MODERN COMPUTATIONAL FINANCE

AAD AND PARALLEL SIMULATIONS

with professional implementation in C++

ANTOINE SAVINE

Preface by Leif Andersen

WILEY

Modern Computational Finance

Modern Computational Finance

AAD and Parallel Simulations

with professional implementation in C++

ANTOINE SAVINE

Preface by Leif Andersen

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data

Names: Savine, Antoine, 1970- author.
Title: Modern computational finance : AAD and parallel simulations / Antoine Savine.
Description: Hoboken, New Jersey : John Wiley & Sons, Inc., [2019] | Includes bibliographical references and index. |
Identifiers: LCCN 2018041510 (print) | LCCN 2018042197 (ebook) | ISBN 9781119539544 (Adobe PDF) | ISBN 9781119539520 (ePub) | ISBN 9781119539452 (hardcover)
Subjects: LCSH: Finance—Mathematical models. | Finance—Computer simulation. | Automatic differentiation.
Classification: LCC HG106 (ebook) | LCC HG106 .S28 2019 (print) | DDC 332.01/5195—dc23
LC record available at https://lccn.loc.gov/2018041510

Cover Design: Wiley
Cover Image: ©kentarcajuan/E+/Getty Images

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my wife Audrey, who taught me to love.

To my children, Simon, Sarah, and Anna, who taught me to care.

To Bruno Dupire, who taught me to think.

To Jesper Andreasen, who believed in me when nobody else would.

And to my parents, Svetlana and Andre Savine,
who I wish were with us to read my first book.

—Antoine Savine

Contents

Modern Computational Finance xi

Preface by Leif Andersen xv

Acknowledgments xix

Introduction xxi

About the Companion C++ Code xxv

PART I
Modern Parallel Programming 1

Introduction 3

CHAPTER 1
Effective C++ 17

CHAPTER 2
Modern C++ 25
2.1 Lambda expressions 25
2.2 Functional programming in C++ 28
2.3 Move semantics 34
2.4 Smart pointers 41

CHAPTER 3
Parallel C++ 47
3.1 Multi-threaded Hello World 49
3.2 Thread management 50
3.3 Data sharing 55
3.4 Thread local storage 56

3.5 False sharing 57
3.6 Race conditions and data races 62
3.7 Locks 64
3.8 Spinlocks 66
3.9 Deadlocks 67
3.10 RAII locks 68
3.11 Lock-free concurrent design 70
3.12 Introduction to concurrent data structures 72
3.13 Condition variables 74
3.14 Advanced synchronization 80
3.15 Lazy initialization 83
3.16 Atomic types 86
3.17 Task management 89
3.18 Thread pools 96
3.19 Using the thread pool 108
3.20 Debugging and optimizing parallel programs 113

PART II
Parallel Simulation **123**

Introduction **125**

CHAPTER 4
Asset Pricing **127**
4.1 Financial products 127
4.2 The Arbitrage Pricing Theory 140
4.3 Financial models 151

CHAPTER 5
Monte-Carlo **185**
5.1 The Monte-Carlo algorithm 185
5.2 Simulation of dynamic models 192
5.3 Random numbers 200
5.4 Better random numbers 202

CHAPTER 6
Serial Implementation **213**
6.1 The template simulation algorithm 213
6.2 Random number generators 223
6.3 Concrete products 230
6.4 Concrete models 245

6.5 User interface 263
6.6 Results 268

CHAPTER 7
Parallel Implementation 271
7.1 Parallel code and skip ahead 271
7.2 Skip ahead with mrg32k3a 276
7.3 Skip ahead with Sobol 282
7.4 Results 283

PART III
Constant Time Differentiation 285

Introduction 287

CHAPTER 8
Manual Adjoint Differentiation 295
8.1 Introduction to Adjoint Differentiation 295
8.2 Adjoint Differentiation by hand 308
8.3 Applications in machine learning and finance 315

CHAPTER 9
Algorithmic Adjoint Differentiation 321
9.1 Calculation graphs 322
9.2 Building and applying DAGs 328
9.3 Adjoint mathematics 340
9.4 Adjoint accumulation and DAG traversal 344
9.5 Working with tapes 349

CHAPTER 10
Effective AAD and Memory Management 357
10.1 The Node class 359
10.2 Memory management and the Tape class 362
10.3 The Number class 379
10.4 Basic instrumentation 398

CHAPTER 11
Discussion and Limitations 401
11.1 Inputs and outputs 401
11.2 Higher-order derivatives 402

11.3 Control flow 402
11.4 Memory 403

CHAPTER 12
Differentiation of the Simulation Library 407
12.1 Active code 407
12.2 Serial code 409
12.3 User interface 417
12.4 Serial results 424
12.5 Parallel code 426
12.6 Parallel results 433

CHAPTER 13
Check-Pointing and Calibration 439
13.1 Check-pointing 439
13.2 Explicit calibration 448
13.3 Implicit calibration 475

CHAPTER 14
Multiple Differentiation in *Almost* Constant Time 483
14.1 Multidimensional differentiation 483
14.2 Traditional Multidimensional AAD 484
14.3 Multidimensional adjoints 485
14.4 AAD library support 487
14.5 Instrumentation of simulation algorithms 494
14.6 Results 499

CHAPTER 15
Acceleration with Expression Templates 503
15.1 Expression nodes 504
15.2 Expression templates 507
15.3 Expression templated AAD code 524

Debugging AAD Instrumentation 541

Conclusion 547

References 549

Index 555

Modern Computational Finance

Computational concerns, the ability to calculate values and risks of derivatives portfolios practically and in reasonable time, have always been a major part of quantitative finance. With the rise of bank-wide regulatory simulations like CVA and capital requirements, it became a matter of survival. Modern computational finance makes the difference between calculating CVA risk overnight in large data centers and praying that they complete by morning, or in real-time, within minutes on a workstation.

Computational finance became a key skill, now expected from all quantitative analysts, developers, risk professionals, and anyone involved with financial derivatives. It is increasingly taught in masters programs in finance, such as the Copenhagen University's MSc Mathematics - Economics, where this publication is the curriculum in numerical finance.

Danske Bank's quantitative research built its front office and regulatory systems combining technologies such as model hierarchies, scripting of transactions, parallel Monte-Carlo, a special application of regression proxies, and Automatic Adjoint Differentiation (AAD).

In 2015, Danske Bank demonstrated the computation of a sizeable CVA on a laptop in seconds, and its full market risk in minutes, without loss of accuracy, and won the In-House System of the Year Risk award.

Wiley's Computational Finance series, written by some of the very people who wrote Danske Bank's systems, offers a unique insight into the modern implementation of financial models. The volumes combine financial modeling, mathematics, and programming to resolve real-life financial problems and produce effective derivatives software.

The scientific, financial, and programming notions are developed in a pedagogical, self-contained manner. The publications are inseparable from the professional source code in C++ that comes with them. The books build the libraries step by step and the code demonstrates the practical application of the concepts discussed in the publications.

This is an essential reading for developers and analysts, risk managers, and all professionals involved with financial derivatives, as well as students and teachers in Masters and PhD programs in finance.

ALGORITHMIC ADJOINT DIFFERENTIATION

This volume is written by Antoine Savine, who co-wrote Danske Bank's parallel simulation and AAD engines, and teaches volatility and computational finance in Copenhagen University's MSc Mathematics - Economics.

Arguably the strongest addition to numerical finance of the past decade, Algorithmic Adjoint Differentiation (AAD) is the technology implemented in modern financial software to produce thousands of accurate risk sensitivities within seconds on light hardware. AAD is one of the greatest algorithms of the 20th century. It is also notoriously hard to learn.

This book offers a one-stop learning and reference resource for AAD, its practical implementation in C++, and its application in finance. AAD is explained step by step across chapters that gently lead readers from the theoretical foundations to the most delicate areas of an efficient implementation, such as memory management, parallel implementation, and acceleration with expression templates.

The publication comes with a self-contained, complete, general-purpose implementation of AAD in standard modern C++. The AAD library builds on the latest advances in AAD research to achieve remarkable speed. The code is incrementally built throughout the publication, where all the implementation details are explained.

The publication also covers the application of AAD to financial derivatives and the design of generic, parallel simulation libraries. Readers with working knowledge of derivatives and C++ will benefit most, although the book does cover modern and parallel C++.

The book comes with a professional parallel simulation library in C++, connected to AAD. Some of the most delicate applications of AAD to finance, such as the differentiation through calibration, are also explained in words, mathematics, and code.

Preface by Leif Andersen

It is now 2018, and the global quant community is realizing that size does matter: big data, big models, big computing grids, big computations – and a big regulatory rulebook to go along with it all. Not to speak of the *big* headaches that all this has induced across Wall Street.

The era of "big finance" has been creeping up on the banking industry gradually since the late 1990s, and got a boost when the Financial Crisis of 2007–2009 exposed a variety of deep complexities in the workings of financial markets, especially in periods of stress. Not only did this lead to more complicated models and richer market data with an explosion of basis adjustments, it also emphasized the need for more sophisticated governance as well as quoting and risk managements practices. One poster child for these developments is the new market practice of incorporating portfolio-level funding, margin, liquidity, capital, and credit effects (collectively known as "XVAs") into the prices of even the simplest of options, turning the previously trivial exercise of pricing, say, a plain-vanilla swap into a cross-asset high-dimensional modeling problem that requires PhD-level expertise in computing and model building. Regulators have contributed to the trend as well, with credit capital calculation requirements under Basel 3 rules that are at the same level of complexity as XVA calculations, and with the transparency requirements of MiFID II requiring banks to collect and disclose vast amounts of trade data.

To get a quick sense of the computational effort involved in a basic XVA calculation, consider that such a computation typically involves path-wise Monte Carlo simulation of option trade prices through time, from today's date to the final maturity of the trades. Let us say that 10,000 simulations are used, running on a monthly grid for 10 years. As a good-sized bank probably has in the neighborhood of 1,000,000 options on its books, calculating a single XVA adjustment on the bank's derivatives holding will involve in the order of $10^3 \cdot 10 \cdot 12 \cdot 10^6 \approx 10^{11}$ option re-pricings, on top of the often highly significant effort of executing the Monte Carlo simulation of market data required for pricing in the first place. Making matters significantly worse is then the fact that the traders and risk managers looking after the XVA positions will always require that sensitivities (i.e., partial derivatives) with respect to key risk factors in the market data are returned along with the XVA number itself. For complex portfolios, the number of sensitivities that

one needs to compute can easily be in the order of 10^3; if these are computed naively (e.g., by finite difference approximations), the number of option repricings needed will then grow to a truly unmanageable order of 10^{14}.

