View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Edinburgh Research Archive

THE UNIVERSITY of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Applying
High Performance Computing
to
Profitability and Solvency Calculations
for
Life Assurance Contracts

Mark Tucker

Doctor of Philosophy
The University of Edinburgh

2016

“As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise – By what course of calculation can these results be arrived at by the machine in the *shortest time*?”

Charles Babbage, 1864. [1]

Abstract

Throughout Europe, the introduction of Solvency II is forcing companies in the life assurance and pensions provision markets to change how they estimate their liabilities. Historically, each solvency assessment required that the estimation of liabilities was performed once, using actuaries' views of economic and demographic trends. Solvency II requires that each assessment of solvency implies a 1-in-200 chance of not being able to meet the liabilities. The underlying stochastic nature of these requirements has introduced significant challenges if the required calculations are to be performed correctly, without resorting to excessive approximations, within practical timescales.

Currently, practitioners within UK pension provision companies consider the calculations required to meet new regulations to be outside the realms of anything which is achievable. This project brings the calculations within reach: this thesis shows that it is possible to perform the required calculations in manageable time scales, using entirely reasonable quantities of hardware. This is achieved through the use of several techniques: firstly, a new algorithm has been developed which reduces the computational complexity of the reserving algorithm from $O(T^2)$ to $O(T)$ for T projection steps, and is sufficiently general to be applicable to a wide range of non unit-linked policies; secondly, efficient *ab-initio* code, which may be tuned to optimise its performance on many current architectures, has been written; thirdly, approximations which do not change the result by a significant amount have been introduced; and, finally, high performance computers have been used to run the code.

This project demonstrates that the calculations can be completed in under three minutes when using 12,000 cores of a supercomputer, or in under eight hours when using 80 cores of a moderately sized cluster.

Lay Summary

Throughout Europe, the introduction of Solvency II is forcing companies in the life assurance and pensions provision markets to change how they estimate their liabilities. Historically, each solvency assessment required that the estimation of liabilities was performed once, using actuaries' views of economic and demographic trends. Solvency II requires that each assessment of solvency implies a 1-in-200 chance of not being able to meet the liabilities. The underlying complexity of these requirements has introduced significant challenges if the required calculations are to be performed correctly, without resorting to excessive approximations, within practical timescales.

It is estimated that performing the required calculations for a representative portfolio of half a million annuity policies, using commercially available software, running on hardware which the industry is comfortable with, would take about 2800 years on a single core of a desktop PC. The elapsed time can be reduced by splitting the calculations over multiple cores on many PCs. However, it would require more than 35,000 quad-core desktop PCs, running at full capacity for over a week, to estimate the liabilities for this portfolio. Therefore, practitioners within UK pension provision companies consider the calculations required to meet the new regulations to be outside the realms of anything which is practical.

The work in this thesis shows that it is possible to perform the calculations required by the new regulations in far shorter time scales, using far smaller quantities of hardware. This is achieved by a) writing efficient code, rather than using commercially developed software; b) changing the underlying computational method; and, c) using high performance computers.

This project demonstrates that the calculations can be completed either during a coffee break when using a supercomputer, or overnight when using a small collection of multi-core CPUs.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own, except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification, except as specified in the text.

Some of this work has been published as follows:

- parts of Chapters 3 and 4 have appeared in
  Tucker, M. and Bull, J.M., *An efficient algorithm for the calculation of reserves for non unit-linked life policies.*, Algorithmic Finance (2014), 3:3-4, 143-161.
- parts of Chapter 5 have appeared in
  Tucker, M. and Bull, J.M., *The Application of High Performance Computing to Solvency and Profitability Calculations for Life Assurance Contracts.*, in proceedings of *5th Workshop on High Performance Computational Finance, SuperComputing, 2012.* (Published in *High Performance Computing, Networking, Storage and Analysis 2012 (SC Companion)*).
- parts of Chapter 8 have appeared in
  Tucker, M. and Bull, J.M., *Fulfilling Solvency II Regulations using High Performance Computing.*, in proceedings of *8th Workshop on High Performance Computational Finance, SuperComputing, 2015.* (Published in *High Performance Computing, Networking, Storage and Analysis 2015 (SC Companion)*).

Mark Tucker,
December 2016.

Acknowledgements

I would like to thank Mark Bull: as my supervisor, his patience, guidance, and suggestions have made this project what it is.

I would also like to thank Fiona Reid: her tutorials on the use of EPCC's high performance computers were invaluable.

I would also like to thank Gavin Conn: his explanation of Solvency II gave this project a worthwhile goal.

I must acknowledge Moose, Bert, and Ernie, the three German Shepherd dogs which have been in my life during this project: they have been a constant source of stress relief, and the time spent walking with them has solved many problems, and brought new insights.

Finally, I must thank my wife, Nicola: without her constant support throughout the entire period leading up to the submission of this thesis the project could not have been completed.

Contents

Abstract i

Lay Summary ii

Declaration iii

Acknowledgements iv

Contents v

1 Introduction 1

1.1 Demonstrating Solvency ... 1

1.2 Profitability ... 2

1.3 Annuities ... 2

1.4 Software ... 3

1.5 Change in Regulations ... 4

1.6 Aims of This Project ... 5

1.7 Thesis Layout ... 7

2 Background 9

2.1 Actuarial Concepts ... 9

2.2 Domain Specific Languages ... 19

2.3 Use Cases ... 23

2.4 Computer Processors ... 31

2.5 High Performance Computing ... 35

2.6 Related Work ... 50

2.7 Platforms Used 55

3 Change of Reserving Algorithm 58

3.1 Motivational Examples 58

3.2 The General Case 63

3.3 Zero Reserve States 70

3.4 Implementational Considerations 73

3.5 Summary 77

4 Use of the Recurrence Algorithm 78

4.1 Single Life Policies 78

4.2 Two-Life Policies 85

4.3 Extension to Other Policies 90

4.4 The General Case 92

4.5 Summary 94

5 Improving the Performance of Profitability Calculations 95

5.1 Initial Code 95

5.2 Optimisations 99

5.3 Performance Implications 106

5.4 Summary 106

6 Simplifying Interpolation in the Mortality Table 108

6.1 Overview 109

6.2 Force of Mortality 110

6.3 Linear Interpolation in the l_x's 114

6.4 Implementation in a Commercial Environment 117

6.5 Linear Interpolation in the q_x's 119

6.6 Alternative Implementation 121

6.7 Effect of Interpolation Method 124

6.8 Summary 127

7 Re-drawing Parameters 128

7.1 To Re-draw or Not To Re-draw 128

7.2 Mathematical Representation 129

7.3 Algorithms and Their Complexity 135

7.4 Relative Merits of Re-drawing or Not 138

7.5 Summary 139

8 Fulfilling Solvency II Requirements 144

8.1 Initial Code 145

8.2 Optimisations 149

8.3 Simplifications 159

8.4 Performance when Re-Drawing Parameters 161

8.5 Performance when Not Re-Drawing Parameters 177

8.6 Summary 181

9 Future Work 183

9.1 Further Work with CPUs 183

9.2 The General Case 186

9.3 Alternative Technologies 188

9.4 Other Uses 191

9.5 Nested Stochastic Projections 193

9.6 Summary 194

10 Conclusion 195

10.1 Review 195

10.2 Summary 200

Bibliography 203

List of Figures 211

List of Tables 213

List of Algorithms 216

A Estimation of Run Time for Brute Force Solvency II in the Commercial Environment 217

A.1 Initial Estimate 217

A.2 Allowing for Technological Advances 220

B Hardware Specifications 221

B.1 Features of a Xeon CPU 221

B.2 Nodes on the Xeon Cluster 222

B.3 Features of an NVIDIA GPU 223

B.4 Comparisons 223

C Number of States for Assurances 225

D Regression Analyses 227

D.1 Summation *vs.* Recurrence Relation 227

D.2 Single Life and Reversionary Annuities 229

E Probability of Transition before Time t 230

E.1 Single Life Policies 230

E.2 Two-Life Policies 235

E.3 Continuous Time 246

Chapter 1

Introduction

1.1 Demonstrating Solvency

All insurance businesses operating within the UK are regulated to ensure that, among other things, they demonstrate, on a regular basis, that they are solvent. At its simplest, solvency boils down to holding sufficient assets (cash, stocks, bonds, Gilts, etc.) so that the income from those assets (both regular interest and dividends, etc., and the proceeds from their disposal), together with future premium income, is sufficient to meet the liabilities arising from, and the expenses attributable to, the policies to which those assets relate. This may be interpreted simplistically as

$$\text{solvency} \Leftrightarrow \begin{cases} \text{present value of (assets and future income)} \\ \text{exceeds} \\ \text{present value of (liabilities and future expenses)} \end{cases}$$

In many commercial environments, the assets and liabilities are valued separately, on completely different systems. Accordingly, this investigation only considers liabilities within a life assurance office, i.e. the payments to the policyholder, and the associated expenses.

The present value of the future payments to the policyholder can be calculated using methods which are elementary to actuarial science: the basic concept is to allow for interest accruing on the funds currently held, and for the probability of making each payment – this will be discussed further in Section 2.1.

The amount which is currently required to be held to meet these liabilities is the *reserve*: the reserve is a fundamental quantity in determining solvency. The available assets must exceed the liabilities: it is usual to hold a *solvency margin*, in excess of the reserve, which acts as an additional cushion.

1.2 Profitability

Further interest in the values of the assets and liabilities lies in the fact that the timings of the cash flows, and the order in which they happen, can be used to estimate the future profit arising from the policies. Each year payments will be made to policyholders, expenses will be paid, premiums will be received from policyholders, and income will be received on the funds held: these items form the basis of the estimation of profitability. Therefore, the estimation of profit which can emerge from each policy also requires knowledge of future reserves.

There are two main categories of policies which are common. Firstly, those where premiums are received from the policyholder over time in order to accrue a benefit at a later time, for example endowments to cover a mortgage: the premiums build up so that, throughout a year, under normal investment conditions, the reserve increases. Secondly, those where a large premium is paid at the outset in return for a sequence of smaller benefits in the future: the amount remaining for each policy will be smaller as a result of each payment, leading to a decrease in reserves. Crucially, this *change of reserves* dictates that all reserves do need to be calculated in each future step of the projection: the formulae for the calculation of reserves may have quite complicated forms; for example, see Equation 2.1.5.

Moreover, to increase the accuracy of the information available for running the business, and to allow for the fact that most policies have cash flows which occur more frequently than yearly, the whole projection of all of this business is often performed using monthly steps rather than yearly steps. While this does add to the accuracy, it also adds to the calculation count, and hence to the run time.

1.3 Annuities

This project focuses on calculations relating to life annuities since the financial well-being of an increasing proportion of the population depends on companies' abilities to

make pension payments. An annuity portfolio within a representative life assurance office will consist of several types of policy: policies where payments are currently being made to the policyholder may be represented as single life annuities (see Section 2.1.5.1); policies where payments could, at some future point, be made to the spouse of a policyholder currently receiving payments may be represented as reversionary annuities (see Section 2.1.5.2); and, policies where payments are currently being made, so long as a specified collection of a group of lives remain alive may be represented as either joint life annuities or last survivor annuities. To allow for this diversity, this project considers all four of these policy types.

The conceptual simplicity of annuity policies makes them ideal for explorational implementations of new programming techniques. The calculations involved in the processing of these policies expose ample parallelism, allowing the investigation of both the combination of existing programming techniques and emerging technologies.

1.4 Software

Within the industry, current standard practice is that the calculation of reserves, and estimation of profitability, are performed using computer programs which are generated by specialist valuation software packages. These packages are generally crafted for ease of use, rather than performance of execution, and there is usually only limited scope for improving the performance of the programs they produce.

These packages require the relationships between variables to be entered in a pseudo-code style language, and the package then produces source code, compiles and links it, and launches the resulting executable. The clear advantage of these packages is that quite complex programs can be developed by users who are not trained programmers, allowing users who understand the intricacies of the policies to produce the programs: this is clearly beneficial when complicated contracts are being modelled. The fact that the users are usually not trained programmers is also one of the main disadvantages of these packages: the art of programming is removed by the package, and this usually results in the executable having sub-optimal performance.

The financial results required are the totals across all policies. However, in general the calculations performed for one policy do not affect the calculations for any other policy. This independence leads to the possibility of using HPC techniques and emerging technologies to process the policies in parallel.

An initial performance benchmark was obtained using a standard valuation package within a life assurance office. Using that package, on reasonably modern PCs, the times taken for a program produced by the package to perform a profitability estimate on annuities were about 1.0 seconds per policy for single life annuities, and roughly 2.5 seconds per policy for reversionary annuities: it took roughly 35 CPU core hours to produce results, using monthly projection steps, for a portfolio of about 129,000 single life annuity policies, and about 22.5 CPU core hours to process about $35,000$ reversionary annuity policies.

1.5 Change in Regulations

Historically, the calculations used for the demonstration of solvency were based on a single, 'best estimate', basis, which is a set of assumptions about future interest and mortality rates: these assumptions were set by actuaries and the calculations were usually performed using software from commercial valuation packages, running on PCs. It is currently not uncommon for life assurance offices to have a few hundred PCs as dedicated 'slaves' to perform these calculations. However, new regulations are being introduced and these require that the best estimate approach is replaced with Monte Carlo simulation of a range of scenarios based on different assumptions.

Solvency II [14] requires that an additional capital requirement, based on a 1-in-200 worst-case scenario in each future time step, is found. The minimum required to fulfil the regulations would be to perform 200 simulations at each step, and take the worst result. However, statistically, this is not particularly robust in terms of outliers. Therefore, one widely-adopted approach to attempt to satisfy Solvency II is to generate 1000 Monte Carlo scenarios in each future time step, and calculate the liabilities using each scenario; from these 1000 scenarios, the one which produces the 5th largest liability value is sought since that corresponds to a 0.5% chance of the actual liabilities being larger than the value reported. Of necessity, this approach dictates that the results of 99.9% of all calculations performed are discarded.

These new requirements, if implemented naively, lead to the need for a far greater volume of calculations: the number of calculations is so vast that the use of commercially available software to obtain the results is currently beyond contemplation. The fact that reserves must be calculated in each future time step means that existing profitability calculations may be used to estimate the run time. By using the benchmark times as a base, considering a moderately sized portfolio of half a million policies, consisting of

300,000 single life annuities, 100,000 reversionary annuities, 50,000 joint life annuities, and 50,000 last survivor annuities, and assuming monthly steps for a period of 60 years, an estimate (derived in Appendix A) of the run time for a projection with 1000 scenarios in each future time step, using 1 CPU core, is about 9130 years.

The estimate of 9130 core years is based on the CPU in mid-range PCs and commercial software available in life offices at the time this project was first mooted. However, as discussed in Section A.2 (in Appendix A), advances in CPU technology, from chips used within commercial environments at the start of this project, to those used in supercomputers at the end of this project, and improvements in compilers over the same period, lead to a speedup of about 3.25×. It would therefore be reasonable to expect that the calculations would take around 2810 CPU core years if an up-to-date CPU core and modern compiler were to be used. An alternative view of this is that, in order to meet reporting deadlines of two weeks, so that the maximum acceptable run time was one week, the processing would require in excess of 145,000 CPU cores continuously running at full capacity. The prospect of running more than 35,000 quad-core desktop PCs at full capacity for a week leads most practitioners in the UK to conclude that the calculations are beyond contemplation.

As an alternative to using PCs as slaves (as outlined in Section 1.4), some life offices now have clusters consisting of a few hundred cores. However, even with this amount of hardware, their processing capacity is of the order of a few percent of the estimated requirement. Therefore, it is generally acknowledged among practitioners within UK life offices that, using current commercially available valuation software on hardware which the industry is currently comfortable with, it is impossible to fully comply with the incoming Solvency II regulations.

Although the estimate of run time derived in Appendix A is based on the approach used within the life industry, it is rather naive. Chapter 7 demonstrates that the correct choice of algorithm, if it could be implemented using standard commercial software, could reduce the estimate by a factor of around 100. However, solely changing the algorithm does not reduce the time to one which is practical in a commercial environment.

1.6 Aims of This Project

The underlying stochastic nature of the requirements for the Solvency II regulations has introduced significant challenges if the required calculations are to be performed

correctly, without resorting to excessive approximations, within practical timescales.

The inability to fulfil the incoming Solvency II regulations means that it is necessary to seek efficient algorithms and implement those algorithms on hardware capable of highly parallel threading: that is the focus of this project. The use of 1000 Monte Carlo scenarios for each future time step has become colloquially known in the UK as the "brute force" approach. This project aims to show that it is possible to perform the full brute force calculation (with a full valuation for each policy, for each scenario, for each step) within practical timescales using massively parallel architectures.

A significant amount of work has been done by others on changing the way in which the stochastic processes underlying the assets or liabilities are modelled, with those changed processes being implemented on small scale parallel computers. That work generally focuses on the development of parallel algorithms which are then implemented on multi-core CPUs. Although that work improves the way in which the underlying parameters are modelled, it does nothing to aid the way in which the liabilities are calculated from those parameters.

This project is based on the philosophy that any of the range of beneficial techniques may be used. Ordering them by increasing difficulty of implementation, the techniques under consideration in this project are i) compiler optimisations, ii) manual optimisations, iii) implementation on different hardware, e.g. GPUs, and iv) implementation of different algorithms. However, this is not necessarily the order in which the techniques have been applied in this project; the approach here has been to utilise techniques which lead to the greatest improvement in performance.

Although the ability to comply with the regulations is the *prima facie* driver for this project, the underlying benefit of this work comes from the fact that the demonstrations of solvency and future profitability are the foundations of the ability of the assurance company to continue in business, and this is clearly essential to people who have their pensions provided by such companies. As the current trend to move away from 'final salary' (or defined benefit) pensions provided by employers continues, and more people are moved to 'money purchase' (or defined contribution) schemes, the ability of pension providers to continue in business becomes paramount to the financial well-being of an increasing portion of the population.

1.7 Thesis Layout

Chapter 2 contains a review of annuities, including a derivation of the formula required to estimate the value of the liabilities relating to an annuity policy. It also discusses current hardware, used in High Performance Computing, which is relevant to this project, and contains a discussion of the performance of that hardware. Finally, it introduces the hardware used in this project.

Chapter 3 presents the derivation of a novel algorithm which may be used to estimate the liabilities within many types of non unit-linked life assurances, pensions, and other policies. This algorithm brings the calculations required for Solvency II within the realms of practicality for most life offices.

Chapter 4 shows how the algorithm developed in Chapter 3 is applied to several specific types of policies, and introduces a means of extending the use of the algorithm to several other types of policies. This work, and that in Chapter 3, has been published in [88].

Chapter 5 presents optimisations made to code used in situations where only one scenario is required. Such calculations include: the current method of estimating liabilities for demonstrating solvency; estimating profitability arising from an existing book of business; and estimating the cash amount to be dis-invested for a cohort of annuities. This work has been published in [87].

Chapter 6 discusses a simplification to the method used for interpolation in a mortality table. Because interpolation is an extremely heavily used operation, this simplification leads to a significant reduction in calculation time, but only a small change in financial results.

Chapter 7 considers the application of Monte Carlo simulations to situations where parameters are drawn from a time series, as required by Solvency II: the parameters drawn in any particular time step should be valid for use in each future time step and so re-sampling may not be necessary. Whilst re-sampling in each step reduces the variance of the result, it may be possible to achieve sufficient accuracy by not re-sampling the parameters and increasing the number of scenarios instead.

Chapter 8 presents *ab-initio* code developed to implement the brute force approach to demonstrating solvency under the Solvency II regulations. Although the shortest run times were obtained using a Cray supercomputer, timings for an implementation on a modest Unix cluster are also presented: these show that life offices with modest amounts of hardware are able to perform the calculations within practical timescales.

Some of this work has been published in [89].

Chapter 9 presents some ideas for future work which could be based on this project. These ideas are split into the two fields covered by this investigation: some ideas concern different hardware, and further software implementations, whilst other ideas concern further actuarial uses for high performance codes.

Chapter 10 contains a summary and closing remarks.

Chapter 2

Background

This chapter puts this project in context: it presents the background behind the problem, together with an overview of the actuarial theory underlying the problem and the computing concepts used in the solution of the problem.

Section 2.1 presents the actuarial concepts which underpin this project, starting with some actuarial notation. Then annuities are described, and mortality is discussed, before these concepts are combined to introduce life annuities. Section 2.2.1 considers actuarial valuation systems, which are the standard tool within life offices at the current time. The general concepts of these systems are presented, and an overview of their benefits and problems is considered. Section 2.3 presents some of the uses for which programs produced by valuations systems are required. These uses may stem from either regulatory requirements, or management actions. Section 2.4 gives an overview of computer processing hardware: CPUs and GPUs are both introduced. Section 2.5 presents an introduction to high performance computing. Initially the different types of current HPC machine are introduced, and then an overview of programming techniques used on these machines is discussed. Section 2.6 discusses work related to this project: there is a distinct shortage of directly relevant work, and so literature from other areas is discussed. Finally, Section 2.7 presents the platforms used in this project.

2.1 Actuarial Concepts

In order to estimate the liabilities relating to, and profit emerging from, a block of life annuity policies it is necessary to explain the formulae used. This section covers the notation used, and the basic concepts of the theory of annuities and mortality, before

combining them to produce the summation formulae which are used to estimate the relevant liabilities.

It is common in actuarial work for the unit of time to be 1 year, but this is not a requirement. Therefore, unless stated otherwise, time units are assumed to be years.

2.1.1 Actuarial Notation

As far as reasonably practical, standard International Actuarial Notation [39] is used throughout this thesis. Therefore,

$$ {}_tp_x = \Pr[\text{life currently aged precisely } x \text{ survives until age } x+t] $$

and

$$ \begin{aligned} {}_tq_x &= 1 - {}_tp_x \\ &= \Pr[\text{life currently aged precisely } x \text{ dies before reaching age } x+t] \end{aligned} $$

and if $t = 1$ then the prefix is dropped so that $p_x = {}_1p_x$ and $q_x = {}_1q_x$. Also following standard notation, for $t \geq 0$, l_{x+t} is the number of lives expected to be alive at age $x + t$, given that there are l_x lives at age x: see Section 2.1.4 for further details.

In theoretical work, (x) is standard notation for "a life currently aged precisely x". However, it is generally accepted that, when there is no possibility of ambiguity, the parentheses may be omitted so that x denotes 'a life currently aged x'.

Many parts of this thesis consider annuities: standard notation uses

- a_x to represent the expected present value of an annuity where payments of amount 1 are made at the *end* of a year to a life aged x at the time of valuation, so long as the life is alive at the time of payment, and
- $\ddot{a}_x$ to represent the expected present value of an annuity where payments of amount 1 are made at the *start* of a year to a life aged x at the time of valuation, so long as the life is alive at the time of payment.

In this project, a'_x is used to highlight that a general payment stream is assumed, i.e. the payments of amount 1 are made at some fraction $f \in [0, 1]$ through the year, and hence a_x and $\ddot{a}_x$ are simply special cases of a'_x with $f = 1$ and $f = 0$ respectively.

In a similar manner, standard notation uses $a_{x|y}$ and $\ddot{a}_{x|y}$ to represent two-life reversion-

ary annuities where the payments are made at the end or start of a year, respectively, to (y) after the death of (x). Here, $a'_{x|y}$ is used to indicate that payments are made some fraction $f \in [0, 1]$ through the year so that $a_{x|y}$ and $\ddot{a}_{x|y}$ are just special cases of $a'_{x|y}$.

2.1.2 Interest and Discounting

Interest is usually considered to be 'the reward to the lender for making a loan'. When bank loans and mortgages are considered, the role of the lender and borrower are familiar: this illustrates the requirement that the lender will require some recompense for making the loan, and the inherent risk that the loan will not be repaid. In general, the greater the risk undertaken in making the loan, the greater the rate of interest charged.

Savings accounts attract interest: in such accounts the customer has, effectively, lent the bank the money. It is straightforward to calculate how much will be in the account after a certain period of time if interest is earnt at a known fixed rate. The converse is also true: it is possible to calculate how much needs to be invested for a certain period at a known interest rate to be able to withdraw a required amount at the end of that period of time. The value invested under such conditions is known as the *discounted present value*, or simply the *present value* (PV), of the amount required at the later date [59, Section 2.5], and the process of obtaining the present value is known as *discounting*.

Suppose the interest rate is fixed at rate i per period. Then the *discount factor* which applies over the period of unit length is

$$v = \frac{1}{1+i}$$

and v^{t+f} is the discount factor which applies from time 0 to a cash flow some fraction $f \in [0, 1]$ through the period from t to $t+1$ for $t \in \mathbb{Z}^+$. When the rate of interest varies as a function of time, standard practice [59, Section 2.4] is to consider the discount factor from 0 to t as

$$v(t) = \exp\left(-\int_0^t \delta(r)\, dr\right)$$

where $\delta(r)$ is the *force of interest* at time r.

To allow generality, removing all assumptions about variability of interest rates, it is possible to denote the discount factor as v_t which allows for either a fixed rate of interest, or a variable rate of interest, over a time interval of length t: this is the approach taken

in Section 2.3.3.1.

2.1.3 Annuities

At its most basic, an annuity is just "a stream of payments". Very common examples in everyday use are bank loans and mortgages, where the bank (or other lender) 'gives' the customer some money now in return for a series of future repayments; effectively the loan is the purchase of an annuity (from the customer) by the lender, and the repayments form the annuity. Most bank loans have fixed interest rates, and are for fixed periods of time, whereas mortgages often have a variable rate of interest.

In general, the payments are made at fixed regular intervals, although this is not a necessity. Many of the most common forms of annuity are payable monthly or yearly, although in some rare situations, payments may be made weekly, but this is becoming extremely uncommon.

By considering an annuity as a stream of payments, the PV of the entire annuity is the sum of the PV's of the individual payments. Consider, for simplicity, an annuity where n repayments, each of amount 1, are to be made annually, the first being one year from outset, so that the last payment is due at time n. Suppose that the interest rate is fixed at i *per annum*, and let the present value of the annuity be a. Then, following the derivation in [59, Section 3.3],

$$a = \frac{1 - v^n}{i} \tag{2.1.1}$$

where v is the discount factor corresponding to the annual interest rate i.

For the case where payments are not made annually, but the interest is quoted as an annual rate, an adjustment is required in order to obtain the *effective interest rate* over the period between payments. Consider the common case where payments are made monthly: suppose the annual rate of i is applied to monthly payments, then the equivalent monthly rate j is such that $(1 + j)^{12} = (1 + i)$ and, to maintain consistency, rate j needs to be applied over the number of months for which the annuity is payable, not the number of years.

So far, the annuities discussed have been payable at the end of the period. However, life annuities (introduced in Section 2.1.5) are a particular class of annuity where it is common for the payment to be made part-way through each month or year. The same basic principles apply to these annuities, but it is necessary to make an adjustment for

the timing of the payments. Consider an annuity where n payments, each of amount 1, are to be made annually, the first being some fraction $f \in [0, 1)$ through the first year. The PV of this annuity is

$$\begin{aligned} a' &= \frac{1 - v^n}{i} v^{f-1} \\ &= a\,(1 + i)^{(1-f)} \end{aligned}$$

where a is as defined in Equation (2.1.1). Note that it does not matter which end of the interval is considered to be closed: if the interval is $f \in (0, 1]$ and the payment is actually made at $f = 1$ then the payment stream is *an annuity payable in arrear*, whereas if the interval is $f \in [0, 1)$ and the payment is actually made at $f = 0$ then the payment stream is *an annuity payable in advance* [59, Page 45].

Therefore, the only difference between the PV of this annuity and the PV of the initial simple annuity is the factor $(1 + i)^{(1-f)}$, which adjusts for the 'addition of interest' to the original annuity value; this is equivalent to allowing for the loss of interest between time f, when the payment is made, and the end of the step, where the payment was made in the introductory case.

Since multiplication is linear, standard actuarial practice recognises that a useful monetary value to use in the derivations is 1. This allows *annuity factors*, tabulated by interest rate and term, to be published. With the advent of computers, these factors are becoming less useful in numerical calculations, but they are invaluable in theoretical work.

2.1.4 Mortality

The rate of mortality is a concept which is of fundamental concern to actuaries in life assurance and pensions businesses. At its simplest, "the rate of mortality at age x" is a measure of the proportion of people currently aged x who are expected to die before reaching age $x + 1$. The actual mechanics of estimating mortality rates are outside the scope of this project, but are well explained in [6, Chapter 2].

When estimating mortality rates, actuaries within life offices usually assume that lives are independent. This assumption has several advantages, primarily by simplifying the derivation of the mortality rates (by not needing covariances between the variables which represent the survival of the lives) and by simplifying calculations using the resulting, estimated, mortality rates. The assumption is not really realistic in cases of

spouses or business partners because there is a possibility of death in same accident. However, in populations which are observed for the mortality rate to be estimated, the assumption is reasonable.

The mortality estimates obtained are $\{q_x\}_{x=0}^{\infty}$ where q_x is the probability that a person currently aged precisely x will die before reaching age $x+1$. In general, the x's are tabulated at integral values since that has the simple interpretation of being the xth birthdays.

These probabilities are used to form a 'life table', which shows the number of lives expected to be alive at a particular age, given a particular number of new-born infants. The table is built in an intuitive manner: let l_x be the number of lives aged exactly x; then l_{x+1} is the number of lives who were aged exactly x and did not die within the year to age $x+1$, and the expected number of lives reaching age $x+1$, given that they were alive at age x, is $l_{x+1} = l_x\,(1-q_x)$.

A natural result of this form of producing the table is that the x's are integers, and this is often regarded as beneficial since people have a tendency to arrange major life events on, or particularly close to, significant birthdays, e.g. they retire at age 65.

A life table is used to calculate the probabilities of survival: the probability of surviving from age x to age y is simply

$$Pr[\text{Survive from age } x \text{ to age } y] = \frac{l_y}{l_x}$$

Using standard actuarial notation, let ${}_tp_x$ be the probability that a life currently aged x survives for t years. Then the standard result for survival probability can be stated as

$${}_tp_x = \frac{l_{x+t}}{l_x} \tag{2.1.2}$$

The use of this relationship is not restricted to integers. Since the relationship expresses the fact that 'the probability of surviving from one age to another is the same as the ratio of the expected number of lives at those ages', it works equivalently well for any ages: the only increase in complexity comes from the fact that l_x at fractional ages needs to obtained from the life table by interpolation.

2.1.5 Life Annuities

Although the regulations which are being introduced apply to all types of policy, this project focuses on life annuities, partly because they are conceptually simple, but mainly because the ability for an annuity provider to demonstrate that they are solvent is of prime importance to a sizeable portion of the population who rely on these annuity providers. Also, for many assurance companies, life annuities form the largest part of their liabilities: by providing a means for dealing with annuities, this project goes a long way towards demonstrating solvency for the entire book of business. Despite the focus on life annuities, the major advances this project has made can be applied to a wider variety of assurance contracts, as will be shown in Chapter 4.

A 'life annuity' is an annuity where the payments depend on the survival, or otherwise, of a pre-specified life, or collection of lives. Section 2.1.5.1 considers single-life annuities certain; they depend on the survival, or death, of only one life, and it is certain that they *will* start to be paid. Other types of life annuity could depend on two or more lives, and yet others may not even start to be paid. The most common example of a single life annuity is the payment of a pension to a pensioner; the pension stops being paid when the pensioner dies. A less common example of a single life annuity is a child's annuity: this is a rider benefit to a temporary assurance of the parent and will only become payable if that parent dies within the period specified in the contract and the annuity payments are made to a dependant child so long as that child is alive at the time of payment, with the payments terminating at a specified age (often 18 or 21).

Although a pension may also have a spouse's pension, the payment of that spouse's pension depends on the survival of the spouse, rather than the pensioner. It is therefore a different type of annuity: it is a two-life (reversionary) annuity, as covered in Section 2.1.5.2. Spouse's pensions are the contracts which provided the motivation for developing a vector form of a recurrence relation for calculating reserves: this is introduced in Section 3.1.3.

It is possible to estimate the present value of life annuities by adding survival probabilities to the present value calculations used for non-life annuities. Including the survival probabilities of the form of Equation (2.1.2) in Equation (2.1.1) allows for both the interest earnt up to the point of payment, and for the possibility of not making the payment.

In principle, the possibility that the recipient could live forever needs to be acknowledged. However, the fact that immortality is unlikely is usually allowed for by setting

${}_tp_x$, or equivalently l_x, to zero for older ages, e.g. $l_x = 0$ for $x > 120$. Hence, the usual situation is that in theoretical work all lives have the possibility of infinite survival, while in numerical work a *limiting age* is set, and all lives are assumed to suffer a terminal event at that age.

2.1.5.1 Single Life Annuities

A payment at a particular future date will only be made if the life is alive at that date. Therefore, for a life aged x at the date of valuation, a payment due t years from the valuation date will need to allow for interest at rate i p.a., using a factor of $v^t = (1+i)^{-t}$, and for the probability of survival, using a factor of ${}_tp_x$. Using standard actuarial notation, let a_x be the expected present value of the annuity, with an annual payment of 1, payable at the end of each year, to a life aged x at the time of valuation, so long as the life is alive at the time the payment is due. Then a_x can be considered as the sum of the present values of the individual payments, and so, following [65, Section 2.2], the total present value can be shown to be

$$a_x = \sum_{t=1}^{\infty} {}_tp_x \, v^t \tag{2.1.3}$$

Life annuities, like all other annuities, can be paid other than yearly, and at fractions of the way through the yearly or monthly period. It is straightforward to derive a formula for the present value of a life annuity which is payable some fraction of the way through a projection step. For a life annuity where payments are to be made annually to a life aged x at the time of valuation, the first payment being some fraction $f \in [0, 1)$ through the first year, the relevant adjustment is to use $(t + f)$, rather than t, so that the summation becomes

$$a'_x = \sum_{t=0}^{\infty} {}_{(t+f)}p_x \cdot v^{(t+f)}$$

A further adjustment can be made to allow for the payments being made some fraction of the way through a month, rather than a year: it is only necessary to adjust the index variable to allow for the fact that there will be twelve times as many steps between tabulated values in the life table. Incorporating the adjustment to recognise that the index variable k is now in months, and $f \in [0, 1)$ is a fraction of a month, requires the use of $\left(\frac{k+f}{12}\right)$; i.e. when k is a non-negative integer representing the number of months into the projection, $\left(\frac{k+f}{12}\right)$ is the number of years from the start of the projection to the payment of the benefit.

It is not uncommon for annuitants (recipients of life annuities) to take their policies out in such a way that the amount payable increases by a fixed escalation rate e on each policy anniversary and it is therefore necessary to include a further adjustment to the valuation formula to allow for escalation. Using standard notation, let $\lfloor m \rfloor$ be the integer part of m. Allowing for the fact that, in general, the valuation of the life annuity will happen some fraction g through the policy year, so that the next payment is at time $g + \frac{f}{12}$ since the most recent policy anniversary, the adjustment is made by incorporating a factor of $(1+e)^{\lfloor g + (\frac{k+f}{12}) \rfloor}$.

a_x is standard notation for a level annuity payable in arrear: since the derivation in this section is for an annuity which is not necessarily level, nor in arrear, different notation is needed, and so a'_x is used. Therefore, the annuity factor for a general single life annuity is

$$a'_x = \sum_{k=0}^{\infty} {}_{(\frac{k+f}{12})}p_x \times v^{(\frac{k+f}{12})} \times (1+e)^{\lfloor g + \frac{k+f}{12} \rfloor} \tag{2.1.4}$$

The PV obtained from this equation is the *expected present value* of payments, of amount 1 at outset, to be made to the policyholder. Assuming that these life annuities are pensions, the pension provider will need to hold this amount in some form (often Treasury bonds) in order to meet the payments in future. Effectively, the amount has been reserved to make the payments to this policy, and hence a'_x is referred to as a *reserve factor*, and the monetary amount which is the product of the reserve factor and the current payment amount is the *reserve*. Finally, supposing that, for the jth policy, ϕ_j is the amount payable at the first payment following the valuation date, the payments are made at a fraction f_j through the month, the policy holder is aged x_j at the time of valuation, the time of valuation is a fraction g_j since the last policy anniversary, and the escalation rate is e_j, the total reserve for all policies is the sum of the individual reserves across relevant policies, i.e.

$$\begin{aligned} \begin{matrix}\text{Total}\\ \text{Reserve}\end{matrix} &= \sum_{j \in \text{ policies}} \phi_j \, a'_{x_j} \\ &= \sum_j \left(\phi_j \sum_{k=0}^{\infty} {}_{\left(\frac{k+f_j}{12}\right)}p_{x_j} \times v^{\left(\frac{k+f_j}{12}\right)} \times (1+e_j)^{\lfloor g_j + \frac{k+f_j}{12} \rfloor} \right) \end{aligned} \tag{2.1.5}$$

2.1.5.2 Two Life Annuities

Life annuities are not restricted to the survival or otherwise of a single life: several forms of multiple life annuities exist within UK life assurance offices, but this project considers

some of the more common examples. For each of these, the summation formulae only differ from Equation 2.1.4 in their probability of payment. For clarity of notation, the lives are denoted x and y, rather than x_1 and x_2.

