costs are then quantified by the *transaction time*, which includes the time necessary to offload the input and the time it takes for the remote server to process and return the results to the client.

Scenario	Variation	Always offload	Real Options (LSM)	LSPI
(I)	channel delay= 30[ms]	34%	33.356%	33.351%
	channel delay= 60[ms]	65.776%	33.4%	33.41%
(II)	channel delay= 30[ms]	38.34%	35.6693%	35.011%
(III)	channel type = LTE	39.01799%	38.3491%	37.0125%

Table 5.3: Summary of the experimental results

I compare my method against three strategies: two fixed (a client that at each iteration offloads to the same hard-coded cloud server, and a client that never offloads), and a dynamic, MDP-informed strategy. The summary of experiments is contained in Table 5.3. The running time in the table is reported as a fraction of the running time of the strategy that never offloads, which I use as a baseline. We can see that for all the scenarios, the "elastic" strategies result in running times that are consistently and significantly reduced. Here I specify that the always-offload client always resorts to a specific server in the cloud, that is, Internet-accessible. This approach accounts for some of its increased running time when compared to the more flexible mechanisms, which can make use of "closer" computational nodes. The never-offload client runtimes are computed assuming that the workload of interest is not the only one running in the device, which introduces some variation in the running time. The computational task is simulated by scheduling replies assuming that each work unit carries out the same computation (simulated as an empty loop of 242×10^6 loops, taking around 3 sec in big cores and 9 sec in a dual-core Atom at 1.5 GHz), under the current load of the hosting server. The battery life b_r is set to be a linear function of the transaction time. The experiment was run 10 times and averages are presented. Using the never-offload client as a baseline, the average saving times is 6 sec/iteration, i.e., two thirds of the running time. Depending on the total run of the task, this can represent important savings both in time and in battery life.

The offload decisions in the home environment are clear: even though the Internet-accessible servers are available, LAN and WLAN "local" nodes are closer, and their computing capabilities are not significantly different from those in the cloud. A consistent

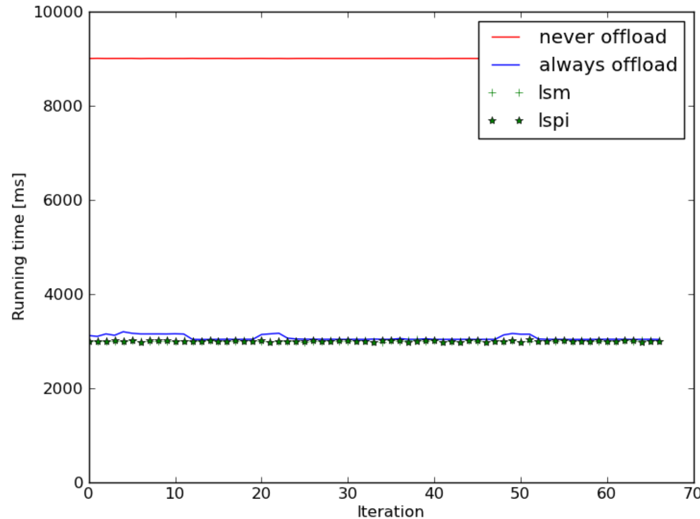


Figure 5.4: Distributed task load under ROA and fixed-host policies in the home network environment

theme in these results is that both the Real Options approach and LSPI have very similar results because both algorithms find the same optimal solution. This can be explained by the fact that they both look for the best available server, or different servers of similar characteristics. It is worthwhile noting that only a limited number of servers are offered as candidates for the algorithm. Subsequent studies on how many choices is worthwhile maintaining are planned. I manipulate the channel delay of the point-to-point connection between home network and Internet Service Provider in the second variation of the home environment (I). The effect of this change is most noticeable in the “always-offload” strategy. In other approaches, the effect is more contained, since the offload targets are consistently local nodes. These conclusions are informed by the behaviour of the systems as illustrated by Figure 5.4.

Figure 5.5 summarizes the comparative behaviour of the different offloading strategies in a Wifi hotspot situation. For the hotspot scenario we can appreciate that the adaptive