There are many interesting ways of chipping away at the practical problems of XVA calculations, but let us focus on the burdens associated with the computation of sensitivities, for several reasons. First, sensitivities constitute a perennial problem in the quant world: whenever one computes some quantity, odds are that somebody in a trading or governance function will want to see sensitivities of said quantity to the inputs that are used in the computation, for limit monitoring, hedging, allocation, sanity checking, and so forth. Second, having input sensitivities available can be very powerful in an optimization setting. One rapidly growing area of "big finance" where optimization problems are especially pervasive is in the machine learning space, an area that is subject to enormous interest at the moment. And third, it just happens that there exists a very powerful technique to reduce the computational burden of sensitivity calculations to almost magically low levels.

To expand on the last point above, let us note the following quote by Phil Wolfe ([1]):

> *There is a common misconception that calculating a function of n variables and its gradient is about $n+1$ times as expensive as just calculating the function. This will only be true if the gradient is evaluated by differencing function values or by some other emergency procedure. If care is taken in handling quantities, which are common to the function and its derivatives, the ratio is usually 1.5, not $n+1$, whether the quantities are defined explicitly or implicitly, for example, the solutions of differential equations...*

The "care" in "handling quantities" that Wolfe somewhat cryptically refers to is now known as *Algorithmic Adjoint Differentiation* (AAD), also known as *reverse automatic differentiation* or, in machine learning circles, as *backward propagation* (or simply *backprop*). Translated into our XVA example, the promise of the "cheap gradient" principle underpinning AAD is that computation of all sensitivities to the XVA metric – no matter how many thousands of sensitivities this might be – may be computed at a cost that is order $\mathcal{O}(1)$ times the cost of computing the basic XVA metric itself. It can be shown (see [2]) that the constant in the $\mathcal{O}(1)$ term is bounded from above by 5. To paraphrase [3], this remarkable result can be seen as somewhat of a "holy grail" of sensitivity computation.

The history of AAD is an interesting one, marked by numerous discoveries and re-discoveries of the same basic idea which, despite its profoundness,[1]

[1]Nick Trefethen [4] classifies AAD as one of the 30 greatest numerical algorithms of the 20th century.

has had a tendency of sliding into oblivion; see [3] for an entertaining and illuminating account. The first descriptions of AAD date back to the 1960s, if not earlier, but did not take firm hold in the computer science community before the late 1980s. In Finance, the first published account took another 20 years to arrive, in the form of the award-winning paper [5].

As one starts reading the literature, it soon becomes clear why AAD originally had a hard time getting a foothold: the technique is hard to comprehend; is often hidden behind thick computer science lingo or is buried inside applications that have little general interest.[2] Besides, even if one manages to understand the ideas behind the method, there are often formidable challenges in actually implementing AAD in code, especially with management of memory or retro-fitting AAD into an existing code library.

The book you hold in your hands addresses the above challenges of AAD head-on. Written by a long-time derivatives quant, Antoine Savine, the exposition is done at a level, and in an applications setting, that is ideal for a Finance audience. The conceptual, mathematical, and computational ideas behind AAD are patiently developed in a step-by-step manner, where the many brain-twisting aspects of AAD are demystified. For real-life application projects, the book is loaded with modern C++ code and battle-tested advice on how to get AAD to run *for real*.

Select topics include: parallel C++ programming, operator overloading, tapes, check-pointing, model calibration, and much more. For both newcomers and those quaint exotics quants among us who need an upgrade to our coding skills and to our understanding of AAD, my advice is this: start reading!

[2]Some of the early expositions of AAD took place in the frameworks of chemical engineering, electronic circuits, weather forecasting, and compiler optimization.

Acknowledgments

The Computational Finance series is co-authored by Jesper Andreasen, Brian Huge, and Antoine Savine, who worked together in Danske Bank's quantitative research, and wrote its award winning systems, together with Ove Scavenius, Hans-Jorgen Terp Flyger, Jakob Nielsen, Niels Sonderby, Marco Haller Schultz and the rest of the department. Brian Fuglsbjerg conducted a last-minute review of the text and suggested meaningful improvements in the interest of clarity and correctness.

We extend thanks to Danske Bank for providing a work environment that encouraged and nurtured the development of the advanced technologies described in these publications. For avoidance of doubt, the code provided with the books is not the one implemented in the bank's systems and was developed independently for the purpose of the publications.

Irrespective of its intrinsic quality, quantitative research is only as effective as its relationship to business. We are extending special thanks to Danske Bank's exotics and xVA trading desks, who, under the particularly enlightened leadership of Peter Honore, Martin Linderstrom, and Nicki Rasmussen, offered valuable feedback, effective partnership, and unabated support during the development of the bank's systems. We could not have done it without them.

The publications are also based on our articles and talks, as well as our lectures at Copenhagen University. We extend special thanks to Professor Rolf Poulsen, head of the MSc Mathematics–Economics, for his continuous support, valuable feedback, and the multiple discussions that helped us tremendously throughout our work. Rolf also kindly reviewed and helped improve the more theoretical Chapters 4 and 5.

Before publication, the books were submitted for review to Leif Andersen, global head of quantitative research at BAML, and co-author, with Vladimir Piterbarg, of the three volumes of Interest Rate Models – in our opinion, the clearest, most comprehensive and useful reference in financial models [6]. Leif and his teams thoroughly reviewed our publication and found hundreds of language, grammar, style, mathematical, and programming mistakes. We are extending a million thanks to Leif and his researchers who performed this considerable work and saved us some serious embarrassment. The mistakes that remain are of course our own responsibility.

Finally, we extend thanks to our multiple friends and colleagues from other institutions who contributed through lively discussions and debates. All these brainstorms sharpened our understanding of the field and made our work a particularly enjoyable one. Specifically, we are extending very special thanks to our dear friends, Bruno Dupire from Bloomberg, Guillaume Blacher from BAML, and Jerome Lebuchoux from Goldman Sachs, for all the hours spent in such meaningful, rich, and pleasant discussions, of which they will certainly find traces in this publication.

Introduction

In the aftermath of the global financial crisis of 2008, massive regulations were imposed on investment banks, forcing them to conduct frequent, heavy regulatory calculations. While these regulations made it harder for banks to conduct derivatives businesses, they also contributed to a new golden age of computational finance.

A typical example of regulatory calculation is the CVA (Counterparty Value Adjustment),[1] an estimation of the loss subsequent to a future default of a counterparty when the value of the sum of all transactions against that counterparty (called netting set) is positive, and, therefore, lost. The CVA is actually the value of a real option a bank gives away whenever it trades with a defaultable counterparty. This option is a put on the netting set, contingent on default. It is an exotic option, and a particularly complex one, since the underlying asset is the netting set, consisting itself in thousands of transactions, some of which may be themselves optional or exotic. In addition, the netting set typically includes derivatives transactions in different currencies on various underlying assets belonging to different asset classes. Options on a set of heterogeneous underlying assets are known to the derivatives industry and called hybrid options. Investment banks' quantitative research departments actively developed hybrid models and related numerical implementations in the decade 1998–2008 for the risk management of very profitable transactions like Callable Reverse Power Duals (CRPD) in Japan.

The specification, calibration, and simulation of hybrid models are essentially well known; see, for example, [7], [8], and [9]. What is unprecedented is the vast number of cash flows and the high dimension of the simulation. With a naive implementation, the CVA on a large netting set can take minutes or even hours to calculate.

In addition, the market and credit risk of the CVA must be hedged. The CVA is a cost that impacts the revenue and balance sheet of the bank, and its value may change by hundreds of millions when financial and credit markets move. In order to hedge the CVA, it is not enough to compute it.

[1] We have a lot of similar regulatory calculations, collectively known as xVA. The capital charge for derivatives businesses and the cost of that capital are also part of these calculations.

All its sensitivities to market variables must also be produced. And a CVA is typically sensitive to thousands of market variables: all the underlying assets that affect the netting set, all the rate and spread curves and volatility surfaces for all the currencies involved, as well as all the foreign exchange rates and their volatility surfaces, and, of course, all the credit curves. In order to compute all these risks with traditional finite differences, the valuation of the CVA must be repeated thousands of times with inputs bumped one by one.

This is of course not viable, so investment banks first implemented crude approximations, at the expense of accuracy, and distributed calculations over large data centres, incurring massive hardware, development, and maintenance costs.

Calculation speed became a question of survival, and banks had to find new methodologies and paradigms, at the junction of mathematics, numerics, and computer science, in order to conduct CVA and other regulatory calculations accurately, practically, and quickly on light hardware.

That search for speed produced new, superior mathematical modeling of CVA (see, for instance, [10]), a renewed interest in the central technology of scripting derivatives cash flows (see our dedicated volume [11]), the systematic incorporation of parallel computing (Part I of this volume), and exciting new technologies such as Algorithmic Adjoint Differentiation (AAD, Part III of this volume) that computes thousands of derivatives sensitivities in constant time.

In the early 2010s, head of Danske Markets Jens Peter Neergaard was having lunch in New York with quantitative research legend Leif Andersen. As Leif was complaining about the administration and cost of data centres, JP replied:

We calculate our CVA on an iPad mini.

In the years that followed, the quantitative research department of Danske Bank, under Jesper Andreasen's leadership, turned that provocative statement into reality, by leveraging cutting-edge technologies in a software efficient enough to conduct CVA risk on light hardware without loss of accuracy.

In 2015, at a public Global Derivatives event in Amsterdam, we demonstrated the computation of various xVA and capital charge over a sizeable netting set, together with all risk sensitivities, within a minute on an Apple laptop. That same year, Danske Bank won the In-House System of the Year Risk award.

We have also been actively educating the market with frequent talks, workshops, and lectures. These publications are the sum of that work.

Modern quantitative researchers must venture beyond mathematics and numerical methods. They are also expert C++ programmers, able to leverage modern hardware to produce highly efficient software, and master key

technologies like algorithmic adjoint differentiation and scripting. It is the purpose of our publications to teach these skills and technologies.

It follows that this book, like the other two volumes in the series, is a new kind of publication in quantitative finance. It constantly combines financial modeling, mathematics, and programming, and correspondences between the three, to resolve real-life financial problems and improve the accuracy and performance of financial derivatives software. These publications are inseparable from the professional source code in C++ that comes with them. The publications build the libraries step by step and the code demonstrates the practical application of the concepts discussed in the books.