A *reversionary annuity* becomes payable to a second life on the death of the first life: a common example is a spouse's pension, which becomes payable when the member of the pension scheme dies. The payment in the kth step is made so long as the first life has died and the second life is alive: allowing for escalation and the possibility of payment a fraction f through a step, the summation for the reserve factor for a single policy is

$$a'_{x|y} = \sum_{k=0}^{\infty} \left({}_{\left(\frac{k+f}{12}\right)}q_x \; {}_{\left(\frac{k+f}{12}\right)}p_y \right) \times \; v^{\left(\frac{k+f}{12}\right)} \; \times \; (1+e)^{\lfloor g+\frac{k+f}{12} \rfloor} \tag{2.1.6}$$

A *joint life annuity* is payable so long as both lives remain alive: the corresponding reserve factor is therefore

$$a'^{\,JL}_{xy} = \sum_{k=0}^{\infty} \left({}_{\left(\frac{k+f}{12}\right)}p_x \; {}_{\left(\frac{k+f}{12}\right)}p_y \right) \times \; v^{\left(\frac{k+f}{12}\right)} \; \times \; (1+e)^{\lfloor g+\frac{k+f}{12} \rfloor} \tag{2.1.7}$$

A *last survivor annuity* remains payable so long as at least one of the two lives remain alive: an alternative view of this is that payments are made so long as 'not both lives are dead' and the corresponding reserve factor is

$$a'^{\,LS}_{xy} = \sum_{k=0}^{\infty} \left(1 - {}_{\left(\frac{k+f}{12}\right)}q_x \; {}_{\left(\frac{k+f}{12}\right)}q_y \right) \times \; v^{\left(\frac{k+f}{12}\right)} \; \times \; (1+e)^{\lfloor g+\frac{k+f}{12} \rfloor} \tag{2.1.8}$$

2.1.5.3 Reserves

Calculations involving reserves need to distinguish between in-force reserves and per-policy reserves [22, Section 12.3]. The factor a'_{x+t} discussed in Section 2.1.5.1 is the factor for the *in-force reserve*: it is the reserve required at time t, per £1 of benefit, for a life aged x at $t = 0$, *given that the policy is in force at time* $t > 0$. By multiplying the in-force reserve by the probability of survival to that time, using a realistic mortality basis, rather than the reserving mortality basis (i.e. ${}_tp_x$ is calculated using, possibly, a different mortality table), the reserve obtained is ${}_tV_x$, which is the *per-policy reserve*: it is the reserve required for a policy in the data set at the valuation date, $t = 0$ (i.e. not necessarily still in force at time $t > 0$).

Similarly, the factors $a'_{x|y}$, $a'^{\,JL}_{xy}$, and $a'^{\,LS}_{xy}$ discussed in Section 2.1.5.2 are factors for

the in-force reserve for the relevant policy type: they are the reserve, per £1 of benefit, for lives aged x and y at $t = 0$, required at time t *given that the policy is in force at that time.* To obtain the per-policy reserve factors for two life annuities, it is necessary to allow for the probability of the survival of the lives to time t, and hence the fact that the policy could have migrated to a different type depending on which of the lives survive: i.e.

annuity type	per-policy reserve factor
Reversionary	${}_tV'_{x\|y} = {}_tp_x\ {}_tp_y \cdot a'_{x+t\|y+t} + {}_tq_x\ {}_tp_y \cdot a'_{y+t}$
Joint Life	${}_tV'^{JL}_{x,y} = {}_tp_x\ {}_tp_y \cdot a'^{JL}_{x+t,y+t}$
Last Survivor	${}_tV'^{LS}_{x,y} = {}_tp_x\ {}_tp_y \cdot a'^{LS}_{x+t,y+t} + {}_tp_x\ {}_tq_y \cdot a'_{x+t} + {}_tq_x\ {}_tp_y \cdot a'_{y+t}$

The distinction between in-force reserves and per-policy reserves is important because a life office only needs to hold the per-policy reserve which, due to the allowance for the probability of still requiring to hold the reserve, will be less than the in-force reserve.

2.2 Domain Specific Languages

Over the last couple of decades, software packages which reduce the load on the programmer have appeared: this section considers such 'domain specific languages', or DLSs.

2.2.1 Actuarial Valuation Packages

Within the life assurance industry, DLSs are known as 'valuation packages', and several are available, e.g. Prophet [31], Algo Financial Modeler[1] [38] and [77], Mo.net [75], and MoSes [94]. The idea behind these valuation packages is analogous to the spreadsheet and, as such, these valuation packages are, in concept, very good. These packages are usually crafted for flexibility and ease of use, rather than performance of the programs they produce.

At their most basic level these packages require the relationships between variables to be entered in a high level pseudo code: see Figure 2.1 for an example of a form used to input these relationships. In the example shown, PVRE is the *present value of retained earnings* – a measure of how much profit is expected to emerge from the policy in a particular step. Hence, the formula for 'CumulativePVRE' is simply either the PVRE

[1] This changed names: VIP → VIP*itech* → Algo Financial Modeler → RNA 'R^{3}S'

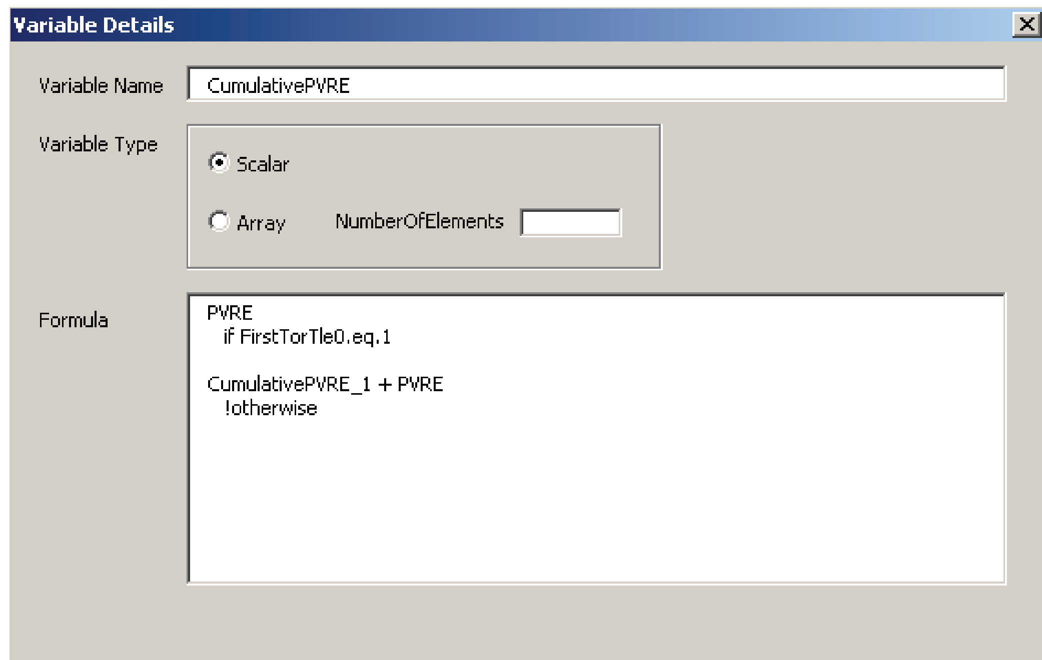


Figure 2.1 Representation of a typical *Variable Input Screen*, as used in a standard actuarial valuation software package.

in the first step, or the existing CumulativePVRE with the PVRE in the current step added.

When the required relationships have all been entered, the valuation software performs the relevant dependency analysis and produces source code in a 'normal' programming language, which is often C, C++ or Fortran, although Visual Basic has now also appeared as the language used by one valuation package. The valuation package which was used for the initial timings for this project produces Fortran code: the source created for the relationship in the example in Figure 2.1 is

```
if (FirstTorTle0 .eq. 1) then
   CumulativePVRE = PVRE
else
   CumulativePVRE = CumulativePVRE + PVRE
endif
```

This code is then compiled and linked into an executable using standard compilers and linkers which are controlled by the valuation package issuing command-line instructions internally. The executable produced by this method is then launched by the valuation package, and when execution is complete the valuation software ensures that any memory used has been correctly freed.

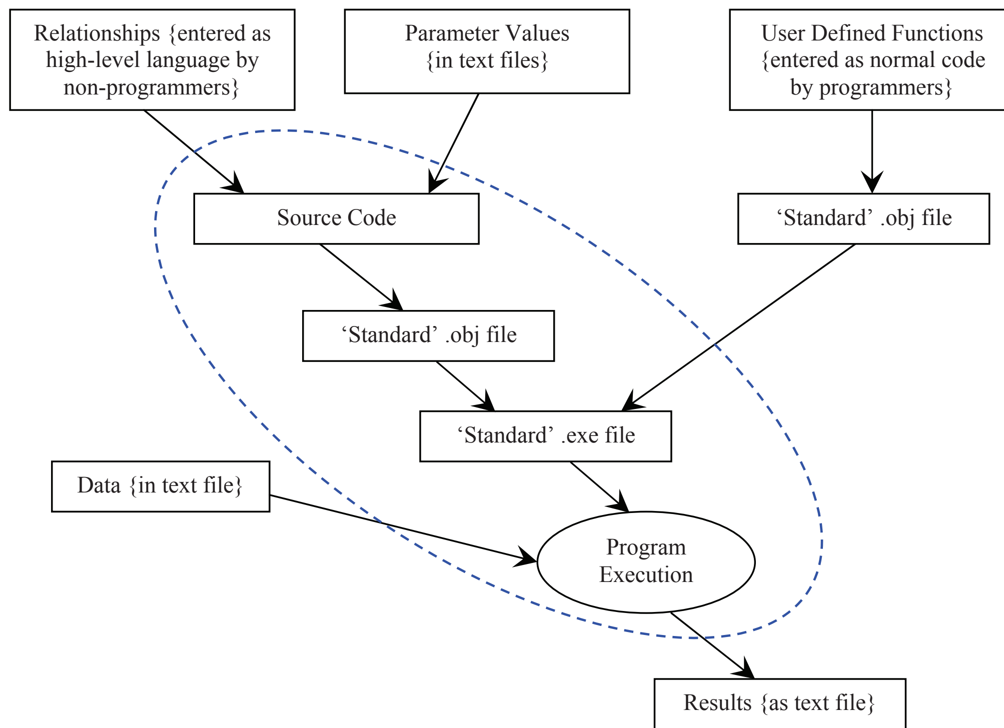


Figure 2.2 Schematic overview of a standard actuarial valuation software package.

In general, the functionality of a spreadsheet can be enhanced using add-ins, macros and user-functions. Analogously, it is generally possible to create user-defined functions for these financial valuation packages. While these functions can be built to perform calculations of arbitrary complexity, they have a minor drawback in that they need to be written in one of the standard languages mentioned above. Since these functions lie outside the normal functionality of the package, they need to be written and compiled by a competent programmer before they can be used by the valuation package. They are only integrated into the system at the point at which the package performs its linking stage. For clarification, Figure 2.2 represents the workings of these systems, with the dashed oval representing the boundary of the valuation package.

Given their mode of operation, the advantages to these systems are clear. For normal operation, they allow non-programmers to produce some really complex programs which can be used to estimate the current value of, or profitability within, some quite complicated assurance contracts: the user simply needs to input the correct relationships and the valuation package does the rest. Software developers only need to get involved when relationships between variables become too complex to enter into the system, and user-defined functions need to be built.

An added advantage to storing the relationships between variables, rather than actual

source code, is that it becomes reasonably straightforward for non-programmers to change the way policies are processed: the user simply needs to change the relevant relationships between the variables and, assuming that no errors were introduced, the valuation package will produce a new executable. The advantage here is that, for this mode of use, users do not need to become involved in the details of variable types, or the shapes of arrays, etc.

However, while the disadvantages of these systems are not quite so obvious, they are as equally fundamental as the advantages. A slight issue arises from the fact that even if the value of only one parameter is changed in only one place, then the system will generate totally new source code for all the relationships entered by the non-programmers (i.e. not the user-defined functions) re-compile all of that source, and link the resultant object file. This is clearly far less efficient than being able to simply re-compile only those parts of the source code which have changed, especially when only a numeric value in a parameter file has altered so that there are actually no changes to the code.

Another disadvantage is that, for the relationships entered by the non-programmers, there is no possibility of being able to tune compiler optimisations. Neither can the linker options be tuned, even when user-defined functions have been created, and then compiled with a high level of optimisation. As a consequence of this, all programs depend on compiler and linker settings which were ordained by the writers of the valuation package at the time it was developed. While these standard options may be beneficial in some situations, they are certainly not optimal for all codes.

A far more fundamental problem is that the user is constrained by pre-configured parameters which were included in the package at the time it was built, e.g. some of these valuation packages only allow a forward progression through time, while others consider all numeric variables to be double precision even when integers would suffice. The only way to work around such constraints and deficiencies is to build relationships which are at a far more fundamental level, i.e. move away from the text-based relationships and utilise the skill of the programmers. It is this problem which is the predominant motivation for the investigation in this report.

2.2.2 Other Areas of Finance

In addition to the actuarial valuation systems just mentioned, similar DSLs exist for other fields. For example, SciFinance [80] is available for derivatives pricing: one case

study [81], which appears on the SciFinance website, states that SciFinance "eliminates programming by automatically translating model specifications for any financial derivative that can be priced using any series of partial differential equations (PDEs) or stochastic differential equations (SDEs) into fully documented C-family/CUDA source code ...". This *modus operandi* is similar to that of Algo Financial Modeler and MoSes, etc. (mentioned in Section 2.2.1), which translate relationships between variables into code – albeit for CPUs rather than GPUs. However, SciFinance is not used in actuarial valuations.

2.3 Use Cases

Having covered the derivation of the formulae in Section 2.1, and given an overview of actuarial valuation packages in Section 2.2.1, this section covers some of the situations in which these calculations are used.

2.3.1 Parameters

Most actuarial calculations are performed to estimate the current value of something which relies on future events and so, in order for this to be possible, assumptions need to be made about what could happen in the future. These assumptions take the form of parameters to the calculation: as with other fields of computing, these parameters may be varied according to their use. Parameters typically fall into two categories, economic and demographic.

Examples of economic parameters are interest rates, inflation rates, levels of expenses, or monetary exchange rates. The majority of these have intuitive explanations, the possible exception being the expenses: these are the proportion of the company's running costs which are attributable to each policy, and recovered from the policyholder, either as an addition to the premium or a reduction in benefit payable. Typical values for interest rates, inflation rates, and monetary exchange rates may be found in newspapers and financial publications, e.g. interest rates are currently around 0.5% with some addition for risk, so that a bank loan attracts a rate of around 5%. Typical values for expenses attributable to policies in a moderately sized life office would currently be around £30 per policy per year, although this will fluctuate yearly, as the operating expenses change.

Examples of demographic parameters would be mortality rates (i.e. deaths), morbidity

rates (i.e. sickness) and birth rates. Mortality rates vary depending on what is being measured, and therefore which table is being used. Consider 65 year old males: according to ELT17 [70] for an entire population, it could be expected that 12.40‰ of the those reaching age 65 would die before their 66th birthday; according to PNML00 [42] for pensioners (who are not impaired lives), it could be expected that 12.85‰ of the those reaching age 65 would die before their 66th birthday; according to ANS00 [41] for assured lives who smoke, it could be expected that 15.42‰ of the those reaching age 65 would die before their 66th birthday.

Within a particular set of parameters, the value of each item may vary with time: for example, in order to allow for lightening of mortality (i.e. the fact that people are living to older ages), any realistic simulation would allow for fewer people dying at a particular age as time progresses.

2.3.2 Single Scenario Calculations

This section introduces uses where only one set of parameters are required because the calculations are only performed once.

2.3.2.1 Calculation of Reserves

As mentioned in Section 1.1, the calculation of reserves is performed as part of the regulatory requirement to assess of solvency. Historically, the reserves were calculated once, using a set of assumptions based on actuaries' views of future economic and demographic trends. For annuities, the calculation is a straightforward implementation of the summation formulae derived in Section 2.1.5.

The assumptions used necessarily introduce subjectivity because different actuaries will have different views. However, the basis used is often quite prudent, e.g. if the expected future interest rate is 4% then the valuation of the liabilities will often use 3.95% so that the resulting estimate of the liabilities is higher than necessary. This produces the benefit of including an implicit solvency margin in addition to the explicit margin that the regulators require.

However, as interest rates have fallen over the last decade, the rates used in calculations are closer to their expected values so that implicit margins are reducing (in an attempt to demonstrate that the company remains solvent in the harsher climate). This reduction in implicit margin, and the resulting reduction in reserves, also leads to a reduction

in the explicit margin (which, under the outgoing regulations, is a fixed percentage of reserves).

2.3.2.2 Estimation of Profitability

These calculations are used to estimate the future profit which will emerge from a block of policies. The company's managers will use these estimates to inform decisions as to whether policies should continue to be sold (because they are profitable), whether a policy type should be withdrawn (because it appears to make a loss), or whether a policy type should be reformulated so that the level of profit may be increased, e.g. by adding an explicit management charge.

Because these calculations are for internal purposes only, the bases used do not necessarily have to closely match market conditions: the bases could mirror management's expectations. There is generally one 'best estimate' basis for calculating reserves and one 'realistic' basis for projecting profitability calculations. This has the advantage of producing future reserves which may be checked against the calculation of statutory reserves (as discussed in Section 2.3.2.1) to ensure that the bases on which the management are basing their decisions are not dramatically different from the bases which are used to create reserves which appear in statutory reports.

Profitability calculations are based on the reserve calculations together with the change in reserves and interest on reserves, against which payments and expenses are offset. Under current practice, reserves need to be re-calculated at each future time step and the change in reserve over each step, and the interest earnt on the reserve during each step, are obtained in a straightforward manner.

The entire process is conceptually straightforward. However, implementations using commercial valuation packages are less than ideal as a means of estimating such profitability: as mentioned in Section 1.4, the time to process the largest data set within a single cohort of single life annuities is roughly 35 CPU core hours for about 129,000 policies: this is a processing rate of approximately one policy per second. Using the same valuation system, which produces single threaded programs, it took about 22.5 CPU core hours to process about $35,000$ reversionary annuity policies. This use of a commercial valuation package was the original motivation for the investigation in this project; the conceptual simplicity of the contracts is not reflected by run time.

2.3.2.3 Disinvestments

In order to run a business which administers annuities, it is necessary to estimate how much cash is paid out to policyholders, and hence how many of the assets backing the portfolio of annuities need to be liquidated, in each future month. These calculations are for internal, management purposes only and so are not required to satisfy any regulatory requirements.

The estimation of cash required is often achieved by performing a profitability estimate and ignoring the change in reserves, interest on reserves and expense-related items: the remaining items are the cash flows which approximate the value of the assets which need to be disinvested at each time step.

2.3.3 Multiple Scenario Calculations

This section gives an overview of the use which motivated this project: the values of the parameters used in the estimation of liabilities must be repeatedly drawn from some underlying distribution, with each set of parameters being used only once.

2.3.3.1 Demonstrating Solvency

Historically, solvency regulations required that a *solvency margin* be held in excess of the reserves; for many types of policy this solvency margin was simply 4% of the reserve. Therefore, demonstrating solvency was simply showing that available assets exceeded 104% of estimated liabilities, where those liabilities were calculated on a best estimate basis.

The new solvency regulations require that the additional capital required to ensure solvency is calculated in a more complex, and hence realistic, manner. For every step, the assurer must calculate the additional capital at that time as the difference between the capital required using the office's 'best estimate' and the capital required to meet a '1-in-200' event, assuming that the latter exceeds the former: these differences are then discounted to obtain their PV, and that PV is the *additional capital requirement* at the valuation date.

2.3.3.1.1 The Process

In order to ensure that the calculations are resilient, and correctly allow for only 0.5%

chance of not being able to meet future liabilities, there is no deduction from the capital required in the unlikely event that the $\left(\frac{1}{200}\right)^{\text{th}}$ worst scenario leads to a smaller reserve than the best estimate. In this way, the capital requirement seeks to ensure that the office holds the optimum amount of capital to cover the liabilities.

It might be possible to use statistical techniques, such as moment generating functions, to derive the distribution of the reserve at each time step: this would require knowledge of the distribution of each of the relevant parameters, the number of policies, the distribution of ages of the policyholders, and the payment characteristics of each policy. However, this is an extremely difficult problem, and it may be that no closed form for the distribution of reserves exists: to date, the derivation of a distribution of reserves has not been achieved. Without knowing the distribution, it is impossible to find an analytical expression for the reserve which infers a '1-in-200' chance of not being able to meet all liabilities and a Monte Carlo approach is therefore required.

In order to estimate a 1-in-200 event, an absolute minimum of 200 simulations are required. However, using only 200 simulations provides very little statistical robustness: several times this number of simulations are required to reduce the error of the estimate. Within the actuarial profession, a generally accepted approach to obtaining the 1-in-200 worst reserve is to obtain 1000 simulations of the reserve, with the parameter values for each simulation drawn from some distribution. However, 999 of those simulations are then discarded; the one kept is that which infers a $\frac{1}{200}$ chance of the required reserve being greater than that obtained in the corresponding scenario. To allow for changes in mortality, and other trends in different parameters, our implementation allows for the parameter values to be drawn from different distributions in each time step. This Monte Carlo approach to assessing solvency under the Solvency II regulations infers a run time which is beyond contemplation using currently available commercial software on hardware which the industry is comfortable with: as stated in Section 1.5, a naive estimation of the run time is around 2800 CPU core years.

The main focus of this project is the consideration of a brute force approach to fulfilling the Solvency II requirements, the ultimate requirement of which is to obtain the *additional capital requirement*, or ACR. This work is based on the interpretation of the ex-modelling actuary from Aegon UK [15] which is as follows:

- let $t \in \{1, 2, \cdots, T\}$ be the index for future months where the maximum, T, typically has a value of 600, 660, 720, or 780, depending on how far into the future the projection is required to run: note that $t = 0$ is not the index of a *future* month and so $t = 0$ may be considered to be the index of the step to which

additional capital requirement relates, i.e. "now";

– let v_t be the discount factor from now (i.e. $t = 0$) to time t, allowing for either a fixed rate of interest or, more realistically, variable rates of interest;
– let B_t be the total (over all policies) of the best estimate per-policy reserves at time t;
– let J be the number of scenarios required (so that, for the situation described in the preceding paragraph, the value of J is 1000);
– let $j \in \{1, 2, \cdots, J\}$ be the scenario index;
– let $V_{t,j}$ be the total (over all policies) of the simulated per-policy reserves at time $t \in \{1, 2, \cdots, T\}$, for scenario $j \in \{1, 2, \cdots, J\}$;
– let V_t' be the 99.5^{th} percentile in the sequence obtained by sorting the set of simulated reserves at time t, $\{V_{t,j}\}_{j=1}^{J}$: note that when $J = 1000$, V_t' is the fifth-highest reserve;
– let C_t be the additional capital required at time t if V_t' exceeds the best estimate at that time, i.e. $C_t = \max(V_t' - B_t, 0)$ for $t \in \{1, 2, \cdots, T\}$;
– let θ be a fixed percentage (whose value is to be mandated by the regulator);

then the whole calculation may be described by Algorithm 2.1.

Obtaining the scalar value obtained in Line 26 of Algorithm 2.1 is the ultimate aim of the entire process: i.e. calculation of the in-force reserves is part of the process of obtaining the ACR – obtaining the reserves is not the final objective. Note that $t = 0$ corresponds to the valuation date: under the construction above, reserves at $t = 0$ do not contribute to the ACR.

In Line 14 of Algorithm 2.1, the $a_{t,j,p}$'s are the annuity factors introduced in Section 2.1.5. These factors are for the in-force reserves[2]: the complexity of the calculation of these factors is discussed in Section 3.4.1 where a new algorithm is compared to the existing approach. The complete calculation of the ACR for Solvency II is considered in Section 7.3 which contains a discussion of the complexity of four possible implementations of the algorithm.

It is not entirely clear which interest rates should be used to obtain the discount factors in Line 21 of Algorithm 2.1: possible candidates are

a) the rates used to obtain the best estimate reserve: this has the advantage that the same rates will be used for a particular time, but the use of fixed rates is not consistent with a Monte Carlo approach;

[2]The per-policy reserves are calculated in Line 16.

Algorithm 2.1 Additional capital requirement for single life annuities using a brute force approach.

```
1: set T = maximum projection step number
2: obtain best estimate economic and demographic parameters for all future steps
3: for t=1 to T do
4:    initialise the overall best estimate per-policy reserve at time t: B_t = 0
5:    for p=1 to number of policies do
6:       calculate the best estimate per-policy reserve at time t for policy p ...
7:       ... and increment B_t by the per-policy reserve for policy p
8:    end for
9:    set J = number of scenarios
10:   for j=1 to J do
11:      obtain economic and demographic parameters by sampling
12:      initialise V_{t,j} = 0
13:      for p=1 to number of policies do
14:         calculate the in-force reserve at time t for policy p using scenario j: a_{t,j,p}
15:         calculate the probability that policy p is in force at time t: _t p*_x
16:         calculate the per-policy reserve: V_{t,j,p} = a_{t,j,p} × _t p*_x
17:         increment V_{t,j} by the individual per-policy reserve V_{t,j,p}
18:      end for
19:   end for
20:   obtain V'_t by sorting {V_{t,j}}_{j=1}^J and taking the 99.5th percentile
21:   obtain the discount factor to time t: v_t
22:   calculate the discounted best estimate reserve at time t: B̂_t = v_t × B_t
23:   calculate the discounted 99.5th percentile reserve at time t: V̂'_t = v_t × V'_t
24:   calculate the discounted additional capital at time t: Ĉ_t = max(V̂'_t − B̂_t, 0)
25: end for
26: calculate the Additional Capital Requirement: ACR = θ Σ_{t>0} Ĉ_t
```

b) the rates used in the scenario which generated the reserve: this is consistent with the Monte Carlo philosophy and removes a potential performance loss caused by the need to broadcast a single value to all processes;

c) rates which are different to anything used elsewhere: this has no clear advantages, but it may become mandated by the regulator at a future date.

2.3.3.1.2 Parameter Usage

Line 11 in Algorithm 2.1 requires that economic and demographic parameters (as exemplified in Section 2.3.1) be obtained. There are two distinct possibilities for the origin of these parameters: either the parameters may be re-drawn from the underlying distribution at each step, or those parameters already drawn might be re-used. If the parameters are re-drawn then $a_{t,r,p}$ in Line 14 of Algorithm 2.1 will need to be re-calculated in every future step using the new parameters: if the parameters are not re-drawn, and $a_{t,r,p}$ is calculated using Algorithm 3.2, then $a_{t,r,p}$ will not need to be

re-calculated in every future step – the values in the sequence obtained through use of the recurrence relation will be precisely those values which are required.

However, it is not clear from the regulations whether obtaining the economic and demographic parameters for time step t would be dependent on the result of the reserve calculation for step $t-1$. If the distributions from which the parameters are drawn do change at each time step, e.g. to allow for management actions, then it is necessary to re-draw the parameters: i.e. those drawn in the previous step cannot be reused, so that the in-force reserves do need to be re-calculated at every time step. Therefore, to allow for a worst-case calculation count, the majority of this project assumes that there is no such dependence. However, to acknowledge the possibility that there could be a dependence, parallelisation over the t step is not exploited: the implementation of this approach is discussed in Chapter 8.

2.3.3.1.3 Alternative Methodology

If it can be guaranteed that the distributions from which the parameters are drawn do not change at each time step, then it may not be considered necessary to re-draw the parameters: i.e. the values for each t, as drawn in the first step, may be considered as valid for the entire projection. Under this approach, it is possible to calculate in-force reserves at $t = 0$ in a manner which calculates the in-force reserves for all future t as a by-product: see Chapter 3 for details. However, by adopting this approach, it is necessary to increase the number of scenarios at each future step in order to reduce the statistical error in the final result: details of the increased number of scenarios, and an implementation of this approach, are discussed in Chapter 7.

2.3.3.1.4 Parameter Generation

Parameters could be produced using standard techniques, such as sampling against the cumulative distribution function and calculating the inverse [78]. In practice, there are two fundamental options for doing this:

a) read the parameters from files on the disk: this allows all companies to use the same values (if such files were supplied by the regulator), and therefore valuations, and estimates of additional capital, would be truly market consistent. However, the obvious detrimental effect is that the time to read parameters could form a significant part of the processing. This project reads parameters from disk to allow the worst case run times to be obtained and evaluated: see Section 8.1.3.2 which discusses the use of 1000 files, each containing parameters for 780 future time steps.
b) generate parameters as part of the processing: this is likely to be faster than reading

from disk, but assumes that there is a valid inverse transformation of the distribution function. However, under Solvency II it is possible for companies to apply to use an 'internal model' (rather than use whichever model is ordained by the regulator): this could lead to companies using different distributions for a particular parameter. If this were to happen, because the distributions used are not necessarily consistent across the market, the whole concept of market-consistent valuations (which was one of the initial drivers for Solvency II) would be invalidated.

2.4 Computer Processors

There are several different categories of computer processors, each having been developed for a different purpose. This section initially discusses CPUs because those are the basis of the investigation in this thesis. It then covers GPUs because they could be used as the basis of a further investigation.

2.4.1 Central Processing Units

CPUs are the general purpose calculation engines within computers: they are designed to perform a multitude of types of task, and are very good at being a jack-of-all-trades. CPUs can provide good performance when working on a single task but their performance starts to deteriorate when multitasking, and the resultant context switching, is involved.

2.4.1.1 Recent History

"Moore's Law" has many interpretations: in the original paper [64], Moore talked about the 'number of components changing over a five year period'. This is often reinterpreted as 'the number of transistors doubling every two years', although this is also sometimes reported as 'the density of transistors on a chip doubling every 18 months'. Other interpretations come in terms of the speed of a chip doubling every two years (or 18 months).

Whatever the interpretation, the Law initially translated into the clock frequencies of CPUs roughly doubling every 18 months or so: this was a result of the production and use of smaller transistors which could be switched more quickly. However, smaller transistors lead to voltage leakage and, when the silicon wafers get thin enough, attempting

to build circuits at the size of atoms becomes a limiting factor. Therefore, physical limitations mean that this interpretation of Moore's Law is coming to an end: whilst huge advances in performance have been achieved by harnessing this improvement in the speed of CPUs, since about 2005 clock frequencies have stopped increasing significantly.

For the last 10 years, the increase in number of transistors has resulted in more cores per chip, rather than each core having a faster clock. Currently, Intel's 6-core, 12-thread, Xeon L5640 chips are available in small cluster machines used within the pensions industry, and the Intel 24-core, 48-thread, Xeon E7-8890 v4 chips are available to build into servers and clusters from mid 2016 [46] and [92].

From developments made in recent years by both Intel and AMD, it looks likely that the current trend towards incorporating more cores, rather than faster clocks, will continue for the near future. These multi-core chips provide a good starting point for the investigation in this project: investigating the performance of highly optimised CPU code provides a benchmark to measure other technologies against.

2.4.1.2 Programming CPUs

There are many references providing details on how to improve the performance of codes, e.g. [2], [3], [5], [32], and [44]. Many of the techniques in these books have been incorporated in the current phase of this project: for example, ordering if-statements in decreasing order of probability of meeting the criteria, ensuring that multidimensional arrays are traversed in the correct order, extracting common sub-expressions, and using multiplication-by-reciprocal rather than division have all been included. Bacon [5] suggests that 'strip mining' is unlikely to have significant effect on performance until the data have been tuned to fit within cache, or "other transformations have been performed first". Strip mining has been used as part of this project: it forms the basis for vectorisation, and is discussed further in Section 2.5.5.1.

Another feature of modern CPUs which may have a major benefit for this project is the availability of vector units; these allow similar calculations, on adjacent elements of arrays, to be performed very efficiently. While utilisation of vector units can enhance performance of calculation-intensive codes [8, Table 9.6], the future direction in which vector units will develop is uncertain: it is not clear whether manufacturers will increase the number of units in each chip, or increase the width of each unit without changing the number of units.

The benefit of vector units can be harnessed by improving the vectorisation oppor-

tunities within the code, although the price to be paid for improved performance is usually increased programmer effort. Increasing the amount of vectorised calculations is likely to become essential to obtaining best performance on CPUs, and is therefore a significant part of this project. However, in order to extract a high level of performance, the code in Chapter 8 has been restructured so that nested loops have the outer loops parallelised and the inner loop vectorised: this is how Intel's compiler handles auto-parallelisation and auto-vectorisation [45, Page 1462].

Current standard software within the pensions industry uses CPUs. However, the extent to which that software can benefit from recent advances and features of modern CPUs is limited. Most valuation packages produce code which runs on only a single thread, although some packages have recently added 'more advanced' options in some packages to allow for up to eight threads to be used. Conversely, one commercial package, released in 2008, almost defeats recent advances in hardware by employing version 5.5.1 of Borland's C++ compiler from 2000.

2.4.1.3 Future Developments

Given the speed, and variety, of developments in computing over the past decade, trying to predict the future is likely to be prone to error. However, there are a few observations which may be relevant to the performance required for the processing requirements in the actuarial world.

Firstly, it seems reasonable to believe that the multi-core revolution will continue. Given that the introduction of more cores, rather than higher clock frequencies, was partly driven by physical limitations within chip manufacture, and that such limitations seem unlikely to be overcome (otherwise they would probably have been overcome by now), it seems unlikely that there will be a move towards fewer cores per chip in the foreseeable future. Therefore, programming techniques which benefit from higher numbers of cores will become increasingly important if the potential of future chips is to be realised.

Secondly, as the number of cores increases, it is likely that it becomes necessary to move away from current CPUs, e.g. Intel's latest Xeon Phi has about 70 cores, each capable of four-way multi-threading. Although the majority of HPC programming techniques may be used on these many-core chips, it is unlikely that they will be in desktop machines, on desks, within the office space: this leads to the possibility that they will be perceived as 'special' in some way and therefore they are unlikely to be

targeted by the creators of valuation packages.

2.4.2 Graphics Processing Units

GPUs are more specialised than CPUs: their original purpose was for rendering and manipulating pixels. As long as the final image is correct, the processing of each pixel can be performed independently of all others, and hence the order of the processing is irrelevant. Further, so long as moving images appear continuous, it does not matter how quickly the data is processed: the human brain considers a rate of slightly less than 30 frames per second to be continuous [53] and so there is no need to produce images at a much higher rate than that. The combination of independence and lack of need for outstanding speed means that GPUs have evolved to have large numbers of relatively slow processing units. Small GPUs, such as NVIDIA's 640-core GTX950M, are available in current laptops while the larger graphics cards have a few thousand cores: for example, the K40c, as used in NVIDIA's CUDA Centre at Durham University [26], has 2880 stream cores. However, on all of these GPUs, each thread is processed more slowly than would be possible on a CPU of equivalent age: this is simply a result of GPUs having lower clock frequencies than CPUs.

GPUs started to be used for general purpose programming when NVIDIA introduced CUDA [67] in 2007: having been developed by NVIDIA, CUDA may only be used to write code which runs on their GPUs. CUDA was originally a set of extensions to C which allowed data to be transferred on and off the GPU, and also controlled how the compute kernels were launched. As CUDA has evolved, further functionality has been added and extensions to Fortran have been incorporated in the PGI compiler.

OpenCL [51] was introduced in 2009 as an alternative means of allowing general purpose C code to run on GPUs. Like CUDA, OpenCL is an extension to the language which places the correctness requirements in the programmer's hands, and it continues to evolve as recent C++ standards are supported [51]. In contrast to CUDA, OpenCL may be used to create code which runs on GPUs from a wide range of manufacturers: the costs of this increased applicability are increased complexity in setting up the code to control the run time environment to account for the specific GPU being used and, possibly, poorer performance.

GPUs may also be programmed using directive-based languages analogous to OpenMP. OpenACC was introduced in 2011 by a consortium consisting of NVIDIA, Cray, The Portland Group, and CAPS [71]. This simplified the programming of GPUs, but limited

options to the use of either the PGI compiler suite or the Cray compiler suite.

In 2013 OpenMP was extended to cover GPU programming with the release of Version 4.0 of the standard [73]. The advantage of this is that OpenMP is implemented by many compilers, removing the tether to a specific vendor. OpenMP's support for GPUs was further extended by the release of Version 4.5 in 2015 [74]: this effectively made OpenACC redundant, and it is not expected that OpenACC will evolve further.

The fact that compilers implementing OpenMP 4.x are available means that it should be relatively straightforward to program GPUs. However, the recentness of the release of these OpenMP standards means that the implementations in many compilers are not yet mature, leading to the possibility that the compilers contain bugs resulting in either an inability to produce code for a particular GPU or, worse, the production of incorrect code.

2.5 High Performance Computing

Over the last 25 years, conventional wisdom within life offices has been to improve computing performance by obtaining more up-to-date PCs: it has been sufficient to rely on chip manufacturers increasing the processing power of computers by increasing the CPU's clock frequency roughly in line with Moore's law. However, it is no longer possible to rely on the speed of CPUs increasing significantly: instead, chip manufacturers have moved to placing more compute cores on each chip in order to increase their processing power. Therefore, to benefit from modern processors, it is necessary to embrace programming paradigms which utilise multi-core chips. Whilst some commercial valuation packages do now have some parallel computation capability, these capabilities are often limited and so much of the benefit of multi-core CPUs is not harnessed.

This project uses standard parallel programming techniques to develop software which is capable of running on modern machines which have multi-core chips. This project also uses several techniques which are used in serial programming but are also of benefit in a parallel environment.

2.5.1 Applicability of High Performance Computing

The problem of demonstrating solvency, as set out in Section 2.3.3.1 is, at first sight, one of high throughput computing, rather than high performance computing. It would be possible, using the current industry standard software, to split the calculations across vast numbers of PCs to perform the calculations for each of the simulations for each future time step. However, there are drawbacks to this approach: firstly, the results need to be collated, and the quantity of intermediate results files involved would lead to concerns about the accuracy of the collation; secondly, hardware reliability considerations suggest that using several thousand PCs is likely to lead to at least some of those machines failing during a run; and, thirdly, practical considerations of housing (and cooling) several thousand PCs is likely to require a bespoke building, and the cost of providing electricity for such a large number of machines is a definite perceived problem. These issues mean that adopting such a process for calculating the additional capital requirement is impractical and untenable.