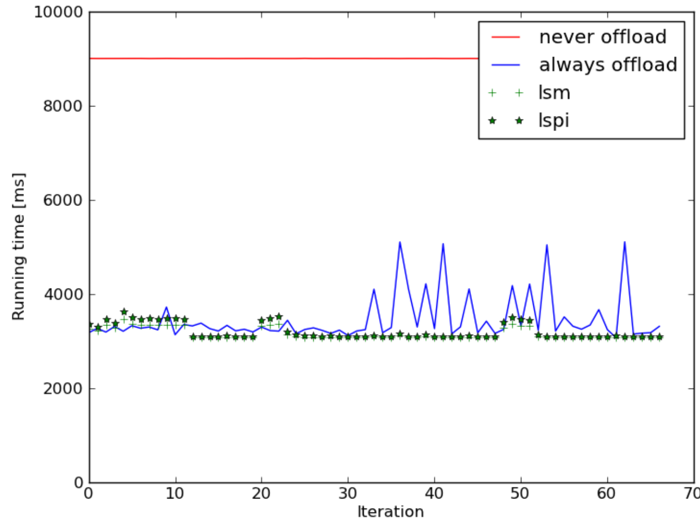
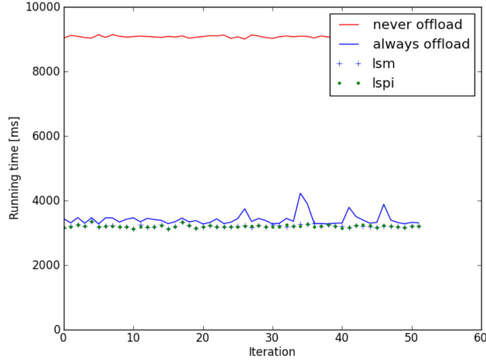


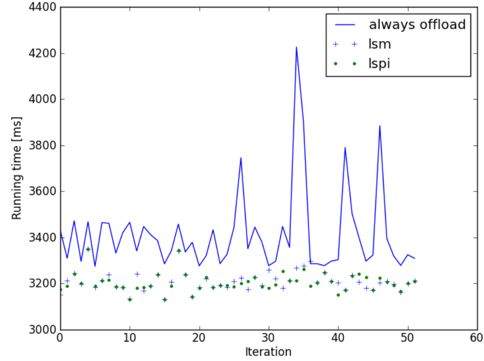
Figure 5.5: Distributed task load under ROA and fixed-host policies in the "hotspot" environment

mechanisms now choose servers from the Internet pool as offload targets. In this way, their effect is closer to the "always offload" policy than for the home network environment. However, the added flexibility of switching servers still results in improved running times. Both adaptive techniques LSM and LSPI have similar effects. Part of the increased running time in this environment is attributable to the increased traffic in the shared wireless access points in a Hotspot. A working assumption in the simulation of this environment is that the wireless nodes are fairly stationary in that they do not change APs. The variations observed in the results of the "always offload" strategy can be explained by variations in the load on the (fixed) target server, as well as the effects of other mobile clients in the shared channel.

The final scenario (illustrated in Figure 5.6) depicts servers accessible only through the cellular network. This brings the performance of the adaptive policies even closer to the original performance of the "always offload" strategy. In this case, the performance



(a) Against the "never offload" policy



(b) Detailed view of the offloading (fixed and dynamic) policies

Figure 5.6: Distributed task load under ROA and fixed-host policies in the "cloud" environment

of LSPI and LSM is more differentiated, slightly favouring LSPI by less than 2%. This may be attributable to the extra time that LSPI spends in improving the policy, which may reap better results upon less-differentiated choices. This is better appreciated on the "zoomed-in" Figure 5.6 (b), which depicts the results without the "never offload" policy. As mentioned above, the variations in the "always offload" policy are also the result of changes in the load on the target server and bandwith sharing.

5.5 Related Work

The efficient operation of mobile nodes is becoming an increasingly active field of research due to its obvious relevance to current computing environment. An important data point in this body of research are Samsung's "elastic applications"[168]. In contrast with our work, elastic applications carry out full application migration (code and data) to a virtual machine hosted on a Internet-connected server. Originally, the decision of whether to

migrate or not is done heuristically, but more recent research has studied the principled semi-Markov decision processes (SMDP) formalism [97]. The approach taken here follows a more restrictive class of MDPs that completely captures the problem, while casting it in an economic model.

Market metaphors for management of computer resources are as old as time-shared systems but most recent work has been done in the context of Grid computing, including the Tycoon system [137], whose design is a direct inspiration for the one presented here.

The market models that underlie systems like Tycoon are most often "spot" markets, where services are transacted for immediate delivery. Applications of financial markets of general and derivative instruments in particular to distributed systems include the use of options to price computational resources [12] (in the computing-as-service model) or in hedging the risks of distributed computation via mechanisms like advance reservation [105]. The present work uses option pricing in its "real option" variant in a different way: under the assumption of fully-decentralized operation, I provide means for a mobile application to rationally value the cost/benefit of offloading, thereby achieving "elasticity" by allowing mixed local/remote execution. The system requires minimal infrastructure (no pricing of resources, no advance reservation facilities, etc.), which makes it well-suited for the widely heterogeneous environment of mobile wireless settings.

5.6 Summary and Future Work

As computation and sensing devices are becoming smaller and more widely available, it is increasingly clear that different consideration have to be taken into account when designing systems. In particular, a more distributed operation is likely to be the norm, and it is unrealistic to assume that all the computing nodes involved will be under user control. A more 'opportunistic' regime, where computation takes place between otherwise unrelated nodes is worth exploring as an alternative strategy. In this setting, market-informed mechanisms provide both a mathematical formalism and a viable implementation approach for

resource-sharing schemes. Furthermore, market analogies allow the efficient and flexible use of those resources following user-defined high-level strategies.