Another unique aspect of these publications is that they are not about models. The definitive reference on models was already written by Andersen and Piterbarg [6]. The technologies described in our publications: parallel simulations, algorithmic adjoint differentiation, scripting of cash-flows, regression proxies, model hierarchies, and how to bring them all together to better risk manage derivatives and xVA, are all model agnostic: they are designed to work with all models. We develop a version of Dupire's popular model [12] for demonstration, but models have little screen time. The stars of the show are the general methodologies that allow the model, any model, to compute and differentiate on an iPad mini.

This volume, written by Antoine Savine, focuses on algorithmic adjoint differentiation (AAD) (Part III), parallel programming in C++ (Part I), and parallel simulation libraries (Part II). It is intended as a one-stop learning and reference resource for AAD and parallel simulations, and is complete with a professional implementation in C++, freely available to readers in our source repository.

AAD is a ground-breaking programming technique that allows one to produce derivative sensitivities to calculation code, automatically, analytically, and most importantly *in constant time*. AAD is applied in many scientific fields, including, but not limited to, machine learning (where it is known under the name "backpropagation") or meteorology (the powerful improvement of expression templates, covered in Chapter 15, was first suggested by a professor of meteorology, Robin Hogan). While a recent addition, AAD has quickly become an essential part of quantitative finance and an indispensable centrepiece of modern financial libraries. It is, however, still misunderstood to a large extent by a majority of finance professionals.

This publication offers a complete coverage of AAD, from its theoretical foundations to the most elaborate constructs of its efficient implementation. This book follows a pedagogical logic, progressively building intuition and taking the time to explain concepts and techniques in depth. It is also a complete reference for AAD and its application in the context of (serial and parallel) financial algorithms. With the exception of Chapters 12 and

13,[2] Part III covers AAD in itself, without reference to financial applications. Part III and the bundled AAD library in C++ can be applied in various contexts, including machine learning, although it was tested for maximum performance in the context of parallel financial simulations.

A second volume [11], co-authored by Jesper Andreasen and Antoine Savine, is dedicated to the scripting of derivatives cash flows. This central technology is covered in detail, beyond its typical usage as a convenience for the structuring of exotics. Scripts are introduced as a practical, transparent, and effective means to represent and manipulate transactions and cash flows in modern derivatives systems. The publication covers the development of scripting in C++, and its application in finance, to its full potential. It is also complete with a professional implementation in C++. Some advanced extensions are covered, such as the automation of fuzzy logic to stabilize risks, and the aggregation, compression, and decoration of cash flows for the purpose of xVA.[3]

A third (upcoming) volume, written by Jesper Andreasen, Brian Huge, and Antoine Savine, explores effective algorithms for the computation and differentiation of xVA, and covers the details of the efficient implementation, applications, and differentiation of the LSM[4] algorithm.

[2] As well as part of 14.

[3] We briefly introduced CVA. The other value adjustments banks calculate for funding, liquidity, cost of capital, etc. are collectively known as xVA and the computation techniques detailed for CVA are applicable with some adjustments.

[4] Least Squares Method, sometimes also called the Longstaff-Schwartz Model, invented in the late 1990s by Carriere [13] and Longstaff-Schwartz [14], and briefly introduced in Section 5.1.

About the Companion C++ Code

This book comes with a professional implementation in C++ freely available to readers on:

www.wiley.com/go/computationalfinance

In this repository, readers will find:

1. All the source code listed or referenced in this publication.
2. The files AAD*.*[1] constitute a self contained, general-purpose AAD library. The code builds on the advanced techniques exposed in this publication, in particular those of Chapters 10, 14, and 15, to produce a particularly fast differentiation library applicable to many contexts. The code is progressively built and explained in Part III.
3. The files mc*.*[2] form a generic, parallel, financial simulation library. The code and its theoretical foundations are described in Part II.
4. Various files with support code for memory management, interpolation, or concurrent data structures, such as threadPool.h, which is developed in Part I and used throughout the book to execute tasks in parallel.
5. A file main.h that lists all the higher level functions that provide an entry point into the combined library.
6. A Visual Studio 2017 project wrapping all the source files, with project settings correctly set for maximum optimization. The code uses some C++ 17 constructs, so the project setting "C++ Language Standard" on the project property "C/C++ / Language / C++ Language Standard" must be set to "ISO C++ 17 standard." This setting is correctly set on the project file xlComp.vcxproj, but readers who compile the files by other means must be aware of this.
7. A number of xl*.* files that contain utilities and wrappers to export the main functions to Excel, as a particularly convenient front end for the library. The project file xlComp.vcxproj is set to build an xll, a file that is opened from Excel and makes the exported library functions callable from Excel like its standard functions. We wrote a tutorial that explains

[1] With a dependency on gaussians.h for the analytic differentiation of Gaussian functions, and blocklist.h for memory management, both included.
[2] With dependency on various utility files, all included in the project.

how to export C++ code to Excel. The tutorial ExportingCpp2xl.pdf is available in the folder xlCpp along with the necessary source files. The wrapper xlExport.cpp file in our project precisely follows the directives of the tutorial and readers can inspect it to better understand these techniques.

8. Finally, we provide a pre-built xlComp.xll[3] and a spreadsheet xlTest.xlsx that demonstrates the main functions of the library. All the figures and numerical results in this publication were obtained with this spreadsheet and this xll, so readers can reproduce them immediately. The computation times were measured on an iMac Pro (Xeon W 2140B, 8 cores, 3.20 GHz, 4.20 max) running Windows 10. We also carefully checked that we have *consistent* calculation times on a recent quad core laptop (Surface Book 2, i7-8650U, 4 cores, 1.90 GHz, 4.20 max), that is, (virtually) identical time in single threaded mode, twice the time in multi-threaded mode.

The code is entirely written in standard C++, and compiles on Visual Studio 2017 out of the box, without any dependency to on a third-party library.

[3]To run xlComp.xll, readers may need to install Visual Studio redistributables VC_redist.x86.exe and VC_redist.x64.exe, also included in the repository.

PART

I

Modern Parallel Programming

Introduction

This part is a self-contained tutorial and reference on high-performance programming in C++, with a special focus on parallel and concurrent programming. The second part applies the notions and constructs developed here to build a generic parallel financial simulation library.

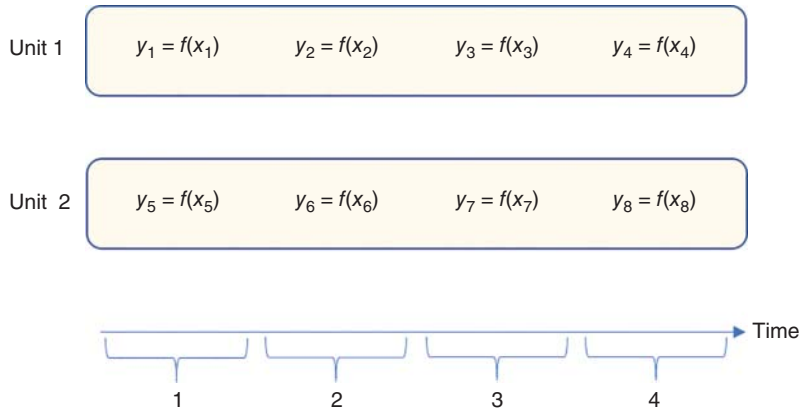
A working knowledge of C++ is expected from readers. We cover modern and parallel C++11, and we illustrate the application of many STL (standard template library) data structures and algorithms, but we don't review the basic syntax, constructs, and idioms of C++. Readers unfamiliar with basic C++ should read an introductory textbook before proceeding. Mark Joshi's website http://www.markjoshi.com/RecommendedBooks.html#C contains a list of recommended C++ textbooks.

Readers already familiar with advanced C++ and concurrent programming, or those using different material, such as Anthony Williams' [15], may skip this part and move on to Parts II and III. They will need the *thread pool*, which we build in Section 3.18 and use in the rest of the publication. The code for the thread pool is in ThreadPool.h, ThreadPool.cpp, and ConcurrentQueue.h in our repository.

PARALLEL ALGORITHMS

Parallel programming may allow calculations to complete faster by computing various parts simultaneously on multiple processing units. The improvement in speed is at best linear in the number of units. To hit a linear improvement (or any improvement at all), parallel algorithms must be carefully designed and implemented. One simple, yet fundamental example is a *transformation*, implemented in the *transform*() function of the C++ standard library, where some function f is applied to all the elements $\{x_i\}$ in a source collection S to produce a destination collection $D = \{y_i = f(x_i)\}$. Such transformation is easily transposed in parallel with the "divide-and-conquer" idiom: partition S into a number of subsets $\{S_j\}$ and process the transformation of the different subsets, simultaneously, on different units.

The chart below illustrates the idiom for the transformation of 8 elements on 2 units.

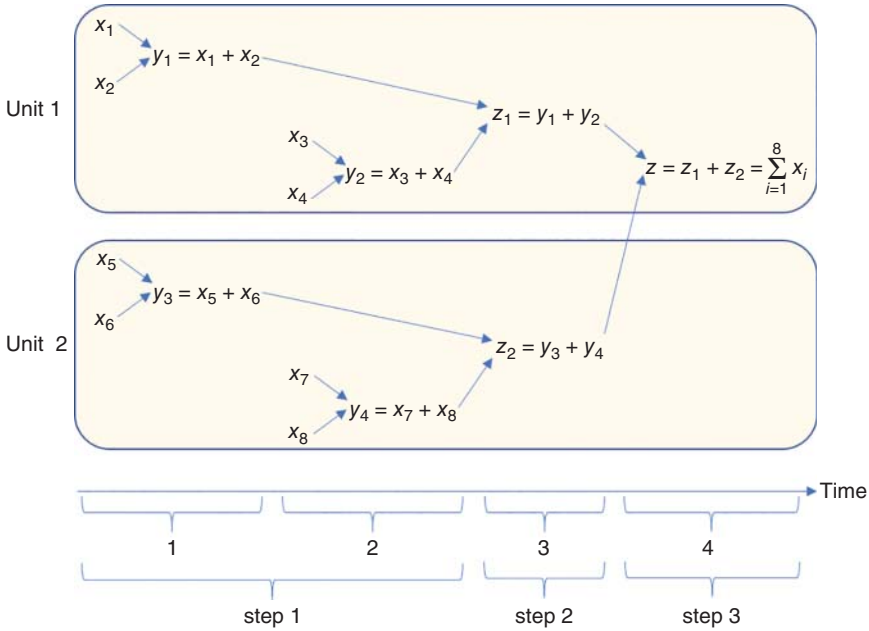


The 8 transformations are processed in the time of 4. Parallel processing cuts time by the number of units, a *linear* speed-up. With n transformations on p units, each unit evaluates f n/p times, and the units work simultaneously, so that the entire transformation completes in the time of n/p evaluations, as opposed to n evaluations in the serial version. In addition, this linear improvement is easily achieved in practice, assuming f computes an output y out of an input x without any side effect (like logging into a console window or a file, or updating a static cache).[1] Such problems that are trivially transposed and implemented in parallel are called "embarrassingly parallel."