In order to correctly demonstrate solvency, an HPC solution is favoured since such an approach could incorporate the collation of the results within the calculation process. That is the main focus of the investigation in this project: a high performance solution is used, so that the required high throughput is achieved.

There is ample parallelism at various levels in this problem; the Monte Carlo simulations within the overall calculation are independent of each other, although their results must be combined; each policy within each simulation may be regarded as independent of all others; and, all probability and present value calculations for each payment within each policy could be calculated independently of all others. This abundance of parallelism allows the investigation of various approaches to performing the calculations, using various paradigms.

2.5.2 Hardware

High performance computers run parallel programs, i.e. programs which perform different parts of the same calculation at the same time. These machines are usually classified as *distributed memory* or *shared memory*: the differences between these machines are described in the following sections.

2.5.2.1 Distributed Memory Machines

A *distributed memory* machine has more than one node: in the case of large supercomputers, there could be several thousand nodes, while small clusters may only have a handful of nodes. All the nodes are able to communicate with each other via a network. Each node has its own computing capability, usually with one or more CPUs on each node, or a CPU and a GPU on the same node. Each node has its own memory: in order for a node to access a variable in another node's memory, the two nodes need to communicate. It is the fact that not all the memory is on the same node which makes these distributed memory machines.

These machines allow separate parts of a program to run on each node, leading to multi-node parallelism. They also have large quantities of memory, usually consisting of a reasonable amount (often a few tens of giga-bytes) on each node: for a machine with several thousand nodes, this results in many tera-bytes of memory.

The main reason this project uses a distributed memory machine is to harness the number of compute cores it has, rather than to use the volume of memory available: the cores are used as a convenience over using a collection of several hundred PCs. However, as described in Section 8.2.1, the processing regime in this project requires roughly 1 GB of RAM per scenario, so that running 1000 scenarios at same time requires 1 TB of RAM. The distributed memory machines used in this project are described in Sections 2.7.2 and 2.7.3. Although this project uses a distributed memory machine with thousands of nodes, these large machines are not common within the industry: this is largely due to the power and cooling costs associated with these machines outweighing their usefulness.

2.5.2.2 Shared Memory Machines

A *shared memory* machine has more than one core on a node: all cores on that node can directly access the memory on that node. There is no standard layout for a node of a shared memory machine: each node may have one or more CPUs; each CPU may have one or more dies; and each die may have one or more cores. However, all machines share common features, and the fact that the node has memory which may be accessed by all cores on the node allows multi-core parallelism within each node.

The memory of these machines has a hierarchical structure which approximately reflects the layout of the cores: the relative nearness of a level of memory to the computational

ability is reflected in the time to access a value from that memory, rather than any geographical interpretation (although physical considerations usually result in the faster memory being located nearer the arithmetic units). In the following descriptions, the figures quoted relate to Intel Xeon Ivy Bridge processors: these are the CPUs in the Cray system used in this project – that system is introduced in Section 2.7.2, and the following figures are taken from Appendix B.

Starting 'closest' to the arithmetic units, there is a small cache which is only available to the core which it is associated with, and allows data to be accessed quickly. There are often two levels of core-specific cache: Level 1 cache is 32 kB and the time to retrieve data is around 1.5 nanoseconds; Level 2 cache is 256 kB and the time to retrieve data is around 5.9 nanoseconds. In addition, there is a Level 3 cache which is shared between the cores on a chip: this is 30 MB and the time to retrieve data is around 17 nanoseconds. Finally, there is the main memory which is accessible by all cores on a node: the size of this is not dependant on the CPU and is often in the region of 64 GB. The time to retrieve data is around 81 nanoseconds, and it is therefore desirable to access this memory as infrequently as possible. Clearly, being able to perform calculations on the data in caches closest to the arithmetic units will lead to the calculations completing more quickly.

However, this memory structure admits the possibility of a core in one CPU on a node needing to read data which was written by a core on a different CPU. This highlights two problems when programming shared memory machines: firstly, the fact that the write must happen before the read leads to synchronisation intricacies; secondly, the fact that the required data could have been written by a core in a different CPU means that the caches need to be kept aware of updates by other cores, although the caches do not necessarily need to be kept identical.

A related, but independent issue, is *non-uniform memory access* (NUMA): this is the situation where a core on one CPU needs to access data stored in either the cache of a different core, or the RAM attached to a different CPU. In the case of data being in another CPU's memory, it is necessary for the data to be transferred across the interconnect between CPUs: this need to copy the data back through the memory hierarchy, and the resulting bookkeeping, takes a not insignificant amount of time meaning that calculations take longer when the most up-to-date data are spread over the different memory levels of several CPUs. NUMA effects, caused by running a program across several CPUs, often degrade performance noticeably: it is possible to reduce these effects by ensuring that, as far as possible, any threads involved in a particular calculation are all running on the same CPU.

It is perfectly possible for small, standalone machines with multiple cores to be programmed using shared memory techniques: multi-core laptops could therefore serve as development platforms for codes which are to be run on larger shared memory machines. Similarly, it is perfectly possible to use shared memory techniques to program machines which have several CPUs, each with multiple cores, accessing the same shared memory.

The shared memory machine used in this project is described in Section 2.7.1. This machine has mechanisms which allow the programmer to dictate the level to which threads are tied to cores: throughout this project, all code is written and run in a manner which ensures that threads only access the memory allocated to the CPU on which that thread is running, i.e.

- the environment variable, `OMP_PROC_BIND`, is set to `true`, and
- the number of threads used does not exceed the number of cores available.

2.5.2.3 Intra-Core Parallelism

There are several forms of parallelism which may be applied within a single core: the more important ones are discussed here.

2.5.2.3.1 Pipelining

Pipelining is a form of instruction level parallelism: parts of consecutive instructions are executed at same time. Multiple independent instructions are executed simultaneously, and a new instruction is started every cycle so long as the required resources are available. Independent instructions keep the processor busy, while dependent instructions may stall the processor. Branch prediction could be a problem for pipelines: if the prediction is incorrect, then results in flight in the pipeline are wasted. Pipelining is implemented in hardware: there is no need for the programmer to intervene.

2.5.2.3.2 Superscalar Processing

Superscalar processing is another form of instruction level parallelism: instructions may be issued to different functional units in the same cycle. Superscalar processing is also implemented in hardware: as with pipelining, there is no need for programmer intervention.

2.5.2.3.3 Vectorisation

Vectorisation is a form of data parallelism: the same arithmetic operation is performed

on several pieces of data at same time. For this reason, vectorisation is also referred to as SIMD (single instruction, multiple data) processing. In order to perform SIMD processing, the compiler will produce instructions specifically for the CPU's SIMD instruction set, although different compilers may produce different instructions.

The number of data items which can be operated on depends on the size of the vector processing unit: vector units in modern CPUs are typically 256 or 512 bits wide, allowing four or eight double precision numbers to be processed at the same time – in particular, the Xeons in the Cray (which will be introduced in Section 2.7.2) have 256-bit wide vector units. However, the size of these units is increasing, meaning that vectorisation within program code may become more important.

Although vector processing is a feature of the hardware, unlike pipelining and superscalar processing, there is much the programmer can do to increase the amount of code which is vectorised: this is a result of the fact that the compiler is often unable to perform adequate analysis of the code to be able to determine that it may safely be vectorised: see Section 2.5.5 for details.

2.5.2.3.4 Simultaneous Multi-Threading

SMT is a cross between instruction level parallelism and data parallelism: it allows instructions from different threads, which are likely to be working on different data items, to execute in any given pipeline stage at the same time. This means that the architecture needs to be modified slightly: the ability to fetch instructions from multiple threads needs to be added, and the register files need to be larger (to hold data from several threads). The number of threads which may run at the same time depends on the hardware: for example Intel Xeon CPUs generally allow 2 threads per core, whereas the Xeon Phi allows 4 threads per core. The fact that instructions from different threads could be working on different data items at different stages in the pipeline means that there is likely to be contention for hardware resources such as memory bandwidth, cache bandwidth, cache capacity, logic units or floating point units.

SMT tends to fall into two categories: firstly, 'red/black processing' where alternate threads are processed on alternate clock cycles, for example the Knights Corner Xeon Phi chips; or secondly, processing different threads in the same clock cycle, e.g. Intel's Xeon CPUs, or Knights Landing Xeon Phi chips. In either case, the functional units are shared by the multiple threads, leading to a possibility of the processing rate reducing.

The benefit of SMT comes from the possibility of filling instruction slots which would otherwise be empty. The slots could be empty either because there are dependencies

between variables which need to be calculated, or because the CPU is stalled waiting for data from a lower level of cache or RAM. Therefore SMT particularly benefits codes which have large amounts of pointer chasing, and would not be expected to be of so much benefit for calculation-intensive codes.

SMT also increases the programmer's workload as a result of needing to expose more parallelism at the thread level in order to keep the functional units busy. However, with SMT the threads may be competing for hardware features such as memory bandwidth, cache bandwidth, or TLBs, etc. and such contentions are likely to adversely affect performance. Therefore, overall, it is not clear, in advance, whether a code will benefit from SMT: the effect of SMT on the code developed in this project is discussed in Section 8.2.6.1.

2.5.3 Scalability of HPC Codes

One feature of any HPC code which is of interest is its *scalability*: this considers how well the code performs as the amount of processing capacity varies. The amount of computation required to solve a particular problem is the 'workload' and, because the size of any particular problem is fixed, the workload is also referred to as the 'problem size'.

There are two forms of scalability which are commonly investigated: firstly, *strong scaling* considers how the run time for a fixed problem size varies as the number of cores varies; secondly, *weak scaling* considers a fixed workload per core, and investigates how the run time varies as the number of cores, and the resulting workload, varies. Demonstrating strong scaling involves showing that the problem can be solved more quickly as more cores are used, whereas demonstrating weak scaling involves showing that larger problems may be solved within a set time by using more cores. If a code shows strong scaling on a machine with a modest number of cores, it is often worth transferring that code to a machine with a larger number of cores. This allows more cores to be used so that the problem might be solved in a shorter time.

For profitability and solvency calculations, interest lies in strong scaling. Any particular life office will have a fixed number of policies and hence a fixed work load: if the monthly variation in the number of policies is small[3], then the workload will not vary dramatically so the overall run time will not differ significantly; if the monthly variation in the number of policies is large[4], then a different decomposition is likely to be needed

[3]Small changes will result from new business, and/or policies lapsing/maturing.

[4]Large changes will result from acquisition of a block of policies from a different office (or the sale

anyway.

For Solvency II, the workload may also change as a result of the number of scenarios changing. However, running 10^4 scenarios would not provide a much more accurate estimate of the 1-in-200 liability than just running 10^3 scenarios does: the accuracy of the estimation of the distributions from which the parameters are drawn is likely to provide a greater source of variation than the number of Monte Carlo simulations anyway.

2.5.4 Programming Techniques

Distributed memory machines require different programming techniques to shared memory machines: the different techniques are discussed in this section.

2.5.4.1 MPI Programs

It is not possible for cores on different nodes of a distributed memory machine to access variables in another node's memory without communication between the nodes. In order to facilitate such communication, programs for distributed memory machines often use MPI, the Message Passing Interface standard [62].

The MPI standard defines a specification which any MPI library should adhere to: the standard does not define how the library should be implemented, nor how any program using MPI should be written. This has the clear advantage that any program which conforms to the specification will run on any machine which has a library which implements the specification: this 'write once and run anywhere' approach allows code to be run on several machines, with only recompilation (to allow for the different architecture) being required.

When running 'a program', the MPI environment actually consists of several processes, each running its own copy of the same program, and each having its own address space. The number of processes running is referred to as the *size*, and the position of a particular process within the scheme is its *rank*. The fact that each process has its own address space means that individual instances of the program do not need to be at identical points in their execution, nor have identical values for variables. This means that, for example, separate files may be read from the disk by different processes since the values of the handles may differ across the processes.

of a block of policies to a different office).

In order to communicate, an MPI program sends messages containing data between nodes, either collectively or on a one-to-one basis: the other side of the communication is the receipt of the message and utilisation of the data contained in the message. The MPI library handles the low level communication, but it is the programmer's responsibility to ensure that the correct data are sent to the correct place, and that the program does not proceed past the point the data are required before they are received.

The data which are moved need not necessarily be fundamental types: it is often useful to pass data in a form which is more convenient to the data structures in the program. Therefore MPI contains the concept of derived data types: these are defined by the programmer, and must be available in the program which is to use them.

This project uses MPI solely as a convenient way to control the launching of several processes which all perform the same processing, albeit on different data. The MPI functions `MPI_Comm_rank` and `MPI_Comm_size` are used to determine which scenarios are processed by which ranks, i.e. there is no communication involving the movement of data from one rank to another, and consequently derived data types are not required. However, the fact that separate files may be read on different processes is taken advantage of in order to read the parameter files for different scenarios on each of the processes.

2.5.4.2 OpenMP Programs

When programming a shared memory machine, it is common to use OpenMP [72]: this is an API consisting of a collection of directives, functions, and environment variables which are added to the program code to control the number, and execution, of threads in the program.

An OpenMP program consists of a series of serial regions and parallel regions. Execution of the program starts on a single thread, performing tasks in serial. When a parallel region is encountered, a team of threads is created and the team remains until the end of the parallel region, at which point they must synchronise to ensure that they are all at the same point before the team is disbanded: whether the team is actually destroyed, or only 'retired', depends on considerations such as how long the program could run before encountering the next region (if one exists) relative to the time taken to create the team. This creation, synchronisation and destruction of threads is similar to the fork-join model used by other threaded programming paradigms, e.g. *Pthreads* [66]. Conceptually, there is no limit to the number of parallel regions which may be

encountered: for example, parallel regions could be entered within a loop, with the fork-execute-synchronise-join process being carried out for each iteration of the loop.

Within many programs, parallelism comes from the ability to concurrently perform separate iterations of loops. Although OpenMP allows the possibility of performing tasks in parallel, the code in this project uses only the loop form of parallelism. When a loop is executed in parallel, it is necessary to ensure that individual iterations do not depend on other iterations in a way which is not controlled: most simply, the iterations should be independent; in more complex cases, the iterations require some form of synchronisation.

The way in which the iterations of a loop are split across the threads is known as *loop scheduling*. There are several forms of loop scheduling, each distributing the loop iterations to threads in a different way: if each iteration requires the same amount of work, then a *static* schedule may be used, but if the amount of work varies, then it may be beneficial to use a *dynamic* schedule to try to achieve a balanced amount of work, or 'load'. If the load is not balanced, some threads may be waiting, doing nothing, while other threads are working: this is not as desirable as having all threads working, but each working for a shorter time.

Within a parallel region, variables may be classified as *shared* or *private*: these classifications determine which threads may access which variables. Shared variables may be accessed by all threads, whereas private variables may only be accessed by the thread which 'owns' them, meaning that there will be a separate instance of each private variable for each thread. The main use for the distinction in the classifications lies in the fact that private variables may be used on a particular thread to perform independent calculations, the results of which are used to update shared variables. In order to ensure that the updates to the shared variables are not corrupted by several threads attempting to perform the update at the same time, it is necessary to ensure that the updates are protected by one of the synchronisation techniques which OpenMP provides. Both the correct categorisation of variables as shared or private, and any necessary synchronisation, are the programmer's responsibility. The programmer is also responsible for ensuring that all variables accessed by different threads are accessed in the correct order. For example, a shared variable must be correctly populated by one thread before is it read by another thread: if the read occurs before the write, it is likely that the wrong value will be used by the reader. Therefore, although the OpenMP runtime handles the low level thread creation and destruction, the program must contain the correct directives and/or calls to the library in order that the threads are manipulated, and perform the required calculations, correctly.

The OpenMP standard allows the implementation to specify the upper limit on the number of threads, but that limit is usually far higher than the number of computational cores available. For example, this project uses a Cray XC30 (which is described more fully in Section 2.7.2): when using the Cray programming environment on that machine, the limit is approximately 2^{32} threads, which exceeds anything which most applications will require. There is, therefore, no reason why the number of threads used must match the number of cores available. If the number of threads is lower than the number of cores, then there will be idle cores: this is useful in applications which have a high memory requirement per thread, allowing all the memory to be utilised by the smaller number of threads. If the number of threads is greater than the number of cores (or SMT threads, where supported by the hardware), then some of the OpenMP threads will be held in an idle state while others use the cores available, before being placed on a core which was hosting a thread which has finished its work: this swapping of threads on and off cores is managed by the operating system and the OpenMP implementation – the programmer may be aware of it, but has no control over it. Using a number of threads which exceeds the number of cores available generally does not enhance performance because there will be delays while the threads, and their relevant data, are switched.

This project makes heavy use of shared memory techniques: the main use is in processing separate parts of large arrays after the data have been read from the disk. Within this project, the number of OpenMP threads only ever exceeds the number of available cores when testing whether using SMT, with two threads per core, is beneficial: the results of this investigations are discussed in Section 8.2.6.1.

2.5.4.3 Hybrid Programs

It is not uncommon, when programming large distributed memory machines, for the distributed memory model to be used in the same program as a shared memory model to create *mixed-mode*, or *hybrid*, programs: MPI is used to communicate between the nodes, and OpenMP is used for the shared memory within a node.

The MPI specification allows four levels of thread safety, which are differentiated by their symbolic values: these are defined as

- `MPI_THREAD_SINGLE`: this allows only a single thread of execution
- `MPI_THREAD_FUNNELED`: this allows the programmer to create threads, but only

the master thread[5] can call MPI routines

- `MPI_THREAD_SERIALISED`: this allows the programmer to create threads, and any thread can call MPI routines, but routines can only be called by one thread at a time
- `MPI_THREAD_MULTIPLE`: this allows threads to be created, and any number of threads can call MPI routines at any time

The level of safety allowed is determined by which of these constants is passed to the MPI initialisation routine, `MPI_Init_thread()`, which takes *required* as an input, and produces *provided* as an output. Since not all implementations of the MPI library provide all levels of safety, the result must be queried to ensure that the desired level of safety has been granted, in order that the program will work as intended.

This project uses mixed-mode programming for the brute force code discussed in Chapter 8: MPI is used to coordinate which scenarios are run on which nodes, and OpenMP is used to process the policies in parallel within a node. Given this usage, there are no MPI calls inside parallel regions, and so thread safety is not a problem: it therefore suffices to use `MPI_Init()`.

Section 8.2.5 considers which combination of processes per CPU and threads per process leads to best performance for the machines used[6].

2.5.5 Aiding Vectorisation

Section 2.5.2.3.3 mentioned vectorisation: this is a form of parallelism that improves the speed at which computationally intensive loops may be processed. Loops which benefit from vectorisation have a few specific features: they have short bodies; they have a single entry and a single exit; there must be no branches (jumps out of the loop are forbidden, but 'if-then-else' constructs are allowable); and there can be no function calls (other than to intrinsic functions). In order to increase the level of vectorisation in a code, several techniques may be used: in particular, strip mining has been used to improve the performance of the code in this project.

[5] The master thread is the one which initialised MPI.

[6] There is no reason to expect that any machine will have the same optimum combination of factors as any other machine (unless the two machines are perfectly identical).

2.5.5.1 Strip Mining

The work at the centre of many calculation-intensive codes involves iterating over a collection of items, which are usually stored in arrays or lists: the actual storage method is purely an implementational detail, and therefore not of prime importance. The collection of items, or 'iteration space', which may contain any number of dimensions, can usually be partitioned to aid processing.

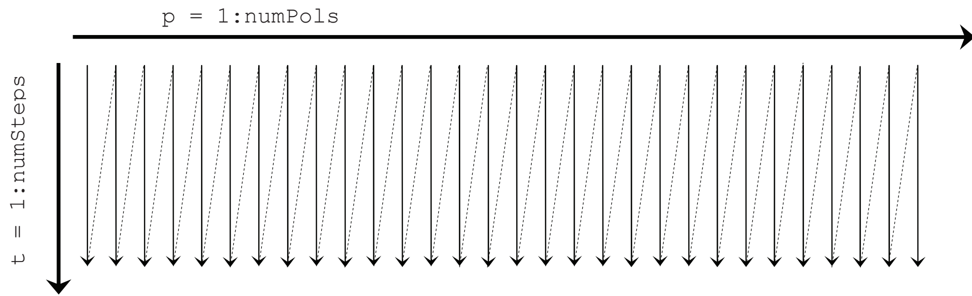
Strip mining is the processing of a large iteration space in smaller chunks: each chunk is a part of the overall space. In one dimension, the iteration space may be envisaged as a strip, which is split into smaller strips. In two dimensions, the rectangular space is split into smaller rectangles which are often long thin strips. In multi-dimensional spaces, the strips may be taken over each dimension at the same time: for a three dimensional space, this leads to the production of small blocks, and so strip mining is also known as 'blocking'. Strip mining has two advantages in relation to vectorisation: firstly, it may allow vectorisation of loops which could not otherwise be vectorised; secondly, it allows the length of the block to be changed so that the code may be tuned to improve performance on each machine on which it is run.

With vectorisation, the same operation is performed on several pieces of data at the same time: this means that the data must be independent, and so dependencies must be removed before vectorisation can be applied. Consider a skeleton for a code to value annuities: the code contains a loop over independent policies, and a loop over time steps within policies. This initial situation is depicted in Figure 2.3 (a) and the pseudo code to describe it is

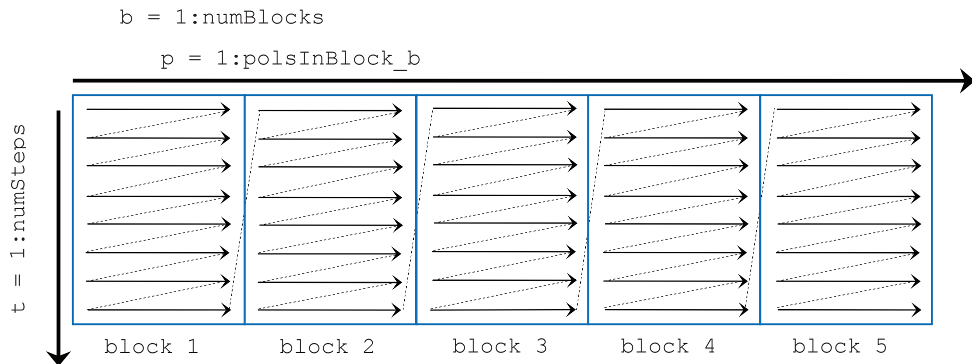
```
for p=1:numPols
    for t=1:numSteps
       process step t of policy p
```

The second loop is inherently sequential because of the dependencies between time steps: these dependencies mean that the loop cannot be vectorised. By adding another loop which processes the policies in blocks, the pseudo code becomes

```
for b=1:numBlocks
    for p=1:polsInBlock_b
        for t=1:numSteps
          process step t of policy p
```



(a) Processing each time step within each policy.



(b) Processing a particular time step for each policy within a block.

Figure 2.3 Traversing the 'policy *vs.* time step' iteration space.

It is then possible to swap the order of the loops so that, for each policy within the block, the same time step is processed. This situation is depicted in Figure 2.3 (b) and the pseudo code to describe it is

```
for b=1:numBlocks
    for t=1:numSteps
       for p=1:polsInBlock_b
          process step t of policy p
```

There are now no dependencies in the innermost loop, and so vectorisation may be applied to the `p` loop. Further, since policies are processed independently, there can be no dependencies between blocks, and so the `b` loop over blocks may be processed in parallel, using OpenMP for example. It is necessary to tune the length of the block so that there is enough work to keep the vector units occupied, but not too much work so that locality of data is lost, leading to more data retrieval than necessary.

The use of strip mining to allow the removal of dependencies is an optimisation technique which could be made for many codes, not just actuarial valuation codes. It

is used here because of the need for the highest level of performance possible to achieve the throughput necessary for the brute force approach to Solvency II. Blocking techniques have been used in all the calculation intensive parts of the code. Section 4.4.2 discusses the actual implementation used in this project, and Section 8.2.6 discusses the performance gains from the use of vectorisation.

2.5.5.2 Compiler Directives

Modern compilers usually permit a choice between several levels of optimisation: the higher levels often contain 'auto-vectorisation', whereby the compiler vectorises loops which i) it recognises to be vectorisable, and ii) it determines to be beneficial. However, there may be cases where the compiler cannot decide whether vectorisation is beneficial, or even possible: in these cases, the addition of directives, or pragmas, is beneficial.

The Intel compiler, for example, uses `#pragma SIMD` to tell the compiler that the loop may be vectorised. The addition of these pragmas is an assertion by the programmer that there are no data dependencies within the loop, and hence that the concurrent processing of several elements of the loop will not contravene the semantics of the code. Other compilers may have different directives. However, since Version 4.0, OpenMP has provided a standardised set of vectorisation-related directives: this standardisation will enhance the portability of the code, but is unlikely to improve performance of the code produced by a particular compiler.

Neither automatic vectorisation, nor the use of pragmas, require the use of loop blocking: it is possible that a loop of arbitrary length may be vectorised. However, there may be performance benefits for being able to select a particular block length within a loop which is to be vectorised: those benefits come from efficiently using the cache and having a number of loop iterations which is divisible by the vector length. Given that different CPUs have different cache configurations, the block length must be tuned to each machine to obtain optimum performance: see Section 8.2.5 for details.

2.5.5.3 Redundant Calculations

The optimum vector length depends on the size of the vector unit in the chip. However, it is possible that the length of the data array is not an integer multiple of the optimum vector length. The processing of the 'fractional block' will decrease the performance of the code, and it is therefore desirable to ensure that all blocks have this same optimum length. A simple way of equalising the lengths of the block is to pad the last block with

‘null data’: such data should be zero for additive calculations, and one for multiplicative calculations. These data are used for null calculations which allow the vector processor to operate on a full vector: the calculations are ‘redundant’ since they are carried out solely to enhance the machine’s performance.

An extreme instance of this case is the situation where the entire data set is smaller than the width of the vector processor. In this case, there is only one trip through the loop, and it becomes debatable whether padding the data is beneficial: it may be better to just use a non-vectorised version of the code.

2.5.5.4 Removing Conditional Tests

The data required for profitability and solvency calculations are a block of policies, and a set of assumptions. Each policy has a policyholder associated with it and, due to the nature of population dynamics, there will be a spread of dates of birth of the policyholders within a block. In general, life tables are populated with mortality values up to age 120. The projection step in which the end of the life table is reached will differ with the date of birth. Therefore, a naive approach is to implement the lookup in the life table using a test to see whether the life is above age 120: this test will need to be performed for every policy for every step. Although most vector units do now allow conditional assignments, such tests degrade performance.

A better approach is to remove these tests by populating the mortality table in the implementation with ‘certain death’, i.e. set $q_x = 1$ for $121 < x < \omega$, where ω is the maximum age implied by the data set for the current projection. There is then no need to perform the conditional test, but the complete set of calculations are performed for all policies for all steps even when, for example, the policyholder is age 135. This technique has been used in the code developed in this project in order to allow the removal of the test, thereby permitting maximum calculation within the vectorised code.

2.6 Related Work

There is a distinct shortage of peer-reviewed literature relating to the application of HPC to actuarial calculations for assurance policies: to the best of our knowledge, this is the first published work in this area. Therefore, this section summarises available actuarial references to HPC and then considers the application of HPC to other areas

of finance.

2.6.1 HPC in the Actuarial World

The actuarial world has been slow to adopt HPC: a selection of results from simple searches reveals the following:

- Brackett and Renzi's article for the Milliman website [61]: this discusses the fact that "Milliman worked with Microsoft to integrate its financial modeling tool, MG-ALFA (Asset Liability Financial Analysis), with Windows Compute Cluster Server 2003." However, MG-ALFA is not used in policy-level actuarial valuations. Whilst there is little of academic interest in this article, it does state that "a 1000-scenario model with reserves and capital based on 1000 paths at each valuation point for a 30-year monthly projection requires the cash flows for each policy to be projected 360 million times", which reinforces the scale of the problem that this project is attempting to address.
- Newberry *et. al*'s article for the Microsoft website [60]: this states that it "is a preliminary document and may be changed substantially . . . ". It does however restate that "a stochastic-on-stochastic 30-year projection requires hundreds of millions of individual policy valuations." Page 15 states that "Towers Perrin produces MoSes actuarial software." and ". . . HPC-compatible version of MoSes was launched in 2008": these statements are both simply advertising for MoSes, which could be expected from the fact that one of the authors is a Towers Perrin employee. Also, the 'HPC' to which this is compatible is Microsoft's HPC Server, which is not commonly used in large HPC systems.
- Pledge's article for the Society of Actuaries [83]: this discusses the fact that Microsoft presented Milliman with an award for their MG-ALFA software which, as previously noted, runs in a Microsoft environment.
- The Royal Society of Edinburgh's conference, 'Computation in Finance & Insurance, post-Napier' (April 2014) for which none of the papers are available. The minutes [79] indicate that:
    a) in Smith's talk about Solvency II, he states that "The performance of the FTSE 100 is key to calculating SCR." and hence (according to the record of this talk) 'the interest rate must be re-drawn every step.';
    b) Keenan and Withy, two Milliman employees, discussed the performance of their proprietary software running in the Microsoft cloud; and
    c) Phillips discussed the use of GPUs with python as a scripting language,

although there is nothing to state how the calculations were done on the GPU.

2.6.2 HPC in the Financial World

In recent years, HPC has been applied to a number of financial applications. However, much of this work has related to the computational requirements of securities and options trading, and is therefore well outside the arena of actuarial valuations: for example

- Bethel *et al.* [7] investigated the application of HPC to the possibility of suspending trading when a 'flash crash' occurs in the market.
- Lindeman [55] investigated the use of shared memory parallelism in the solution of 'an inverse problem', represented by a sparse matrix, generated in the calibration of pricing models, and concluded that using Intel's *Threaded Building Blocks* and two 4-core CPUs led to better performance than a commercial Levenberg-Marquardt solver when considering realistic problems.
- Dozsa *et al.* [25] looked at whether the high-bandwidth and low-latency of Blue Gene systems provide sufficient performance to allow real-time stream processing of large volumes of data, concentrating their work on the financial sector.
- Doan *et al.* [24] implemented a parallel version of American-Bermudan options pricing on a Grid using Java.
- Irturk *et al.* [48] implemented an FPGA version of Markowitz's mean variance framework for optimal asset allocation.
- Daly *et al.* [21] demonstrate that, using a Blue Gene machine, real-time time series analysis can be performed on a multivariate basis, rather than a simpler univariate basis.

Despite this variety of areas to which HPC have been applied, there is, as indicated in Section 2.6.1, a lack of research into the application of HPC to actuarial valuations, or to solvency and profitability calculations for life assurance policies. However, the work in this project does have some similarities with other work previously undertaken in other areas of finance. Firstly, Dixon *et al.* [23] worked on market value-at-risk estimation, and by changing algorithms, and using other techniques, achieved a speedup of 148× for their GPU calculations; this project implements a different algorithm, and uses other techniques to improve performance, whilst transferring the calculations to GPUs is suggested as future work. Secondly, Williams [93] presents a discussion on the

distribution of calculations using coarse-grained parallelism to spread the calculations for fixed income securities over multiple machines; this project harnesses the fine-grained parallelism which is inherent in the portfolio of policies, whilst future work could exploit coarse-grained parallelism through the use of clusters of GPUs.

2.6.3 Use of GPUs

Over the last few years, there has been much interest in discovering whether there is any performance benefit in the approach of applying the large numbers of slower cores to non-graphics codes: this approach has become known as "general purpose computing on GPUs". A wide variety of scientific and engineering codes have been ported for execution on GPUs, and the performance of these codes compared to the CPU codes: for example

- *molecular dynamics*: Lui *et al.* [58] obtained speedups of the order of 16× when comparing a simulation running on a single Pentium4 3 GHz CPU against a system with an AMD Opteron 2210 1.8 GHz with an NVIDIA GeForce 8800 GTX 512.
- *molecular dynamics*: Ufimtsev and Martinez [90] used a workstation with a single Core 2 quad-core 2.66 GHz CPU and two NVIDIA GeForce GTX 295 cards, and obtained speedups of the order of 6× for small molecules, roughly 20× for medium molecules, and approximately 100× for large molecules.
- *fluid dynamics*: Corrigan *et al.* [16] achieved speedups of 9.4× for single precision and 1.56× for double precision, when compared to OpenMP running on 4 cores.
- *genetics*: Chang *et al.* [12] achieved a speedup of between 40× and 90× for one aspect of their work, and roughly 35× for another aspect.
- *linear algebra*: Lahabar [52] obtained speedups of between 3× and 8× over Intel's MKL and between 3× and 59× over Matlab, depending on the size of the problem.
- *astronomy*: Harris *et al.* [35] achieved a speedup of two orders of magnitude when compared to a traditional CPU-based approach.

In addition to these non-financial arenas, there has been interest in applying GPUs to the pricing of options. Options are financial instruments which are traded and give rise to a profit or loss depending on the price of some other commodity at some future date [37, Section 1.3]; for example it is possible to buy an option which depends on the future price of IBM shares [9, Section 14.1]. The pricing of options involves simulation of non-trivial random walks, and is therefore computationally intensive. In recent years, there has been interest in the performance of option pricing codes on GPUs: for example

- Joshi [50] achieved a speedup of about 150× for pricing Asian options.
- Tian *et al.* [85] achieved a speedup of 22× when implementing a new CPU-GPU version the Longstaff-Schwartz method.
- Lui *et al.* [57] achieved a speedup of about 43× for pricing Exotic options, and about 18x for pricing Vanilla options.
- Jauvion and Nguyen [49] achieved a speedup of 30× when parallelising the Cox-Ross-Rubinstein pricing model on a GPU.
- Cvetanoska and Stojanovski [20] implemented a GPU algorithm for pricing American options which was "6500 times faster than the serial algorithm implemented on the CPU".

2.6.4 Performance of GPUs

Many of the comparisons of performance between CPUs and GPUs are misleading. From the comparisons in Appendix B.4, GPUs have about 11× the Flop rate and roughly 8× memory bandwidth of CPUs: GPUs cannot, therefore, be more than about 10× faster than CPUs. Pakin [76] highlights some major concerns which should be considered when performance comparisons are being undertaken. These observations include

- the need to ensure that the code is run on systems which are not doing anything else at the same time (or, at least, ensure that any other work does not interfere with the work being considered),
- ensuring that the same level of arithmetic precision is used on both systems, rather than using double precision on CPUs and only single precision on GPUs,
- reporting the time required for the entire routine, rather than just the calculation kernel which, when run on a GPU, may not include the time required to transfer the data to/from the GPU,
- ensuring that CPU code has been as heavily optimised for the relevant CPU architecture as the GPU code has been for its architecture,
- ensuring that the algorithm used in the two architectures are not fundamentally different, and
- comparing the size of problem which solves the problem, rather than running more iterations than necessary on the GPU in order to reduce the cost of transferring the data to the GPU.

The relative performance of the two architectures, allowing for the considerations just

outlined, has been scrutinised by Lee *et al.* [54], and their observations and conclusions reinforce the need for consistency. The first observation made is that performance of CPUs is good so long as they are only doing one thing at a time, whereas GPUs can perform switching between elements when long latency events occur, thereby aiding the GPU's perceived performance. The second observation is that the two architectures have different designs, and this usually results in the optimal data layout for the GPU code being different to that which would be optimal for the CPU.

The overall conclusion is that fair and unbiased comparisons between the performance of CPUs and GPUs are extremely difficult to obtain. Certainly, Cvetanoska and Stojanovski's paper on pricing American options [20] seems to have benefited from both a different algorithm and performing artificially high numbers of iterations on the GPU to emphasise its parallel capabilities.

Giles *et al.* [33] have taken these factors into account in their investigation of the use of GPUs for finite difference solvers. They used both one-factor models of the algorithm, which require a small amount of data, leading to a compute intensive situation, and three-factor models, which require more data, leading to more transfer of data between CPU and GPU: these models are therefore ideal as comparators for performance. Their results show that, whether for the one-factor or the three-factor models, the times for 2 Xeon processors are in the region of only $3\times$ to $7\times$ the times obtained using an NVIDIA K40, indicating that the GPU performance is consistent with the factors derived at the start of this section.

As with references to HPC in the actuarial world, references to the application of GPUs to actuarial valuations and profitability calculations within the life assurance and pensions industry are elusive.

2.7 Platforms Used

This section details the platforms used in this project, and gives an indication of what they were used for.

2.7.1 48-core Opteron SMP

This is a server hosted by Edinburgh Parallel Computing Centre, which is a department within The University of Edinburgh. This machine has: four 12-core AMD Magny-

Cours[7] 1.9 GHz CPUs which can be used as a single 48-core SMP[8], using OpenMP across all 48 cores in a single program; a total of 64 GB of RAM; Scientific Linux V6.5; Version 12.1 of the Intel compiler suite, which implements V3.1 of the OpenMP standard. This machine is used to measure performance and scalability of code which calculates profitability arising from a block of life annuities, as discussed in Chapter 5.

Although multi-core machines are becoming available in the life and pensions industry, they tend to run Microsoft HPC Server, and it is unusual that these machines are configured such that more than one CPU can be treated as an SMP. This machine is representative of those machines which could be used in life offices, if such machines were configured appropriately. In this thesis, this machine is referred to as 'the Opteron cluster'.

2.7.2 Cray XC30

This machine is the current UK academic supercomputer: it is provided by EPSRC and run by Edinburgh Parallel Computing Centre. It has 4920 compute nodes [30]; each node has two 2.7 GHz 12-core Intel E5-2697 Ivy Bridge processors; each core can support two hardware threads, allowing SMT (as discussed in Section 2.5.2.3.4); the majority of the nodes have 64 GB of RAM, while a few have 128 GB of RAM. There is a large selection of software on this machine: this project uses V5.2.56 of the Cray programming environment, which implements V4.0 of the OpenMP standard, and Cray `mpich` V7.2.6, which can be used to coordinate jobs across several nodes.