Following one such analogy, I presented in this chapter a system that applies the insights of ROA for capital budgeting decisions to applications running on mobile clients, where the decision in question is whether and to which server to offload. I validate the ROA-based offloading algorithm by simulation and find that is both effective and flexible. Future work includes the application of the technique to real applications with varying utility functions.

It is worth noting that the computational cost of running an offload-manager is itself not negligible, so the strategy proposed here may work better for medium form-factor devices, such as tablets, rather than the smallest devices available, such as smartphones. This situation is likely to change in the near future, should the trends in on-board processors continue. In fact, the presence of multi-core processors on smartphones and tablets may enable the continuous execution of a generic "offload managing service", pinned to a single core and available to all registered applications. The exploration of such a design, as well as the study of a wider source of variation in our simulations, is also the subject of future research.

Chapter 6

Conclusions and future work

In this chapter I conclude the thesis by summarizing main research results and proposing future work.

6.1 Conclusions

The three research topics presented in this thesis are about one thing: adaptability. Either adaptability to input data (as in Chapter 3), to programmer's specifications (as in Chapter 4), or to operating conditions (as in Chapter 5). To some extent or another, each of these applications carries out a pattern recognition task, and then, through the use of either dynamic code generation or some other means of self-behaviour modification, reconfigures itself to be best suited to the workload.

A summary of the results presented in this thesis is as follows:

- The idea of "code factory" as a means for program adaptation was proposed in Chapter 3, with a concrete illustration in the form of the Input-adaptive Kalman filter (IAKF). The IAKF carries out an approximate filtering task on a non-linear

dynamic system. In contrast with traditional methods, the IAKF adapts to the input, running as many filters as necessary to best fit the input data. By introducing an adapting pre-processing stage, the accuracy of the state estimates is improved and the robustness of the filter to different input data is increased when compared to its non-adaptive counterparts. Furthermore, in the face of changing underlying computational environment, the run-time code generation capability enables the more effective use of computational resources.

- In Chapter 4 I studied the effects of the interposition of a sensitivity analysis step on simulation and inference tasks. This task, in the form of the functional ANOVA decomposition, results in an evaluation of the factors of the input that contribute most to the variance of the model. This knowledge is used to tailor the QMC sampling regime on-the-fly, to improved estimates in the corresponding task. This method results in estimators with significantly lower variance than mechanisms that use the "blind" (i.e., non-adaptive) QMC-sampling parameters.

- I studied in Chapter 5 a distributed system designed to operate in a wireless/mobile environment. Under such circumstances, it is unrealistic to assume that the reachable computing nodes will all be under the mobile user's control, but it is equally limiting to believe that they cannot be used. The system adopts an 'opportunistic' offloading regime, where computation is delegated should it be evaluated to be cost-effective to do so (where considerations of what is the cost and what are the criteria to determine whether a cost is acceptable are left to the end user). To carry out the cost/benefit analysis, I make use of a market-informed analogy of the distributed system, and bring the expressive technique of Real Option pricing to bear. Real Options analysis allows the efficient and flexible evaluation of the suitability of carrying out the computation elsewhere under a variety of circumstances. The use of this technique results in better use of the computational resources as a whole, and the better meeting of the mobile user's performance goals.

From this work, some over-arching conclusions present themselves: from the almost

accidental inclusion of the ability to generate code as part of a task-specification mechanism for a run-time system, a powerful form of metaprogramming can be used to effectively adapt programs to their environment. Furnishing the end-programmer with this or other means to tailor data-driven programs in response to late-arriving information allows for simple and straightforward implementation of programs that deal better with realistic scenarios. Furthermore, the evaluation and monitoring of the operating environment of a program can be represented, reasoned about, and acted upon through the use of the Dynamic Bayesian Network formalism.

6.2 Future work

The current trend in computing electronics to furnish mobile nodes with an increasing number of sensing devices makes for a desirable platform to apply the techniques explored in this thesis. In effect, the majority of computer users may not even be aware of their being permanently connected, both to each other and to the Internet, but that expect more perceptiveness (over sensor data) from the programs they interact with than ever before. The input of an application consists now of the whole environment of the user: their geographical location, the landmarks in their field of vision, their activities history and a myriad other pieces of information available to the application. And it is the application that is expected to decide what to do with this information, both for the advancement of the user's purposes as to ensure the correct and continued operation of the device the application is running on. The DBN modeling of these sources of information, coupled with the adaptive execution of inferencing and decision making algorithms seem like a perfect fit.

In particular, the exploration of distributed systems like that of Chapter 5, where the offloading is not only of the data that the remote server is to process, but for full code migration, where the code is being dynamically generated either in the mobile node or in a reachable computing node. I believe that the not-so-distant future will allow any single

application to enlist the help of other agents within network range, most likely running on specialized devices, with specific resource limitations where on-the-fly code generation will become an imperative.