Another classical family of fundamental algorithms is the *reduction*, implemented in the standard C++ *accumulate*() function, which computes an aggregate value (sum, product, average, variance...) from a collection of elements. One simple example is the sum $z = \sum_{i=1}^{n} x_i$. In order to compute z in parallel, we can compute partial sums over subsets, which leaves us with a lower number of partial sums to sum-up. A parallel reduction is therefore recursive in nature: we compute partial sums, then partial sums of partial

[1] We call such a function "thread safe." It is a fundamental principle of functional programming that functions should not be allowed to have side effects. This is enforced in pure functional languages like Haskell, where all functions are thread safe and suitable for parallel application. C++ allows side effects, so thread safe design is the developer's responsibility. In particular, it should be made very clear which functions have side effects and which don't. Thread safe design is closely related to const correctness, as we will see in the next chapters.

sums, and so forth, until we get a final sum. The chart below illustrates the sum of 8 terms on 2 units:



A reduction of 8 elements is, again, completed in the time of 4. A parallel reduction also produces a linear improvement. But the algorithm is no longer trivial; it involves 3 steps in a strict sequential order, and logic additional to the serial version. In general, to reduce n elements in parallel on p units, assuming n is even, we partition $S = \{x_i, 1 \leq i \leq n\}$ into $n/2$ pairs and sum the 2 elements of the $n/2$ pairs in parallel. To process $n/2$ adds in parallel on p units takes the time of $n/2p$ serial adds. Denoting δ the time of a serial add, we reduced S into $n/2$ partial sums in $(n/2p)\delta$ time. Assuming again that $n/2$ is even, we repeat and reduce the $n/2$ partial sums into $n/4$ pairwise sums in a time $(n/4p)\delta$. By recursion, and assuming that n is a power of 2: $n = 2^m$, the entire reduction completes in $m = log_2 n$ steps, where the step number i consists in $n/2^i$ pairwise adds on p units and takes $(n/2^i p)\delta$ time. The total time is therefore:

$$\sum_{i=1}^{m} \frac{n}{2^i p}\delta = \frac{n-1}{p}\delta$$

as opposed to $(n-1)\delta$ for the serial reduction, hence an improvement by p.

In a more general case where n is not necessarily a power of 2, we still achieve a linear improvement. To demonstrate this, we note that n can be uniquely decomposed into a sum of powers of 2:

$$n = \sum_{i=1}^{M} n_i, n_i = 2^{m_i}$$

This is called the *binary decomposition* of n.[2] We conduct parallel reductions over the M subsets of $n_i = 2^{m_i}$ elements, each one in time $\delta(n_i - 1)/p$, as explained above, a total time of $\delta(n - M)/p$. We repeat and reduce the M partial sums in subsets of sizes the binary decomposition of M, in a time $\delta(M - M_2)/p$, where M_2 is the number of terms in the binary decomposition of M, reduce the M_2 results, and repeat recursively until we obtain one final reduction of the whole set, in a total time $\delta(n - 1)/p$.

As for the transformation, we get a linear improvement, but with a less trivial algorithm, both to comprehend and design, and to implement in practice. All units must complete a step before the next step begins, so units must wait on one another, and the algorithm must incorporate some form of *synchronization*. Synchronization overhead, combined with the fact that the last steps involve fewer elements to sum-up than available units, makes a linear speed-up hard to achieve in practice.

Of course, this classical algorithm applies to all forms of reduction, not only sums, but also products, dot products, variance and covariance estimates, etc. Because it applies to dot products, it also applies to matrix products, where every cell in the result matrix is a dot product.

We have introduced two fundamental examples. One is embarrassingly parallel and fully benefits from parallelism without difficulty. The other one takes work and rarely achieves full parallel efficiency. In finance, derivatives risk management mainly involves two families of valuation algorithms: Monte-Carlo (MC) simulations and finite difference methods (FDM). MC simulations compute the average outcome over many simulated evolutions of the world. FDM computes the same result as the solution of a partial differential equation, numerically, over a grid. MC simulations are slow, heavy, and embarrassingly parallel. They are also applicable in a vast number of contexts. MC is therefore, by far, the most widely used method in modern finance; it is easy to implement in parallel, and the linear speed improvement makes a major difference due to the slow performance of the serial version. Part II is dedicated to the implementation of parallel MC. In contrast, FDM is light and fast, but applicable in a limited number of situations. FDM is

[2]To implement a binary decomposition is a trivial exercise, and an unnecessary one, since the chip's native representation of (unsigned) integers is binary.

less trivial to implement in parallel, and the resulting acceleration may not make a significant difference, FDM performing remarkably fast in its serial form. FDM and its parallel implementation are not covered in this text.

THE MANY FLAVORS OF PARALLEL PROGRAMMING

Distributed and concurrent programming

Parallel programming traditionally comes in two flavors: distributed and concurrent. The distributed flavor divides work between multiple *processes*. Processes run in parallel in separate memory spaces, the operating system (OS) *scheduling* their execution over the available processing units. Distributed computing is safe by design because processes cannot interfere with one another. It is also scalable to a large number of processing units, because different processes may run on the same computer or on multiple computers on a network. Since processes cannot communicate through memory, their synchronization is achieved through *messaging*. Distributed computing is very flexible. The cooperation of different nodes through messaging can accommodate many parallel designs.

Distributed programming is not implemented in standard C++ and requires specialized hardware and software.[3] The creation and the management of processes takes substantial overhead. Processes cannot share memory, so the entire context must be copied on every node. Distributed programming is best suited for *high-level* parallelism, like the distribution of a derivatives book, by transaction, over multiple machines in a data center. A lighter form of parallel computing, called *shared memory parallelism* or *concurrent programming*, is best suited for the implementation of parallelism in the lower level valuation and risk algorithms themselves.

Concurrent programming divides work between multiple *threads* that run on the same process and share common memory space. Threads are light forms of processes. Their execution is also scheduled by the OS over the available processing units, but limited to a single computer. The overhead for their creation and management is orders of magnitude lower. They share context and communicate directly through memory. The management of threads in C++ is standard and portable since C++11. All modern compilers, including Visual Studio since 2010, incorporate the Standard Threading Library out of the box. The threading library provides all the necessary tools and constructs for the manipulation of threads. It is also well documented, in particular in the book [15] written by one of its developers.

[3] A number of frameworks exist to facilitate distributed programming, the best known being the Message Passing Interface (MPI).

Concurrent programming is generally the preferred framework for the implementation of parallel algorithms. This text focuses on concurrent programming and does not cover distributed programming. Concurrent computing is not without constraints: it is limited to the processing units available on one computer, and, contrary to the distributed memory model, it is not safe by design. Threads may interfere with one another in shared memory, causing severe problems like crashes, incorrect results, slowdowns, or deadlocks, and it is the developer's responsibility to implement thread safe design and correct synchronization to prevent these. The Standard Threading Library provides a framework and multiple tools to facilitate correct concurrent programming. This library and its application to the development of safe, efficient parallel algorithms, is the main subject of this part.

CPU and GPU programming

A particular form of concurrent computing is with GPU (Graphic Processing Unit) parallelism. Modern GPUs provide a massive number of processing units for limited hardware costs. GPUs were initially designed to process the computation of two- and three-dimensional graphics, and later evolved to offer general-purpose computing capabilities. Nvidia, in particular, offers a freely available C++-like language, CUDA (Compute Unified Device Architecture), for programming their chips. GPU programming is implemented today in many scientific and commercial applications besides graphic design and video games. In particular, GPU accelerates many machine learning and financial risk management softwares. Reliable sources in the financial industry report speed improvements of order 50× for production Monte Carlo code; see for instance [16].

GPU programming is evidently not standard C++. In addition to specialized hardware and software, effective GPU programming requires algorithms to be written in a specific manner, down to low level, to accommodate the design and constraints of GPUs. GPU parallelism may come with unbeatable hardware cost; it is also subject to prohibitive development cost. For this reason, GPUs somewhat fell out of favor recently in the financial industry; see for instance [17].

CPU (Central Processing Unit) manufacturers like Intel have been systematically increasing the number of cores on their chips in recent years, so that CPUs may compute with GPUs, without the need for specialized GPU programming. High-end modern workstations now include up to 48 cores (up to 18 on Apple's slim iMac Pro). CPU cores are truly separate units, so different cores may conduct independent, unrelated work concurrently. GPU cores are not quite that flexible. CPU cores are also typically substantially faster than their GPU counterparts. More importantly, CPU parallelism is programmed in standard C++.

In addition, it is very challenging to effectively run memory-intensive algorithms, like AAD, on GPUs. Algorithmic adjoint differentiation (AAD) produces risk sensitivities in constant time (see our dedicated Part III), something no amount of parallelism can achieve. AAD is the most effective acceleration implemented in the financial industry in the past decade.[4] It is not particularly difficult to implement AAD on parallel CPU. We do that in Part III. On the other hand, to our knowledge, AAD has not yet been convincingly implemented on GPU, despite recent encouraging results by Uwe Naumann from RWTH Aachen University.

For those reasons, we don't explore GPU parallelism further and focus on CPU concurrency. Readers interested in learning GPU programming are referred to one of the many available CUDA textbooks.

Multi-threading and SIMD programming

Finally, concurrent CPU programming itself splits into two distinct, mutually compatible forms of parallelism: multi-threading and SIMD.

Multi-threading (MT) consists in the concurrent execution of different parts of some calculation on different threads running on different cores. The cores are effectively independent CPUs, so they may run unrelated calculations concurrently without loss of performance.[5] For example, the transformation of a collection can be multi-threaded by transforming different subsets over different threads. In this case, the different cores will execute the same sequence of instructions, but over different data. In a simple video game, the management of the player's star ship and the management of the enemies can be processed on two different threads. In that case, the two cores that execute the two threads conduct different, independent work. The game still runs twice as fast, providing a more satisfactory experience.

Multi-threading is a particularly flexible, efficient form of concurrent CPU programming, and the subject of Chapter 3. In Part II, it is applied to accelerate financial simulations.