This machine is used to measure performance and scalability of the brute force code, as discussed in Section 2.3.3.1. Although supercomputers are rare within the life and pensions industry, interest in this machine arises from the High Performance Computing aspect of this project. In this thesis, this machine is referred to as 'the Cray XC30'.

2.7.3 16-core Xeon Cluster

This is a server hosted by Edinburgh Compute and Data Facility, which is a department within The University of Edinburgh. This machine has a myriad of types of nodes, which are described in Appendix B.2. Despite this variety, the code developed in this project does not require vast amounts of memory per core. Therefore, all runs were done on standard nodes: each node has two 2.4 GHz 8-core Intel E5-2630 V3 processors,

[7]Opteron 6168

[8]SMP = Symmetric Multi-Processor.

and 64 GB of RAM; scientific Linux V7.2; Intel 16.0.0 compiler suite, which implements V4.0 of the OpenMP standard; MPI functionality is provided via openMPI V1.6.5.

Small numbers of nodes (i.e. up to 5) are used to represent modern hardware which could be available in industry: it is used to measure the performance of the code which performs the brute force approach to solvency, as described in Section 2.3.3.1. In this thesis, this machine is referred to as 'the Xeon cluster'.

Chapter 3

Change of Reserving Algorithm

The simplest way to reduce a program's execution time is make it do less work. In general, this may be achieved either by reducing the amount of I/O, or by performing fewer operations. For the solvency and profitability calculations considered here, the amount of input data is already very small, and the amount of output data is negligible, so the only realistic opportunity to reduce the workload is to do fewer calculations. This chapter derives a vector form of a recurrence algorithm for calculating reserves: that recurrence algorithm significantly reduces the volume of calculations required, leading to a dramatic increase in speed.

Section 3.1 provides motivational examples by producing recurrence relations for simple annuities using straightforward manipulation of the fundamental summation formulae. It also considers the change in representation from a policy-based approach to a survival-based approach. Section 3.2 derives the recurrence for a general policy, indicating that a recurrence may be found for any policy type. Section 3.3 introduces a simplification which reduces the implementational complexity of the recurrence by removing those survival states which do not require a reserve: these states have been christened 'Zero Reserve States'. Section 3.4 discusses features which must be considered when the recurrence is implemented. Section 3.5 contains a summary of the chapter.

3.1 Motivational Examples

In actuarial valuations, it is usual to think in terms of the underlying contract: this generally makes things simple to understand because there is a direct relation back to

the circumstances under consideration. However, in the derivation of the algorithm considered in this chapter, it becomes helpful to think in terms of the survival state of the lives to which the policy relates. This section provides an overview of the transition from a policy based approach to a survival based approach.

3.1.1 Single-Life Annuities

As discussed in Section 2.1.5, annuities may be paid in arrear, or in advance, although many annuities are paid part way through each period; obvious examples are pensions where the payments are often made on the '*monthiversary*' of the policy inception date, which is often the policyholder's 60$^{\text{th}}$ or 65$^{\text{th}}$ birthday. In Section 2.1.3 it was noted that the advance and arrear cases are just special cases of annuities where payments are made part way through each period; it is therefore appropriate that only cases where payments are made at a fraction $f \in [0,1]$ through the step are considered.

Level, single-life annuities serve as an introduction to several concepts which become useful later, particularly when considering other types of annuity. For a single-life annuity with level payments which are made at some fraction f through the period, the summation formula for the reserve factor is

$$a'_x = \sum_{t=0}^{\infty} {}_{t+f}p_x \times v^{t+f} \quad \text{for } f \in [0,1] \tag{3.1.1}$$

The derivation of a recurrence relation for these policies is straightforward:

$$\begin{aligned} a'_x &= \sum_{t=0}^{\infty} {}_{t+f}p_x \times v^{t+f} \\ &= {}_fp_x \times v^f + \sum_{(s+1)=1}^{\infty} {}_{((s+1)+f)}p_x \times v^{((s+1)+f)} \\ &= {}_fp_x \, v^f + p_x \, v \sum_{s=0}^{\infty} {}_{(s+f)}p_{x+1} \times v^{(s+f)} \end{aligned}$$

and, by comparing the summation with Equation 3.1.1, the recurrence relationship is

$$a'_x = {}_fp_x \, v^f + p_x \, v \, a'_{x+1} \tag{3.1.2}$$

Note that this derivation assumes that mortality rates and interest rates are constant in time: in Section 3.4.3 these assumptions will be removed. Note also that, when written in the form of Equation 3.1.2, the recurrence runs backwards in time: this is

convenient since natural boundary conditions exist (or may be assumed) at the end of the policy, e.g. $p_{120} = 0$.

A by-product of the recurrence running backwards in time is that if the distributions from which the parameters for a particular step are drawn do not change over time, then the reserves calculated at older ages in the process of obtaining a'_x are valid for those older ages. Hence, a reduction in the overall amount of computation may be achieved by simply storing those intermediate results: this is the approach taken in Section 8.5 which considers the case where parameters do not need to be re-drawn from their distributions.

For an annuity, with level payments of amount 1, payable annually in advance, Equation 3.1.2 reduces to

$$\ddot{a}_x = 1 + v\, p_x\, \ddot{a}_{x+1}$$

A useful side-effect of using the recurrence is that it removes the need to use the power function to compute v^{t+f} in Equation 3.1.1, and replaces it with multiplication, which is about 20 times cheaper in modern hardware.

3.1.2 Two-Life Reversionary Annuities

Suppose that two lives are aged x and y, and that they are independent. Assume, without loss of generality, that the payment is made to (y) after the death of (x). Then, for a reversionary annuity,

$$\begin{aligned}
\Pr[\text{payment at time } t \text{ is made}] &= \Pr[x \text{ is dead at time } t \text{ and } y \text{ is alive at time } t] \\
&= \Pr[x \text{ dies within time } t] \times \Pr[y \text{ survives for time } t] \\
&= (1 - {}_t p_x) \times {}_t p_y
\end{aligned}$$

For the case where level payments of amount 1 are made part-way through an interval, at some fraction f from the start of each interval, the summation formula for the reserve factor is

$$a'_{x|y} = \sum_{t=0}^{\infty} (1 - {}_{t+f}p_x) \times {}_{t+f}p_y \times v^{t+f} \quad \text{for } f \in [0, 1] \tag{3.1.3}$$

Again, the recurrence relation may be derived straightforwardly:

$$a'_{x|y} = \sum_{t=0}^{\infty} (1 - {}_{t+f}p_x) \cdot {}_{t+f}p_y \cdot v^{t+f}$$

$$= {}_fq_x \cdot {}_fp_y \cdot v^f \quad + \sum_{(s+1)=1}^{\infty} {}_{(s+1)+f}p_y \cdot v^{(s+1)+f}$$

$$- \sum_{(s+1)=1}^{\infty} {}_{(s+1)+f}p_x \cdot {}_{(s+1)+f}p_y \cdot v^{(s+1)+f}$$

$$= {}_fq_x \cdot {}_fp_y \cdot v^f \quad + v\, p_y \cdot a'_{y+1}$$

$$- v\, p_x\, p_y \sum_{s=0}^{\infty} [1 - {}_{s+f}q_{x+1}] \cdot {}_{s+f}p_{y+1} \cdot v^{s+f}$$

i.e.

$$a'_{x|y} = {}_fq_x\, {}_fp_y\, v^f + q_x\, p_y\, v\, a'_{y+1} + p_x\, p_y\, v\, a'_{x+1|y+1} \tag{3.1.4}$$

In turn, the three components of the sum represent

1) the payment at time f which is made only if x has died but y is still alive,
2) the (single life) reserve factor which is required at the start of the next step if x has died but y is still alive, and
3) the (reversionary) reserve factor which is required at the start of the next step if both x and y are still alive.

3.1.3 Vector Recurrence Relation

The derivation of the vector form of the recurrence relation is based on the observation that the recurrence relation for the reserve factor for the reversionary annuity in Equation 3.1.4 also involves the reserve factor for the single life annuity.

3.1.3.1 Contract-Based Presentation

When expressed as a vector, the pair of recurrence relations in Equations 3.1.4 and 3.1.2 become

$$\begin{pmatrix} a'_{x|y} \\ a'_y \end{pmatrix} = \begin{pmatrix} {}_fq_x\, {}_fp_y\, v^f + p_y\, q_x\, v\, a'_{y+1} + p_x\, p_y\, v\, a'_{x+1|y+1} \\ {}_fp_y\, v^f + p_y\, v\, a'_{y+1} \end{pmatrix}$$

which can be written as

$$\begin{pmatrix} a'_{x|y} \\ a'_y \end{pmatrix} = v^f \begin{pmatrix} {}_fq_x\, {}_fp_y & 0 \\ 0 & {}_fp_y \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + v \begin{pmatrix} p_x\, p_y & p_y\, q_x \\ 0 & p_y \end{pmatrix} \begin{pmatrix} a'_{x+1|y+1} \\ a'_{y+1} \end{pmatrix} \tag{3.1.5}$$

Hence, for a relatively simple contract, it is possible to find a vector expression for the recurrence relation where that vector expression brings together the relations for all contract types which may be involved in the reserve calculations for that policy type.

3.1.3.2 Survival-Based Presentation

Although the two matrices in Equation 3.1.5 are the same shape, they are populated differently: this results from the derivation being based on the type of the policy at each step. An alternative presentation is to consider the survival state of each life at each time point. Using binary indexing for each life being either alive (state 0) or dead (state 1) leads to a simple representation of all the possibilities of survival over the step: the states and labelling for two lives, currently aged x and y, are therefore

State	Binary Label	(x)	(y)
0	00	Alive	Alive
1	01	Alive	Dead
2	10	Dead	Alive
3	11	Dead	Dead

Using the ordering which results from this binary labelling, it is possible to construct a matrix of probabilities of the lives surviving for time g, which could be either f (when considering the probability of payment), or 1 (when considering the probability of requiring a reserve). For two lives, the relevant matrix is

$$\begin{pmatrix} {}_gp_x\,{}_gp_y & {}_gp_x\,{}_gq_y & {}_gq_x\,{}_gp_y & {}_gq_x\,{}_gq_y \\ 0 & {}_gp_x & 0 & {}_gq_x \\ 0 & 0 & {}_gp_y & {}_gq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.1.6}$$

The states of the lives can be considered as a time-inhomogeneous Markov chain, where matrix 3.1.6 is the transition matrix, also known as a Markov matrix or stochastic matrix [95]. Note that no transitions are possible in a step of length zero so that, when $g = 0$, the transition matrix is the identity, $\mathbf{I}$.

Using this matrix to combine Equations 3.1.4 and 3.1.2, and setting $g = f$ or $g = 1$,

leads to the following recurrence for a reversionary annuity

$$\begin{pmatrix} a'_{x|y} \\ 0 \\ a'_y \\ 0 \end{pmatrix} = v^f \begin{pmatrix} {}_fp_x\,{}_fp_y & {}_fp_x\,{}_fq_y & {}_fq_x\,{}_fp_y & {}_fq_x\,{}_fq_y \\ 0 & {}_fp_x & 0 & {}_fq_x \\ 0 & 0 & {}_fp_y & {}_fq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + v \begin{pmatrix} p_x\,p_y & p_x\,q_y & q_x\,p_y & q_x\,q_y \\ 0 & p_x & 0 & q_x \\ 0 & 0 & p_y & q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a'_{x+1|y+1} \\ 0 \\ a'_{y+1} \\ 0 \end{pmatrix} \quad (3.1.7)$$

This vector expression for the reserve factors under consideration appears more complex than the version in Equation 3.1.5: however, this expression involves only one matrix, which must be evaluated at two points in time, rather than requiring two different matrices, one for each of the time points.

Note that although binary labelling using 'zero for alive' may initially seem counter-intuitive, its use leads to a desirable property: since the number of dead people cannot decrease, this labelling naturally leads to the transition matrix being upper triangular.

3.2 The General Case

This section presents a completely general vector recurrence relation which may be used for different forms of life assurance contract: after the initial presentation, there is a formal derivation of the relation using a component-wise approach. The generic nature of the components in the recurrence relation means that this vector relation can be used for any non unit-linked policy of arbitrary complexity, so long as the cash flows and probabilities can be isolated and expressed in the form required by the vector notation presented here.

The generality of the recurrence relation means that the derivation is not entirely trivial. Therefore, the first four sub-sections establish some preliminary results which are required in the overall derivation in Section 3.2.5.2.

3.2.1 Lives

In theoretical work, it is usually assumed that all lives involved in a policy are independent. For the most general case, there could be any number of lives, so it is necessary to consider $\mathbf{x} + \mathbf{t}$, which is a vector of length m, representing a collection of m independent lives aged precisely $x_1 + t, x_2 + t, \cdots, x_m + t$: n.b. $\mathbf{t}$ is a vector, of length m, whose elements are all equal to t. This notation reinforces the fact that the ages of the lives are a function of the time t since the start of the projection. Using this construction, the ages of a given set of lives are just a function of the duration from valuation to the time under consideration.

3.2.2 Cash Flows

We are interested in a sequence of cash flows which may be made at fixed future times, according to the survival state of $\mathbf{x}$ at those future times: due to their nature, cash flows to holders of annuity policies are also known as 'payments' or 'benefits'. The nominal amount of each cash flow is fixed, but the expected amount of each cash flow depends on the probability that the payment is made and hence on the survival state, at the point of payment, of the collection of lives.

Let $t \in \mathbb{R}^+$ and let $\beta \in (0, t)$. Let $c_{\mathbf{x},t,j}$ be the nominal cash flow which happens at the fraction $f \in [0, 1]$ from time t, if $\mathbf{x}$ is in state j at that time. Then

the cash flow $c_{\mathbf{x},t,j}$ happens at time $t + f$ if the lives are in state j at that time

$\Rightarrow$ the cash flow $c_{\mathbf{x},t,j}$ happens at time $\beta + (t - \beta) + f$ if the lives are in state j at that time

$\Rightarrow$ the cash flow $c_{\mathbf{x},t,j}$ may also be denoted $c_{\mathbf{x}+\boldsymbol{\beta},t-\beta,j}$

i.e.

$$c_{\mathbf{x},t,j} = c_{\mathbf{x}+\boldsymbol{\beta},t-\beta,j} \qquad \forall\, \mathbf{x} \qquad 0 < \beta < t$$

This is the mathematical formulation of the statement that "a payment of known nominal amount, made at some time in the future, will be of the same nominal amount at that future date, irrespective of how the time in the future is determined".

Therefore, in particular, let $\beta = 1$ and let $t = s + 1$. Then

$$c_{\mathbf{x},s+1,j} = c_{\mathbf{x}+\mathbf{1},s,j} \tag{3.2.1}$$

It should be noted that the cash flows are a completely general, arbitrary function of

the state of the lives at the time the payment is made: see Section 3.4.3.1 for further discussion about assumptions relating to cash flows.

3.2.3 Discount Factors

In standard notation, if i is a constant annual rate of interest which applies over a period of t years then the discount factor which applies for that period is $v^t = (1+i)^{-t}$. There is an implicit assumption that the period starts at time 0 and ends at time t; if that is not the case then the discount factor remains $v^t = (1+i)^{-t}$ wherever the difference in time is t. Hence, when the rate of interest is constant, $v^t = v^{t_1}\, v^{t-t_1}$ for any $0 \leq t_1 \leq t$.

When the rate of interest varies with time, standard practice [59, Section 2.4] is to consider the discount factor from 0 to t as

$$v(t) = \exp\left(-\int_0^t \delta(r)\cdot dr\right)$$

where $\delta(r)$ is the *force of interest* at time r.

Let $0 \leq t \leq t_1 \leq t_2$, and let $\delta(r)$ be the force of interest at time $r \in [0, t_2]$. Let $d_{\mathbf{x},t,t_1,t_2}$ be the discount factor which applies from time $t + t_1$, and lasts for time $t_2 - t_1$; i.e. it applies from the point at which the lives are aged $\mathbf{x} + \mathbf{t} + \mathbf{t_1}$ to the point the where lives are aged $\mathbf{x} + \mathbf{t} + \mathbf{t_2}$. Then

$$\begin{aligned}
d_{\mathbf{x},t,t_1,t_2} &= \exp\left(-\int_{t_1}^{t_2} \delta(r)\cdot dr\right) \\
&= \exp\left(-\left[\int_{t_1}^{t'} \delta(r)\, dr + \int_{t'}^{t_2} \delta(r)\, dr\right]\right) \\
&= \exp\left(-\int_{t_1}^{t'} \delta(r)\, dr\right) \times \exp\left(-\int_{t'}^{t_2} \delta(r)\, dr\right)
\end{aligned}$$

i.e.

$$d_{\mathbf{x},t,t_1,t_2} = d_{\mathbf{x},t,t_1,t'} \cdot d_{\mathbf{x},t,t',t_2} \qquad \forall\, \mathbf{x} \qquad t_1 \leq t' \leq t_2$$

This is the mathematical formulation of the statement that "discounting an amount from t_2 to t_1 is the same as discounting that amount from t_2 to an intermediate time t' and then discounting that (discounted) amount from t' to t_1".

Therefore, in particular, let $t_1 = 0$, $t' = 1$ and $t_2 = s+1+f$ so that $t_2 - t_1 = s+1+f$,

$t' - t_1 = 1$ and $t_2 - t' = s + f$. Then

$$d_{\mathbf{x},t,0,s+1+f} = d_{\mathbf{x},t,0,1} \cdot d_{\mathbf{x},t,1,s+f} \tag{3.2.2}$$

Notice that these discount factors are independent of the state of the lives, a phenomenon which is consistent with reality since mortality rates are generally independent of investment returns. Notice also that the use of force of interest removes any assumption that the interest rate remains constant: see Section 3.4.3.2 for further discussion about assumptions relating to interest rates.

3.2.4 Survival Probabilities

In standard notation, ${}_tp_x$ is the probability that a life aged precisely x survives for time t: as mentioned in Section 2.1, unless specified as otherwise, time units are assumed to be years. Hence, the probability that two independent lives aged precisely x and y both survive for time t is ${}_tp_x \cdot {}_tp_y$. By extension, the probability that a collection of m independent lives all survive for time t is simply the product of the survival probabilities of each of the individual lives.

It is necessary to consider the probability of moving from any state to any other. Let $w_{\mathbf{x},t,t_1,t_2,j,i}$ be the probability of the set lives $\mathbf{x}$ being in state i at time $t+t_2$, given that it is in state j at time $t + t_1$ for $t_1 \leq t_2$ (so that the time for the possible transition from state j to state i is $t_2 - t_1$). Then, using the Partition Theorem, and conditioning on the state at time $t' \in [0, t_2 - t_1]$,

$$w_{\mathbf{x},t,t_1,t_2,j,i} = \sum_k w_{\mathbf{x},t,t_1,t',j,k} \cdot w_{\mathbf{x}+\mathbf{t}',t',t',t_2,k,i}$$

where the sum is over all possible states to which the lives could migrate. This is the mathematical formulation of the statement that "the probability that the set of lives moves from state j to state i in time $t_2 - t_1$ is the same as the probability of moving from state j to any other state at some intermediate time t' and then moving to state i in the remaining time".

Therefore, in particular, let $t_1 = 0$, $t' = 1$ and $t_2 = s+1+f$ so that $t_2 - t_1 = s+1+f$, $t' - t_1 = 1$ and $t_2 - t' = s + f$. Then

$$w_{\mathbf{x},t,0,s+1+f,j,i} = \sum_k w_{\mathbf{x},t,0,1,j,k} \cdot w_{\mathbf{x}+\mathbf{1},t,1,s+f,k,i} \tag{3.2.3}$$

Notice that there is no assumption of constant mortality: see Section 3.4.3.3 for further

discussion about assumptions relating to mortality.

3.2.5 Recurrence Relation for Reserve Factors

Using the results in the preceding sections it is possible to derive a general recurrence relation where benefits may be payable, depending on the survival state of the lives. This is a two stage process: first, a relation is derived for lives being in a particular state, and then the general relation is obtained by considering all possible states that the lives may be in.

3.2.5.1 Reserve for Lives in a Particular State

From Section 3.2.2, any cash flows which happen in a step happen at some fraction $f \in [0,1]$ through the step. Let $r_{\mathbf{x},t,j}$ be the reserve which must be held, at time t, for a set of lives $\mathbf{x}$ in state j at that time. Then, because the lives could have migrated to state i by time s, the nominal amount of the cash flow to be made in step $t+s$ to the lives in $\mathbf{x}$, if they are in state i at that time, is $c_{\mathbf{x},t+s,i}$. Therefore, the expected amount of a cash flow to be made in step $t+s$ to the lives in $\mathbf{x}$, given that they are currently in state j, and allowing for migration to any other state i, is

$$\sum_i \left[w_{\mathbf{x},t,0,s+f,j,i} \cdot c_{\mathbf{x},t+s,i}\right]$$

Allowing for discounting to that time, the present value of such a cash flow is

$$d_{\mathbf{x},t,0,s+f} \sum_i \left[w_{\mathbf{x},t,0,s+f,j,i} \cdot c_{\mathbf{x},t+s,i}\right]$$

Therefore, using an infinite sum to allow for possible cash flows in all future time steps, the reserve required at time t, given that the lives $\mathbf{x}$ are in state j at that time, is

$$r_{\mathbf{x},t,j} = \sum_{s=0}^{\infty} \left(d_{\mathbf{x},t,0,s+f} \sum_i \left[w_{\mathbf{x},t,0,s+f,j,i} \cdot c_{\mathbf{x},t+s,i}\right] \right) \tag{3.2.4}$$

for $t = 0, 1, 2, \cdots, \infty$. Note that, for the brute force approach, the 1000 simulations at each time step t will each be an instance of $r_{\mathbf{x},t,j}$.

3.2.5.2 Relation for Lives in a Particular State

Starting from Equation 3.2.4, the recurrence relation for lives in a particular state may be derived in a straightforward manner;

$$r_{\mathbf{x},t,j} = \sum_{s=0}^{\infty} \left(d_{\mathbf{x},t,0,s+f} \sum_i \left[w_{\mathbf{x},t,0,s+f,j,i} \cdot c_{\mathbf{x},t+s,i} \right] \right)$$

which, splitting off the first term in the sum, is

$$\begin{aligned} &= d_{\mathbf{x},t,0,f} \sum_i w_{\mathbf{x},t,0,f,j,i} \cdot c_{\mathbf{x},t,i} \\ &\quad + \sum_{s=1}^{\infty} \left(d_{\mathbf{x},t,0,s+f} \sum_i w_{\mathbf{x},t,0,s+f,j,i} \cdot c_{\mathbf{x},t+s,i} \right) \end{aligned}$$

and, shifting the index (so that $s = s' + 1$), this is

$$\begin{aligned} &= d_{\mathbf{x},t,0,f} \sum_i w_{\mathbf{x},t,0,f,j,i} \cdot c_{\mathbf{x},t,i} \\ &\quad + \sum_{s'=0}^{\infty} \left(d_{\mathbf{x},t,0,s'+1+f} \sum_i w_{\mathbf{x},t,0,s'+1+f,j,i} \cdot c_{\mathbf{x},t+s'+1,i} \right) \end{aligned}$$

which, using the results from Equations 3.2.1, 3.2.2, and 3.2.3, is

$$\begin{aligned} &= d_{\mathbf{x},t,0,f} \sum_i w_{\mathbf{x},t,0,f,j,i} \cdot c_{\mathbf{x},t,i} \\ &\quad + \sum_{s'=0}^{\infty} \left(\left[d_{\mathbf{x},t,0,1} \cdot d_{\mathbf{x},t,1,s'+f} \right] \sum_i \left[\sum_k w_{\mathbf{x},t,0,1,j,k} \cdot w_{\mathbf{x+1},t,1,s'+f,k,i} \right] \left[c_{\mathbf{x+1},t+s',i} \right] \right) \end{aligned}$$

and, by reordering the summations and factoring terms which are sum independent, this is

$$\begin{aligned} &= d_{\mathbf{x},t,0,f} \sum_i w_{\mathbf{x},t,0,f,j,i} \cdot c_{\mathbf{x},t,i} \\ &\quad + d_{\mathbf{x},t,0,1} \sum_k w_{\mathbf{x},t,0,1,j,k} \left[\sum_{s'=0}^{\infty} d_{\mathbf{x},t,1,s'+f} \sum_i \left(w_{\mathbf{x+1},t,1,s'+f,k,i} \cdot c_{\mathbf{x+1},t+s',i} \right) \right] \end{aligned}$$

Finally, recognising that $d_{\mathbf{x},t,1,s'+f} = d_{\mathbf{x+1},t,0,s'+f}$, and then comparing the sum in square brackets to the summation required for $r_{\mathbf{x+1},t,k}$, gives

$$r_{\mathbf{x},t,j} = d_{\mathbf{x},t,0,f} \sum_i w_{\mathbf{x},t,0,f,j,i} \cdot c_{\mathbf{x},t,i} + d_{\mathbf{x},t,0,1} \sum_k w_{\mathbf{x},t,0,1,j,k} \cdot r_{\mathbf{x+1},t,k} \tag{3.2.5}$$

where the i and k sums are over all possible states of the lives.

Various policy types could fit a particular instance of this formula just by changing the values of $c_{\mathbf{x},t,i}$ which may or may not be zero, depending on the state j. Notice that a policy which has a limited term trivially fits the use of an infinite sum by setting the cash flow amounts after the end of the policy term to zero.

3.2.5.3 Relation Considering all Possible States

Equation 3.2.5 relates to the jth possibility of a set of possible states. Combining all of the possibilities for $r_{\mathbf{x},t,j}$ into a vector, the relationship becomes

$$\mathbf{r}_{\mathbf{x},t} = d_{\mathbf{x},t,0,f}\ \mathbf{W}_{\mathbf{x},t,f}\ \mathbf{c}_{\mathbf{x},t} + d_{\mathbf{x},t,0,1}\ \mathbf{W}_{\mathbf{x},t,1}\ \mathbf{r}_{\mathbf{x+1},t}$$

where

- $\mathbf{r}_{\mathbf{x},t}$ is a column vector, of length n, where the jth entry is $r_{\mathbf{x},t,j}$ and n is the total number of states which could be occupied by $\mathbf{x}$,
- $d_{\mathbf{x},t,0,g}$ is the discount factor from time t to time $t+g$ for $g \in \{f, 1\}$,
- $\mathbf{W}_{\mathbf{x},t,g}$ is an $n \times n$ Markov transition matrix where the entries relate to the lives surviving from time t to the time $t+g$ for $g \in \{f, 1\}$; the $(i, j)^{\text{th}}$ entry of $\mathbf{W}_{\mathbf{x},t,g}$ is $w_{\mathbf{x},t,0,g,j,i}$,
- $\mathbf{c}_{\mathbf{x},t}$ is a column vector, of length n, where the ith entry is $c_{\mathbf{x},t,i}$.

Replacing the discount factors $d_{\mathbf{x},t,0,g}$ with v_t^g, which is closer to the equivalent standard notation, recognises that interest rates do not depend on the state of any lives, but explicitly allows the possibility that the underlying interest rate varies through the projection, the equation becomes

$$\boxed{\mathbf{r}_{\mathbf{x},t} = v_t^f\ \mathbf{W}_{\mathbf{x},t,f}\ \mathbf{c}_{\mathbf{x},t} + v_t\ \mathbf{W}_{\mathbf{x},t,1}\ \mathbf{r}_{\mathbf{x+1},t} \qquad \forall\, \mathbf{x} \in \mathbb{R}^{m^+} \qquad f \in [0,1]} \tag{3.2.6}$$

If there are two or more changes of state in a step, then only the overall change is of interest. For example, if a reversionary annuity changes to a single life annuity on the death of the first life, and becomes ceased on the death of the second life within the same time step, then we are only interested in the fact that it has gone from being a reversionary annuity to being ceased, and the fact that it was temporarily a single life annuity is of no consequence.

Note that Section 3.1.1 stated that, for the motivating example, the reserves calculated

for future ages are applicable at those ages so long as the basis on which they were calculated does not change. That observation is true for the full recurrence relation derived here: each element of the vector $\mathbf{r}_{\mathbf{x},t+\alpha}$ is valid for $\alpha > 0$ so long as the basis does not change in the time interval $[0, \alpha]$.

3.2.5.4 Interpretation

Using a standard definition [65, Section 4.2],

> A (prospective) reserve is the present value of all future cash flows, allowing for discounting and the probability of those cash flows being made.

The vector approach presented here is confirmation of an intuitive interpretation of that definition, i.e.

> *The reserve 'now' is the present value of 'any cash flows which may occur during the first period' together with the present value of 'any reserve which is required at the end of the first period, so long as that reserve is then required'.*

3.3 Zero Reserve States

The recurrence relation for the motivational example was derived in its natural form in Equation 3.1.5: this was expanded to a form which had a full stochastic matrix in Equation 3.1.7. This increase in complexity was necessary to show how the motivational example progressed, but it is not a necessary part of performing the calculations.

For complex policies, the full matrix $\mathbf{W}_{\mathbf{x},t,g}$ has a large number of zeroes, and the proportion of zeroes grows rapidly with the number of lives and number of states. Also, for many policies, the cash flow vector will contain a large proportion of zeroes, e.g. 75% of the entries in the cash flow vector are zero for even the relatively simple example of a joint life annuity: see Section 4.2.1.2 for further details.

A naive implementation of the recurrence relation, using the natural form of the transition matrix and cash flow vector, will therefore lead to large proportion of the operations being 'multiply by zero'. Clearly, these operations will result in zero (mathematically, if

not implementationally[1]) and so can be omitted if the sections of the relation in which they arise can be identified.

3.3.1 Overview

We call a state a 'zero reserve state' (ZRS) if it is a state for which all future cash flows are zero and there is no path to a state which has any future non-zero cash flows. There is no need to keep track of ZRSs (since they do not contribute to the liabilities) and hence calculations which relate to ZRSs are redundant and may be removed. For example, a reversionary annuity payable to y after the death of x has a zero cash flow if x is still alive, but the state where x and y are both alive is not a ZRS because there is a path to a state where there are future cash flows, i.e. the path to the state where x dies before y.

A ZRS is not necessarily a sink state for the Markov chain because it is possible to move out of one ZRS, into another ZRS: for example, a reversionary annuity has no future cash flows when y dies so that the state where y dies first is a ZRS from which it is possible to move to another ZRS (on the death of x). Allowing for ZRSs, Equation 3.2.6 may be written as

$$\overline{\mathbf{r}}_{\mathbf{x},t} = v_t^f \, \overline{\mathbf{W}}_{\mathbf{x},t,f} \, \overline{\mathbf{c}}_{\mathbf{x},t} + v_t \, \overline{\mathbf{W}}_{\mathbf{x},t,1} \, \overline{\mathbf{r}}_{\mathbf{x+1},t} \tag{3.3.1}$$

where

- $\overline{\mathbf{r}}_{\mathbf{x},t}$ is $\mathbf{r}_{\mathbf{x},t}$ with rows which correspond to ZRSs removed,
- $\overline{\mathbf{c}}_{\mathbf{x},t}$ is $\mathbf{c}_{\mathbf{x},t}$ with rows which correspond to ZRSs removed, and
- $\overline{\mathbf{W}}_{\mathbf{x},t,g}$ is $\mathbf{W}_{\mathbf{x},t,g}$ with rows and columns which correspond to ZRSs removed.

3.3.2 Example

For single life annuities, nothing further will be paid once the life has entered the dead state so that dead is a ZRS. Therefore, Equation 3.3.1 can be interpreted directly as

$$a'_x = v^f \, {}_f p_x \, c_{x,t} + v \, p_x \, a'_{x+1}$$

where the j index on the cash flow has been dropped since there is only one non-trivial cash flow. This results in the initial motivational example in Equation 3.1.2 when the

[1] Any implementational values which are not truly zero will arise as a result of rounding in the machine.

cash flow amount is set to 1, as is the case for a level annuity after the payment amount has been factored out.

3.3.3 Derivation of General Case

$\mathbf{W}$ is the stochastic matrix containing probabilities of changing state in one transition. Therefore, $\mathbf{W}$ may be used to identify ZRSs.

Let $\mathbf{B}$ be a matrix containing indicator variables (i.e. 1 or 0) showing *other* possible states to which it is possible to migrate in one transition, i.e.

$$\mathbf{B}_{ij} = \begin{cases} 0 & \text{if } w_{ij} = 0 \ \forall\, t \quad \text{or} \quad i = j \\ 1 & \text{otherwise} \end{cases}$$

Then $\mathbf{B}$ has size $n \times n$ where, as stated in Section 3.2.5.3, n is the total number of states which could be occupied by $\mathbf{x}$, and $\mathbf{B}^k$ shows the states which it is possible to reach in k transitions. Note that, since $k = 0$ corresponds to zero transitions, it is equivalent to staying in the same state and hence $\mathbf{B}^0$ is the identity matrix.

Let

$$\mathbf{U} = \sum_{k=0}^{n-1} \mathbf{B}^k$$

so that $\mathbf{U}$ is a matrix containing the number of distinct routes through the transition diagram, from any state to any other, in $n-1$ or fewer transitions: since there are n states, the longest possible path (ignoring retracing) between any two states is of length $n-1$. Note that if $\mathbf{U}_{ij} = 0$, then state j cannot be reached from state i. Note also that $\mathbf{U}$ is constant for each particular combination of i) number of states and ii) number of lives.

Finally, let $\mathbf{d}$ be a vector containing indicator variables showing states which could have cash flows in any time step, i.e.

$$d_i = \begin{cases} 1 & \text{where } c_i \neq 0 \\ 0 & \text{where } c_i = 0 \end{cases}$$

Then, under this construction,

$$\mathbf{e} = \mathbf{U}\mathbf{d}$$

is a vector containing indicators showing states which i) have cash flows and ii) can be

reached from the current state. Hence, ZRSs can be identified as those states which correspond to zero-valued elements of $\mathbf{e}$.

3.4 Implementational Considerations

The recurrence relation developed in Section 3.2 was completely general, ensuring that the relation is applicable in as many situations as possible. This section considers some practicalities, and discusses the severity of the restrictions implied by the assumptions in the derivation.

3.4.1 Algorithm Complexity

A useful indicator of the performance of an algorithm is its complexity: it is therefore necessary to consider how the complexity of the approach to calculating reserves using the recurrence relation developed in Section 3.2 differs from the complexity using the original summation approach.

3.4.1.1 Summation Approachto Calculating In-Force Reserves

In order to calculate the in-force reserve at each future time using the standard summation approach, two forward loops are required. Let T be the index of the maximum projection step, which may either be calculated from the data or set as a parameter. Then the algorithm to calculate the in-force reserves is as shown in Algorithm 3.1.

Algorithm 3.1 Calculation of in-force reserves using a summation approach.

1: **for** $t = 0, 1, \ldots, T$ **do**
2: obtain $\overline{\mathbf{c}}_{\mathbf{x},t}, v_t, v_t^f, \overline{\mathbf{W}}_{\mathbf{x},t,f}, \overline{\mathbf{W}}_{\mathbf{x},t,1}$
3: **end for**
4: **for** $t = 0, 1, \ldots, T$ **do**
5: set $\overline{\mathbf{r}}_{\mathbf{x},t} = \mathbf{0}$
6: set $v_{\text{cash flow}} = 1$
7: set $v_{\text{end step}} = 1$
8: **for** $s = t, \ldots, T$ **do**
9: set $v_{\text{cash flow}} = v_{\text{end step}} \times v_s^f$
10: set $v_{\text{end step}} = v_{\text{end step}} \times v_s$
11: increment $\overline{\mathbf{r}}_{\mathbf{x},t}$ by $v_{\text{cash flow}} \cdot \left(\prod_{r<t} \overline{\mathbf{W}}_{\mathbf{x},r,1} \right) \cdot \overline{\mathbf{W}}_{\mathbf{x},s,f} \cdot \overline{\mathbf{c}}_{\mathbf{x},s}$
12: **end for**
13: **end for**

Hence, from the loop nest over t and s, it is apparent that obtaining the sequence $\{\overline{\mathbf{r}}_{\mathbf{x},t}\}_{t=0}^{T}$ is $O(T^2)$, i.e. the computational complexity is quadratic in the number of projection steps.

3.4.1.2 Recurrence Approach to Calculating In-Force Reserves

As stated in Section 3.1.1, calculating the in-force reserve at each future time using the form of a recurrence relation, as derived in Section 3.2.5, allows the use of backward substitution. Therefore, the nest of two forward loops in the approach in Section 3.4.1.1 may be replaced by a single backward loop. Again, let T be the index of the maximum projection step. Then the algorithm to calculate the in-force reserves is as shown in Algorithm 3.2. Hence, obtaining the sequence $\{\overline{\mathbf{r}}_{\mathbf{x},t}\}_{t=0}^{T}$ is $O(T)$, i.e. the computational

Algorithm 3.2 Calculation of in-force reserves using the recurrence approach.

1: **for** $t = 0, 1, \ldots, T$ **do**
2:     obtain $\overline{\mathbf{c}}_{\mathbf{x},t}, v_t, v_t^f, \overline{\mathbf{W}}_{\mathbf{x},t,f}, \overline{\mathbf{W}}_{\mathbf{x},t,1}$
3: **end for**
4: set $\overline{\mathbf{r}}_{\mathbf{x},T+1} = \mathbf{0}$
5: **for** $t = T, \ldots, 0$ **do** {descending}
6:     set $\overline{\mathbf{r}}_{\mathbf{x},t} = v_t^f \, \overline{\mathbf{W}}_{\mathbf{x},t,f} \, \overline{\mathbf{c}}_{\mathbf{x},t} + v_t \, \overline{\mathbf{W}}_{\mathbf{x},t,1} \, \overline{\mathbf{r}}_{\mathbf{x+1},t}$
7: **end for**

complexity is linear in the number of projection steps.