Finally, another area that I believe is worth exploring is the self-examination of computer code through the techniques explored in this thesis. For example, a program could monitor its own execution via a DBN expression of its performance model, and dynamically recompile/reconfigure itself to better respond to changes in the operating environment (for example to limit the offloading of code to a GPU or other accelerator in the light of dwindling battery supply), or changes in its input data (through some provisions for change detection in a filtering solution). Such a self-aware application fits the rapidly changing computer environment we seem to be trending to.

In summary, dynamic code generation, dynamic computation offloading and Monte Carlo-based algorithms over Dynamic Bayesian network representations constitute a powerful set of tools used individually or in combination for the advancement of novel computing applications that demand adaptability.

References

[1] *Mathematical Systems Theory I: Modelling, State Space Analysis, Stability and Robustness.* 18

[2] The ns-3 network simulator. `http://www.nsnam.org/`. 119

[3] *On the Approximation of Correlated Non-Gaussian Noise Pdfs using Gaussian Mixture Models.*

[4] When are quasi Monte Carlo algorithms efficient for high dimensional integrals? *Journal of Complexity*, 14:1–33, 1998. 100

[5] Computational investigation of low-discrepancy sequences in simulation algorithms for Bayesian networks. 2000. 87

[6] GINAC homepage. `http://www.ginac.de`, 2011.

[7] Intel Array Building Blocks homepage. `http://software.intel.com/en-us/articles/intel-array-building-blocks`, 2011. 62

[8] OpenCV – open source computer vision. `http://opencv.willowgarage.com/wiki`, 2012. 36

[9] Nayef Bassam Abu-Ghazaleh. *Optimizing communication performance of Web services using differential deserialization of SOAP messages.* PhD thesis. 40

[10] W. Adam, R. Frühwirth, A. Strandlie, and T. Todorov. Reconstruction of electron tracks with the Gaussian-sum filter. Technical Report CERN-CMS-RN-2003-001, CERN, 2003. 54

[11] Charalambos D. Aliprantis and Kim C. Border. *Infinite-dimensional analysis: a hitchhiker's guide*. Springer-Verlag, 2006. 8

[12] David Allenotor, Rupa Thulasiram, and Parimala Thulasiram. A novel application of option pricing to distributed resource management. In *IEEE International Symposium on Parallel and Distributed Processing*, 2009. 124

[13] Daniel L. Alspach and Harold W. Sorenson. Nonlinear Bayesian estimation using Gaussian sum approximations. *IEEE Transactions on automatic control*, 17(4), 1972. 51, 54

[14] J. An and Art B. Owen. Quasi-regression. *Journal of Complexity*, 17:588–607, 2001. 78, 79, 85

[15] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2), 2003. 40

[16] Roger Bakeman and John M. Gottman. *Observing interaction: an introduction to Sequential Analysis*. Cambridge University Press, 2 edition, 1997. 7

[17] Olav Beckmann, Peter Fordham, Alastair Houghton, and Paul Kelly. A library for explicit dynamic code generation and optimisation in C++. In *CPC 2003: Tenth International Workshop on Compilers for Parallel Computers*, 2003. 41

[18] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A scalable approach to interactive global illumination. *Computer Graphics Forum*, 22:621–630, 2003.

[19] Gal Berkooz, Paul Chew, Jim Cremer, Rick Palmer, and Richard Zippel. Metaprogramming for the creation of scientific software. Technical report, AAAI, 1992. `http://www.aaai.org`. 39

[20] H.W. Bode and Claude E. Shannon. A simplified derivation of linear least square smoothing and prediction theory. 1950. 8

[21] A. Bossenbroek, A. Tirado-Ramos, and P.M.A Sloot. Grid resource allocation by means of option contracts. *IEEE Systems Journal*, 3(1):49–64, 2009.

[22] Paul Bratley and Bennett L. Fox. Algorithm 659: Implementing Sobol''s quasi-random sequence generator. *ACM Transactions on Mathematical Software*, 14(1):88–100, 1988. 33, 87

[23] L. Busoniu, A. Lazaric, M. Ghavamzadeh, R. Munos, R. Babuska, and B. De Schutter. Least-squares methods for policy iteration. In *Reinforcement Learning: State of the Art*, number 12 in Adaptation, Learning, and Optimization, pages 75–109, 2012. 115

[24] Rajkumar Buyya. *Economic-Based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, 2002. 104

[25] R. E. Caflisch, W. Morokoff, and A. B. Owen. Valuation of mortgage-backed securities using Brownian bridges to reduce effective dimension. *Journal of Computational Finance*, 1:27–46, 1997. 81

[26] Yen-Kuang Chen, J. Chhugani, P. Dubey, C.J. Hughes, Kim Daehyun, S. Kumar, V. W. Lee, A. D. Nguyen, and M Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5). 103

[27] John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Little languages and their programming environments. In *Monterey Workshop on Engineering Automation for Software Intensive System Integration*, 2001. 45