By contrast, SIMD (Same Instruction Multiple Data) refers to the ability of every core in a modern CPU (or GPU) to apply the same instruction to multiple data, simultaneously. In a transformation, for instance, one single core could apply the function f to multiple xs simultaneously. SIMD applies CPU instructions to a vector of data at a time, as opposed to a single data. For this reason, it is also called "vectorization." Modern mainstream CPUs (implementing AVX2) can process four doubles (32 bytes, 256 bits) at a time.

[4] In our opinion, AAD is the most effective algorithm implemented in finance since FDM.

[5] Although they may interfere through shared memory and cache as we will see later in this part.

Higher-end CPUs (AVX512) can process eight doubles (64 bytes, 512 bits) in a single instruction.

SIMD may be combined with multi-threading, in a process sometimes called "GPU on CPU" because GPUs implement a similar technology. For the transformation of a collection, multiple subsets can be processed on different threads running on different cores while every core applies SIMD to process multiple elements of the subset in a single instruction.

The theoretical acceleration from combining MT with SIMD is the product of the number of cores by the SIMD vector width. In a standard 8-core workstation with AVX2, this is 32. On an 18-core high-end iMac Pro (AVX512), this is a vertiginous 144, well above the acceleration reportedly obtained on GPU. But this is theoretical of course. MT usually achieves linear acceleration as long as the code is correctly designed, as we will see in the rest of this book. SIMD acceleration is a different story.

SIMD is very restrictive: only simple functions applied to simple types may be vectorized. In addition, the data must be coalescent and properly aligned in memory. For example, a simple transformation $x \to \cos x$ of a set $S = \{x_i, 1 \leq i \leq n\}$ can be vectorized, and Visual Studio would indeed *automatically* vectorize a loop implementing this transformation. With a different function, like $x \to \log x$ if $x > 0$ or $-\log -x$ otherwise, the transformation cannot (easily) be vectorized: the effective CPU instruction that applies to each data depends on whether the number is positive or negative; we are no longer in a *same* instruction multiple data context. Visual Studio would not vectorize the transformation.

SIMD is low level, close to the metal parallelism. It is not implemented in standard C++ or supported in standard libraries. SIMD is implemented in the compiler. Contrary to multi-threading, it is not scheduled at run time. It is at compile time, when the C++ code is turned into CPU instructions, that vectorization occurs or not. Standard compilers like Visual Studio offer little support for SIMD. Visual Studio automatically vectorizes simple loops, and may be set to reports which loops are vectorized and which are not. Other compilers, like Intel's, offer much more extensive support for vectorization in the form of settings and pragmas.[6]

Intel also offers its Integrated Performance Primitives (IPP) and Math Kernel Library (MKL) as a free download. These libraries include a vast number of vectorized mathematical functions and algorithms. With direct relevance for Monte-Carlo simulations, IPP offers a vectorized implementation of the inverse Normal distribution *ippsErfcInv_64f_A26*().

[6]Special instructions inserted in the body of the code that are not part of the C++ code to be compiled, but some directives for the compiler about how code is to be compiled. Pragmas are therefore compiler specific.

Hence, and even though Visual Studio supports SIMD for the occasional auto-vectorization of loops, an effective, systematic implementation of vectorization takes a specialized compiler and specialized third-party libraries.

In addition, in order to systematically benefit from SIMD, code must be written in a special way, so that mathematical operations are applied simultaneously to multiple data at low level. SIMD acceleration is *intrusive*. We will see in Part II that this is not the case for MT. We implement parallel simulations without modification of the model or the product code (as long as that code is *thread safe*). To vectorize simulations, we would need to rewrite the simulation code in models and products entirely.

A full SIMD acceleration is only achieved when the exact same CPU instructions are applied to different data. Whenever we have control flow in the code (if, while and friends), SIMD may be partially applied at best. Partial parallelism is a known pitfall in parallel programming. When perfect parallelism (acceleration by the number of units p) is achieved in a proportion μ of some algorithm, a proportion $1 - \mu$ remains sequential, resulting in a global acceleration by a factor $p/[\mu + (1 - \mu)p]$,[7] very different from the perhaps intuitive but completely wrong $p\mu$. The resulting acceleration is typically counterintuitively weak. For instance, the perfect parallel implementation of 75% of an algorithm over 8 units results in an acceleration by a factor 2.9, terribly disappointing[8] and far from a perhaps mistakenly expected factor 6. Even when 90% of the algorithm is parallel with efficiency 100%, the resulting global acceleration is only 4.7, a parallel efficiency below 50%. Naive parallel implementation often causes bad surprises, and partial parallelism, in particular, generally produces disappointing results. Therefore, to achieve a linear acceleration with SIMD over complex, structured code is almost impossible. Besides, that theoretical limit is only 4× on most current CPUs. By contrast, a linear acceleration of up to 24× is easily achieved with MT. As long as the multi-threaded code is thread safe and lock-free, high-level multi-threading is, by construction, immune to partial parallelism. Of course, MT code is also vulnerable to sometimes challenging flaws and inefficiencies, which we will encounter, and resolve, in the chapters of this book. We do achieve a linear acceleration for our simulation code, including differentiation with AAD.

Finally, and most importantly, it is extremely challenging to combine SIMD with AAD. Vectorized AAD may be the subject of a future paper, but

---

[7] This result is known as Amdahl's law. It is immediately visible that, when the serial calculation time is Δ, the parallel calculation time is $\Delta[\mu/p + (1 - mu)]$, and the result follows.

[8] The ratio of the speed-up over the number of parallel units is called "parallel efficiency." Parallel efficiency is always between 0 (excluded) and 1 (included). The purpose of parallel algorithms is to achieve an efficiency as close as possible to 1.

it is way out of the scope of this book. In contrast, MT naturally combines with AAD, as demonstrated in detail in Chapter 12, where we apply AAD over multi-threaded simulations.

We see that the combination of MT with SIMD offers similar performance to GPU but suffers from the same constraints. This is not a coincidence. GPUs are essentially collections of a large number of slower cores with wide SIMD capabilities. Systematic vectorization requires rewriting calculation code, down to a low level, with a specialized compiler and specialized libraries. Of course, all of this also requires specialized skills. We will not cover SIMD in detail, and we will not attempt to write specialized code for SIMD acceleration. On the contrary, we will study MT in detail in Chapter 3, and implement it throughout the rest of this text.

We will also apply *casual* vectorization, whereby we write loops in a way to encourage the compiler to auto-vectorize them, for instance, by using the general algorithms of the standard C++ library, like *transform*() or *accumulate*(), in place of hand-crafted loops, whenever possible. Those algorithms are optimized, including for vectorization when possible, and typically produce faster code. In addition, it makes our code more expressive and easier to read. It is therefore a general principle to apply STL (Standard Template Library) algorithms whenever possible. C++ programmers must know those algorithms and how to use them. We use a number of STL algorithms in the rest of the publication, and refer to [18] for a complete reference.

Readers interested in further learning SIMD are referred to Intel's documentation. For an overview and applications to finance, we refer to [19]. To our knowledge, there exists no textbook on SIMD programming. MT programming is covered in detail in [15], and, in a more condensed form, in our Chapter 3.

Multi-threading programming frameworks

There exist many different frameworks for concurrent MT programming. The responsibility for the management of threads and their *scheduling* on hardware cores belongs to the operating system, and all major OS provide specific APIs for the creation and the management of threads. Those APIs, however, are generally not particularly user friendly, and obviously result in non-portable code. Vendors like Intel or Microsoft released multiple libraries that facilitate concurrent programming, but those products, while excellent, remain hardware or OS dependent.

Prior to C++11, two standardization initiatives were implemented to provide portable concurrent programming frameworks: OpenMP and Boost.Thread. OpenMP offers compile time concurrency and is supported by all modern compilers on all major platforms, including Visual Studio on

Windows. Its consists of a number of compiler directives, called pragmas, that instruct the compiler to automatically generate parallel code for loops. OpenMP is particularly convenient for the parallelization of simple loops, and we provide an example in Chapter 1. OpenMP is available out of the box with most modern compilers, and therefore results in portable code. Using OpenMP is also amazingly simple, as demonstrated in our example.

For multi-threading complex, structured code, however, we need the full flexibility of a threading library. Boost.Thread provides such a library, with run-time parallelism, in the form of functions and objects that client code uses to create and manage threads and send tasks for concurrent execution.[9] Boost is available on all major OSs, and offers a consistent API for the management of threads and parallel tasks while encapsulating OS and hardware-specific logic. Many developers consider Boost as "almost standard," and applications written with Boost.Thread are often considered portable. However, compilers don't incorporate Boost out of the box. Applications using Boost.Thread must include Boost headers and link to Boost libraries. This is not a particularly simple, user friendly exercise, especially on Windows. It is also hard work to update Boost libraries in a vast project, for instance, when upgrading to a new version of Visual Studio.

Since C++11, C++ comes with a Standard Threading Library, which is essentially a port of Boost.Thread, better integrated within the C++ language, and consistent with other C++11 innovations and design patterns. Contrary to Boost.Thread, the Standard Threading Library is provided out of the box with all modern compilers, including Visual Studio, without the need to install or link against a third-party library. Applications using the Standard Threading Library are fully portable across compilers, OS, and hardware. In addition, as part of C++11 standard, this library is well documented, and the subject of numerous dedicated books, the best in our opinion being [15]. The Standard Threading Library is our preferred way of implementing parallelism. With very few exceptions, all the algorithms and developments that follow use this library.

WRITING MULTI-THREADED PROGRAMS

So, there exists many flavors of parallel computing and a wide choice of hardware and software environments for a practical implementation of each. Distributed, concurrent, and SIMD processing are not mutually exclusive. They form a hierarchy from the highest to the lowest level of parallelism. In Chapter 1, we implement a matrix product by multi-threading over the

[9] Before Boost.Thread, the pthread C library offered similar functionality in plain C.

outermost loop and vectorizing the innermost loop. In a financial context, the risk of a bank's trading books can be split into multiple portfolios distributed over many computers. Each computer that calculates a portfolio may distribute multiple tasks concurrently across its cores. The calculation code that executes on every core may be written and compiled in a way to enable SIMD. Similarly, banks using GPUs distribute their trading book across several machines, and further split the sub-books across the multiple cores on a computer, each core controlling one GPU that implements the risk algorithm. Maximum performance is obtained with the combination of multiple levels of parallelism. For the purpose of this publication, we made the choice, motivated by the many reasons enumerated above, to cover multi-threading with the Standard Threading Library.