3.4.2 Step Lengths

In all parts of the derivation in Section 3.2, the relationships are based on time steps of unit length: in theoretical work, it is usually assumed that the default length of a step is a year. Cases where cash flows happen more frequently need to be adjusted to allow for the frequency of payments.

Using this vector form of the recurrence there is no need for such adjustments; the sequence $\{\mathbf{c}_{\mathbf{x},t}\}$ indicates the nominal amount of each cash flow. Hence, in the case of escalating payments, for example, the stream of payments which populate the sequence $\{\mathbf{c}_{\mathbf{x},t}\}$ must already include the allowance for escalation and the timing of the cash flows. Similarly, if a projection is being performed using monthly steps and the cash flows occur yearly, then the sequence $\{\mathbf{c}_{\mathbf{x},t}\}$ would have a non-zero value for every twelfth step only.

3.4.3 Required Assumptions

The derivations in the preceding sections require only the simplest of assumptions which, for practical purposes, are not particularly restrictive.

3.4.3.1 Cash Flows

The monetary amounts of all cash flows (whether they are premiums, benefits or expenses) must be known in advance of their use. Also, more as a requirement to be able to use vector arithmetic than an assumption, there must be a one-to-one correspondence between time steps and cash flows (so that there can be a maximum of one cash flow of any particular type in each projection step). Therefore, if a policy has monthly cash flows then monthly projection steps are required: using monthly projection steps is fine for policies which have annual cash flows since 11 of any 12 consecutive steps will have a cash flow of amount zero.

3.4.3.2 Interest

The discount factor being used for a particular step must be known in advance of reaching the time step being simulated. Since the derivation in Section 3.2.3 is based on the force of interest, there is no need to assume that the interest rate is constant. There is, however, a requirement that the time interval under consideration can be split appropriately and, for all practical purposes, this should be possible. Treating inflation as 'the other side of the interest coin' requires that an equivalent assumption applies to inflation of expenses.

There is a great body of research into modelling future interest rates; see for example [13], [17] and [91]. Any interest rate model could be used to derive the sequence of discount factors $\{v_t\}$ required for the recurrence relation in this project because the relation simply requires that interest rates are derivable, and available when required for use in the calculations. Notice that $\{v_t\}$ is independent of the lives.

3.4.3.3 Mortality

The mortality table being used for a particular step must be known in advance of reaching the time step being simulated. There is no need to assume that the underlying mortality cannot change, provided that it is possible to derive a set of survival

probabilities from whatever mortality model is applicable throughout the period up to the point that the transition matrix is used.

There is a great body of work on the modelling of mortality rates; for example [11] provides a comprehensive discussion on recent models. Much of this work shows that mortality is currently improving over time, so that

> ‘the probability of a life currently aged 75 surviving for a year’

is less than

> ‘the probability of a life currently aged 65 surviving from age 75 to age 76, assuming they first survive for 10 years to reach age 75’.

While it is noted that these improvements in mortality exist, they are not of fundamental relevance to the workings of the equation derived in Section 3.2.4. Any mortality model could be used to derive the sequence of survival probabilities $\{\mathbf{W}_{\mathbf{x},t,g}\}$ required for the recurrence relation because the relation simply requires that mortality rates are derivable, and available when required for use in the calculations.

It is important to note that $\mathbf{W}_{\mathbf{x},t,g}$ depends on the lives in $\mathbf{x}$, and therefore on the properties (such as gender, age, medical history, etc.) of those lives. Hence, in principle, it would be possible to have different mortality tables for each life, allowing the mortality to be completely general. It should also be noted that the granularity of the improvements in mortality might mean that the underlying tables only need to be changed every twelfth step, effectively using annual improvements in a projection which uses monthly steps.

3.4.3.4 Complexity

From Algorithm 3.2, the complexity of the recurrence approach is $O(T)$: this is based on the second loop over t. The first t loop calculates cash flows, interest rates and mortality rates which are required in the second loop. Therefore, provided that the derivation of these items can be achieved with complexity $O(T)$ or better, the overall complexity of the algorithm remains $O(T)$.

3.4.4 Number of States

The derivation in Section 3.2.5 was based on the existence of two states; 0 for alive, and 1 for dead. However, there are several policy types where more than two states are required. Examples of the need for more than two states include life assurances, where the benefit is paid on the transition from alive to dead (rather than the continuance in the alive state), and it is possible to show that a third state, say died-in-step, is required for the derivation to be mathematically correct: see Appendix C for details.

Also, for policies such as permanent health instance, where the benefit is payable in the 'ill' state and it is possible to make many transitions from able to ill and back again, before death, it is clear that states able, ill and dead are required. Notice, however, that because some transitions are reversible, the stochastic matrix for such a policy will not be upper triangular.

The derivation in Section 3.2.4 is cast in terms of survival probabilities and hence there were only two states. However, the generality of that derivation means that any number of survival states could be considered and hence that section could be renamed 'Transition Probabilities'. Therefore, the fact that more than two states might be required is not contrary to the derivation of Equation 3.2.6 – in fact, the use of two states is a particular instance of the fact that the recurrence relation relates to any number of states.

3.5 Summary

This chapter has shown how to change the algorithm for calculating the in-force reserves from one which has complexity $O(NT^2)$ for a projection of N policies over T steps to one which has complexity $O(NT)$. For a projection using monthly steps over a term of 50 years, this change would be expected to yield a speedup of two orders of magnitude: the actual performance gain for a single-scenario case will be discussed in Section 5.2.4.

The derivation has been completely general and so will work for a wide variety of non unit-linked policies. Chapter 4 discusses the application of this recurrence algorithm to several policy types in order to demonstrate how widely it may be applied.

This chapter has also considered ZRSs, the removal of which is essential to any efficient implementation of the algorithm.

Chapter 4

Use of the Recurrence Algorithm

The derivations in Chapter 3 led to the stochastic matrix $\mathbf{W}_{\mathbf{x},t,g}$, where the indices emphasise the dependence of the transition probabilities on the age of the lives, the projection step number, and the fraction of the step where cash flows occur. Further, the matrix allowing for the removal of zero reserve states is $\overline{\mathbf{W}}_{\mathbf{x},t,g}$ to emphasise the same dependencies. In contrast to Chapter 3, which was a formal derivation, this chapter shows how the recurrence relation may be used. Therefore, throughout this chapter these stochastic matrices are denoted by $\mathbf{W}$ and $\overline{\mathbf{W}}$ for clarity: the dependence on age, projection step number, and fraction of the step are implicit.

Section 4.1 considers the simplest policies to which the recurrence derived in Chapter 3 may be applied, i.e. single life policies. Section 4.2 considers the extension to two-life policies: the extension is conceptually simple, and the implementation is only marginally more complex. Section 4.3 discusses the extension to other types of policy, both those having more lives, and those with more complicated conditions under which cash flows are made. Section 4.4 proposes the applicability of the relation to completely arbitrary policies, and introduces the framework for the implementation of such cases. Section 4.5 contains a summary.

4.1 Single Life Policies

The derivation in Section 3.2 led to a survival based consideration of policies being a more natural approach. Therefore, the simplest illustrations of the recurrence algorithm are those which relate to only one life.

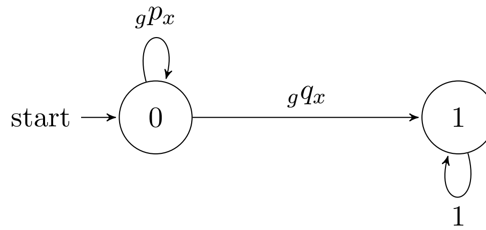


Figure 4.1 State transition diagram for one life and two states.

4.1.1 Two States

For a single life, the simplest model has only two states, alive and dead. As mentioned in Section 3.1.3.2, binary labelling is used, with 0 for alive and 1 for dead, so that the transition diagram is as shown in Figure 4.1. This leads to the stochastic matrix

$$\mathbf{W} = \begin{pmatrix} {}_g p_x & {}_g q_x \\ 0 & 1 \end{pmatrix}$$

and the matrix of indicators for possible transfers to other states becomes

$$\mathbf{B} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Therefore, since $\mathbf{B}$ is a 2×2 matrix, the upper limit of the sum is $r = \dim(\mathbf{B}) - 1 = 1$ so that

$$\mathbf{U} = \sum_{k=0}^{1} \mathbf{B}^k = \mathbf{I} + \mathbf{B} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

4.1.1.1 Single Life Annuities

Different values of f lead to annuities where the timing differs, i.e. in advance, in arrear, or part-way through the step. Varying $c_{\mathbf{x},t}$ leads to the recurrences for other differences in types of annuity;

- for a *level annuity*, $c_{\mathbf{x},t} = \theta$ where θ is constant,
- for an *increasing annuity*, $c_{\mathbf{x},t} = \phi_t$ where ϕ_t increases in arithmetic progression,
- for an *escalating annuity*, $c_{\mathbf{x},t} = \omega_t$ where ω_t increases in geometric progression,
- for a *limited term annuity*, $c_{\mathbf{x},t} = 0$ for all time steps after the end of the policy term.

For any of the four payment types, payments are only made so long as the life remains alive, i.e. so long as the life remains in state 0. Therefore, the indicator of cash flows is

$$\mathbf{d} = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$$

so that

$$\mathbf{e} = \mathbf{U}\mathbf{d} = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$$

from which it is apparent that state 0 is not a ZRS, but state 1 is a ZRS.

The general form of the recurrence relation for one life and two states is

$$\begin{pmatrix} r_{0,x} \\ r_{1,x} \end{pmatrix} = v^f \begin{pmatrix} {}_fp_x & {}_fq_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_{0,x} \\ c_{1,x} \end{pmatrix} + v \begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} r_{0,x+1} \\ r_{1,x+1} \end{pmatrix} \tag{4.1.1}$$

Hence, for single life annuities, where the payment is made if the life is in state 0, the recurrence relation, after the ZRSs are removed, becomes

$$\begin{pmatrix} r_{0,x} \end{pmatrix} = v^f \begin{pmatrix} {}_fp_x \end{pmatrix} \begin{pmatrix} c_{0,x} \end{pmatrix} + v \begin{pmatrix} p_x \end{pmatrix} \begin{pmatrix} r_{0,x+1} \end{pmatrix}$$

and, for level payments (where the value of $c_{0,x}$ is always 1), in actuarial notation, this is

$$a'_x = v^f \, {}_fp_x + v \, p_x \, a'_{x+1}$$

which agrees to the motivational example in Equation 3.1.2. Hence, for a level annuity, with payment amount 1, payable in advance (so that $f = 0$), the relation reduces to

$$\ddot{a}_x = 1 + v \, p_x \, \ddot{a}_{x+1}$$

For whole life annuities an appropriate boundary condition is ${}_tp_{120} = 0$ for $t > 0$.

Standard notation uses $a_{x:\overline{n}|}$ to denote the reserve factor for an annuity of amount 1, payable at the end of each period, to a life aged x at inception, for a maximum of n payments, so long as the life is alive at the time of the payment. Therefore, this thesis uses $a'_{x:\overline{n}|}$ to denote the reserve factor for an annuity of amount 1, payable at some fraction f through each period, to a life aged x at inception, for a maximum of n payments, so long as the life is then alive. Using this notation, an appropriate boundary condition for such a policy is $a'_{x+n:\overline{0}|} = 0$.

4.1.1.2 Endowments

A pure endowment may be viewed as an annuity where the amounts of all cash flows except one are zero, the non-zero cash flow being at the time the endowment is payable. Using this interpretation, the usual value for the cash flow vector is $\mathbf{c}_{\mathbf{x},t} = \begin{pmatrix} 0 & 0 \end{pmatrix}^T$, and the non-zero cash flow for a policy effected by a life aged x at inception with an original term of n is $\mathbf{c}_{\mathbf{x+n},0} = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$.

Standard actuarial notation uses $A_{x:\overline{n|}}^{\;\,1}$ to denote an endowment of amount of 1 payable to x, after a period of length n, if x is alive at time n. Using this notation, the required interpretation Equation 4.1.1 is

$$\begin{pmatrix} A_{x:\overline{n|}}^{\;\,1} \\ 0 \end{pmatrix} = v^f \begin{pmatrix} {}_f p_x & {}_f q_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + v \begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} A_{x+1:\overline{n-1|}}^{\;\,1} \\ 0 \end{pmatrix}$$

where $f \in [0,1]$ is arbitrary[1]. Removing the ZRS leads to

$$A_{x:\overline{n|}}^{\;\,1} = v \, p_x \, A_{x+1:\overline{n-1|}}^{\;\,1}$$

with the boundary condition $A_{x+n:\overline{0|}}^{\;\,1} = 1$.

4.1.2 Three States

It is possible to show that assurances cannot use a Markov transition matrix with only two states: see Appendix C for details. This is because the benefit is payable on the transition from one state to another, rather than the continuance in a particular state, and hence the assumption that the cash flows are computable as a function of x, t and state does not hold.

A straightforward solution is to introduce a third state, say 'died in step', from which the life transfers to the dead state in the next step with probability 1. Under this construction, ternary labelling must be used for the states, and the states under consideration are 0 for alive, 1 for 'died in step', and 2 for dead. The transition diagram for this situation is as shown in Figure 4.2, and the stochastic matrix derived from that diagram is

$$\mathbf{W} = \begin{pmatrix} {}_g p_x & {}_g q_x & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

[1] Because the cash flow is zero, it does not matter where in the step is is considered to occur.

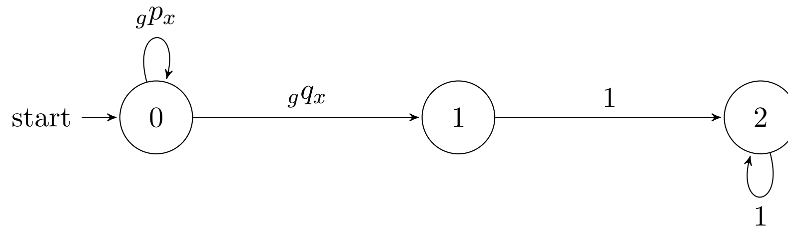


Figure 4.2 State transition diagram for one life and three states.

and the matrix of indicators for possible transfers to other states is

$$\mathbf{B} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Since $\mathbf{B}$ is a 3×3 matrix, the upper limit of the sum in $\mathbf{U}$ is 2 so that

$$\mathbf{U} = \sum_{k=0}^{2} \mathbf{B}^k = \mathbf{I} + \mathbf{B} + \mathbf{B}^2 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

4.1.2.1 Assurances (Whole Life and Term)

The benefit is payable when the life dies, i.e. when the state 'died-in-step' is entered. Therefore the vector for indication of cash flow is

$$\mathbf{d} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$$

and hence

$$\mathbf{e} = \mathbf{Ud} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

from which it is apparent that states 0 and 1 are not ZRSs, but state 2 is a ZRS.

In standard notation, A_x denotes the reserve factor for a whole life assurance, where a benefit of amount 1 is payable at the end of the step in which the death of a life currently aged x occurs. Using this notation, the recurrence relation is equivalent to

$$A_x = q_x \, v + p_x \, v \, A_{x+1}$$

and an appropriate boundary condition is $p_{120} = 0$, or $q_{120} = 1$.

In standard notation, $A^1_{x:\overline{n}|}$ denotes the reserve factor for a term assurance, issued to a life aged x at inception, with an original term of n, and the benefit of amount 1 is payable at the end of the step in which death occurs. Using this notation, the recurrence relation is equivalent to

$$A^1_{x:\overline{n}|} = q_x\, v + p_x\, v\, A^{\ 1}_{x+1:\overline{n-1}|}$$

and an appropriate boundary condition is $A^{\ 1}_{x+n:\overline{0}|} = 0$ because if the life has not died by the end of the original term then nothing is payable, so no reserve is required.

4.1.2.2 Critical Illness Policies

Under these policies, a lump sum benefit is payable if a life contracts a devastating, incurable illness, e.g. cancer or Alzheimer's. The policies may be modelled using three states; 0 for undiagnosed, 1 for diagnosed in step, and 2 for previously diagnosed. Under this construction, the transition diagram for these policies has transitions in the same directions as that in Section 4.1.2, although the descriptions of the states are different. It is therefore possible to apply identical logical derivations as in Section 4.1.2.1 and hence deduce that states 0 and 1 are not ZRSs, but state 2 is a ZRS.

4.1.3 Three States with One Reversible Transition

Consider a Permanent Health Insurance policy: depending on the policy contract, this pays out when the policyholder becomes unable to work, either through any sickness, or because of some pre-specified sickness. In either case, cash flows only happen when the policyholder is in the ill state. Using ternary labelling for the states leads to 0 for able (i.e. fit to work), 1 for ill, and 2 for dead. Let ${}_gp_x^{rs}$ be the probability of a life aged x moving from state r to state s in time g, where $r, s \in \{0, 1, 2\}$. Then the state transition diagram is as shown in Figure 4.3.

Since it is not possible for a life to leave the dead state, so that ${}_gp_x^{22} = 1$, the stochastic matrix is

$$\mathbf{W} = \begin{pmatrix} {}_gp_x^{00} & {}_gp_x^{01} & {}_gp_x^{02} \\ {}_gp_x^{10} & {}_gp_x^{11} & {}_gp_x^{12} \\ 0 & 0 & 1 \end{pmatrix}$$

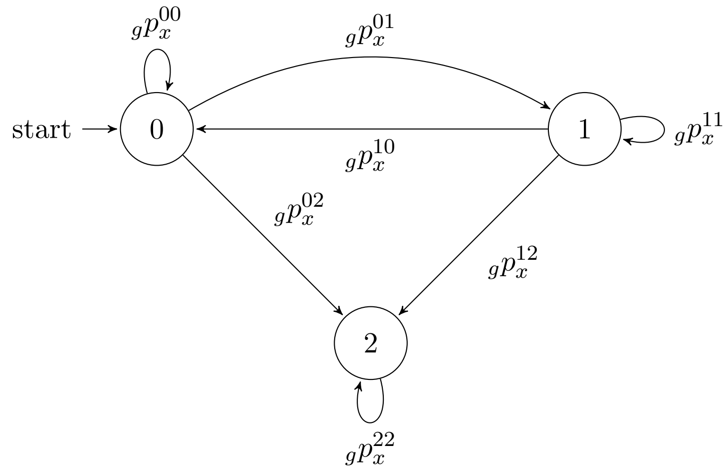


Figure 4.3 State transition diagram for one life and three states, where one transition is reversible.

and the matrix of indicators for possible transfers to other states becomes

$$\mathbf{B} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Since $\mathbf{B}$ is a 3×3 matrix, the upper limit of the sum for $\mathbf{U}$ is 2 so that

$$\mathbf{U} = \sum_{k=0}^{2} \mathbf{B}^k = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

4.1.3.1 Permanent Health Insurance

Since payments are made while the policyholder is in the ill state, the vector of indicators for a cash flow happening is

$$\mathbf{d} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$$

so that

$$\mathbf{e} = \mathbf{Ud} = \begin{pmatrix} 1 & 2 & 0 \end{pmatrix}^T$$

from which it is apparent that 'dead' is the only ZRS. This agrees to the fact that a payment could be made on transition into the ill state from either the able state or the ill state: n.b. the transition ill-to-ill is, effectively, continuing in the ill state.

From this example, it is clear that the elements of $\mathbf{e}$ may be interpreted as the number of transitions required to enter the benefit-paying state from the current state: the first element indicates that only one transition is need to reach the ill state from the able state; the second element indicates that two transitions are need to reach the ill state from the ill state, i.e. ill $\rightarrow$ able $\rightarrow$ ill.

4.1.3.2 Income Protection

These policies have their benefits payable when the policyholder becomes out of work. Using ternary labelling, the states are 0 for employed, 1 for unemployed, and 2 for 'withdrawn from workforce' (i.e. retired or dead). Under this construction, these policies may be considered using a three-state model with one reversible transition and hence follow the same derivation as Section 4.1.3.1, leading to retired and dead being ZRSs while employed and unemployed are not ZRSs.

4.2 Two-Life Policies

In order to demonstrate the applicability of the algorithm to a wider collection of situations it is necessary to consider the next level of complexity, i.e. policies with two lives.

4.2.1 Two States

Representing the lives as x and y rather than x_1 and x_2, the states to which it is possible to migrate can be tabulated as

State	Binary	x	y	Possible Future States
0	00	alive	alive	0 1 2 3
1	01	alive	dead	1 3
2	10	dead	alive	2 3
3	11	dead	dead	3

and the corresponding transition diagram is as shown in Figure 4.4.

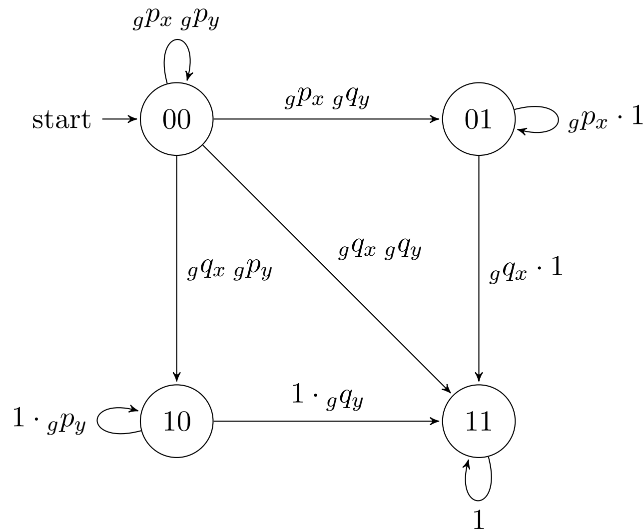


Figure 4.4 State transition diagram for two lives and two states.

The stochastic matrix is

$$\mathbf{W} = \begin{pmatrix} {}_gp_x\ {}_gp_y & {}_gp_x\ {}_gq_y & {}_gq_x\ {}_gp_y & {}_gq_x\ {}_gq_y \\ 0 & {}_gp_x & 0 & {}_gq_x \\ 0 & 0 & {}_gp_y & {}_gq_y \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and the matrix of indicators for possible transfers to other states is

$$\mathbf{B} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and, since $\mathbf{B}$ is 4×4, the upper limit of the sum of $\mathbf{U}$ is 3, so that

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & 1 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4.2.1.1 Reversionary Annuities

Payments are made to the second life, y, after the death of the first, x, so that the payments are made in state 2 and the vector of indicators for a cash flow being made

is therefore

$$\mathbf{d} = \begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}^T$$

so that

$$\mathbf{e} = \mathbf{Ud} = \begin{pmatrix}1 & 0 & 1 & 0\end{pmatrix}^T$$

from which it is apparent that payments can only be made from states which can be reached from state 0 or state 2. Therefore states 0 and 2 are not ZRSs, but states 1 and 3 are ZRSs.

In actuarial notation, the vector for reserve factors $\mathbf{r}_{\mathbf{x},t}$ can be interpreted as $\begin{pmatrix}a'_{x|y} & 0 & a'_y & 0\end{pmatrix}^T$

4.2.1.2 Joint Life and Last Survivor Annuities

Payments are made on a joint life annuity so long as both lives are alive, whereas payments on a last survivor annuity made so long as at least one of the lives survives. Therefore, for a joint life annuity

$$\mathbf{d} = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix}^T$$
$$\Rightarrow \mathbf{e} = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix}^T$$

and for a last survivor annuity

$$\mathbf{d} = \begin{pmatrix}1 & 1 & 1 & 0\end{pmatrix}^T$$
$$\Rightarrow \mathbf{e} = \begin{pmatrix}3 & 1 & 1 & 0\end{pmatrix}^T$$

Hence, for a joint life annuity states 1, 2 and 3 are ZRSs and for a last survivor annuity state 3 is the only ZRS.

4.2.2 Three States

For the construction where the three states are alive, died-in-step and dead, any of the three survival states could be occupied by either of the two lives. Therefore, the relevant transition diagram, using ternary labelling, is as shown in Figure 4.5.

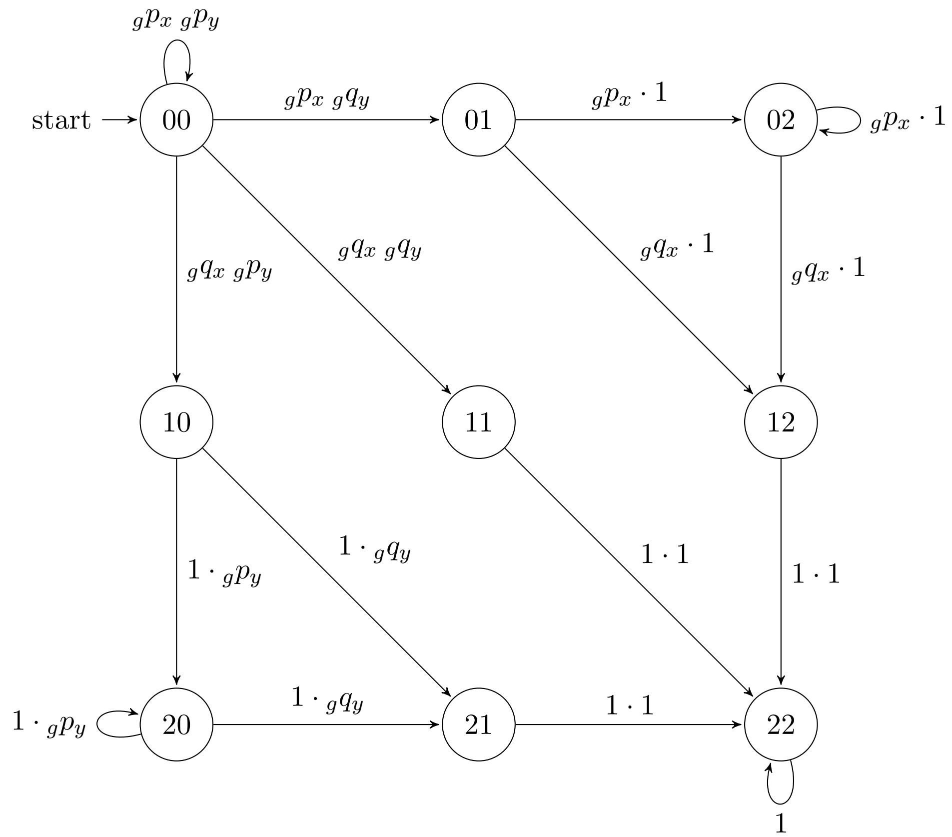


Figure 4.5 State transition diagram for two lives and three states.

4.2.2.1 Assurances

There are 5 states, transition into which might lead to payment, i.e. 01, 10, 11, 12, and 21. Whether payment is made on the first or second death will be detailed in the policy document, making it necessary to consider the two cases separately: the first death happens on transition into states 01, 10 and 11, i.e. both lives are not still alive, and the second death happens on transition into states 11, 12, and 21. However, transition into states 12 and 21 can only happen from states which indicate that one of the lives is already dead, i.e. those transitions happen from a state which is, effectively, a single life policy anyway.

The transition matrix is

$$\mathbf{W} = \begin{pmatrix} {}_gp_x & {}_gq_x & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} {}_gp_y & {}_gq_y & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{where } \otimes \text{ is the Kronecker product}$$

$$= \begin{pmatrix} {}_gp_x\,{}_gp_y & {}_gp_x\,{}_gq_y & 0 & {}_gq_x\,{}_gp_y & {}_gq_x\,{}_gq_y & 0 & 0 & 0 & 0 \\ 0 & 0 & {}_gp_x & 0 & 0 & {}_gq_x & 0 & 0 & 0 \\ 0 & 0 & {}_gp_x & 0 & 0 & {}_gq_x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_gp_y & {}_gq_y & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_gp_y & {}_gq_y & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

from which the matrix of indicators for possible transitions to other states is

$$\mathbf{B} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and hence

$$\mathbf{U} = \sum_{k=0}^{8} \mathbf{B}^k = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 2 & 1 & 2 & 5 \\ 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Considering the two cases:

a) when the benefit is paid on the first death, the vector of indicators for a cash flow being made is

$$\mathbf{d} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

so that

$$\mathbf{e} = \mathbf{Ud} = \begin{pmatrix} 3 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

and hence all states other than 00, 01, 10 and 11 are ZRSs.

b) when the benefit is paid on the second death, the vector of indicators for a cash flow being made is

$$\mathbf{d} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}^T$$

so that

$$\mathbf{e} = \mathbf{Ud} = \begin{pmatrix} 5 & 2 & 1 & 2 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}^T$$

and hence only state 22, where both lives have already died, is a ZRS.

4.3 Extension to Other Policies

The derivation in Chapter 3 was completely general, suggesting that the algorithm could be used for a vast array of policy types. So far, this chapter has only considered those policies which are relatively common. Therefore, this section gives an overview of considerations required to apply the recurrence algorithm to a far wider selection of policies.

4.3.1 More than Two Lives

Although cases which involve more than two lives are uncommon, they are included here for completeness.

The two-life models in Section 4.2 are straightforward extensions of the single life models in Section 4.1. For larger numbers of lives, the transition matrices may be obtained by induction, using tensor products. Let $\mathbf{W}^{(m)}$ be the transition matrix required for m lives, and let $\otimes$ be the Kronecker product. Then

a) for the two state model (used for annuities and endowments),

$$\mathbf{W}^{(m)} = \mathbf{W}^{(m-1)} \otimes \begin{pmatrix} {}_g p_{x_m} & {}_g q_{x_m} \\ 0 & 1 \end{pmatrix}$$

with $\mathbf{W}^{(0)} = (1)$. There are 2^m possible states so that $\mathbf{W}^{(m)}$ has 4^m elements, of which 3^m are non-zero, and

b) for the three state model (used for assurances),

$$\mathbf{W}^{(m)} = \mathbf{W}^{(m-1)} \otimes \begin{pmatrix} {}_g p_{x_m} & {}_g q_{x_m} & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

with $\mathbf{W}^{(0)} = (1)$. There are 3^m possible states so that $\mathbf{W}^{(m)}$ has 9^m elements, only 4^m of which are non-zero.

4.3.2 With-Profit Policies

The recurrence algorithm from Chapter 3 can be applied to with-profit policies so long as the cash flows can be determined in advance of their use so that the assumption in Section 3.4.3.1 holds. This should not pose any practical problems because the cash flows can be calculated in the forward loop in Algorithm 3.2 and then used in the backward loop. Even if bonuses are applied on a two-tier basis, this algorithm can be used if estimated bonus rates are available in advance of calculating the bonuses attaching at a particular time.

4.4 The General Case

A good implementation of the process for calculating reserves using the relation derived in Chapter 3 would generate sufficient code that any policy can be processed by the addition of minimal further code. This is a straightforward procedure: from the number of states and number of lives, determine the transition diagram and hence the ZRSs, as outlined in Section 3.3.3; from a separate process, which must be identified for each new policy type, generate the cash flows; from the transition diagram and the cash flows, calculate the reserves.

4.4.1 Theory

Reserves must be calculated using overall matrix formulation, but once ZRSs have been identified it is possible to omit relevant columns and rows from the matrix multiplication. For example, for a single life whole life assurance, which is a three state case,

$$\mathbf{W} = \begin{pmatrix} {}_g p_x & {}_g q_x & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

so that, omitting the entire rows and columns which result from the identification of ZRSs,

$$\overline{\mathbf{W}} = \begin{pmatrix} {}_g p_x & {}_g q_x \\ 0 & 0 \end{pmatrix}$$

Hence, the required multiplication with the reserve vector is

$$\overline{\mathbf{W}}\,\overline{\mathbf{r}} = \begin{pmatrix} {}_g p_x \cdot r_0 + {}_g q_x \cdot r_1 \\ 0 \cdot r_0 + 0 \cdot r_1 \end{pmatrix}$$

Therefore, by omitting other cases where $\overline{\mathbf{W}}_{ij} = 0$, irrespective of whether or not they are ZRSs, it is possible to circumvent multiplication by zeroes.

4.4.2 Implementation

A naive implementation of the algorithm would be

```
for p = 1 to number_of_policies
```

```
for s  = 1 to number_of_future_time_steps
   for r = 1 to number_of_rows_in_W
      for c = 1 to number_of_columns_in_W
         reserve(r,c,p) =
```

However, using the standard loop optimisation technique of introducing blocks within the loop over policies, the loop nest becomes

```
for b = 1 to number_of_blocks
   for p = 1 in number_of_policies_in_block
      for s  = 1 to number_of_future_time_steps
         for r = 1 to number_of_rows_in_W
            for c = 1 to number_of_columns_in_W
               reserve(r,c,p) =
```

It is then possible to perform permutation of the loops so that the loop over policies is innermost and hence in the ideal position for vectorisation to be applied. The loop nest for this structure is

```
for b = 1 to number_of_blocks
   for s  = 1 to number_of_future_time_steps
      for r = 1 to number_of_rows_in_W
         for c = 1 to number_of_columns_in_W
            for p = 1 in number_of_policies_in_block
               reserve(r,c,p) =
```

4.4.3 Masked Implementation

To reduce the potentially excessive number of multiply-by-zero operations, it may be sensible to create a mask which contains Boolean indicators which are zero where either a row or column leads to a ZRS, or a row or column has zero probability of a cash flow in this step: the mask would be populated as

```
for i=1:num_rows
   for j=1:num_cols
      mask[i,j] = 1
for i=1:num_rows
```

```
if ( row(i)  IMPLIES  ZRS )
{
   mask[i,:] = 0
   mask[:,i] = 0
}
```

It is then possible to insert a test against the mask within the permuted loop nest, viz;

```
for b = 1 to number_of_blocks
   for s  = 1 to number_of_future_time_steps
      for r = 1 to number_of_rows_in_W
         for c = 1 to number_of_columns_in_W
            if ( mask[r,c] ) then
               for p = 1 in number_of_policies_in_block
                  reserve(r,c,p) =
            endif
```

Note that the performance is not particularly adversely affected by the existence of the if-test because that test is performed outside the loop over policies which, under this permutation, is innermost. Note also that, since the policies in the innermost loop are independent, it should be possible for that loop to be vectorised.

4.5 Summary

This chapter has shown how the theory developed in Chapter 3 may be applied to a wide variety of contracts, and how that application produces relations which may be used as the basis for an efficient implementation for the calculation of each contract's reserves. It has also shown how these relations, derived in a matrix context, relate to formulae which are recognisable in the actuarial landscape.

This chapter has also intimated that, since several contracts may use the same state transition diagram, and hence the same stochastic matrix, efficient code can be produced for specific cases so that only a few changes, which relate to cash flows, need to be made for each particular contract. An even more general implementation is discussed in Section 9.2.

Chapter 5

Improving the Performance of Profitability Calculations

Section 2.3.2 introduced several cases where programs produced by a valuation system are currently run, with only a single set of parameters being needed. The fact that only one set of parameters is required means that the required calculations can be performed, albeit in a long time. This chapter considers optimisations which were performed on a code which performs a typical single scenario calculation and the effects of those optimisations on the run time.

Section 5.1 describes the purpose and implementation of the initial code: it describes the policy data and parameters used in the project. Regression testing and variability of timings are also considered. Section 5.2 considers the optimisations applied to the code: initially optimisations from a previous project are described since they form the basis of the work in this project and then the optimisations performed in this project, and their effect, are presented. Section 5.3 contains a discussion of the performance implications of the optimisations performed, and Section 5.4 contains a summary of the chapter.

5.1 Initial Code

In order to present the benefits of the optimisations applied to the code during this project, it is necessary to explain what the initial code for the investigation was, and how it compared to code used in real situations.

5.1.1 Initial Code

At the time this investigation was first mooted, the package used within the commercial environment which spawned the investigation transformed the relationships within the package into Fortran77 source, and compiled it using Compaq Visual Fortran V6.6 (CVF). This led to two major problems when producing a piece of code to replicate that produced within the commercial environment; firstly, because CVF is no longer available, a licensed copy could not be obtained to use outside the office; secondly, and more fundamentally, the code produced within that environment had intellectual rights attached to it, so its use for anything other than its original intent, i.e. running the business, is prohibited. The combination of these two factors meant that the starting point was some Fortran90 code which ran on a laptop after having been compiled using V10.1 of Intel's Fortran compiler.

The code used as the starting point for this project was developed during an MSc in HPC [86]. The original code was a mirror of the code from the commercial environment in two respects: firstly, for the same combination of initial parameters and data, it produced the same financial results as the system used within the office; secondly, it originally ran at similar speed to code from commercial software. The similarity in performance was a direct result of its implementation being similar to code produced by a commercial package. At the end of the MSc project, the code had been through several stages of serial optimisation and had been parallelised using OpenMP, and so had several desirable characteristics: it produced the correct financial results, it ran significantly more quickly than the original code, and scaled reasonably well over small numbers of cores.

Originally, only code to estimate profitability within single life annuities was developed. This limited code was sufficient to show that the principles work, i.e. optimisation of *ab-initio* code produces significantly faster code than that resulting from commercial packages. The assumption that the principles are transferable to other policy types was validated, as an early stage of this project, by the production of code to estimate profitability of reversionary annuities.

5.1.2 Data

A standard interpretation of the UK's Data Protection Act is that *'data may only be used for the purpose for which it was collected'.* This, allied to client confidentiality, precludes the use of real data from the commercial environment and so this investigation

uses data created specifically for this purpose: having been artificially created, this data will be referred to as 'synthetic data'.