[28] The `sympy` Team. `sympy` project. `http://sympy.org/en/index.html`. 91

[29] Intel Corporation. Intel Integrated Performance Primitives. `http://software.intel.com/en-us/articles/intel-ipp`, 2012. 36

[30] R. Cranley and T. N. L. Patterson. Randomization of number-theoretic methods for multiple integration. *SIAM Journal on Numerical Analysis*, 13:904–914, 1976. 34

[31] Jack Herrington D. *Code Generation in Action*. Manning Publications, 2003. 38

[32] Nando de Freitas, Kevin Murphy, Arnaud Doucet, and Stuart Russell. Rao-Blackwellised particle filtering for dynamic bayesian networks. In *Uncertainty in Artificial Intelligence*, 2000. 24

[33] Thomas Dean, Sonia Leach, and Hagit Shatkay. Graphical models for learning dynamical systems. `ftp://ftp.cs.brown.edu/u/tld/postscript/DeanShatkayandLeach.ps`, 1995. 12

[34] Josef Dick and Friedrich Pillichshammer. *Digital nets and sequences*. Cambridge University Press, 2010. 33

[35] Arnaud Doucet, Nando de Freitas, Neil Gordon, and A. Smith, editors. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001. 75

[36] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. Rao-Blackwellised particle filtering for Dynamic Bayesian Networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, 2000. 14

[37] Arnaud Doucet and Adam M. Johansen. *Oxford handbook of nonlinear filtering*, chapter A tutorial on particle filtering and smoothing: fifteen years later. 2009. 24

[38] Arnaud Doucet and Xiaodong Wang. Monte Carlo methods for signal processing. *IEEE Signal Processing magazine*, 22(6), 2005. 23

[39] Darrell Duffie. *Dynamic Asset Pricing Theory*. Princeton University Press, 2001. 105

[40] William L. Dunn and J. Kenneth Shultis. *Exploring Monte Carlo methods*. Elsevier Science, 2011. 25

[41] B. Efron and C. Stein. The jackknife estimate of variance. *The Annals of Statistics*, 9(3), 1981. 83

[42] Karl Entacher. Discrepancy estimates based on Haar functions. *Mathematics and Computers in Simulation*, 55:49–57, 2001. 28

[43] Rodolfo G. Esteves, Christiane Lemieux, and Michael McCool. Input-adaptive QMC-Kalman filters for track fitting. In *International Converence on Adaptive and Self-adaptive systems and applications*, 2011. 5

[44] Rodolfo G. Esteves, Christiane Lemieux, and Michael McCool. Real options for mobile communication management. In *Workshop on Ubiquitous Computing and Networks*, 2011. 5

[45] Rodolfo G. Esteves, Christiane Lemieux, and Michael D. McCool. Run-time generation of QMC-Kalman filters for track fitting (abstract). In *Monte Carlo and Quasi-Monte Carlo methods*, 2010. 5

[46] G. Fasano and A. Franceschini. A multidimensional version of the Kolmogorov-Smirnov test. *Monthly Notices of the Royal Astronomical Society*, 225:155–170, 1987. 28

[47] Henri Faure. Van der Corput sequences towards general (0, 1)-sequences in base b. *J. Théor. Nombres Bordeaux 19*, 1, 2007. 33

[48] Henri Faure and Christiane Lemieux. Generalized Halton sequences in 2008: A comparative study. *ACM Transactions on Modeling and Computer Simulation*, 19(4), 2009. 96

[49] Bernd Fischer, Johann Schumann, and Thomas Pressburger. Generating data analysis programs from statistical models. In *Workshop on Semantics, Applications, and Implementation of Program Generation*, 2000. 75

[50] Florence Forbes and Gersende Fort. Combining Monte Carlo and mean-field-like methods for inference in hiddenMarkov random fields. *IEEE transactions on image processing*, 16:824, 2007.

[51] B.L. Fox. Algorithm 647: Implementation and relative efficiency of quasirandom sequence generator. *ACM Trans. Math. Software*, 12:362–376, 1986.

[52] Ronald F. Fox. Stochastic calculus in Physics. *Journal of Statistical Physics*, 46(5–6). 16

[53] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *International Conference on Acoustics, Speech, and Signal Processing*, 1998. 41

[54] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005. 41

[55] Rudolf Frühwirth. Track fitting with long-tailed noise: a Bayesian approach. *Computer Physics Communications*, 85, 1995. 54

[56] Andrea Gamba. Real options valuation: a Monte Carlo approach. In *EFA 2002 Berlin Meetings*, 2003.

[57] Zoubin Ghahramani and Geoffrey E. Hinton. Parameter estimation for linear dynamical systems. Technical report, University of Toronto, 1996. 113

[58] Paul Glasserman. *Monte Carlo methods in financial engineering*. Springer, 2003.