As Moore's law has been showing signs of exhaustion over the past decade with CPU speeds limiting around 4GHz, chip manufacturers have been multiplying the number of cores in a chip as an alternative means of improving performance. However, whereas a program automatically runs faster on a faster chip, a single-threaded code does not automatically parallelize over the available cores. Algorithms other than embarrassingly parallel must be rethought, code must be rewritten with parallelism in mind, extreme care must be given to interference between threads when sharing memory, and, more generally, developers must learn new skills and rewrite their software to benefit from the multiple cores. It is often said that the exhaustion of Moore's law terminated a free option for developers. Modern hardware comes with better parallelism, but to benefit from that takes work. This part teaches readers the necessary skills to effectively program parallel algorithms in C++.

PARALLEL COMPUTING IN FINANCE

Given such progress in the development of parallel hardware and software, programmers from many fields have been writing parallel code for years. However, this trend only reached finance in the very recent years. Danske Bank, an institution generally well known for its cutting-edge technology, only implemented concurrent Monte-Carlo simulations in their production systems in 2013. This is all the more surprising as the main algorithm in finance, Monte-Carlo simulations, is embarrassingly parallel. The reason is there was little motivation for parallel computing in investment banks prior to 2008–2011. Quantitative analysts almost exclusively worked on the valuation and risk management of exotics. Traders held thousands of exotic transactions in their books. Risk sensitivities were computed by "bumping" market variables one by one, then recomputing values. Values and risk sensitivities were additive across transactions. Hence, to produce the value and risk for a portfolio, the valuation of the transactions was

repeated for every transaction and in every "bumped" market scenario. A large number of small valuations were conducted, which encouraged high-level parallelism, where valuations were distributed across transactions and scenarios but the internal algorithms that produce a value in a given context remained sequential.

That changed with regulatory calculations like CVA (Counterparty Value Adjustment) that estimates the loss incurred from a counterparty defaulting when the net present value (PV) of all transactions against this counterparty is positive. CVA is not additive due to netting effects across different transactions, and must be computed for all transactions at once, in one (particularly heavy) simulation. A CVA is also typically sensitive to thousands of market variables, and risk sensitivities cannot be produced by bumping in reasonable time. The response to this particular challenge, and arguably the strongest addition to computational finance over the past decade, is AAD, an alternative to bumping that computes all derivatives together with the value in constant time (explained in detail in Part III). So we no longer conduct many light computations but a single, extremely heavy one. The time taken by that computation is of crucial importance to financial institutions: slow computations result in substantial hardware costs and the inability to compute risk sensitivities in time. The computations must be conducted in parallel to take advantage of modern hardware. Hence, parallel computing only recently became a key skill for quantitative analysts and a central feature of modern financial libraries. It is the purpose of this part to teach these skills.

We cover concurrent programming under the Standard Threading Library, discuss the challenges involved, and explain how to develop effective parallel algorithms in modern C++. C++11 is a major modernization of C++ and includes a plethora of new features, constructs, and patterns. The Standard Threading Library is probably the most significant one, and it was designed consistently with the rest of C++11. For this reason, we review the main innovations of C++11 before we delve into its threading library.

Chapter 1 discusses high-performance programming in general terms, shows some concrete examples of the notions we just introduced, and delivers some important considerations for the development of fast applications in C++ on modern hardware. Chapter 2 discusses the most useful innovations in C++11, and, in particular, explains some important modern C++ idioms and patterns applied, among other places, in the Standard Threading Library. These patterns are also useful on their own right. Chapter 3 explores many aspects of concurrent programming with the Standard Threading Library, shows how to manage threads and send tasks for parallel execution, and illustrates its purpose with simple parallel algorithms. We also build a *thread pool*, which is applied in Parts II and III to situations of practical relevance in finance.

CHAPTER 1

Effective C++

It is often said that quantitative analysts and developers should focus on algorithms and produce a readable, modular code and leave optimization to the compiler. It is a fact that substantial progress was made recently in the domain of compiler optimization, as demonstrated by the massive difference in speed for code compiled in release mode with optimizations turned on, compared to debug mode without the optimizations. It is also obviously true that within a constantly changing financial and regulatory environment, quantitative libraries must be written with clear, generic, loosely coupled, reusable code that is easy to read, debug, extend, and maintain. Finally, better code may produce a *linear* performance improvement while better algorithms increase speed by orders of magnitude. It is a classic result that 1D finite differences converge in ΔT^2 and ΔX^2 while Monte Carlo simulations converge in $\sqrt{N}$; hence FDM is preferable whenever possible. We will also demonstrate in Part III that AAD can produce thousands of derivative sensitivities for a given computation *in constant time*. No amount of code magic will ever match such performance. Even in Python, which is quite literally *hundreds of times* slower than C++, a good algorithm would beat a bad algorithm written in C++.

However, speed is so critical in finance that we cannot afford to overlook the low-level phenomena that affect the execution time of our algorithms. Those low-level details, including memory cache, vector lanes, and multiple cores, do not affect algorithmic complexity or theoretical speed, but their impact on real-world performance may be very substantial.

A typical example is memory allocation. It is well known that allocations are expensive. We will repeatedly recall that as we progress through the publication and strive to preallocate at high level the memory required by the lower level algorithms in our code. This is not an optimization the compiler can conduct on our behalf. We must do that ourselves, and it is not always easy to do so and maintain a clean code. We will demonstrate some techniques when we deal with AAD in Part III. AAD records every mathematical

operation, so with naive code, every single addition, subtraction, multiplication, or division would require an allocation. We will use custom *memory pools* to eliminate that overhead while preserving the clarity of the code.

Another expensive operation is *locking*. We lock *unsafe* parts of the code so they cannot be executed concurrently on different threads. We call *thread safe* such code that may be executed concurrently without trouble. All code is not always thread safe. The unsafe pieces are called *critical regions* and they may be *locked* (using primitives that we explore later in this part) so that only one thread can execute them at a time. But locking is expensive. Code should be thread safe *by design* and locks should be encapsulated in such a way that they don't produce unnecessary overhead. It is not only explicit locks we must worry about, but also hidden locks. For example, memory allocations involve locks. Therefore, all allocations, including the construction and copy of containers, must be banned from code meant for concurrent execution.[1] We will show some examples in Chapters 7 and 12 when we multi-thread our simulation library and preallocate all necessary memory beforehand.

Another important example is memory caches. The limited amount of memory located in CPU caches is orders of magnitude faster than RAM. Interestingly perhaps, this limitation is not technical, but economical. We *could* produce RAM as fast as cache memory, but it would be too expensive for the PC and workstation markets. We may envision a future where this ultra-fast memory may be produced for a reasonable cost, and CPU caches would no longer be necessary. In the meantime, we must remember caches when we code. CPU caches are a hardware optimization based on a locality assumption, whereby when data is accessed in memory, the same data, or some data stored nearby in memory, is likely to be accessed next. So, every access in memory causes a duplication of the nearby memory in the cache for faster subsequent access. For this reason, code that operates on data stored nearby in memory – or *coalescent* – runs substantially faster. In the context of AAD, this translates into a better performance with a large number of small tapes than a small number of large tapes,[2] despite "administrative" costs per tape. This realization leads us to differentiate simulations pathwise, and, more generally, systematically rely on *checkpointing*, a technique that differentiates algorithms one small piece at a time over short tapes. This is all explained in detail in Part III.

---

[1] Intel, among others, offers a concurrent lock-free allocator with its freely available Threading Building Blocks (TBB) library. Without recourse to third-party libraries, we must structure our code to avoid concurrent allocations altogether.

[2] The *tape* is the data structure that records all mathematical operations.

For now, we make our point more concrete with the extended example of an elementary matrix product. We need a simplistic matrix class, which we develop as a wrapper over an STL vector. Matrices are most often implemented this way, for example, in Numerical Recipes [20].

```
template <class T>
class matrix
{
        //  Dimensions
        size_t      myRows;
        size_t      myCols;

        //  Data
        vector<T>   myVector;

public:

        using value_type = T;

        //  Constructors
        matrix() : myRows(0), myCols(0) {}
        matrix(const size_t rows, const size_t cols)
            : myRows(rows), myCols(cols), myVector(rows*cols) {}

        //  Access
        size_t rows() const { return myRows; }
        size_t cols() const { return myCols; }
        //  So we can call matrix [i][j]
        T* operator[] (const size_t row)
            { return &myVector[row*myCols]; }
        const T* operator[] (const size_t row) const
            { return &myVector[row*myCols]; }
};
```

We test a naive matrix product code that sequentially computes the result cells as the dot product of each row vector on the left matrix with the corresponding column vector on the right matrix. Such code is a direct translation of matrix algebra and we saw it implemented in a vast number of financial libraries.

```
inline void matrixProductNaive(
const matrix<double>& a,
const matrix<double>& b,
matrix<double>& c)
{
    const size_t rows = a.rows(), cols = b.cols(), n = a.cols();

    //  Outermost loop on result rows
    for (size_t i = 0; i < rows; ++i)
    {
        const auto ai = a[i];
        auto ci = c[i];

        //  Loop on result columns
```

```
15          for (size_t j = 0; j < cols; ++j)
16          {
17              //  Innermost loop for dot product
18              double res = 0.0;
19              for (size_t k = 0; k < n; ++k)
20              {
21                  res += ai[k] * b[k][j];
22              }
23              //  Set result
24              c[i][j] = res;
25          }
26      }
27  }
```

This code is compiled on Visual Studio 2017 in release 64 bits mode, with all optimizations on. Note that the following settings must be set on the project's properties page, tab "C/C++":

- "Code Generation / Enable Enhanced Instruction Set" must be set to "Advanced Vector Extensions 2" to produce AVX2 code.
- "Language / OpenMP Support" must be set to "yes" so we can use OpenMP pragmas.

For two random 1,000 × 1,000 matrices, it takes around 1.25 seconds to complete the computation on our iMac Pro. Looking into the innermost loop, we locate the code on line 21, executed 1 billion times:

```
                res += ai[k] * b[k][j];
```

One apparent bottleneck is that $b[k][j]$ resolves into $(\&b.myVector[k * b.myCols])[j]$. The multiplication $k * b.myCols$, conducted a billion times, is unnecessary and may be replaced by an order of magnitude faster addition, at the cost of a somewhat ugly code, replacing the lines 17–22 by:

```
            //  Dot product
            double res = 0.0;
            const double* bkj = &b[0][j];
            size_t r = b.rows();
            for (size_t k = 0; k < n; ++k)
            {
                res += ai[k] * *bkj;
                bkj += r;
            }
```

And the result is *still* 1.25 second! The compiler was already making that optimization and we polluted the code unnecessarily. So far, the theory that optimization is best left to compilers holds. We revert the unnecessary modification. But let's see what is going on with memory in that innermost loop.