In order that the performance of the program investigated is representative of the performance of the valuation system, our synthetic data look like they could be drawn from a population of policyholders with similar characteristics as those in the business. Because this investigation considers annuity policies, payable to pensioners, our data was created so that it represents a similar group of pensioners, even if they are fictitious. This data was created so that all attributes are random, and the main characteristics may be summarised as follows:

1) the date of birth is such that the age at valuation is uniformly distributed between 57 and 67 so that the synthetic data represents a cohort of people who have retired recently, either at normal retirement age, or slightly early;
2) the policy inception date is uniformly distributed over the calendar year prior to the valuation date so that these data represent a cohort of recent business;
3) roughly 73% of the policyholders are male;
4) roughly 81% of the policies have payments made monthly, the remainder have payments made annually;
5) if s is defined as the *modal payment*, i.e. the amount paid at each payment, then s has a log-normal distribution with a mean of roughly 5.0 and standard deviation of about 1.5: more precisely,

$$\ln s \sim N\left(5.01, 1.477^2\right)$$

and
6) the rate of escalation, i.e. the annual increase in the amount paid to the policyholder, is approximately distributed as

escalation rate	0%	3%	4.25%	5%
proportion of policies	95.2%	3.5%	0.8%	0.5%

Standard statistical simulation techniques [78, Section 4.1] were used to to create the data. Using the probability density function for a required distribution, it is possible to determine the cumulative distribution function and hence the probability that a variable from that distribution is less than a specified value. Generating a random number in the range from zero to one produces the required probability, corresponding to the value on the cumulative distribution function, from which the relevant value from the distribution can be determined.

This method clearly relies on the availability of a random number generator: this project used the Fortran `rand()` function. The implementation of `rand()` provided by the Intel Fortran compiler uses a combination of two linear congruential generators [43]. However, the sequence produced by Intel's `rand()` suffices for the production of synthetic data which look like they have similar properties to a real cohort of policyholders.

5.1.3 Parameters

The profitability calculations in a commercial environment use two bases; a *reserving* basis which is used to calculate the reserves in each future time, and a *realistic* basis which is used to project the reserves forward from each time point to the next, allowing for interest earnt on the reserve between the time points. These bases are usually simplistic: there is one interest rate and one inflation rate, both of which are used for the duration of the projection, and there is usually no allowance for improvements in mortality.

In order to reproduce the financial results, these same simplistic bases are used throughout the development and optimisation of the investigation into the single scenario calculations. Whilst these constant parameters are unrealistic, their use has the advantage of truly emulating the commercial environment, thereby allowing valid performance comparisons.

5.1.4 Regression Testing

A naive statement of the aim of a code optimisation project is 'to make the program run more quickly'. However, it is possible to make any program run arbitrarily fast by simply removing large parts of the code, thereby allowing the program to output rubbish. Therefore, a better statement of the aim of an optimisation project is 'to make the program run faster while producing the same results'.

The test data described in Section 5.1.2 were run through the live system in a commercial environment and the results from that run have been compared to the output from running that same test data through the initial version of the new F90 code produced for the investigation. That comparison showed the output for each policy to be the same to 6 significant figures. Within the commercial environment which inspired this investigation any difference in results on different platforms is considered to be immaterial if it is less than 1%. Therefore, an actual relative difference of less

than 10^{-6} provides evidence that the initial program for this investigation does give the same results as the real program. This regression test was performed at the end of each stage of the investigation. The level of agreement maintained throughout the course of optimisation was 6 significant figures. Overall, the result produced by the final code is not materially different to the original results. The final code could therefore be used to replace the code in the commercial environment.

5.1.5 Variability of Timings

As mentioned in Section 1.4, the time to process each policy in a commercial environment is roughly 1 second for single life policies and about 2.5 seconds for reversionary annuities. This chapter reports performance in terms of 'policies per second' and, as changes to the code are made, the processing rate improves dramatically. The difference in processing rates indicates that any variability of run times for a particular optimisation is not significant in relation to the overall time, and so there is no benefit in obtaining confidence intervals for the run times reported in this chapter. Therefore, each run has been performed only once and, whilst variability in times is acknowledged to exist, it is stressed that the times reported are representative, rather than being a mean, mode or median, etc. of the overall time.

5.2 Optimisations

The optimisation of any code is an iterative process. To reflect this, the optimisations applied to the Fortran code are presented separately, allowing the relative effect of each stage to be discussed individually.

5.2.1 Initial Optimisation

As stated in Section 5.1.1, the idea that commercially available valuation packages do not create optimal code was originally investigated for an MSc dissertation [86]. The results of that investigation showed that there is scope for improving performance of code which initially has similar performance to that produced by commercial software. It is not possible to fully appreciate the performance of code produced by valuation packages in comparison to the performance of code developed in this project without considering the initial improvements made during the MSc. Therefore, for completeness, the optimisations from the MSc are discussed here briefly.

The MSc project performed most of the optimisations which would be expected for a scientific code. However, due to the short duration of that project, the improvements made may not have been absolutely optimal.

In the first stage of the MSc, a collection of compiler flags which improved performance was applied: see [86, Section 2]. In order to match the performance of the valuation package it was necessary to disable optimisation and enable debugging code; although the use of unoptimised code is appealing from a correctness viewpoint, it is generally accepted that moderate levels of optimisation will not lead to transformations which dramatically change the value of the results.

In the second stage of the MSc, two dominant routines were manually optimised. Following the selection of the collection of compiler flags, a profile of the program's execution showed that the majority of the execution time was spent in only two routines making them obvious targets for re-factoring. Therefore, i) the routine for the interpolation within a life table to obtain l_x at non-integral ages was optimised in [86, Section 3], and ii) [86, Section 4] optimised the routine for the calculation of the reserve using the summation approach (which was a naive implementation of Equation 2.1.5).

The profile obtained after these optimisations showed that more than 50% of the execution time was spent in routines which had already been optimised, indicating that more performance could only be extracted from the use of other techniques. Therefore, in [86, Section 6], the code was parallelised by the addition of OpenMP which used a single thread to read the data into a shared array, processed all the data using a team of threads, stored the results back to another shared array, and finally wrote the results to disk using a single thread.

As mentioned in Section 5.1.1, the original code was written to mirror that produced by the valuation package: it therefore had global variables, and was written in a single large routine. OpenMP uses the concept of *shared* and *private* variables in order to partition what separate threads may access: to simplify the process of determining the scope of variables, it is useful to have as much of the program split into subroutines as possible. Therefore, as part of the addition of OpenMP, the code was re-engineered in a manner similar to the object oriented philosophy: subroutines were placed within modules relevant to a particular concept, and variables were made local to each routine.

The largest machine available to the MSc project was a 16-core SMP which had the Portland compiler, rather than the Intel compiler used on the laptop: the re-engineering, and the change of hardware and compiler, led to a speedup of 1.5×. Also,

Optimisation	Speedup	Rate
None	N/A	1.0
Increase level of compiler optimisation	3.73×	3.7
Manually optimise interpolation routine	1.68×	6.3
Manually optimise reserving calculation	2.69×	17
Change hardware and re-engineer (to simplify OpenMP)	1.56×	26
OpenMP parallelisation (16 threads on 16 cores)	15.6×	410

Table 5.1 Performance rate (policies per second) for profitability calculations for single life annuities resulting from optimisations in MSc project: the speedup relates to the individual stage, but the rate is cumulative.

the scalability of the OpenMP code was investigated in the MSc [86, Section 7], which showed that it was 96% efficient when running 16 threads on 16 cores: a reduction in efficiency at this relatively low number of threads suggests scope for extracting more performance.

5.2.2 Further Optimisation

The changes discussed in Section 5.2.1 were performed as part of the MSc and the resulting performance improvements are summarised in Table 5.1. The changes discussed below were performed as part of this investigation.

5.2.2.1 Power Calculations

The optimisation of the reserving calculation in the MSc confirmed a generally accepted truth: on modern hardware, a single multiplication operation is faster than a single power calculation. Therefore, if a sequence of values are known to be in geometric progression then, for sufficiently long sequences, obtaining the values by repeated multiplication is to be preferred to obtaining the values by calculating powers. Hence, power calculations were replaced wherever possible: this improved overall performance by a factor of 2.6×.

5.2.2.2 OpenMP Synchronisation

As a result of the short nature of the MSc project, the OpenMP parallelisation implemented there used a critical region in order to accumulate values across policies: a standard OpenMP reduction could not be used because the data structures involved were not intrinsic types. The replacement of this region with bespoke reduction code

reduced the time the threads were waiting to access the accumulation variables: this improved performance for the entire program by a factor of 1.2×.

5.2.3 Scaling of OpenMP Code

The OpenMP parallelisation of the original Fortran code, as developed during the MSc, was not particularly efficient past about 12 threads; when running 40 threads on 48 cores, the efficiency was only about 60%.

The cause of the inefficiency was identified as the locking of variables used as internal files on each thread. These internal files were used to collect the values of several variables, ready to be output as strings to files on the disk. The variables were declared within subroutines which were called by each thread and therefore each thread should have had its own private copy. However, even though those variables were declared as `private` within the OpenMP code, they were being treated by the compiler as 'normal' files and hence being locked during I/O.

The solution was to keep the individual variables separately, and write them to the external file using Fortran's I/O formatting functionality. Having resolved the issue, the code scaled to almost 95% efficiency when using 48 threads on 48 cores of the Opteron cluster.

5.2.4 Change of Algorithm

Chapter 3 discussed a recurrence algorithm which has computational complexity $O(T)$ for a projection over T time steps. For a projection over 600 monthly steps this implies a speedup of two orders of magnitude compared to the summation approach used in the commercial valuation package.

5.2.4.1 Processing Rate

In order to investigate the speedup, the input data were changed: the date of birth, and policy inception date were set to be identical for every policy, but the amount and frequency of the payments were unaltered. The changes to the dates ensure that each policy has the same outstanding term for a particular valuation date, while having the financial data differing across policies ensures that the calculations do need to be performed, rather than just recalling previously calculated values. Using this approach,

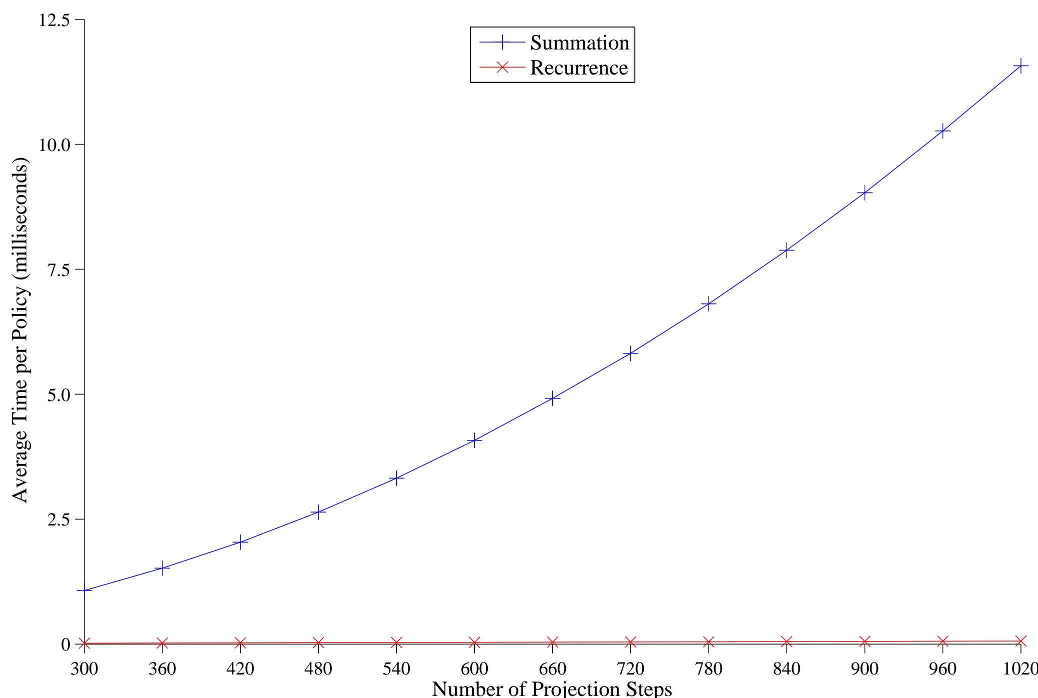


Figure 5.1 Time (in milliseconds) to calculate reserves for single life annuities using naive summation and recurrence algorithm: times were obtained using a single core of the Cray, and are the average over 5000 policies.

the number of steps which are processed is controlled by changing the valuation date.

Figure 5.1 shows the average time to process single-life annuity policies using the summation approach and the recurrence relation. The times measured are solely for the calculation of reserves: all inputs to the algorithm (such as l_x's and payment amounts) are calculated prior to the recurrence being used and so are not included in the measurement[1]. The implementation of each of the algorithms used the same level of compiler optimisation, so that the only difference between the timings was the algorithm. On the scale presented, which demonstrates the quadratic nature of the summation approach, the times for the recurrence approach are almost indistinguishable from the axis. This indicates that as well as having better scaling than the summation approach, the recurrence approach is far faster than the summation approach.

Figure 5.2 indicates that the speedup resulting from changing the algorithm is a linear function of the number of steps: this is consistent with changing the complexity from $O(T^2)$ to $O(T)$ where T is the number of time steps. The speedup for T steps is not T because the process of obtaining the reserve using a summation approach contains some calculations which are $O(T)$: these are unaffected by the change of algorithm and

[1]This also means that there is no I/O in the timings.

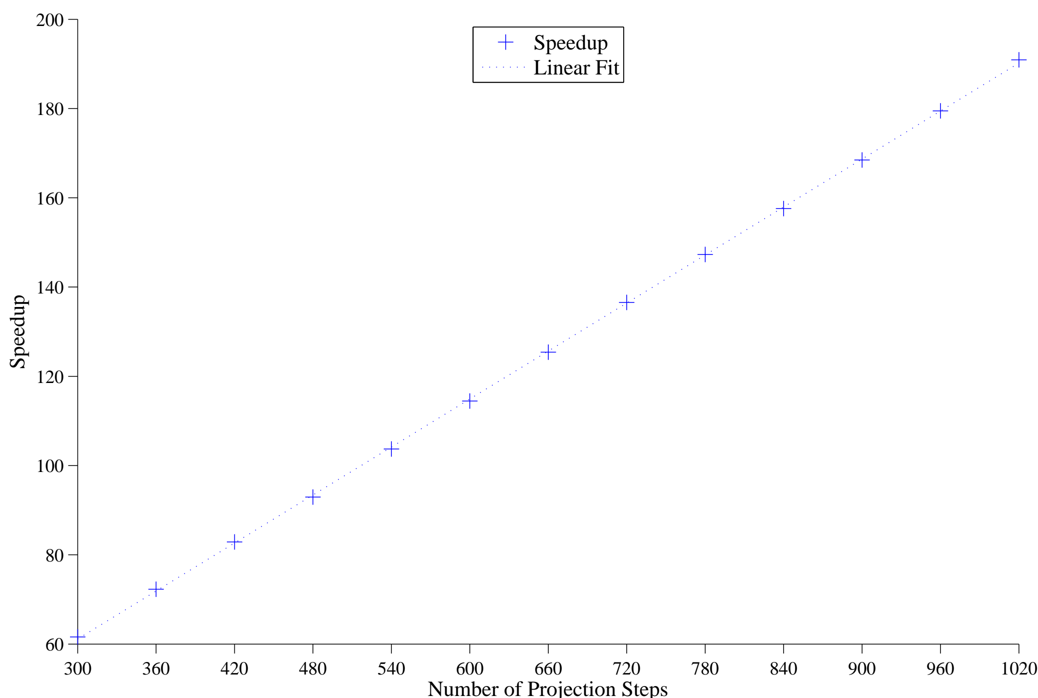


Figure 5.2 Speedup (for Single Life Annuities) resulting from change from naive summation to recurrence relation: times were obtained using a single core of the Cray, and are the average over 5000 policies.

hence do not contribute to the speedup resulting from changing the algorithm's overall complexity: see Appendix D for further details.

Figure 5.3 shows the average time to process single-life and reversionary annuity policies using the recurrence approach to calculating reserves. The linear scaling of the recurrence approach, with increasing number of steps, for both types of annuity, is clear: linear scaling agrees with the complexities derived in Section 3.4.1. The fact that the slope of the line for the reversionary annuity policies is roughly twice the slope for the single-life annuity policies is a result of the fact that, for a reversionary annuity, there are twice as many lives for which the probability of survival needs to be obtained and, per Equation 3.1.4, two reserve factors are required at each step.

5.2.4.2 Verifying Correctness

Section 5.1.4 noted that regression testing was performed as part of each optimisation phase: for the optimisations discussed in Sections 5.2.1, 5.2.2 and 5.2.3, the output for each policy was verified to be the same to 6 significant decimal figures. However, because a new algorithm was introduced here, more stringent testing was performed to

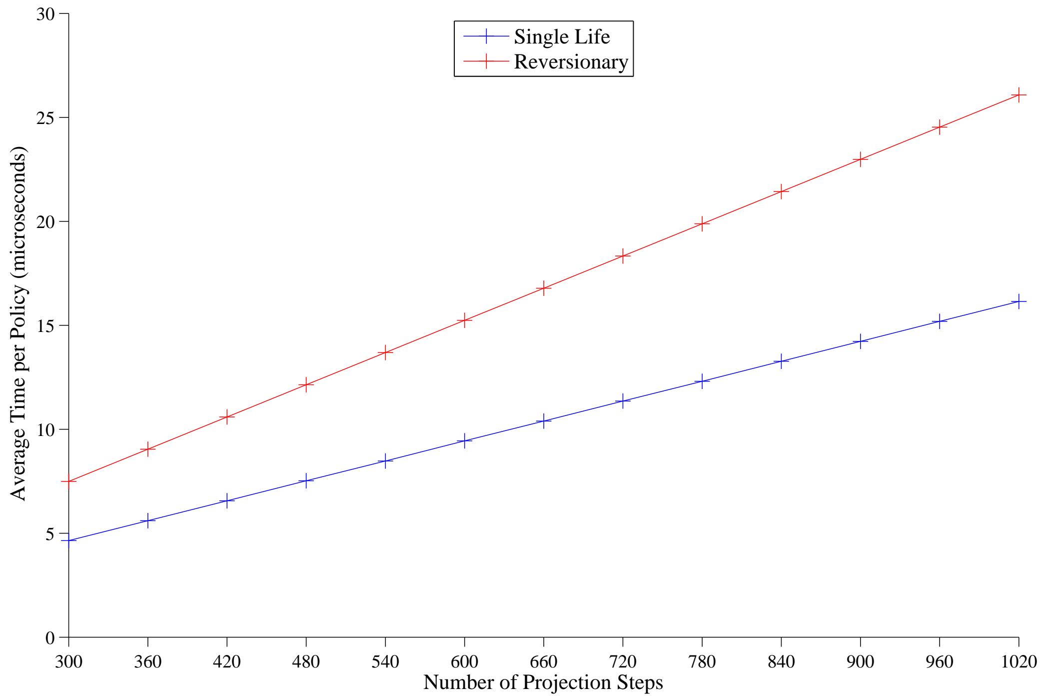


Figure 5.3 Processing times (in microseconds) using recurrence relation for different annuity contracts: times were obtained using a single core of the Cray, and are the average over 100,000 policies.

demonstrate the correctness of the new algorithm.

Ideally, the results produced using the new recurrence algorithm should be the same (to the level of machine precision) as the results produced using the old summation algorithm. This may be verified by producing results from both approaches in binary format, rather than ASCII format, and comparing them using readily available, standard tools. However, there are known failings with computer arithmetic: a common example is summing a collection of real-valued variables of varying magnitudes in different orders producing different answers. These failings become apparent in this project: examples include the recurrence approach's use of repeated multiplication compared to the summation approach's use of the power function, and the accumulation of errors from the recurrence's use of the reserve at the next time step in the calculation of the current reserve.

Therefore, to allow for the fact that it is impossible to verify exact numerical equality of the results from each approach, the output for each policy was verified to be the same to 11 significant decimal figures, rather than just 6. Taken with the mathematical equivalence of the approaches, as derived in Section 3.2, the numerical equivalence at this level of accuracy provides evidence that the differences in reserves are immaterial by

Optimisation	Speedup	Rate
Serial optimisations in MSc (see Table 5.1)		17
Remove calls to power function	2.60×	44
Implement recurrence algorithm	105.3×	4,600
Change to multi-core platform	1.87×	8,600
OpenMP parallelisation (48 threads on 48 cores)	46.6×	390,000

Table 5.2 Performance rate (policies per second) as optimisation of profitability code progressed.

any practical definition of materiality, e.g. the definitions discussed in Section 6.7.2.

5.3 Performance Implications

Table 5.2 summarises the increases in processing rate discussed in the previous sections. The overall improvement of five orders of magnitude is significant: it allows runs which would have previously taken many hours to perform to be completed in a few seconds, potentially allowing a paradigm shift in the way that life offices work.

Figure 5.4 shows the close to linear scaling of the OpenMP parallelisation on the Opteron cluster. The code scales well up to 24 threads on 24 cores and scales reasonably well to 48 threads on 48 cores, being roughly 94.5% efficient at that point.

5.4 Summary

Figures 5.2 and 5.3 show that the speedup resulting from, and linearity of, the recurrence relation agree with the theoretical expectation which results from changing the complexity of the reserving algorithm from $O(T^2)$ to $O(T)$ for T projection steps.

The discussion of the change in algorithm in Section 5.2.4 indicates that experimental timings agree with theoretical expectation, as derived in Section 3.4.1.

The scaling indicated by Figure 5.4 suggests that the OpenMP implementation may not be particularly efficient on more than about 64 cores. However, this is not currently a major concern because SMPs with this number of cores are not common, especially in life offices at the present time.

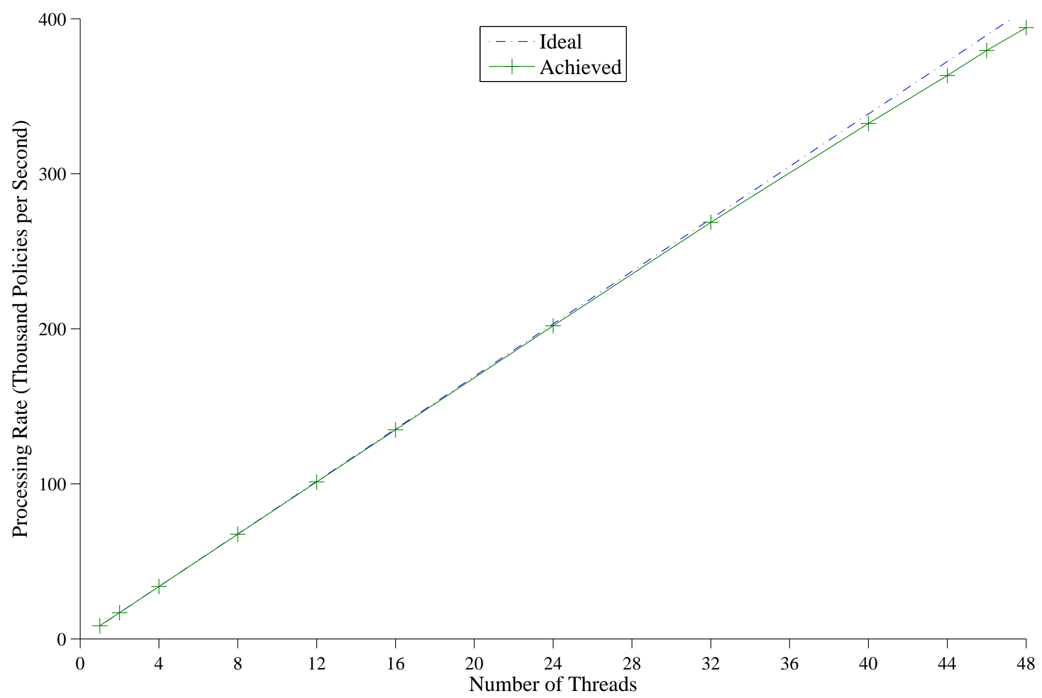


Figure 5.4 Performance using OpenMP on the Opteron cluster: for each thread count, the ideal processing rate is derived from the time taken to process 4×10^5 single life annuity policies on one thread.

Chapter 6

Simplifying Interpolation in the Mortality Table

It is highly unlikely that many policyholders will have their birthdays on the valuation date, so that most policyholders will be a non-integer age at that date: this becomes more relevant when monthly projections are being performed since the ages will be non-integer for at least 11 of the 12 steps. In order to calculate survival probabilities for non-integer ages, it is necessary to obtain l_x or q_x for fractional ages using interpolation between the values in the life table (which are tabulated for integral ages). Interpolation in the life table becomes more prominent for two life annuities, thereby increasing the importance of efficient interpolation. Therefore, this chapter considers simplifications to the interpolation process.

Section 6.1 provides an overview of the significance of interpolation, both in terms of the proportion of execution time taken and the accuracy of the financial results produced. It also contains a brief recap of the important actuarial concepts used in this chapter. Section 6.2 introduces the concept of the force of mortality, which may be considered as the formula underlying the mortality. The section contains a discussion of how to fit a polynomial to data obtained from published life tables, and how the mortality rates derived from the fitted polynomial compare to the rates in those published tables. Section 6.3 discusses what is apparently the simplest form of interpolation, i.e. linear interpolation in the l_x's. As the section shows, there is a major problem with this method, and that makes it wholly unusable. Section 6.4 discusses the interpolation used in the commercial environment, i.e. cubic interpolation in the l_x's. This method solved the problem with linear interpolation, but the method has a high calculational

cost, and that makes its use undesirable. Section 6.5 discusses an alternative simple form of interpolation, i.e. linear interpolation in the q_x's. However, again there is a major problem with this interpolation method which prohibits its use. Section 6.6 introduces an alternative form of deriving mortality rates for fractional step lengths. This method is based on the binomial expansion, and requires a slight adjustment which is also explained. Section 6.7 introduces a measure of goodness-of-fit of each of the various methods: this measure shows that the approximation based on the binomial expansion is almost as good as the complex method used in the commercial approach, and is far better than the simple interpolation methods. Finally, Section 6.8 contains a summary of the chapter.

6.1 Overview

As mentioned in Section 5.2.1, the routine which interpolates within the life table was the first to be manually optimised in an MSc project [86] which preceded this one: that optimisation improved the performance of the interpolation routine by a factor of roughly 43×. Despite that improvement, the profile at the end of the MSc showed the interpolation routine to be the most heavily occupied routine in the program.

In order to perform projections using monthly time steps, monthly survival probabilities are required. However, in general, q_x or l_x are tabulated only at integral ages, and so it is necessary to use interpolation to obtain probabilities at fractional ages, and for fractions of a year. Various methods of interpolation are possible, but these each have their failings: some of these interpolation methods are discussed in this chapter. Furthermore, as this chapter demonstrates, the method used to obtain the monthly death probabilities dramatically influences the progression of these probabilities.

A property of death probabilities which is obviously desirable is that "the probability of dying in any of 12 consecutive months should be the same as the probability of dying within the year comprising those 12 months": this chapter also considers how this requirement guides the method of interpolation.

Terminology

As with previous chapters, standard International Actuarial Notation [39] is used in this chapter. Therefore, for $x \in \mathbb{R}^+$, and $t \in [0, 1]$,

- $_tp_x$ is the probability of a life aged precisely x surviving for time t,

- $_tq_x$ is the probability of a life aged precisely x dying within time t: when time units are years, $_{\frac{1}{12}}q_x$ is the probability of a life aged precisely x dying within one month,
- l_{x+t} is the expected number of lives at age $x + t$, given a particular number of lives at age x.

6.2 Force of Mortality

Any method of interpolating in a life table will lead to some level of approximation: it is therefore necessary to find a (continuous) function that may be used as the ground truth, against which various methods of interpolation may be measured. Although there is no way of directly obtaining fractional survival probabilities from the tabulated values of q_x or l_x, it is possible to use a calculation based on the force of mortality, μ_x. A standard formula [65, Section 1.6] gives

$$_tp_x = \exp\left(-\int_0^t \mu_{x+r}\, dr\right) \tag{6.2.1}$$

Use of the force of mortality allows monthly survival probabilities to be determined exactly; these exact values may be used to gauge the effect of the various interpolation methods.

6.2.1 Survival Probabilities

By modelling the force of mortality as a function of age gives $\mu_r = \mu(r)$ so that

$$_tp_x = \exp\left(-\int_x^{x+t} \mu(r)\, dr\right) \tag{6.2.2}$$

For accuracy in fitting an appropriate model, and to allow for improvements in mortality, the Continuous Mortality Investigation Bureau would fit a curve to $\ln(\mu_r)$ [40]. However, for the purpose of demonstrating the effect of interpolation, it suffices to model to μ_r, rather than $\ln(\mu_r)$.

Values of q_x are tabulated, and so the values of p_x may be readily obtained. However, it is simpler to check goodness-of-fit (and fewer calculations are required) if the exponentiation is removed. Therefore, for the purpose of investigating interpolation, mortality is modelled by fitting a low-order polynomial to the force of mortality obtained from

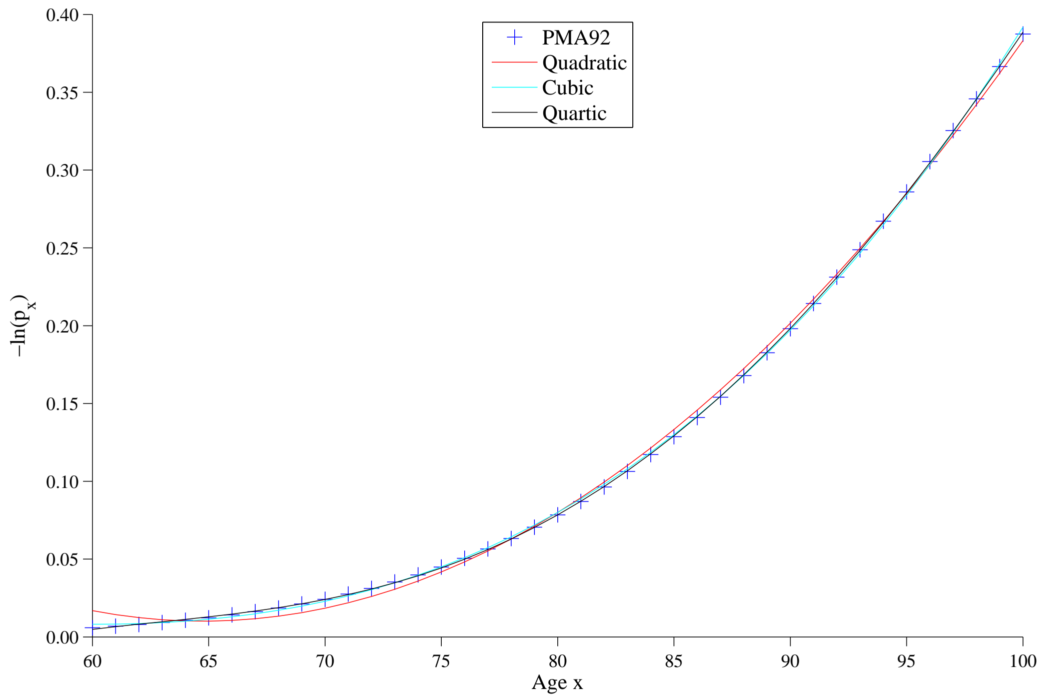


Figure 6.1 Polynomials of low degree fitted to $-\ln(p_x)$ where the q_x's are from PMA92 Ultimate.

rearranging Equation 6.2.2, viz:

$$-\ln({}_tp_x) = \int_x^{x+t} \mu(r)\,dr \tag{6.2.3}$$

To allow for the fact that the tabulated values are for death in the next year, setting $t = 1$ gives

$$-\ln(p_x) = \int_x^{x+1} \mu(r)\,dr$$

6.2.2 Fitting Polynomials

Figure 6.1 shows the effect of fitting low order polynomials to $-\ln(p_x)$ where the corresponding q_x's are from the PMA92 Ultimate table and the fitting process used least squares. The age range in the fitting process was restricted to $60 \leq x \leq 100$ for two reasons: firstly, in a cohort of recent, normal health, retirees there are unlikely to be many policies where the policyholder is below age 60, and secondly, for a representative cohort of retirees, age 100 is likely to be attained in roughly 40 years, by which time the effect of discounting will outweigh any effect caused by inaccuracy in mortality calculations.

It is apparent from Figure 6.1 that a quadratic curve does not provide a very good fit. Closer inspection also shows that the cubic exhibits decreasing mortality at the younger ages: while decreasing mortality is a feature of childhood mortality, mortality at pensioners' ages is generally expected to be a monotonically increasing function of age. Therefore, the lowest order, well fitting, polynomial with desirable properties is the quartic.

6.2.2.1 Quartic Polynomial

To a relatively small degree of precision, the fitted quartic is

$$\begin{aligned} -\ln(p_x) &= -1.048 \times 10^{-7}\, x^4 + 3.649 \times 10^{-5}\, x^3 - 4.395 \times 10^{-3}\, x^2 \\ &\quad + 2.256 \times 10^{-1}\, x - 4.235 \end{aligned} \tag{6.2.4}$$

Using the fitted quartic to obtain monthly probabilities is straightforward:

$$\begin{aligned} -\ln({}_t p_x) &= \int_x^{x+t} \mu(r)\, dr \\ &= \int_x^{x+t} \left(ar^4 + br^3 + cr^2 + dr + e\right)\, dr \end{aligned} \tag{6.2.5}$$

so that, for one year survival probabilities, setting $t = 1$ gives

$$\begin{aligned} -\ln(p_x) &= ax^4 + (2a+b)\, x^3 + \left(2a + \frac{3}{2}b + c\right) x^2 \\ &\quad + (a+b+c+d)\, x + \left(\frac{a}{5} + \frac{b}{4} + \frac{c}{3} + \frac{d}{2} + e\right) \end{aligned} \tag{6.2.6}$$

Equating coefficients between Equations 6.2.4 and 6.2.6 leads to

a	b	c	d	e
-1.048×10^{-7}	3.670×10^{-5}	-4.450×10^{-3}	2.300×10^{-1}	-4.349

For monthly probabilities, using these values, and setting $t = \frac{1}{12}$ in Equation 6.2.5, gives

$$\begin{aligned} -\ln\left({}_{\frac{1}{12}}p_x\right) &= -8.737 \times 10^{-9}\, x^4 + 3.058 \times 10^{-6}\, x^3 - 3.704 \times 10^{-4}\, x^2 \\ &\quad + 1.914 \times 10^{-2}\, x - 3.616 \times 10^{-1} \end{aligned} \tag{6.2.7}$$

Generating monthly death probabilities using Equation 6.2.7 leads to a monotonically increasing, non-negative set of values for ${}_{\frac{1}{12}}q_x$ for $60 \le x \le 100$. Therefore, this quartic

fit will be used to create the 'true' monthly mortality throughout the remainder of this chapter.

6.2.3 Fitted Mortality

As stated at the start of this chapter, the purpose of fitting the polynomial is to compare the effect of various methods of interpolation. To do this, it is necessary to obtain the annual mortality rates which are derived from the fitted monthly rates: the actual monthly death probabilities are

$$ {}_{\frac{1}{12}}q_x = 1 - {}_{\frac{1}{12}}p_x $$

where ${}_{\frac{1}{12}}p_x$ may be obtained from Equation 6.2.7.

The monthly death probabilities are the base-line, against which all approximations in the remainder of this chapter will be compared. The fitted annual mortality rates may be obtained from these fitted monthly death probabilities as

$$ \begin{aligned} q_x = 1 - p_x &= 1 - \prod_{k=0}^{11} {}_{\frac{1}{12}}p_{x+\frac{k}{12}} \\ &= 1 - \prod_{k=0}^{11} \left(1 - {}_{\frac{1}{12}}q_{x+\frac{k}{12}}\right) \end{aligned} \qquad (6.2.8) $$

Figure 6.2 shows the original rates tabulated in PMA92 Ultimate, and the annual rates fitted using the quartic in Equation 6.2.7 and the relationship in Equation 6.2.8: there is clear correspondence between the values at each age, indicating that the fitted quartic *is* appropriate.

6.2.4 Consistency of Probabilities

When interpolating in the l_x's, the probability of surviving for 12 consecutive monthly steps is

$$ \begin{aligned} \prod_{k=0}^{11} {}_{\frac{1}{12}}p_{x+t+\frac{k}{12}} &= \prod_{k=0}^{11} \exp\left(-\int_{x+t+\frac{k}{12}}^{x+t+\frac{k}{12}+\frac{1}{12}} \mu(r)\,dr\right) \\ &= \exp\left(-\sum_{k=0}^{11} \int_{x+t+\frac{k}{12}}^{x+t+\frac{k}{12}+\frac{1}{12}} \mu(r)\,dr\right) \end{aligned} $$

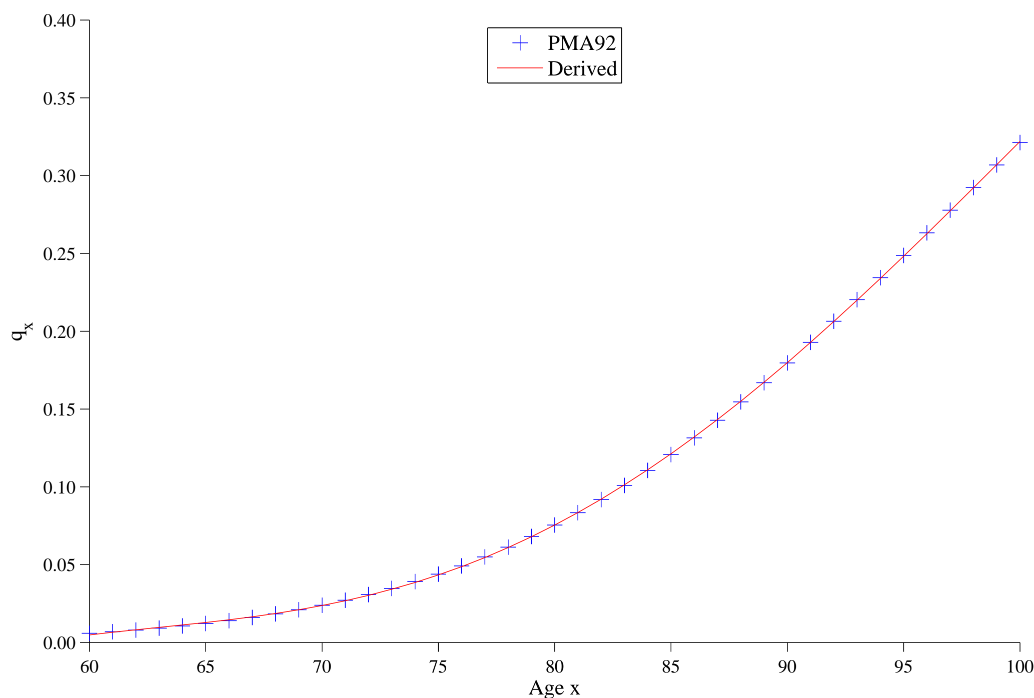


Figure 6.2 Annual death probabilities as tabulated in PMA92 Ultimate, and derived from fitting a quartic polynomial to the force of mortality.

$$= \exp\left(-\int_{x+t}^{x+t+1} \mu(r)\,dr\right) \qquad = {}_1p_{x+t}$$

Hence, the progression of monthly survival probabilities obtained via the force of mortality agrees to the annual survival probability: later parts of this chapter show that the consistency between the two methods of calculating probability of survival for one year is not always present for other methods of obtaining monthly death probabilities.