[59] Rafael Camilo Lozoya Gómez, Pau Martí, Manel Velasco, and Josep M. Fuertes. Wireless network delay estimation for time-sensitive applications. Technical Report ESAII-RR-06-12, Universitat Politècnica de Catalunya, 2006. 112, 113, 119

[60] Sergey Gorbunov and Ivan Kisel. An analytic formula for track extrapolation in an inhomogeneous magnetic field. In *International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, 2005. 64

[61] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. Novel approach to nonlinear/non-gaussian Bayesian state estimation. volume 140, pages 107–113, 1993.

[62] Dong Guo and Xiaodong Wang. Quasi-Monte Carlo filtering in nonlinear dynamic systems. *IEEE Transactions on Signal Processing*, 54(6), 2006. 50, 51, 53

[63] John H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12), 1964. 32

[64] James D. Hamilton. *Time Series Analysis*. Princeton University Press, 1994. 101

[65] James D. Hamilton. *New Palgrave Dictionary of Economics*, chapter Regime-Switching Models. Palgrave McMillan Ltd., 2008. 12

[66] Stefano M. Iacus. *Simulation and inference for Stochastic Differential Equations*. Springer, 2010. 15

[67] Michael Isard and Andrew Blake. CONDENSATION - conditional density propagation for visual tracking. *International Journal of Computer Vision*, 1998.

[68] G. Moore J. W. Barrett and P. Wilmott. Inelegant efficiency. *RISK Magazine*, 5(9):82–84, 1992. 80

[69] F. James. Monte Carlo theory and practice. *Reports on progress in Physics*, 43, 1980. 26

[70] Mark Joshi. *C++ Design Patterns and Derivatives Pricing*. Cambridge University Press, 2008. 17

[71] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999. 45

[72] Simon J. Julier and Jeffrey K. Uhlmann. A new extension of the Kalman filter to nonlinear systems. In *11th International Symposium on Aerospace/Defence Sensing, Simulation and Controls*, 1997. 50, 51, 53

[73] Rudolph E. Kalman. A new approach to linear filtering and prediction problems. *Transaction of the SME-Journal of Basic Engineering*, pages 35–45, 1960. 11

[74] Alexander Keller. Quasi-Monte Carlo methods in computer graphics: The global illumination problem. 27, 80

[75] Zia Khan, Tucker Balch, and Frank Dellaert. A Rao-Blackwellized particle filter for Eigen tracking. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern recognition*, 2004. 14

[76] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2011. 43

[77] Ross Kindermann and J. Laurie Snell. *Markov Random Fields and Their Applications*. American Mathematical Society, 1980. 10

[78] Simon King, Joe Frankel, Karen Livescu, Erik McDermott, Korin Richmond, and Mirjam Wester. Speech production knowledge in automatic speech recognition. *Journal of the Acoustical Society of America*, 121, 2006. 14

[79] Andreas Klöckner. `codepy` python package, 2012. 60

[80] Peter Eris Kloeden, Eckhard Platen, and Henri Schurz. *Numerical solution of SDEs through computer experiments*. Springer, 2003. 21

[81] J. F. Koksma. A general theorem from the theory of uniform distribution modulo 1. *Mathematica, Zutphen. B*, 11, 1942. 29

[82] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009. 14

[83] N. M. Korobov. The approximate computation of multiple integrals. *Doklady Akademii Nauk SSSR*, 124. 31

[84] Nalin Kulatilaka and Lenos Trigeorgis. The general flexibility to switch: real options revisited. *International Journal of Finance*, 1994. 105, 106

[85] Frances Kuo. Lattice rule generating vectors. `http://web.maths.unsw.edu.au/~fkuo/lattice/index.html`. 32

[86] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, 2001. 10

[87] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003. 115

[88] C. Lecot and A. Koudiraty. Quasi-random simulation of linear kinetic equations. *Journal of Complexity*, 17, 2001. 27

[89] Pierre L'Ecuyer and Christiane Lemieux. Recent advances in randomized quasi-Monte Carlo methods. In *Modeling Uncertainty*, volume 46 of *International Series in Operations Research and Management Science*, pages 419–474. Springer New York, 2005. 31

[90] Christiane Lemieux. *Monte Carlo and quasi-Monte Carlo Sampling*. Springer, 2009. xi, 20, 27, 29, 30, 33, 34, 67, 80, 86

[91] Christiane Lemieux, Mikolaj Cieslak, and Kristopher Luttmer. RandQMC, a package for randomized quasi-Monte Carlo methods in c, user's guide. `http://www.math.uwaterloo.ca/~clemieux/randqmc.html`, 2004. 87

[92] Christiane Lemieux and Pierre L'Ecuyer. Randomized polynomial lattice rules for multivariate integration and simulation. *SIAM Journal on Scientific Computing*, 2001.

[93] Christiane Lemieux, Dirk Ormoneit, and David Fleet. Lattice particle filters. In *Uncertainty in Artificial Intelligence*, 2001.