The loop iterates on *k* and each iteration reads *ai*[*k*] and *b*[*k*][*j*] (for a fixed *j*). Data storage on the *matrix* class is row major, so successive *ai*[*k*] are localized in memory next to each other. But the successive *b*[*k*][*j*] are distant by 1,000 doubles (8,000 bytes). As mentioned earlier, CPU caches are based on locality: every time memory is accessed that is not already duplicated in the cache, that memory *and the cache line around it, generally 64 bytes, or 8 doubles*, are transferred into the cache. Therefore, the access to *a* is cache efficient, but the access to *b* is not. For every *b*[*k*][*j*] read in memory, the line around it is unnecessarily transferred into the cache. On the next iteration, *b*[*k* + 1][*j*], localized 8,000 bytes away, is read. It is obviously not in the cache; hence, it is transferred along with its line again. Such unnecessary transfer may even erase from the cache some data needed for forthcoming calculations, like parts of *a*. So the code is not efficient, not because the number of mathematical operations is too large, but because it uses the cache inefficiently.

To remedy that, we modify the order of the loops so that the innermost loop iterates over coalescent memory for both matrices:

```
1  inline void matrixProductSmartNoVec(
2  const matrix<double>& a,
3  const matrix<double>& b,
4  matrix<double>& c)
5  {
6      const size_t rows = a.rows(), cols = b.cols(), n = a.cols();
7
8      //  zero result first
9      for (size_t i = 0; i < rows; ++i)
10     {
11         auto ci = c[i];
12         for (size_t j = 0; j < cols; ++j)
13         {
14             ci[j] = 0;
15         }
16     }
17
18     //  Loop on result rows as before
19     for (size_t i = 0; i < rows; ++i)
20     {
21         const auto ai = a[i];
22         auto ci = c[i];
23
24         //  Then loop not on result columns but on dot product
25         for (size_t k = 0; k < n; ++k)
26         {
27             const auto bk = b[k];
28             //  We still jump when reading memory,
29             //      but not in the innermost loop
30             const auto aik = ai[k];
31
32             //  And finally loop over columns in innermost loop
33             //      without vectorization to isolate impact of cache alone
```

```
34              #pragma loop(no_vector)
35              for (size_t j = 0; j < cols; ++j)
36              {
37                  //  No more jumping through memory
38                  ci[j] += aik * bk[j];
39              }
40          }
41      }
42  }
```

The pragma on line 34 will be explained ahead.

This code produces the exact same result as before, in 550 milliseconds, more than twice as fast! And we conducted just the same amount of operations. The only difference is cache efficiency. To modify the order of the loops is an operation too complex for the compiler to make for us. It is something we must do ourselves.

It is remarkable and maybe surprising how much cache efficiency matters. We increased the speed more than twice just changing the order of the loops. Modern CPUs operate a lot faster than RAM so our software is *memory bound*, meaning CPUs spend most of their time waiting on memory, unless the useful memory is cached in the limited amount of ultra-fast memory that sits on the CPU. When we understand this and structure our code accordingly, our calculations complete substantially faster.

And we are not quite done there yet.

What does this "#pragma loop(no_vector)" on line 34 stand for? We introduced SIMD (Single Instruction Multiple Data) in the Introduction. SIMD only works when the exact same instructions are applied to multiple data stored side by side in memory. The naive matrix product code could not apply SIMD because the data for b was not coalescent. This was corrected in the smart code, so the innermost loop may now be vectorized. We wanted to measure the impact of cache efficiency alone, so we disabled SIMD with the pragma "#pragma loop(no_vector)" over the innermost loop.

Visual Studio, like other modern compilers, *auto-vectorizes* (innermost[3]) loops whenever it believes it may do so safely and efficiently. If we remove the pragma but leave the code otherwise unchanged, the compiler should auto-vectorize the innermost loop[4]. Effectively, removing the pragma

[3]Evidently, given SIMD constraints, innermost loops are the only candidates for vectorization.

[4]Provided the setting "C/C++ / Code Generation / Floating Point Model" is manually set to Fast in the project properties page for the release configuration on the relevant platform, presumably x64. When this is not the case, Visual Studio does not vectorize reductions. Whether the reduction was vectorized or not can be checked by writing "/Qvec-report:2" in the "Additional Options" box in the "C/C++ / Command Line" setting.

further accelerates calculation by 60%, down to 350 milliseconds. The SIMD improvement is very significant, if somewhat short of the theoretical acceleration. We note that the innermost loop is a *reduction*, having explained in the Introduction how parallel reductions work and why they struggle to achieve parallel efficiency. In addition, data must be *aligned* in memory in a special way to fully benefit from AVX2 vectorization, something that we did not implement, this being specialized code, outside of the scope of this text.

What this teaches us is that we must be SIMD aware. SIMD is applied in Visual Studio outside of our control, but we can check which loops the compiler effectively vectorized by adding "/Qvec-report:2" (without the quotes) in the "Configuration Properties/ C/C++ / Command Line/ Additional Options" box of the project's properties. We should strive to code innermost loops in such a way as to encourage the compiler to vectorize them and then check that it is effectively the case at compile time.[5] To fail to do so may produce code that runs at half of its potential speed or less.

Altogether, to change the order of the loops accelerated the computation by a factor 3.5, from 1250 to 350 milliseconds. Over half is due to cache efficiency, and the rest is due to vectorization.

Always try to structure calculation code so that innermost loops sequentially access coalescent data. Do not hesitate to modify the order of the loops to make that happen. This simple manipulation accelerated our matrix product by a factor close to 4, similar to multi-threading over a quad core CPU.

Finally, we may easily distribute the *outermost* loop over the available CPU cores with another simple pragma above line 19:

```
//  OpenMp directive: execute loop in parallel
#pragma omp parallel for
for (int i = 0; i < rows; ++i)
{
    const auto ai = a[i];
    auto ci = c[i];

    for (size_t k = 0; k < n; ++k)
    {
        const auto bk = b[k];
        const auto aik = ai[k];
```

[5] Visual Studio is somewhat parsimonious in its auto-vectorization and frequently declines to vectorize perfectly vectorizable loops (although it would never vectorize a loop that should not be vectorized). Therefore we must check that our loops are effectively vectorized and, if not, rewrite them until such time the compiler finally accepts to apply SIMD. This may be a frustrating process. STL algorithms are generally easier auto-vectorized than hand-crafted loops, yet another reason to prefer these systematically.

```
            for (size_t j = 0; j < cols; ++j)
            {
                ci[j] += aik * bk[j];
            }
        }
    }
```

This pragma is an OpenMP directive that instructs the compiler to multi-thread the loop below it over the available cores on the machine running the program. Note that we changed the type of the outermost counter *i* from *size_t* to *int* so OpenMP would accept to multi-thread the loop. OpenMP's auto-parallelizer is somewhat peculiar this way, not unlike Visual Studio's auto-vectorizer.

This code produces the exact same result in just 40 milliseconds, approximately 8.125 times faster, more than our number (8) of cores! This is due to a so-called "hyper-threading" technology developed by Intel for their recent chips, consisting of two *hardware threads* per core that allow each core to switch between threads at hardware level while waiting for memory access. The OS effectively "sees" 16 hardware threads, as may be checked on the Windows Task Manager, and their scheduling over the 8 physical cores is handled on chip. Depending on the context, hyper-threading may increase calculation speed by up to 20%. In other cases, it may *decrease* performance due to excessive *context switches* on a physical core when execution switches between two threads working with separate regions of memory.

Contrary to SIMD, multi-threading is not automatic; it is controlled by the developer. In very simple cases, it can be done with simple pragmas over loops as demonstrated here. But even in these cases, this is not fully satisfactory. The code is multi-threaded at compile time, which means that it always runs concurrently. But users may want to control that behavior. For instance, when the matrix product is part of a program that is itself multi-threaded at a higher level, it is unnecessary, and indeed decreases performance, to run it concurrently. Concurrency is best controlled at run time. Besides, to multi-thread complex, structured code like Monte-Carlo simulations, we need more control than OpenMP offers. In very simple contexts, however, OpenMP provides a particularly light, easy, and effective solution.

Altogether, with successive modifications of our naive matrix product code, but without any change to the algorithm, its complexity, or the mathematical operations involved, we increased the execution speed by a factor of 30, from 1,250 to 40 milliseconds. Obviously, the results are unchanged. We achieved this very remarkable speed-up by tweaking our code to take full advantage of our modern hardware, including on-chip cache, SIMD, and multiple cores. It is our responsibility to know these things and to develop code that leverages them to their full potential. The compiler will not do that for us on its own.

CHAPTER 2

Modern C++

C++ was thoroughly modernized in 2011 with the addition of a plethora of features and constructs borrowed from more recent programming languages. As a result, C++ kept its identity as the language of choice for programming close to the metal, high-performance software in demanding contexts, and at the same time, adapted to the demands of modern hardware, adopted modern programming idioms borrowed from the field of functional programming, and incorporated useful constructs into its syntax and standard library that were previously available only from third party libraries.

The major innovation is the new Standard Threading Library, which is explored in the next chapter. Since we are using new C++11 constructs in the rest of the text, this chapter selectively introduces some particularly useful innovations. A more complete picture can be found online or in up-to-date C++ textbooks. Readers familiar with C++11 may easily skip this chapter.

2.1 LAMBDA EXPRESSIONS

One of the most useful features in C++11, borrowed from the field of *functional programming*, is the ability to define anonymous function objects on the fly with the *lambda* syntax like in:

```
auto myLambda = [] (const double r, const double t)
{
    double temp = r * t; return exp( -temp);
};
```

The new *auto* keyword provides automatic type deduction, which is particularly useful with lambdas that produce a different compiler-generated type for every lambda declaration. A lambda declaration starts with a *capture* clause [], followed by the list of its arguments (a lambda is after all a function, or more exactly a *callable object*), and its *body*, the sequence of instructions that are executed when the lambda is called, just like the body of functions and methods. The difference is that lambdas are declared

within functions. Once declared, they may be called like any other function or function object, for example:

```
cout << myLambda( 0.01, 10) << endl;
```

One powerful feature of lambdas is their ability to *capture* variables from their environment *on declaration*.

[]	means no capture.
[=]	means capture *by value*, that is, by copy, of all variables in scope *used in the lambda's body*.
[&]	means capture all variables *by reference*.