6.3 Linear Interpolation in the l_x's

A standard method of calculating probabilities is to use ${}_tp_x = \dfrac{l_{x+t}}{l_x}$. A simple method of obtaining l_{x+t} for non-integral ages is linear interpolation in the l_x's. Also, intuitively, the principle quantity is the number of lives, rather than the number of deaths. Therefore linear interpolation in the l_x's is appealing from the perspective of intuitive simplicity. However, as this section shows, there are problems with this form of interpolation.

6.3.1 Consistency of Probabilities

When interpolating in the l_x's, the probability of surviving for 12 consecutive monthly steps is

$$\prod_{k=0}^{11} {}_{\frac{1}{12}}p_{x+t+\frac{k}{12}} = \prod_{k=0}^{11} \frac{l_{x+t+\frac{k}{12}+\frac{1}{12}}}{l_{x+t+\frac{k}{12}}}$$

$$= \frac{l_{x+t+\frac{1}{12}}}{l_{x+t+\frac{0}{12}}} \times \frac{l_{x+t+\frac{2}{12}}}{l_{x+t+\frac{1}{12}}} \times \cdots \times \frac{l_{x+t+\frac{12}{12}}}{l_{x+t+\frac{11}{12}}} = \frac{l_{x+t+1}}{l_{x+t}}$$

$$= {}_1p_{x+t}$$

Hence, interpolation in the l_x's gives a progression of survival probabilities which agrees to that obtained directly from the underlying table, irrespective of the actual method of interpolation: this is because the l_x's used as knot-points for the interpolation *are* the l_x's in the table.

6.3.2 Progression of Probabilities

A standard formula [65, Section 1.6] using l_x's gives

$${}_{\frac{1}{12}}q_{x+t} = 1 - \frac{l_{x+t+\frac{1}{12}}}{l_{x+t}}$$

which, using linear interpolation between l_x's, is

$$\approx 1 - \frac{\left(1 - \left[t + \frac{1}{12}\right]\right)\ l_x + \left(\left[t + \frac{1}{12}\right]\right)\ l_{x+1}}{(1-t)\ l_x + (t)\ l_{x+1}}$$

and, since $l_{x+1} = (1 - q_x)\ l_x$, this is

$$= 1 - \frac{\left(1 - \left[t + \frac{1}{12}\right]\right)\ l_x + \left(t + \frac{1}{12}\right)\ (1 - q_x)\ l_x}{(1-t)\ l_x + t\ (1 - q_x)\ l_x}$$

which simplifies to

$$= 1 - \frac{1 - t\,q_x - \frac{1}{12}\,q_x}{1 - t\,q_x}$$

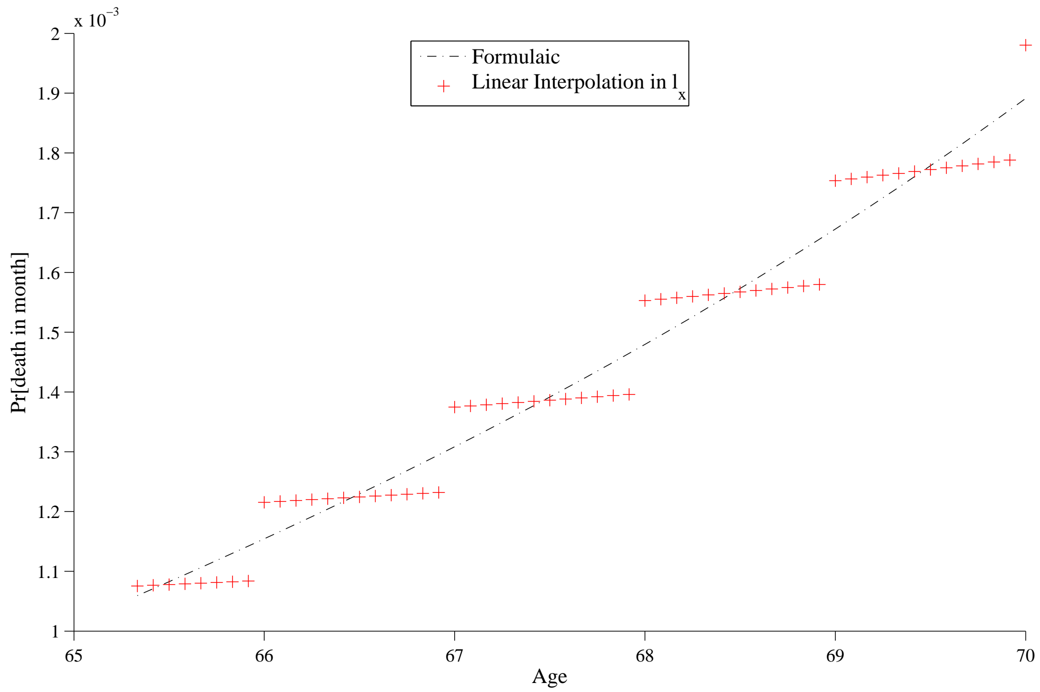


Figure 6.3 Progression of probability of death within the next month, for a life initially aged $65\frac{1}{3}$, increasing by $\frac{1}{12}$th thereafter, using linear interpolation in the l_x's.

and hence, ultimately,

$$_{\frac{1}{12}}q_{x+t} \approx \frac{1}{12\left(\frac{1}{q_x} - t\right)}$$

Since, in this form, $_{\frac{1}{12}}q_{x+t}$ is a function of only one q_x, there are two phenomena which prohibit the use of this form of interpolation. Firstly, for ages between about 55 and about 75, the size of q_x relative to t leads to the probability of dying within one month being an almost constant function of t for $t \in (0, 1)$, as shown in Figure 6.3: an alternative interpretation of this is that, since the values of l_x's are tabulated for $x \in \mathbb{Z}^+$, "the probability of dying within one month only changes significantly on a birthday", which is clearly absurd. Secondly, for ages over about 85, the progression of monthly death probabilities starts to oscillate wildly: for illustration, values derived from using q_x's obtained from Equation 6.2.7 give

$$_{\frac{1}{12}}q_{88\frac{11}{12}} \approx 15.1 \times 10^{-3} \quad , \quad _{\frac{1}{12}}q_{89} \approx 13.9 \times 10^{-3}$$

and

$$_{\frac{1}{12}}q_{94\frac{11}{12}} \approx 28.8 \times 10^{-3} \quad , \quad _{\frac{1}{12}}q_{95} \approx 23.1 \times 10^{-3}$$

The interpretation of this is that "the probability of dying within one month significantly reduces on a birthday", which is also absurd.

6.4 Implementation in a Commercial Environment

Since linear interpolation within the l_x's is not desirable because the resultant probabilities have strange properties, a different approach is required. One particular life office uses cubic interpolation within the l_x's to obtain survival probabilities: that approach is considered in this section.

6.4.1 Degree of Interpolation Polynomial

In a mortality investigation, the fundamental quantity observed is the number of lives at each particular age: the mortality rates are derived from these observations, possibly via the force of mortality [6, Chapter 2]. Therefore valuation actuaries within several life offices consider agreement between the l_x's used in their projection and the l_x's in the life table to be of fundamental importance, and hence interpolation must be within the l_x's: this agrees with Section 6.3.1 because the tabulated l_x's are the knot-points when interpolating in the l_x's.

A more realistic progression of probabilities may be obtained using a higher degree interpolation polynomial. In particular, so as not to bias the interpolation toward either higher or lower ages, it is possible to fit a cubic based on the two integral ages either side of the fractional age at which interpolation is being performed. The knot-points used are the l_x's from the underlying table so that, by construction, the derived l_x's must agree to the table.

Figure 6.5 shows the deviation from the true monthly mortality (obtained using Equation 6.2.7) and various approaches to interpolation: it shows that using cubic interpolation in the l_x's to obtain the probability of death within one month overcomes the problem of stepwise increases in the probability of death (seen in Section 6.3.2) and the values are particularly close to the true values. It therefore appears that using a more computationally intensive interpolation method might give 'correct' probabilities. This is the approach taken in the life office which inspired this investigation.

The original implementation used in this project, which mirrors this commercial practice, is based on fitting a cubic curve to the l_x's which are derived from the q_x's in the

table.

6.4.2 Initial Implementation

The original program in the commercial environment was only ever tested and run using monthly steps throughout the entire projection. To avoid the absurd progression of monthly survival probabilities resulting from linear interpolation in l_x's, and to produce a sequence of l_x's which does agree to the original life table, a somewhat convoluted approach has been implemented.

The stages used in the commercial environment are:

i) use the q_x's in the relevant table to create a series of l_x's: since the q_x's are tabulated for integral ages, the obtained l_x's will relate to integral ages;

ii) use cubic interpolation in the l_x's, based on two integer ages either side of the required age, to obtain the sequence of l's at fractional ages, i.e. $\{l_{x+t+f}\}$ for $x \in \mathbb{Z}^+$, $t \in \mathbb{R}^+$ and $f \in [0,1]$: since interpolation in the l_x's is used, the sequence $\{l_{x+t+f}\}$ will agree to the table when $(x+t+f) \in \mathbb{Z}^+$;

iii) obtain ${}_f q_{x+t} = 1 - \dfrac{l_{x+t+f}}{l_{x+t}}$ for $f \in (0,1)$ from the l's produced in the previous step: this ensures that the progression of l's obtained from the probabilities matches the l_x's in the table.

6.4.3 Complexity of the Commercial Implementation

After the optimisation in the MSc project [86, Section 3], the routine to calculate l_{x+t} using cubic interpolation in the l_x's can be described by the following:

Input: `actualAge` as $x \in \mathbb{R}^+$
Input: `lxTable` (as an array of l_x's at integer ages)

```
1: ageLastBirthday = floor(actualAge)
2: ageFraction = actualAge - ageLastBirthday
3: for k = 0 to 3 do
4:   j = k - 1
5:   lx(k) = lxTable(ageLastBirthday + j)
6:   ageAdj(k) = ageFraction - j
7: end for
8: factor(0) = - ONE_SIXTH * ageAdj(1) * ageAdj(2) * ageAdj(3)
9: factor(1) =   ONE_HALF  * ageAdj(0) * ageAdj(2) * ageAdj(3)
```

```
10: factor(2) = - ONE_HALF  * ageAdj(0) * ageAdj(1) * ageAdj(3)
11: factor(3) =   ONE_SIXTH * ageAdj(0) * ageAdj(1) * ageAdj(2)
12: reqdLx = factor(0) * lx(0) + factor(1) * lx(1)
          + factor(2) * lx(2) + factor(3) * lx(3)
```

Output: `reqdLx`

Overall, therefore, the commercial implementation for the process of calculating monthly death probabilities can be described as

Input: $x + t$, the age at the start of the interval
Input: f, the length of the interval
1: from age $x + t$, use cubic interpolation in the l_x's to obtain l_{x+t}
2: from age $x + t + f$, use cubic interpolation in the l_x's to obtain l_{x+t+f}
3: obtain ${}_fq_{x+t} = 1 - \dfrac{l_{x+t+f}}{l_{x+t}}$
Output: ${}_fq_{x+t}$

Note that `ONE_SIXTH` and `ONE_HALF` have been set as macro constants to remove the need to calculate or retrieve them. Allowing for the high degree of compiler optimisation used, it is possible that many of the intermediate variables are retained in registers: this becomes more probable if the interpolation routine is inlined within the overall routine. Therefore it is likely that the only values which are required to be retrieved from memory are the 8 values from `lxTable(:)`. Also, floating point arithmetic takes longer than integer arithmetic and so it seems reasonable to ignore the operations which relate to manipulation of indices. Finally, because the value of ${}_fq_{x+t}$ is output from the overall function, it seems unlikely that it will be stored. Therefore, the operation count for this approach to obtaining Pr[die within f, starting at age $x + t$] may be estimated as

operation	load	`floor()`	store	$\times$	$+$	$-$	$\div$
count	8	2	0	32	7	11	1

6.5 Linear Interpolation in the q_x's

In general, when a life table is published it does not only contain the l_x's: it also contains the q_x's for integral ages. Therefore, a straightforward alternative to interpolating in the l_x's is using interpolation within the q_x's to obtain monthly probabilities.

6.5.1 Progression of Probabilities

Assuming a uniform distribution of deaths, the standard actuarial result [6, Section 1.47] gives

$$_{\frac{1}{12}}q_{x+t} \equiv \frac{1}{12}\, q_{x+t}$$

which, using linear interpolation between q_x's populated for $x \in \mathbb{Z}^+$, is

$$\approx \frac{1}{12}\left[(1-t)\ q_x + t\, q_{x+1}\right]$$

This is a function of two adjacent q_x's so that, for pensioners' ages, $_{\frac{1}{12}}q_{x+t}$ is a continuously increasing (albeit piecewise linear) function of t. Therefore, linear interpolation in the q_x's leads to a more realistic progression of probabilities than linear interpolation in the l_x's.

However, as Figure 6.5 shows, the probabilities obtained by this method have a bias towards heavy mortality at lower ages. Further, as mortality increases at older ages, the contribution from q_x is not compensated for by the contribution from q_{x+1} so that the interpolated mortality at these older ages suffers a bias towards being too light.

6.5.2 Consistency of Probabilities

Using linear interpolation in the q_x's, the probability of surviving for 12 consecutive monthly steps is

$$\prod_{k=0}^{11} {}_{\frac{1}{12}}p_{x+t+\frac{k}{12}} = \prod_{k=0}^{11}\left(1 - {}_{\frac{1}{12}}q_{x+t+\frac{k}{12}}\right)$$

which, assuming a uniform distribution of deaths, is

$$\approx \prod_{k=0}^{11}\left(1 - \frac{1}{12}\, q_{x+t+\frac{k}{12}}\right)$$

and, by linear interpolation, this is

$$\approx \prod_{k=0}^{11}\left(1 - \frac{1}{12}\left[(1-k)\, q_{x+t} + k\, q_{x+t+1}\right]\right)$$

$$= \prod_{k=0}^{11} \left(1 - \frac{1}{12} \left[(1 - p_{x+t}) + k \left(p_{x+t} - p_{x+t+1} \right) \right] \right)$$

There is no clear path to showing this equal to ${}_1p_{x+t}$ and so it is difficult to see how interpolating in the q_x's produces anything which matches the l_x's in the underlying table. In fact, numerically, there is evidence that interpolation in the q_x's produces a sequence of l_x's which do not agree to the underlying table. This lack of agreement with the underlying table discourages the use of linear interpolation in the q_x's: certainly, this is the reason it is not used in some life offices.

However, by using a different approach it is possible to produce very similar death probabilities, thereby leading to l_x values which are similar to those in the original table.

6.6 Alternative Implementation

Sections 6.3 and 6.5 have demonstrated that linear interpolation is not a plausible approach to obtaining probabilities. However, by considering a further approximation to the probabilities it is possible to improve the performance of the routines which calculate the probabilities.

6.6.1 Interpolation Methodology

The value required is ${}_fq_x$, i.e. the probability that a life currently aged precisely x dies within a fraction f of a year. The binomial expansion may be used to obtain

$$\begin{aligned} {}_fq_x &\approx 1 - (1 - q_x)^f \\ &\equiv f\ q_x \left[1 - \frac{f-1}{2}\ q_x \left[1 - \frac{f-2}{3}\ q_x \left[1 - \frac{f-3}{4}\ q_x \left[\cdots \right] \right] \right] \right] \end{aligned}$$

Using only the first term of the expansion gives

$${}_fq_x \approx f\ q_x$$

which is precisely the approximation obtained by using the assumption of uniform distribution of deaths [6, Section 1.47]. Using the first two terms of the expansion gives

$${}_fq_x \approx f\ q_x \left(1 - \frac{f-1}{2}\ q_x \right) \tag{6.6.1}$$

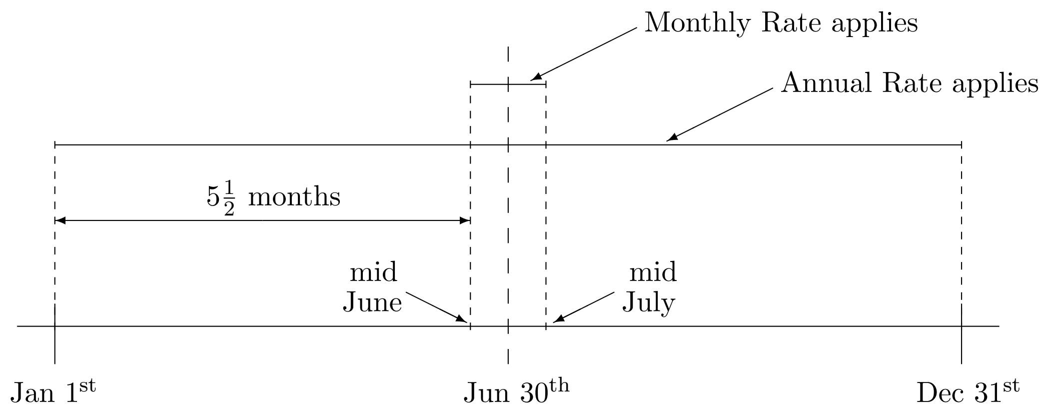


Figure 6.4 Required age deduction to ensure equivalence of average monthly mortality and average yearly mortality.

so that, for a monthly step, where $f = \frac{1}{12}$,

$$_{\frac{1}{12}}q_x \approx \frac{q_x}{12}\left(1 + \frac{11}{2}\,\frac{q_x}{12}\right) \tag{6.6.2}$$

In many statistical applications, a continuity correction is made when applying continuous distributions to discrete data: that correction takes the form of a deduction of $\frac{1}{2}$ from the discrete value. For mortality, the rate which should apply to the middle of a monthly step should be the same as the rate which would apply to the middle of a yearly step, if the mid-points of the intervals coincided: this is depicted in Figure 6.4 which considers the mortality applicable to a calendar year starting on January 1$^{\text{st}}$. Hence, the q_x which should be used in the binomial expansion is that rate which applies to a life $5\frac{1}{2}$ months younger, i.e. $q_{x-5.5/12}$ where x is the actual age of the life. Figure 6.5 shows that the monthly death probabilities, obtained by setting $f = \frac{1}{12}$ in this method, show a slight deviation from the expected values, but that deviation is extremely small compared to the deviations resulting from other interpolation methods.

6.6.2 Complexity of the Simplification

For the simplification, the process is

Input: `actualAge` as $x \in \mathbb{R}^+$
Input: f, the length of the interval
Input: `qxTable` (as an array of q_x's at integer ages)
1: `effectiveAge` $= x' = x - \frac{5.5}{12}$
2: `qxAge = floor(effectiveAge)`

3: `ageFraction = effectiveAge - qxAge`

4: use linear interpolation in the q_x's to obtain $q_{x'}$

i.e. $q_{x'}$ = `(1 - ageFraction) * qxTable(qxAge)`

`+ ageFraction * qxTable(qxAge + 1)`

5: calculate ${}_f q_{x'}$ using Equation 6.6.1

i.e. ${}_f q_{x'}$ = f `*` $q_{x'}$ `* (1 - 0.5 * (`$f-1$`) *` $q_{x'}$`)`

Output: ${}_f q_{x'}$

Using the same assumptions as Section 6.4.3, the complexity of this approach is estimated as

operation	load	`floor()`	store	$\times$	$+$	$-$	$\div$
count	2	1	0	6	1	5	0

Hence, compared to the commercial implementation discussed in Section 6.4, this simplification leads to a significant reduction in data retrieval and computation: the resulting improvement in performance is discussed in Section 8.3.1.

6.6.3 Higher Order Approximation

Using the first three terms of the expansion gives

$$ {}_f q_x \approx f\, q_x \left(1 - \frac{f-1}{2}\, q_x + \frac{f-1}{2}\,\frac{f-2}{3}\, (q_x)^2\right) $$

so that, for monthly death probabilities, where $f = \frac{1}{12}$,

$$ {}_{\frac{1}{12}} q_x \approx \frac{q_x}{12} \left(1 + \frac{11}{24}\, q_x + \frac{11}{24}\,\frac{10}{36}\, (q_x)^2\right) $$

The deviations of the monthly death probabilities (obtained from this approximation) from the expected values are shown in Figure 6.5. Given the size of q_x, and hence of $(q_x)^2$, the probabilities obtained using the first three terms do not really diverge from those obtained using only the first two terms until about 20 years into the projection, by which time the effect of discounting will override the change caused by any error in the estimation of the probability. Therefore, there is an increase in volume of calculations for no significant change in q_x's, and hence no significant improvement in the accuracy of the financial results.

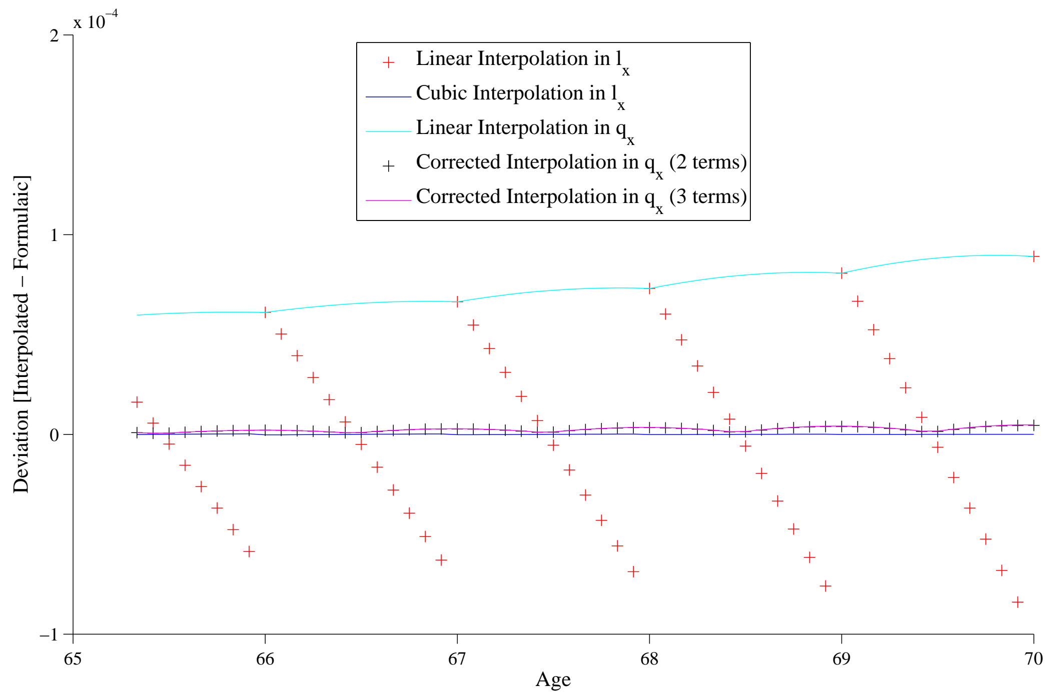


Figure 6.5 Deviation from known formulaic value of monthly probabilities of death, using different interpolation methods, for a life initially aged $65\frac{1}{3}$, with the age increasing by $\frac{1}{12}^{\text{th}}$ thereafter, to a maximum of age 70.

6.7 Effect of Interpolation Method

Although this chapter has discussed various forms of interpolation, there has been no attempt to justify the quality of those interpolations. This section quantifies the relative differences of those methods from the 'true' expected values obtained using Equation 6.2.7.

6.7.1 Goodness of Fit

Figure 6.5 shows the deviation of the interpolated values from the known formulaic values: the deviation at age x is

$$\delta_x = \hat{q}_x - \mathring{q}_x$$

where $\hat{q}_x$ is the interpolated value and $\mathring{q}_x$ is the value obtained from the formula derived in Section 6.2.2. Although Figure 6.5 provides evidence that the different methods of interpolation differ in their closeness to the 'correct' value, it does not quantify the accuracy of each of the methods.

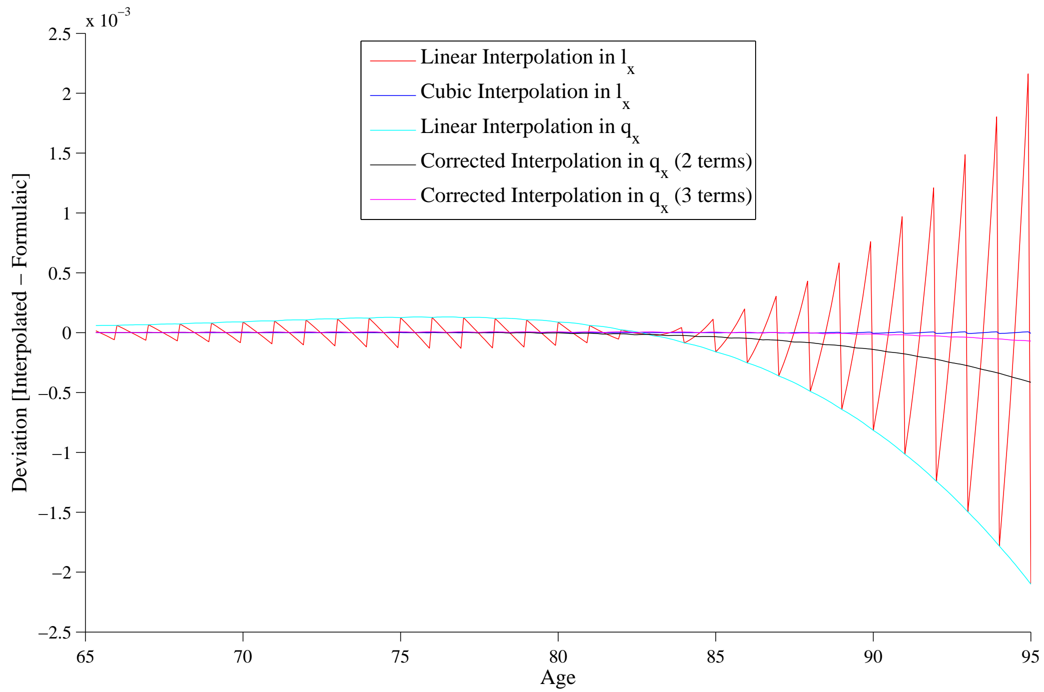


Figure 6.6 Deviation from known formulaic value of monthly probabilities of death, using different interpolation methods, for a life initially aged $65\frac{1}{3}$, with the age increasing by $\frac{1}{12}^{\text{th}}$ thereafter, to a maximum of age 95.

A simple measure of closeness is the RMS value, i.e.

$$\phi = \sqrt{\frac{1}{n}\sum_{x}\left(\hat{q}_x - \mathring{q}_x\right)^2}$$

However, this is skewed by large deviations from large values where, overall, the relative deviation may not be too large: this becomes especially apparent at older ages, as shown in Figure 6.6. Therefore, a better measure is one akin to the chi-squared statistic, as used in goodness-of-fit tests: this leads to the closeness, ξ, being

$$\xi = \sum_{x}\frac{\left(\hat{q}_x - \mathring{q}_x\right)^2}{\mathring{q}_x}$$

where the sum is over monthly ages x_m for $65 \leq x_m \leq 95$, i.e. the same as the range in Figure 6.6. The closeness, using this measure, of each of the interpolation techniques is shown in Table 6.1, from which it is apparent that the commercial approach of using cubic interpolation in the l_x's gives the best results with regard to correctness of probabilities. The alternative implementation from Section 6.6 gives a reasonably good result: there is no real need to use a third term in the expansion since the deviations from the known formulaic values do not start to become significant until after the effect

Interpolation Method	Closeness, ξ
Linear Interpolation in l_x	3.30×10^{-3}
Cubic Interpolation in l_x	8.18×10^{-8}
Linear Interpolation in q_x	8.07×10^{-3}
Corrected Interpolation in q_x (2 terms)	2.56×10^{-4}
Corrected Interpolation in q_x (3 terms)	6.38×10^{-6}

Table 6.1 Closeness of interpolation to expected values for monthly probability of death.

Policy Type	Method	Payments	Renewal Exps	Investment Exps
Single Life	Commercial	1.4847×10^{10}	4.7824×10^{12}	1.5776×10^{11}
	Approximation	1.4844×10^{10}	4.7776×10^{12}	1.5760×10^{11}
Reversionary	Commercial	8.1767×10^{8}	2.1832×10^{11}	7.5464×10^{9}
	Approximation	8.1791×10^{8}	2.1810×10^{11}	7.5390×10^{9}
Joint Life	Commercial	2.0762×10^{9}	2.4533×10^{9}	7.0526×10^{7}
	Approximation	2.0755×10^{9}	2.4482×10^{9}	7.0391×10^{7}
Last Survivor	Commercial	2.9021×10^{9}	1.8547×10^{11}	6.3089×10^{9}
	Approximation	2.9017×10^{9}	1.8528×10^{11}	6.3027×10^{9}

Table 6.2 Initial reserves for representative portfolio of 500,000 policies: the 'commercial' implementation uses cubic interpolation in the l_x's (as described in Section 6.4.2), and the 'approximation' uses the algorithm in Section 6.6.

of discounting has swamped the effect of survival probabilities.

6.7.2 Financial Consequences

The validity of this simplification can be assessed by whether the change in financial results is material. Several 'definitions' of materiality are available: in terms of financial planning, a difference in results is usually considered immaterial if the results differ by less than 1%, and an old definition of materiality, used by auditors, was that a difference in results is immaterial if the difference is less than 5%. However, several life offices are currently moving to a modern concept of materiality, i.e. "a difference in results is immaterial if it does not change an economic decision which would be based on those results" [47, QC11].

The initial reserves for the representative portfolio of 500,000 policies have been obtained using the original commercial implementation based on cubic interpolation in the l_x's, and the first two terms of the expansion in Equation 6.6.1: the results are given in Table 6.2. The magnitude of the change in reserves is always less than $\frac{1}{4}$% which, by any practical definition, is immaterial.

The fact that the method of interpolation does not make a significant difference to the financial results is to be expected from the fact that the basic summation formula for a single life annuity with level payments is

$$a'_x = \sum_t v^t \, {}_tp_x$$

so that a small change in ${}_tq_x$ has no real effect on ${}_tp_x$ which

a) remains ‘imperceptibly close’ to 1 for ages up to about 85, and

b) is overwhelmed by v^t for values of t greater than about 15 so that, when the age implies that ${}_tp_x$ is not particularly close to 1, the discounting dominates anyway.

6.8 Summary

Linear interpolation in l_x’s should be avoided because of the stepwise nature of the increase in probabilities at low ages, and the erratic nature of the increase in probabilities at high ages.

The commercial implementation uses a method based on cubic interpolation in the l_x’s, thereby producing a smooth progression of probabilities at the cost of a large amount of computation. However, the monthly death probabilities produced are reasonably close to those obtained from use of the correct mortality.

Linear interpolation in q_x’s produces a smooth progression of probabilities, but these are biased towards heavy mortality at ages where discounting does not override survival probabilities.

The simplification using the binomial expansion produces probabilities which are smooth and not systematically biased: the amount of computational work required is greatly reduced compared to that required for the commercial implementation and there is no need to use more than a relatively simple implementation of the expansion.

Chapter 7

Re-drawing Parameters

Section 2.6.1 noted that, at the RSE conference in 2014 [79], Smith indicated that it is essential that interest rates be re-sampled every step. Despite this, there is a body of thought which considers that a sequence of parameters drawn from a time series, once drawn, may be re-used within the scenario in which the parameters were drawn, without affecting the validity of the simulation. This chapter considers the rationale behind, and consequences of, not re-drawing economic and demographic parameters at each step.

Section 7.1 considers whether or not parameters need to be re-drawn in a Monte Carlo simulation where the parameters are from a time series. Section 7.2 contains a complete mathematical statement of the interpretation of Solvency II considered in this project. A matrix form of the probability of transferring between states before a particular time is presented for the main classes of policy type considered in Chapter 4: these matrices allow the per-policy reserves to be obtained from the in-force reserves. Section 7.3 presents algorithms for possible implementations of the interpretation of Solvency II considered in this project: there are alternatives for whether the in-force reserves are calculated using the summation or recurrence approaches, and for each of these cases, whether or not parameters are re-drawn. Section 7.4 considers the relative merits of re-drawing parameters. Finally, Section 7.5 contains a summary of the chapter.

7.1 To Re-draw or Not To Re-draw

This chapter considers the paradigm under which parameters that are sampled from a time series need not be re-drawn in each future time step because the values drawn at

outset will be equally valid for the particular time step to which they relate. However, a smaller variance in the calculated values is obtained if the values are re-drawn.

Solvency II is a set of regulations enacted in Law: no law is completely unambiguous. Within the UK, the standard interpretation appears to be that the parameters should be re-drawn in each future step, per Smith's presentation at the 2014 RSE conference [79]. Other countries may have other interpretations, but allowing for re-drawing considers the worst computational case.

If the distributions from which the parameters are drawn do not change at each time step, e.g. in order to reflect changes which result from management actions, changes in legislation or a shortfall in a previous step, then (as noted in Section 3.2.5.3) it is not necessary to re-draw the parameters: i.e. the values for each t, as drawn in the first step, may be considered as valid for the entire projection. Under this approach, it is possible to calculate in-force reserves at $t = 0$ in a manner which calculates the in-force reserves for all future $t > 0$ as a by-product. However, by adopting this approach, it is necessary to increase the number of scenarios at each future step in order to reduce the statistical error in the final result: the increased number of scenarios are discussed in Section 8.5.3.

It is not entirely clear that there is a particular advantage (in terms of accuracy of the answer) in re-drawing from the mortality table every month: since changes in mortality are gradual, re-drawing yearly should be sufficient. However, since the in-force reserves need to be re-calculated in every step of the outer loop (to allow for interest and inflation), it is appropriate to re-draw the mortality.

7.2 Mathematical Representation

Algorithm 2.1, in Section 2.3.3.1, gave an outline of the mathematical statement of the algorithm for satisfying Solvency II requirements. This section gives a more complete mathematical description of the calculations required to obtain Solvency II's additional capital requirement which allows the algorithms in Section 7.3 to be directly related back to the underlying mathematical requirements.

7.2.1 Transitions between States before Time t

Let the matrix of probabilities of transition between states between times 0 and t be

$$\begin{cases} \mathbf{H}_{x,t} & \text{for a single life aged } x \text{ at time 0} \\ \mathbf{H}_{xy,t} & \text{for two lives aged } x \text{ and } y \text{ at time 0} \\ \mathbf{H}_{\mathbf{x},t} & \text{for a collection of lives } \mathbf{x} \text{ at time 0} \end{cases}$$

where $t \in \{0, 1, 2, \dots\}$. Then,

- ⊛ for $t = 0$, i.e. the start step the first step,

  $$\mathbf{H}_{x,0} = \mathbf{H}_{xy,0} = \mathbf{H}_{\mathbf{x},0} = \mathbf{I}$$

  because it is not possible to move between states in a time interval of length zero, and
- ⊛ for $t \geq 1$ and various combinations of numbers of lives and possible states, the matrices are populated as shown in the remainder of this section.

Note that $\mathbf{H}_{\mathbf{x},t}$ corresponds to

$$\Pr[\text{transition between time 0 and time } t]$$

and so is not necessarily equal to $\mathbf{W}_{\mathbf{x},t,1}$, which corresponds to

$$\Pr[\text{transition during step of length 1, starting at time } t]$$

In fact

$$\mathbf{H}_{\mathbf{x},t} = \prod_{s=0}^{t-1} \mathbf{W}_{\mathbf{x},s,1} \qquad t \in \{1, 2, 3, \dots\} \tag{7.2.1}$$

In order to populate the matrix of probabilities of transition between states before time t, it is useful to use more standard notation. There is a possibility that a life survives for some time and then dies within a specified subsequent interval: this is the 'deferred probability of death within that interval' and, by definition,

$$\begin{aligned} {}_{r|s}q_x &= \Pr[(x) \text{ survives for time } r \text{ and then dies within further time } s] \\ &\equiv \Pr[(x) \text{ survives for time } r] \times \Pr[(x+r) \text{ dies within time } s] \\ &= {}_rp_x \; {}_sq_{x+r} \end{aligned}$$

$$= \frac{l_{x+r} - l_{x+r+s}}{l_x}$$

where, per previous sections, (x) denotes a life aged precisely x. As usual, if the period in which death may occur is 1 then the subscript may be dropped, i.e.

$$\begin{aligned} {}_{r|}q_x &= {}_{r|1}q_x \\ &= {}_r p_x \; q_{x+r} \end{aligned}$$

The following sub-sections simply state the stochastic matrices for various combinations of number of lives and states: the proofs of the correctness of these matrices are given in Appendix E.