[94] Christiane Lemieux and Art B Owen. Quasi-regression and the relative importance of the ANOVA components of a function. Technical report, Stanford University, 2001. 84, 85, 90, 91, 95

[95] Christian Lengauer. *Domain-Specific Program Generation*, chapter Program Optimization in the Domain of High-Performance Parallelism. Lecture Notes in Computer Science. Springer-Verlag, 2004. 36

[96] Yuxi Li and Dale Schuurmans. Policy iteration for learning an exercise policy for american options. In *Proceedings of European Workshop on Recent Advances in Reinforcement Learning*, 2008. 115

[97] Hongbin Liang, Dijiang Huang, and Daiyuan Peng. On economic mobile cloud computing model. In *International Workshop on Mobile Computing and Clouds*, 2010. 124

[98] Jun S. Liu and Rong Chen. Sequential Monte Carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93:1032–1044, 1998.

[99] Francis A. Longstaff and Eduardo S. Schwartz. Valuing american options by simulation: A simple least-squares approach. *Review of Financial Studies*, 14:113–147, 2001. 107

[100] Stephen Marsland. *Machine learning: An algorithmic perspective.* CRC Press, 2009. 57, 58, 113, 114

[101] Michael Mascagni and Aneta Karaivanova. What are quasirandom nubers and are they good for for anything besides integration? In *Proceedings of Advances in Reactor Physics and Mathematics and Computation into the next milleniu*, 2000. 33

[102] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for parallel programming.* Addison-Wesley, 2005.

[103] Michael McCool. Intel Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization*, 2011. 62

[104] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Hot Topics in Parallelism (HotPar)*, 2010.

[105] Thomas Meinl. Advance reservation of grid resources via real options. In *IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, 2008. 124

[106] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949. 19

[107] MKL Team. *Intel MKL Vector Statistical Library Notes*. 9

[108] Alberto Molina-Escobar. *Filtering and parameter estimation for electricity markets*. PhD thesis, University of British Columbia, 2009.

[109] Kevin P. Murphy. An introduction to graphical models, 2001. 12

[110] Kevin Patrick Murphy. *Dynamic Bayesian networks: representation, inference and learning*. PhD thesis, University of California, Berkeley, 2002. xii, 13

[111] Stewart C. Myers. Determinants of corporate borrowing. *Journal of Financial Economics*, 5(2):147–175, 1977. 102

[112] Chris J Needham, James R Bradford, Andrew J Bulpitt, and David R Westhead. Inference in bayesian networks. *Nature*, 24, 2006.

[113] D. J. Salmond Neil J. Gordon and A. F. M. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation, 1993.

[114] Harald Niederreiter. *Random number generation and quasi-Monte Carlo methods*. Number 63 in CBMS-NSF regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, 1992. QA298.N54 1992.

[115] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 2007. 41

[116] Frederick Novomestky. `orthopolynom` cran (r) package, 2012. 84

[117] Nvidia Corporation. CUDA: parallel programming made easy, 2011. 43

[118] G. Ökten and M. Willyard. Parameterization based on randomized quasi-Monte Carlo methods. In *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[119] Hoong Boon Ooi. Global sensitivity analysis of fault location algorithms. Master's thesis, University of Adelaide, 2009. 84

[120] Art Owen. The dimension distribution, and quadrature test functions. *Statistica Sinica*, 13:1–17, 2003. 97

[121] Art B. Owen. Randomly permuted (t, m, s)-nets and $(t; s)$-sequences. In Harald Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*. Springer-Verlag, 1995. 34

[122] Pietro Parodi. Computational intelligence techniques for general insurance. `http://www.actuaries.org.uk/sites/all/files/documents/pdf/parodicomputationalintelligence.pdf`, 2009. 15

[123] S. Paskov and J. Traub. Faster valuation of financial derivatives. *The Journal of Portfolio Management*, 22:113–120, 1995. 29, 80, 87

[124] Spassimir Paskov and Joseph F. Traub. Faster valuation of financial derivatives. *Journal of Portfolio Management*, 1995. 27, 78, 80

[125] Judea Pearl. *Probabilistic reasoning in intelligent systems : networks of plausible inference*. Morgan Kaufmann, 1988. 10

[126] Charles Petzold. On-the-fly code generation for image processing. In *Beautiful Code*. O'Reilly and Associates, 2007. 39

[127] Michael K Pitt and Neil Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association*, 94(446):590–599, 1999.

[128] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. *SIGPLAN Notices*, 1997.

[129] The RAVE project team. `RAVE` – Reconstruction in an Abstract, Versatile Environment, 2012. 58

[130] Lawrence R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. 12

[131] Fons Rademakers, Rene Brun, and the ROOT team. The ROOT SMatrix package, 2012.