We may also capture variables selectively with the syntax:

[x]	means capture only x, by value with this syntax or by reference with [&x].
[=, &x, &y]	means capture x and y by reference, and all others by value. Obviously $[\&, x, y]$ means capture x and y by value and all others by reference.
[x, &y]	means capture x by value, y by reference and nothing else.

For instance,

```
1  double mat = 10;
2  auto myLambdaRef = [&mat] (const double r)
3  {
4      return exp( -mat * r);
5  };
6  auto myLambdaCopy = [mat] (const double r)
7  {
8      return exp( -mat * r);
9  };
10
11 cout << myLambdaRef( 0.01) << endl;
12 //  exp( -10 * 0.01)
13 cout << myLambdaCopy( 0.01) << endl;
14 //  exp( -10 * 0.01)
15
16 mat = 20;
17 cout << myLambdaRef( 0.01) << endl;
18 //  exp( -20 * 0.01)
19 cout << myLambdaCopy( 0.01) << endl;
20 //  exp( -10 * 0.01)
```

Behind the scenes, the compiler creates a *function object* when we declare a lambda, that is, an object that defines the operator () (with the arguments of the lambda) and therefore is *callable* (like a function). The captured variables are implicitly declared as data members with a value type when captured by value and a reference type when captured by reference, initialized with the captured data on declaration. Hence, the syntax:

```
void main()
{
    double a, x;
    //  ...
    auto l = [=, &x] (const double y) { return a*x*y; }
    //  ...
    double z = l(y);
}
```

is equivalent to the (much heavier):

```
class Lambda
{
    const double  myA;
    const double& myX;

public:

    Lambda(const double a, const double& x) : myA(a), myX(x) {}

    operator() (const double y) const { return myA * myX * y; }
};

void main()
{
    double a, x;
    //  ...
    Lambda l(a, x);
    //  ...
    double z = l(y);
}
```

As a function object, a lambda can be passed as an argument or returned from functions. Functions that manipulate functions are called *higher-order functions*, and the standard <algorithm> library provides a vast number of these.

Lambdas are also incredibly useful as *adapters* and resolve a constant annoyance C++ developers face when calling functions with signatures inconsistent with their data. The Standard Template Library (STL), for instance, includes a wealth of useful generic algorithms. But to use these algorithms we must respect their functions' signatures. Say we hold a vector of times from today:

```
vector<double> times;
```

and we want to compute an annuity given a constant rate r. We could write a hand-crafted loop, of course:

```
double ann = 0.0;
for(size_t i = 0; i < times.size(); ++i)
{
    ann += exp(-r*times[i]);
}
```

but it is considered best professional practice to apply generic algorithms instead.[1] The computation we just conducted is a *reduction*, where a collection is traversed sequentially and an *accumulator* is updated for each element. The STL algorithm for reductions is *accumulate*(), located in the <numeric> header. The version of interest to us has the following signature:

```
template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );
```

The type *T* of the accumulator in our case is double, as is *InputIt, so the function *op* that updates the accumulator *acc* for each element *x* must be consistent with the form:

```
double op(const double& acc, const double& x);
```

but our instruction for the update of the accumulator is:

```
acc += exp(-r*x);
```

and prior to C++11, it would have been such an annoyance to squeeze that line of code into the required signature that we would probably have ended up with the hand-crafted loop. With the lambda syntax, it takes a line to do that right:

```
1 double ann = accumulate(times.begin(), times.end(), 0.0,
2               [r] (const double& acc, const double& x)
3               {
4                 return acc + exp(-r*x);
5               });
```

There are of course many other uses of lambdas, and we will discuss a few later, but their ability to seamlessly adapt data to signatures is the reason why we use them every day.

C++11 also provides dedicated adapter functions *bind*() and *mem_fn*() in the <functional> header (the latter turning member functions *class.func*() into free functions *func*(*class*)), although lambdas can also do this in more convenient manner. The syntax for *bind*() in particular is rather peculiar and it is easier to achieve the exact same behavior with lambdas.

We will be working with lambdas throughout the book.

2.2 FUNCTIONAL PROGRAMMING IN C++

The introduction of lambdas is part of an effort to modernize C++ with idioms borrowed from the growing and fashionable field of functional programming. Although C++ does not, and never will, support functional programming idioms the way a language like Haskell does, C++ does

[1]For reasons explained for instance in Scott Meyers [18].

support some key elements of functional programming, in particular *value semantics for functions* and *higher-order functions*.

Value semantics means that functions may be manipulated just like other types and in particular they can be assigned to variables and passed as arguments or returned as results by higher-order functions. Note that lambdas are literals for functions, which means that the instruction:

```
auto f = [] (const double x) { return 2*x; };
```

assigns a *function literal* to *f* in the same way we assign number or string literals in:

```
double x = 2.0;
string s = "C++11";
```

C++11 defines the *function* template class in the <functional> header as a unique class for holding functions and *anything callable*. That means that a concrete type like

```
function<double(const double)>
```

can hold anything that may be called with a double to return a double: a C style function pointer, a function object, including a lambda, or a member function bound to an object. An object of that type is itself callable of course, and it has value semantics, in the sense that it can be assigned or passed as an argument, or returned as a result from a higher-order function.

It looks peculiar and at first sight impossible in C++ to define a type based on the behavior rather than the nature of the objects it holds.[2] *function* is implemented with an interesting, advanced design pattern called *type erasure*. Unfortunately, this versatility comes with a cost. Type erasure necessarily involves the storage of the underlying objects on the heap. Hence, to initialize, assigning or copying a *function* object involves an allocation.[3] For this reason, we refrain from using this class despite its convenience, and manipulate functions as template types instead.[4]

Composition

As a first example, we consider the composition of functions, and write a (higher-order) function that takes two functions as arguments and returns the *function* resulting from their composition.

[2]This is sometimes called "duck-typing": if it walks like a duck and quacks like a duck, then it is a duck.

[3]Visual Studio implements a small object optimization (SMO) whereby a small buffer is allocated for every object on the stack to minimize heap allocations.

[4]The implementation of type erasure is outside of our subject; interested readers can find information online, in particular on Stack Overflow.

```
template<class F1, class F2=F1>
auto compose(const F1& f, const F2& g)
{
    return [=](const auto& x) { return f(g(x)); };
}
```

We use the auto keyword so that types are deduced at compile time. Note that it is a function, not a number, that is returned. For instance, the following code creates a function by composing an exponential with a square root:

```
int main()
{
    auto f = compose([](const double x) { return exp(x); },
        [](const double x) { return sqrt(x); });
    cout << f(0.5) << endl;
}
```

Lambdas are obviously unnecessary here; they wrap the functions *exp*() and *sqrt*() without adapting anything. However, the following does not compile on Visual Studio:

```
int main()
{
    auto f = compose(exp, sqrt);
    cout << f(0.5) << endl;
}
```

Standard mathematical functions are overloaded so they work with many different types, and the compiler doesn't know which overload to pick to instantiate the templates. For this reason, we must explicitly state the function types when we compose standard functions, as follows:

```
auto f = compose<double(double)>(exp, sqrt);
```

We are not limited to numerical functions. Any function that takes an argument of type T_1 and returns a result of type T_2 (which we denote $f : T_1 -> T_2$) may be composed with any function $g : T_2 -> T_3$ to create a function $h : T_1 -> T_3$. We can imagine a function that creates a vector $1..n$ out of an unsigned integer:

```
vector<unsigned> generateVec(const unsigned n)
{
    vector<unsigned> result(n);
    generate(result.begin(), result.end(),
        [counter = 0]() mutable { return ++counter; });
    return result;
}
```

where *generate*() is an STL algorithm from the header <algorithm> that fills a sequence by repeated calls to a function, and the lambda is marked

mutable because its execution modifies its internal data *counter*. We can code a function that sums up the values in a vector:

```
template <class T>
T accumulateVec(const vector<T>& v)
{
    return accumulate(v.begin(), v.end(), T());
}
```

where the STL *accumulate*() algorithm was discussed earlier. We could define a (particularly inefficient) way to compute the sum of the first *n* numbers by composition:

```
int main()
{
    auto h = compose(accumulateVec<unsigned>, generateVec);

    cout << h(100) << endl;
}
```

We could even design ways to compose functions of *multiple* arguments, either by binding or currying. We have to stop here and refer interested readers to a specialized publication like [21].

Lifting

Another useful idiom borrowed from functional programming is *lifting*. To lift a function means to turn it into one that operates on compound types. For instance, we may implement a lift that turns a scalar function into a vector function that applies the original function to all the elements of a vector:

```
template <class F>
auto lift(const F& f)
{
     return [f](const vector<double>& v)
     {
         vector<double> result(v.size());
         transform(v.begin(), v.end(), result.begin(), f);
         return result;
     };
}
```

transform() is a generic STL algorithm from header <algorithm> that applies a unary function to all the elements in a collection. What is returned from *lift*() is not a vector but a function of a vector that returns a vector. It can be used as follows (we lift the *exp*() function into a *vExp*() that computes a vector of exponentials from a vector of numbers):

```
int main()
{
    auto vExp = lift<double(double)>(exp);

    vector<double> v = { 1., 2., 3., 4., 5. };
    vector<double> r = vExp(v);

    for_each(r.begin(), r.end(),
        [](const double& x) {cout << x << endl; });
}
```

for_each() is another generic algorithm from the <algorithm> header that sequentially applies an action to all the elements in a collection. We use it to display the entries in the result vector *r*.

As a (slightly) more advanced example, suppose we have a function that implements the Black and Scholes formula from [22]:

```
double blackScholes(const double spot,
    const double strike,
    const double expiry,
    const double vol);
```

We can lift it into a function that computes a vector of option prices from a vector of spots, but we must first turn it into a function of the spot alone by binding the other arguments. That could be done with a lambda, or with the *bind*() function from the header <functional>:

```
#include <functional>
using namespace std;
using namespace placeholders;

int main()
{
    //  Create unary pricing function out of spot alone
    //      by binding the other arguments
    auto BSfromS = bind(
        blackScholes,   //  Function to bind
        _1,             //  spot = 1st arg of bound function
        100.,           //  strike = 100
        1.,             //  maturity = 1
        .10);           //  vol = 10

    //  Lift the unary function into a vector function
    auto vBlackScholes = lift(BSfromS);

    //  Apply the lifted function to a vector of spots
    vector<double> spots = { 50., 75., 100., 125., 150. };
    vector<double> calls = vBlackScholes(spots);

    //  Display results
    for_each(calls.begin(), calls.end(),
        [](const double& x) {cout << x << endl; });
}
```