[132] Julian Richardson and Edward Wilson. Flexible generation of Kalman filter code. In *IEEE Aerospace Conference*, 2006. 75

[133] Christian P. Robert and George Casella. *Introducing Monte Carlo methods with R*. Springer-Verlag, 2009. 27

[134] Artur Rodrigues and Manuel J. Rocha Armada. The valuation of real options with the least squares Monte Carlo simulation method. Technical report, University of Minho. 107

[135] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012. 41

[136] A. Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global Sensitivity Analysis: The Primer*. Wiley-Interscience, 2008. 77, 82

[137] Thomas Sandholm, Kevin Lai, Jorge Andrade Ortiz, and Jacob Odeberg. Market-based resource allocation using price prediction. In *IEEE International Symposium on High Performance Distributed Computing*, 2006. 124

[138] Glenn Shafer and Vladimir Vovk. The origins and legacy of Kolmogorov's *Grundbegriffe*. The game-theoretic probability and finance project working paper 4, 2005. 8

[139] Robert H. Shumway and David S. Stoffer. *Time series analysis and its applications with R examples*. Springer, 2010. 22

[140] Ken Simpson. `pyinline` python package, 2001. 60

[141] D.B. Skillicorn. Structuring data parallelism using categorical data types. In *Programming Models for Massively Parallel Computers*, 1993.

[142] Padhraic Smyth. Belief networks, hidden Markov models and Markov random fields: a unifying view. *Pattern Recognition Letters*, 18(11–13):1261–1268, 1997. 10

[143] I. M. Sobol' and D. I. Asotsky. One more experiment on estimating high-dimensional integrals by quasi-Monte Carlo methods. *Mathematics and Computers in Simulation*, 62(3–6), 2003.

[144] Ilya Sobol'. Uniformly-distributed sequences with an additional uniform property. *USSR Computational Math*, 16:236–242, 1976. 87

[145] Ilya M. Sobol'. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Mathematics and Computers in Simulation*, 55:271–280, 2001. 84, 97

[146] Herb Sutter. Heterogeneous computing and C++ AMP. `http://channel9.msdn.com/posts/AFDS-Keynote-Herb-Sutter-Heterogeneous-Computing-and-C-AMP`,, 2011.

[147] Herb Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54—-62, 2005.

[148] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation*. Springer-Verlag, 2004.

[149] LLVM team. The low-level virtual machine website. `http://llvm.org`. 46

[150] The Quantilb team. Quantilb, a free/open-source library for quantitative finance. `http://quantlib.org`, 2012. 36

[151] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968. 41

[152] J. P. R. Tootill, W. D. Robinson, and D. J. Eagle. An asymptotically random tausworthe sequence.

[153] P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic compilation for energy adaptation. In *IEEE/ACM International Conference on Computer-aided design*. 40

[154] R. van der Merwe, E. A. Wan, and S. Julier. Sigma-point Kalman filters for nonlinear estimation and sensor-fusion: Applications to integrated navigation. In *Proceedings of the AIAA Guidance, Navigation & Control Conference*, 2004.

[155] Rudolph van der Merwe and Eric A. Wan. Efficient derivative-free Kalman filters for online learning. In *European Symposium on Artificial Neural Networks*, 2001.

[156] Eric Veach. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, 1997. 26

[157] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004. 45

[158] Kai Velten. *Mathematical modeling and simulation*. Wiley-VCH, 2009.

[159] Richard Vuduc, James Demmel, and Katherine Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, 2005. 41

[160] E. A. Wan and R. van der Merwe. chapter The Unscented Kalman Filter. 55

[161] Yuan Wang and Fred J. Hickernell. An historical overview of lattice point sets. In *Monte Carlo and Quasi-Monte Carlo Methods*, 2001.

[162] Gary Michael Wassermann. *Techniques and Tools for Engineering Secure Web Applications*. PhD thesis, University of California, Davis, 2008. 40

[163] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Conference on high performance networking and computing*. IEEE Computer Society, 1998. 41

[164] Jon Whittle and Johann Schumann. Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 30, 2004.

[165] Norbert Wiener. *Extrapolation, Interpolation, and Smoothing of Stationary Time Series With Engineering Applications*. MIT Press, 1964. 8

[166] Eric Winsberg. *Science in the Age of Computer Simulation*. University of Chicago Press, 2010. 20

[167] Paul Zarchan, Howard Musoff, and Frank K. Lu. *Fundamentals of Kalman Filtering: A Practical Approach*. American Institute of Aeronautics and Astronautics, 2009. 8

[168] Xinwen Zhang, Sangoh Jeong, Anugeetha Kunjithapatham, and Simon Gibbs. Towards an elastic application model for augmenting computing capabilities of mobile platforms. In *Mobile wireless middleware, operating systems, and applications*, 2010. 103, 123

[169] Lingling Zhao, Peijun Ma, and Xiaohong Su. Multiresolutional quasi-Monte Carlo-based particle filters. In *IEEE International Conference on Computing and Intelligent Systems*, 2009.

[170] Paul Zipkin. Mortgages and Markov chains: a simplified evaluation model. *Management Science*, 39(6), 1993. 17