7.2.1.1 Single Life Policies

For single life policies the matrices are straightforward:

⊛ for two states;

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_t p_x & {}_t q_x \\ 0 & 1 \end{pmatrix} \qquad \forall\, t \in \{1, 2, 3, \dots\}$$

⊛ for three states;

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_t p_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \forall\, t \in \{1, 2, 3, \dots\}$$

⊛ for three states with one reversible transition;

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_t p_x^{00} & {}_t p_x^{01} & {}_t p_x^{02} \\ {}_t p_x^{10} & {}_t p_x^{11} & {}_t p_x^{12} \\ 0 & 0 & 1 \end{pmatrix} \qquad \forall\, t \in \{1, 2, 3, \dots\}$$

7.2.1.2 Two-Life Policies

For two-life policies, it is possible to make use of the mixed product property of Kronecker products for matrices [36, Lemma 4.2.10]. This leads to the following:

⊛ for two states;

$$\mathbf{H}_{xy,t} = \begin{pmatrix} {}_tp_x\ {}_tp_y & {}_tp_x\ {}_tq_y & {}_tq_x\ {}_tp_y & {}_tq_x\ {}_tq_y \\ 0 & {}_tp_x & 0 & {}_tq_x \\ 0 & 0 & {}_tp_y & {}_tq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \forall\, t \in \{1, 2, 3, \dots\}$$

$$= \begin{pmatrix} {}_tp_x & {}_tq_x \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} {}_tp_y & {}_tq_y \\ 0 & 1 \end{pmatrix} \qquad \text{where } \otimes \text{ is the Kronecker product}$$

$$= \mathbf{H}_{x,t} \otimes \mathbf{H}_{y,t}$$

⊛ for three states;

$$\mathbf{H}_{xy,t} = \mathbf{H}_{x,t} \otimes \mathbf{H}_{y,t} \qquad \forall\, t \in \{1, 2, 3, \dots\}$$

$$= \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

7.2.1.3 Calculating Probabilities of Transitions between States

Although $\mathbf{H}_{\mathbf{x},t}$ is the matrix of probabilities for a collection of lives moving between states in t steps, there is no need to perform any matrix multiplications in order to populate $\mathbf{H}_{\mathbf{x},t}$: the fact that each entry is a product of transition probabilities for one or more lives means that it suffices to perform look-ups into mortality tables or, in the case of sickness benefits, morbidity tables.

For example, consider a two-life two-state model: from Section 7.2.1.2, the required matrix is

$$\mathbf{H}_{xy,t} = \begin{pmatrix} {}_tp_x\ {}_tp_y & {}_tp_x\ {}_tq_y & {}_tq_x\ {}_tp_y & {}_tq_x\ {}_tq_y \\ 0 & {}_tp_x & 0 & {}_tq_x \\ 0 & 0 & {}_tp_y & {}_tq_y \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

which, using Equation 2.1.2, is

$$= \begin{pmatrix} \frac{l_{x+t}}{l_x}\frac{l_{y+t}}{l_y} & \frac{l_{x+t}}{l_x}\frac{l_y - l_{y+t}}{l_y} & \frac{l_x - l_{x+t}}{l_x}\frac{l_{y+t}}{l_y} & \frac{l_x - l_{x+t}}{l_x}\frac{l_y - l_{y+t}}{l_y} \\ 0 & \frac{l_{x+t}}{l_x} & 0 & \frac{l_x - l_{x+t}}{l_x} \\ 0 & 0 & \frac{l_{y+t}}{l_y} & \frac{l_y - l_{y+t}}{l_y} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Therefore, so long as the look-ups can be performed in constant time, calculation of $\mathbf{H}_{\mathbf{x},t}$, for any number of lives, and any value of t, is linear in the number of lives.

7.2.2 Mathematical Formulation of ACR

Starting from Line 26 of Algorithm 2.1, the additional capital requirement is

$$\text{ACR} = \theta \sum_{t>0} \max(\hat{V}_t' - \hat{B}_t, 0)$$

As in Algorithm 2.1, let J be the number of scenarios, and let $j \in \{1, 2, \ldots, J\}$ be the index of the individual scenarios. Let $\overset{99.5\%}{\max_j} \{V_{t,j}\}$ be the 99.5^{th} percentile V over the scenarios at step t, and let $j^*(t)$ be the index of the scenario having the 99.5^{th} percentile value of $V_{t,j}$. Finally, let $d_{t_1,t_2,j^*(t)}$ be the discount factor from time t_1 to time $t_1 + t_2$ using interest rates from scenario $j^*(t)$. Then, allowing for sorting to find the 99.5^{th} percentile reserve,

$$\text{ACR} = \theta \sum_{t=1}^{T} \max \left(d_{0,t,j^*(t)} \overset{99.5\%}{\max_j} \{V_{t,j}\} - d_{0,t,j^*(t)} \, B_t, 0 \right) \tag{7.2.2}$$

Therefore, the calculationally-intensive part of Solvency II is finding the discounted 99.5^{th} percentile total per-policy reserve at each time step,

$$d_{0,t,j^*(t)} \overset{99.5\%}{\max_j} \{V_{t,j}\}_{j=1:J}$$

which requires calculation of the set $\{V_{t,j}\}_{j=1:J}$, i.e. the entire set of discounted total per-policy reserves at each time step t where $t \in \{1, 2, \cdots, T\}$.

From Section 3.2.5.3, $\mathbf{W}_{\mathbf{x},t,g}$ is the stochastic matrix containing probabilities of changing states between times t and $t+g$, for $g \in \{f, 1\}$. Also, from Section 7.2.1, $\mathbf{H}_{\mathbf{x},t}$ is the stochastic matrix containing probabilities of changing states before the start of the tth step. The lives are in a particular state at $t = 0$, so let $\mathbf{h}_{\mathbf{x},t}$ be the row of $\mathbf{H}_{\mathbf{x},t}$ which corresponds to the state of the lives at $t = 0$: i.e. $\mathbf{h}_{\mathbf{x},t}$ is the vector of probabilities of the lives migrating to a state at time t, given the state which they are in at $t = 0$.

The per-policy reserve at time t is the sum of in-force reserves over all possible states which the lives could occupy at that time, allowing for the probability of transferring to those states. Hence, adding a j index to emphasise that the reserve is dependent on the scenario from which the parameters are drawn,

$$V_{\mathbf{x},t,j} = \mathbf{h}_{\mathbf{x},t,j}\, \mathbf{r}_{\mathbf{x},t,j}$$

where

- $\mathbf{r}_{\mathbf{x},t,j}$ is a column vector of in-force reserves at time t, calculated using parameters from scenario j,
- $\mathbf{h}_{\mathbf{x},t,j}$ is the row vector of probabilities of the lives migrating to a state at time t, given the state they are in at $t = 0$, and using parameters from scenario j, and
- $V_{\mathbf{x},t,j}$ is a scalar which represents the per-policy reserve, at time t, using parameters from scenario j.

Hence, for scenario j, allowing for the fact that the lives $\mathbf{x}$ relate to policy p, so that $\mathbf{x}$ may be used as summation index, the total per-policy reserve at time t is

$$V_{t,j} = \sum_{\substack{\mathbf{x}\in \\ \text{policies}}} V_{\mathbf{x},t,j}$$

which, using the result in the previous paragraph to relate the per-policy reserve to the in-force reserve for individual policies, is

$$= \sum_{\substack{\mathbf{x}\in \\ \text{policies}}} \mathbf{h}_{\mathbf{x},t,j}\, \mathbf{r}_{\mathbf{x},t,j}$$

and, using the results from Equation 3.2.4, this is

$$= \sum_{\substack{\mathbf{x}\in \\ \text{policies}}} \mathbf{h}_{\mathbf{x},t,j} \sum_{s=0}^{\infty} d_{t,s+f,j}\, \mathbf{W}_{\mathbf{x},t,s+f,j}\, \mathbf{c}_{\mathbf{x},t+s}$$

where

– $\mathbf{h}_{\mathbf{x},t,j}$ is the vector of probabilities of the lives migrating to a state at time t, given the state they are in at time $t = 0$, using parameters from scenario j,

– $d_{t_1,t_2,j}$ is the discount factor from time t_1 to time $t_1 + t_2$ using interest rates from scenario j,

– $\mathbf{W}_{\mathbf{x},t_1,t_2,j}$ is the transition matrix containing probabilities of the lives $\mathbf{x}$ migrating from one state at time t_1 to another (not necessarily different) state at time t_1+t_2, using mortality tables, morbidity tables, etc., from scenario j,

– $\mathbf{c}_{\mathbf{x},t}$ is the vector of cash flows made at time t: these monetary amounts are independent of scenarios.

Substituting this back into Equation 7.2.2, and allowing for the linearity of multiplication of the discount factor, gives

$$\boxed{\mathrm{ACR} = \theta \sum_{t=1}^{T} \max \left(\overset{99.5\%}{\max_{j}} \left\{ \sum_{\substack{\mathbf{x}\in \\ \text{policies}}} \mathbf{h}_{\mathbf{x},t,j} \sum_{s=0}^{\infty} d_{t,s+f,j}\, \mathbf{W}_{\mathbf{x},t,s+f,j}\, \mathbf{c}_{\mathbf{x},t+s} \right\} - B_t, 0 \right) d_{0,t,j^*(t)}} \tag{7.2.3}$$

7.3 Algorithms and Their Complexity

The process of obtaining the additional capital requirement was described in Algorithm 2.1 where, in Line 11, economic and demographic parameters are obtained for each step in the t loop. Parts of Section 2.3.3.1 discussed the possibility of obtaining the additional capital requirement without re-drawing parameters at each future time: it is therefore necessary to consider the complexity of the two cases (re-drawing *vs.* not re-drawing) separately.

Equation 7.2.3 details the mathematical relationship between the policies, the parameters and the additional capital requirement. This section presents different algorithms, each showing a possible implementation of the calculations required by Equation 7.2.3.

Note that, although the estimate of run time derived in Appendix A is used within the life industry, it is rather naive: it is based on an $O(T^2)$ summation at each of T future steps, making the overall complexity of the estimate $O(T^3)$. However, due its naivety, this approach is not one which is pursued in this thesis: the algorithms in this section are such that the worst complexity is $O(T^2)$. Also note that the algorithms in this section are conceptual algorithms: there may be implementational optimisations such as allowing for Zero Reserve States, and fusing loops, but these do not affect

the complexity of the algorithms and so are not considered here. Finally note that, as mentioned in Section 7.2.2, p and $\mathbf{x}$ are equivalent summation indices for the pth policy. Therefore, in the following algorithms, the loops are indexed over p but variables are indexed as $\mathbf{x}$.

7.3.1 Parameters are Re-drawn

The requirement to re-draw parameters came from the initial interpretation of Solvency II by the ex-modelling actuary from Aegon UK [15], as presented in Algorithm 2.1. This interpretation is further enforced by Smith (at the RSE conference in 2014 [79]) where he observed that it is essential that interest rates be re-sampled every step. Given the predominantly actuarial nature of this project, the interpretation of the regulations by actuaries is given prominence in the project: the majority of the analysis in Chapter 8 is therefore conducted on the basis that parameters be re-drawn each time step.

Allowing for re-drawing, the complete set of parameters is $\{\mathcal{P}_{t,j}\}$ where $1 \leq t \leq T$, and $1 \leq j \leq J$. Each element of the set consists of an entire basis: i.e. the discount factors (derived from the interest rate) and the inflation factors for all time steps s, where $t \leq s \leq T$, and mortality rates both sexes, for each of 120 ages, for each of the years of birth covered by the data set. Note that for time step t the discount and inflation factors for $s < t$ are not required and can be omitted from the parameter sets, but the full mortality tables are still needed. Section 8.1.3.2 discusses how the parameter sets are stored in files. The actual values in the files do not affect the amount of computation to be performed and are therefore unimportant in this project which is only concerned with performance. In practice, the regulator will dictate the actual values, or some means of producing them.

7.3.1.1 Naive Summation

Algorithm 7.1 calculates in-force reserves using the summation approach from Algorithm 3.1 after re-drawing parameters at each step: conceptually, this is the simplest intuitive description of the interpretation in Section 2.3.3.1. From the loops over t and s in Lines 9 and 20 of Algorithm 7.1, calculation of the ACR using summation over a maximum of T time steps when re-drawing parameters has complexity $O(T^2)$.

7.3.1.2 Recurrence Relation

Algorithm 7.2 calculates in-force reserves using the recurrence approach from Algorithm 3.2 after re-drawing parameters at each step. From the loops over t and s in Lines 9 and 21 of Algorithm 7.2, calculation of the ACR using the recurrence relation over a maximum of T time steps when re-drawing parameters has complexity $O(T^2)$.

7.3.2 Parameters are not Re-drawn

The alternative interpretation, where parameters are not re-drawn, is also considered in this project. When parameters are not re-drawn, the complete set of parameters is $\{\mathcal{P}_j\}$ where $1 \leq j \leq J$, and J is the number of scenarios. Again, each element of the set consists of an entire basis, i.e. the discount factors and the inflation factors for all time steps s, where $1 \leq s \leq T$, and mortality rates both sexes, for each of 120 ages, for each of the years of birth covered by the data set. Section 8.5.1 discusses how the parameter values are stored in files: that section also contains the associated changes to the program which are required to correctly implement this regime. As with the case where parameters are re-drawn, the actual values in the files are not important in this project.

7.3.2.1 Naive Summation

Algorithm 7.3 calculates in-force reserves using the summation approach from Algorithm 3.1, but does not re-draw parameters at each step. Although parameters are not re-drawn, the loops over t and s in Lines 12 and 22 of Algorithm 7.3 mean that the process of obtaining the ACR for a projection over T time steps has complexity $O(T^2)$.

7.3.2.2 Recurrence Relation

Algorithm 7.4 calculates in-force reserves using the summation approach from Algorithm 3.2, but does not re-draw parameters at each step. Because parameters are not re-drawn, it is possible to rearrange the loops so that there are no nested loops over future time steps. The values of $\mathbf{h}_{\mathbf{x},t,j}$ (the probabilities of transferring to a particular state between time 0 and time t) must be calculated in the forward loop starting at Line 19 in order that the values are known when they are required in the backward

loop, starting at Line 32, which calculates the per-policy reserves, $\mathbf{r}_{\mathbf{x},t,j}$. By storing the values of $V_{t,j}$ for $t > 1$ as the backward loop progresses, it is not necessary to perform any recalculation.

Hence, since there are no nested loops over time, obtaining the ACR for a maximum of T time steps using this approach has complexity $O(T)$. Note that although it is possible to perform loop fusion and permutations in this algorithm, those optimisations do not add to the clarity of the underlying process and have therefore not been shown.

7.4 Relative Merits of Re-drawing or Not

The preceding sections have presented the definition of the ACR, and algorithms to calculate it depending on whether or not parameters are re-drawn. This section considers the relative merits of each approach.

Re-drawing parameters at each future step is generally accepted practice within life offices in the UK, per [84]: it coincides with the view of actuaries that it is necessary to produce a new yield curve in each step of the outer loop of a nested stochastic projection of liabilities. Conversely, not re-drawing parameters requires further work to establish the validity of the approach and hence permit acceptability by actuaries: this is discussed in Section 9.1.4.

Re-drawing parameters at each step requires a large amount of computation. Not re-drawing parameters results in a reduction in computation because the $O(T)$ algorithm can be used. While fewer calculations are required for each scenario, experimentation is required to quantify the robustness of the ACR produced in this way: it is likely that more scenarios will be required. Based on the results in Figure 5.2, assuming a value of $T = 720$, around 140,000 scenarios without re-drawing could be performed in the same time as 1000 scenarios with re-drawing. This is confirmed by the actual performance measurements comparing implementations of Algorithms 7.2 and 7.4, as presented in Section 8.5.3.

When re-drawing parameters, it is difficult to obtain a robust estimator for the ACR with significantly fewer than 1000 scenarios. Conversely, by not re-drawing parameters it is easier to vary the number of scenarios in order to achieve a required variance of the estimator.

7.5 Summary

So long as the values of l_x can be obtained from the relevant life tables in constant time, calculation of the probabilities of transition before any time step is linear in the number of lives; i.e. it is independent of the number of time steps. Therefore calculation of the per-policy reserves has the same complexity (in the number of time steps) as calculation of the in-force reserves.

The complexity of the various algorithms to calculate the ACR over T time steps may be summarised as follows:

Parameters Re-drawn ?	Approach	Algorithm	Complexity
Yes	Summation	7.1	$O(T^2)$
Yes	Recurrence	7.2	$O(T^2)$
No	Summation	7.3	$O(T^2)$
No	Recurrence	7.4	$O(T)$

Hence, the complexity of the implementation is only $O(T)$ when parameters are not re-drawn.

Algorithm 7.1 Calculation of ACR using summation and re-drawing parameters.

1: set J = number of scenarios
2: set T = maximum projection step number
3: **for** p=1 to number of policies **do**
4: **for** t=1 to T **do**
5: calculate nominal cash flows: $\mathbf{c}_{\mathbf{x},t}$ {these are independent of all bases}
6: **end for**
7: **end for**
8: obtain best estimate economic and demographic parameters for all future steps
9: **for** t=1 to T **do**
10: initialise the overall best estimate per-policy reserve at time t: $B_t = 0$
11: **for** p=1 to number of policies **do**
12: calculate the best estimate per-policy reserve at time t for policy p ...
13: ... and increment B_t by the per-policy reserve for policy p
14: **end for**
15: **for** j=1 to J **do**
16: read economic and demographic parameters $\mathcal{P}_{t,j}$ from file
17: initialise the total per-policy reserve: $V_{t,j} = 0$
18: **for** p=1 to number of policies **do**
19: initialise $\mathbf{r}_{\mathbf{x},t,j} = \mathbf{0}$, $d_{t,0,j} = 1$, $\mathbf{W}_{\mathbf{x},t,0,j} = \mathbf{I}$
20: **for** $s = 0, \ldots, T - t$ **do**
21: obtain (from $\mathcal{P}_{t,j}$) the discount factor for step $t + s$: $d_{t+s,1,j}$
22: set $d_{t,s+f,j} = d_{t,s,j} \cdot (d_{t+s,1,j})^f$
23: set $d_{t,s+1,j} = d_{t,s,j} \cdot d_{t+s,1,j}$
24: obtain $\mathbf{W}_{\mathbf{x},t+s,f,j}$ and $\mathbf{W}_{\mathbf{x},t+s,1,j}$ from mortality/morbidity tables in $\mathcal{P}_{t,j}$
25: set $\mathbf{W}_{\mathbf{x},t,s+f,j} = \mathbf{W}_{\mathbf{x},t,s,j}\, \mathbf{W}_{\mathbf{x},t+s,f,j}$
26: set $\mathbf{W}_{\mathbf{x},t,s+1,j} = \mathbf{W}_{\mathbf{x},t,s,j}\, \mathbf{W}_{\mathbf{x},t+s,1,j}$
27: increment $\mathbf{r}_{\mathbf{x},t,j}$ by $d_{t,s+f,j}\, \mathbf{W}_{\mathbf{x},t,s+f,j}\, \mathbf{c}_{\mathbf{x},t+s}$
28: **end for**
29: calculate probabilities of being in a particular state at time t: $\mathbf{h}_{\mathbf{x},t,j}$
30: calculate the per-policy reserve: $V_{\mathbf{x},t,j} = \mathbf{h}_{\mathbf{x},t,j}\, \mathbf{r}_{\mathbf{x},t,j}$
31: increment $V_{t,j}$ by the individual per-policy reserve $V_{\mathbf{x},t,j}$
32: **end for**
33: **end for**
34: obtain V'_t by sorting $\{V_{t,j}\}_{j=1}^{J}$ and taking the 99.5$^{\text{th}}$ percentile
35: obtain the discount factor from time 0 to time t from the basis used to calculate the 99.5$^{\text{th}}$ percentile: $d_{0,t,j^*(t)}$
36: calculate the discounted best estimate reserve at time t: $\hat{B}_t = d_{0,t,j^*(t)}\, B_t$
37: calculate the discounted 99.5$^{\text{th}}$ percentile reserve at time t: $\hat{V}'_t = d_{0,t,j^*(t)}\, V'_t$
38: **end for**
39: calculate the Additional Capital Requirement: $\text{ACR} = \theta \sum_{t=1}^{T} \max\left(\hat{V}'_t - \hat{B}_t, 0\right)$

Algorithm 7.2 Calculation of ACR using recurrence relation and re-drawing parameters.

1: set J = number of scenarios
2: set T = maximum projection step number
3: **for** p=1 to number of policies **do**
4:     **for** t=1 to T **do**
5:         calculate nominal cash flows: $\mathbf{c}_{\mathbf{x},t}$ {these are independent of all bases}
6:     **end for**
7: **end for**
8: obtain best estimate economic and demographic parameters for all future steps
9: **for** t=1 to T **do**
10:     initialise the overall best estimate per-policy reserve at time t: $B_t = 0$
11:     **for** p=1 to number of policies **do**
12:         calculate the best estimate per-policy reserve at time t for policy p ...
13:         ... and increment B_t by the per-policy reserve for policy p
14:     **end for**
15:     **for** j=1 to J **do**
16:         read economic and demographic parameters $\mathcal{P}_{t,j}$ from file
17:         initialise the total per-policy reserve: $V_{t,j} = 0$
18:         **for** p=1 to number of policies **do**
19:             calculate probabilities of being in a particular state at time t: $\mathbf{h}_{\mathbf{x},t,j}$
20:             initialise $\mathbf{r}_{\mathbf{x},T+1,j} = \mathbf{0}$
21:             **for** $s = T - t, \ldots, 0$ **do** {descending}
22:                 obtain (from $\mathcal{P}_{t,j}$) the discount factor for step $t+s$: $d_{t+s,1,j}$
23:                 obtain $\mathbf{W}_{\mathbf{x},t+s,f,j}$ and $\mathbf{W}_{\mathbf{x},t+s,1,j}$ from mortality/morbidity tables in $\mathcal{P}_{t,j}$
24:                 set $\mathbf{r}_{\mathbf{x},t+s,j} = (d_{t+s,1,j})^f \ \mathbf{W}_{\mathbf{x},t+s,f,j} \, \mathbf{c}_{\mathbf{x},t+s} + d_{t+s,1,j} \, \mathbf{W}_{\mathbf{x},t+s,1,j} \, \mathbf{r}_{\mathbf{x+1},t+s,j}$
25:             **end for**
{at this point $s = 0$, so that the reserve is $\mathbf{r}_{\mathbf{x},t,j}$}
26:             calculate the per-policy reserve: $V_{\mathbf{x},t,j} = \mathbf{h}_{\mathbf{x},t,j} \, \mathbf{r}_{\mathbf{x},t,j}$
27:             increment $V_{t,j}$ by the individual per-policy reserve $V_{\mathbf{x},t,j}$
28:         **end for**
29:     **end for**
30:     obtain V'_t by sorting $\{V_{t,j}\}_{j=1}^{J}$ and taking the 99.5^{th} percentile
31:     obtain the discount factor from time 0 to time t from the basis used to calculate the 99.5^{th} percentile: $d_{0,t,j^*(t)}$
32:     calculate the discounted best estimate reserve at time t: $\hat{B}_t = d_{0,t,j^*(t)} \, B_t$
33:     calculate the discounted 99.5^{th} percentile reserve at time t: $\hat{V}'_t = d_{0,t,j^*(t)} \, V'_t$
34: **end for**
35: calculate the Additional Capital Requirement: $\text{ACR} = \theta \sum_{t=1}^{T} \max\left(\hat{V}'_t - \hat{B}_t, 0\right)$

Algorithm 7.3 Calculation of ACR using summation but not re-drawing parameters.

1: set J = number of scenarios
2: set T = maximum projection step number
3: **for** p=1 to number of policies **do**
4:     **for** t=1 to T **do**
5:         calculate nominal cash flows: $\mathbf{c}_{\mathbf{x},t}$ {these are independent of all bases}
6:     **end for**
7: **end for**
8: obtain best estimate economic and demographic parameters for all future steps
9: **for** $j = 1$ to J **do**
10:     read economic and demographic parameters $\mathcal{P}_j$ from file
11: **end for**
12: **for** t=1 to T **do**
13:     initialise the overall best estimate per-policy reserve at time t: $B_t = 0$
14:     **for** p=1 to number of policies **do**
15:         calculate the best estimate per-policy reserve at time t for policy p ...
16:         ... and increment B_t by the per-policy reserve for policy p
17:     **end for**
18:     **for** j=1 to J **do**
19:         initialise the total per-policy reserve: $V_{t,j} = 0$
20:         **for** p=1 to number of policies **do**
21:             initialise $\mathbf{r}_{\mathbf{x},t,j} = \mathbf{0}$, $\quad d_{t,0,j} = 1$, $\quad \mathbf{W}_{\mathbf{x},t,0,j} = \mathbf{I}$
22:             **for** $s = 0, \ldots, T - t$ **do**
23:                 obtain (from $\mathcal{P}_j$) the discount factor for step $t + s$: $d_{t+s,1,j}$
24:                 set $d_{t,s+f,j} = d_{t,s,j} \cdot (d_{t+s,1,j})^f$
25:                 set $d_{t,s+1,j} = d_{t,s,j} \cdot d_{t+s,1,j}$
26:                 obtain $\mathbf{W}_{\mathbf{x},t+s,f,j}$ and $\mathbf{W}_{\mathbf{x},t+s,1,j}$ from mortality/morbidity tables in $\mathcal{P}_j$
27:                 set $\mathbf{W}_{\mathbf{x},t,s+f,j} = \mathbf{W}_{\mathbf{x},t,s,j}\, \mathbf{W}_{\mathbf{x},t+s,f,j}$
28:                 set $\mathbf{W}_{\mathbf{x},t,s+1,j} = \mathbf{W}_{\mathbf{x},t,s,j}\, \mathbf{W}_{\mathbf{x},t+s,1,j}$
29:                 increment $\mathbf{r}_{\mathbf{x},t,j}$ by $d_{t,s+f,j}\, \mathbf{W}_{\mathbf{x},t,s+f,j}\, \mathbf{c}_{\mathbf{x},t+s}$
30:             **end for**
31:             calculate probabilities of being in a particular state at time t: $\mathbf{h}_{\mathbf{x},t,j}$
32:             calculate the per-policy reserve: $V_{\mathbf{x},t,j} = \mathbf{h}_{\mathbf{x},t,j}\, \mathbf{r}_{\mathbf{x},t,j}$
33:             increment $V_{t,j}$ by the individual per-policy reserve $V_{\mathbf{x},t,j}$
34:         **end for**
35:     **end for**
36:     obtain V'_t by sorting $\{V_{t,j}\}_{j=1}^{J}$ and taking the 99.5$^{\text{th}}$ percentile
37:     obtain the discount factor from time 0 to time t from the basis used to calculate the 99.5$^{\text{th}}$ percentile: $d_{0,t,j^*(t)}$
38:     calculate the discounted best estimate reserve at time t: $\hat{B}_t = d_{0,t,j^*(t)}\, B_t$
39:     calculate the discounted 99.5$^{\text{th}}$ percentile reserve at time t: $\hat{V}'_t = d_{0,t,j^*(t)}\, V'_t$
40: **end for**
41: calculate the Additional Capital Requirement: $\text{ACR} = \theta \sum_{t=1}^{T} \max\left(\hat{V}'_t - \hat{B}_t, 0\right)$

Algorithm 7.4 Calculation of ACR using recurrence relation but not re-drawing parameters.

1: set J = number of scenarios
2: set T = maximum projection step number
3: **for** p=1 to number of policies **do**
4:   **for** t=1 to T **do**
5:     calculate nominal cash flows: $\mathbf{c}_{\mathbf{x},t}$ {these are independent of all bases}
6:   **end for**
7: **end for**
8: obtain best estimate economic and demographic parameters for all future steps
9: **for** t=1 to T **do**
10:   initialise the overall best estimate per-policy reserve at time t: $B_t = 0$
11:   **for** p=1 to number of policies **do**
12:     calculate the best estimate per-policy reserve at time t for policy p ...
13:     ... and increment B_t by the per-policy reserve for policy p
14:   **end for**
15: **end for**
16: **for** j=1 to J **do**
17:   read economic and demographic parameters $\mathcal{P}_j$
18: **end for**
19: **for** t=1 to T **do**
20:   **for** j=1 to J **do**
21:     initialise the total per-policy reserve: $V_{t,j} = 0$
22:     **for** p=1 to number of policies **do**
23:       calculate probabilities of being in a particular state at time t: $\mathbf{h}_{\mathbf{x},t,j}$
24:     **end for**
25:   **end for**
26: **end for**
27: **for** j=1 to J **do**
28:   **for** p=1 to number of policies **do**
29:     initialise $\mathbf{r}_{\mathbf{x},T+1,j} = \mathbf{0}$
30:   **end for**
31: **end for**
32: **for** t=T to 1 **do** {descending}
33:   **for** j=1 to J **do**
34:     obtain (from $\mathcal{P}_j$) the discount factor for step $t+s$: $d_{t+s,1,j}$
35:     **for** p=1 to number of policies **do**
36:       obtain $\mathbf{W}_{\mathbf{x},t,f,j}$ and $\mathbf{W}_{\mathbf{x},t,1,j}$ from mortality/morbidity tables in $\mathcal{P}_j$
37:       set $\mathbf{r}_{\mathbf{x},t,j} = (d_{t,1,j})^f \; \mathbf{W}_{\mathbf{x},t,f,j} \, \mathbf{c}_{\mathbf{x},t} + d_{t,1,j} \; \mathbf{W}_{\mathbf{x},t,1,j} \, \mathbf{r}_{\mathbf{x+1},t,j}$
38:       calculate the per-policy reserve: $V_{\mathbf{x},t,j} = \mathbf{h}_{\mathbf{x},t,j} \, \mathbf{r}_{\mathbf{x},t,j}$
39:       increment $V_{t,j}$ by the individual per-policy reserve $V_{\mathbf{x},t,j}$
40:     **end for**
41:   **end for**
42:   obtain V'_t by sorting $\{V_{t,j}\}_{j=1}^{J}$ and taking the 99.5^{th} percentile
43:   obtain the discount factor from time 0 to time t from the basis used to calculate the 99.5^{th} percentile: $d_{0,t,j^*(t)}$
44:   calculate the discounted best estimate reserve at time t: $\hat{B}_t = d_{0,t,j^*(t)} \, B_t$
45:   calculate the discounted 99.5^{th} percentile reserve at time t: $\hat{V}'_t = d_{0,t,j^*(t)} \, V'_t$
46: **end for**
47: calculate the Additional Capital Requirement: $\text{ACR} = \theta \sum_{t=1}^{T} \max\left(\hat{V}'_t - \hat{B}_t, 0\right)$

Chapter 8

Fulfilling Solvency II Requirements

Section 2.3.3 introduced the demonstration of solvency under Solvency II: the Monte Carlo nature of the approach outlined there requires several sets of parameters to be processed. Using standard commercial software, these calculations are beyond contemplation. However, by creating *ab-initio* software which is designed from the outset to be well-performing, the calculations may be performed in a reasonable time. This chapter presents the code developed in this project, its features and its performance characteristics.

Section 8.1 discusses the code, both in terms of its implementation and its functionality: it also introduces the inputs, and the variability of the timings. Section 8.2 discusses the optimisations applied to the code: because the code was written from scratch, these optimisations are part of the design, rather than part of a subsequent speed improvement programme, and are therefore at a higher level than those considered in Chapter 5. Section 8.3 discusses simplifications used in the code, including the approximation covered in Section 6.6: these simplifications result in a small difference to the results, and are therefore different to the optimisations which do not affect the results. Section 8.4 discusses the performance of the code on different platforms: the hardware and software constituents of each platform were introduced in Section 2.7. Section 8.5 discusses performance of a version of the code which does not re-draw parameters from their distributions at each time step: the lack of re-drawing parameters is likely to reduce the statistical robustness of the estimated additional capital requirement but Section 8.5.3 considers how this may by compensated for by

increasing the number of scenarios processed. Section 8.6 contains a summary of the chapter.

8.1 Initial Code

Since the main use of a code which runs multiple scenarios would be fulfilling Solvency II regulations, the most efficient code would be one which only calculates reserves in each future time step: any calculation of profitability-related items is superfluous with respect to solvency. Therefore, if the existing profitability code were to be used, then revenue items would need to be stripped out of it. Whilst removing these items from the existing code is possible, and not particularly daunting, there could be a need to re-factor that code again in order to extract sufficient performance to allow the new regulations to be met. Creating *ab-initio* code means that the code can be designed, from the outset, for high levels of performance: in particular, it ensures that maximum use can be made of vectorisation.

8.1.1 Mechanics of The Code

The code could have been written in either Fortran or C. There is not much performance difference between codes written in these languages if the programs are written with optimal properties of modern hardware in mind and then compiled using modern compilers which target that hardware.

However, the code was written in C because coding techniques for new technologies tend to appear in C before their Fortran equivalents appear so that having a code base in C allows emerging technologies to be embraced more quickly. In particular, optimum performance is generally extracted from GPUs when CUDA is used, and that requires a good C code base to be available.

Because the code was written for modern hardware, it was designed to make use of vector units and so vectorisation was used wherever possible. The Intel compiler, used throughout development, allows the use of `#pragma simd` to encourage the compiler to create code which uses the vector units. This has the advantage of letting the compiler report which loops were vectorised, which loops were not vectorised in their current layout, and which loops could not be vectorised because of data dependencies: clearly, this information is invaluable when developing the code with the aim of maximising vectorisation. These pragmas are ignored by other compilers: there is therefore no need

to remove them when using, for example, the Cray compiler on the XC30. However, since the code was developed, OpenMP has added SIMD directives to the standard: it would be possible to translate the code to use these directives as compilers, which adhere to this version of the OpenMP standard, become available.

The main problem addressed in this project is the ability to perform the brute force calculations required to fulfil Solvency II. The code therefore concentrates on calculations, and data are stored in the simplest structures possible, i.e. arrays. While it would have been possible to create elaborate data types and storage mechanisms, such an approach would require the use of pointers: this is not a problem in itself, but de-referencing pointers takes time which could more usefully be used for performing floating point arithmetic.

8.1.2 Features of the Code

In order to improve the realism of the projections, the code is designed to allow flexibility in the parameters it uses. Both interest rates, and inflation rates, are assumed to vary on a monthly basis, and so the implementation allows different values for each time step. Similarly, to allow for future changes in mortality, the implementation allows for different tables in each time step.

The initial implementation has one purpose – demonstrating that Solvency II regulations can be met. The code therefore only performs the reserving calculations which are necessary to meet this goal. Revenue items may be added later if use for stochastically-based calculation of profitability, such as pricing, is required.

8.1.3 Inputs

The inputs for the single scenario calculations were discussed in Section 5.1. The corresponding values for multiple scenario calculations are similar and are therefore only covered briefly.

8.1.3.1 Data

The synthetic data for single life annuities and reversionary annuities are as described in Section 5.1.2. The fact that those data represent a cohort of recent retirees means that they have long outstanding terms, which makes them ideal for demonstrating that

the processing can be done for policies with non-trivial terms. Therefore, those data are reused, without alteration, for the multiple scenario calculations. Additionally, data for joint life annuities and last survivor annuities were created to have the same characteristics as the reversionary annuity data.

8.1.3.2 Parameters

The main objective of this project is the reduction of total execution time: other than testing for correctness of the optimisations during the course of the project, there is no particular interest in the financial results obtained. Therefore, the basis items take values which allow the reserves obtained at each stage to be easily verified, with no particular regard to the values obtained. For example, the interest rates used show a linear progression from month to month, rather than following any particular model which may be applied.

The values are read from files to simulate a situation where regulatory requirements might mean that the same values are used for all competitors in a particular market, thereby allowing comparisons of solvency positions on a predetermined basis. Also, using known values ensures that the financial results obtained from different platforms are identical: if the values had been dynamically generated during the execution of the program, then the use of different compilers, which may implement the random number generators differently, could lead to different sets of parameter values and hence to a different set of results. However, because the basis items would be from a known distribution, these differences in actual parameter values would be eroded by the use of sufficiently large numbers of simulations in the Monte Carlo process.

To account for improvements in future mortality, a separate mortality table is used for each monthly step within each of the 1000 scenarios. To allow for the synthetic data used in this project, there are separate q_x values for each of 2 sexes, for each of 40 years of birth, for each of 120 ages, for each of 780 future time steps, for each of the 1000 scenarios: for the implementation in this project, these mortality rates are read from disk, as are all interest and inflation rates. There are therefore 1000 parameter files which will be discussed further in Section 8.2.2.

8.1.3.3 Algorithm

In order to conform to Smith's interpretation that 'it is essential that interest rates are re-sampled every step' [79], the majority of the results in this chapter are based on

the requirement that parameters are re-drawn at each time step. Also, Section 5.2.4 discussed the fact that the recurrence relation was two orders of magnitude faster than the summation approach for a projection over several hundred monthly steps.

Section 7.3 presented four possible algorithms for determining the ACR under Solvency II, the differences being dependent on whether parameters are re-drawn at each time step, and whether the recurrence relation is used. Algorithm 7.2 calculates in-force reserves using the recurrence approach after re-drawing parameters at each step: this is therefore the algorithm used to produce the majority of the performance results presented in this chapter. Section 8.2.1 presents the implementation of the algorithm and Section 8.4 discusses the performance of that implementation. Performance of an implementation based on Algorithm 7.4 (using the recurrence relation) where, per the possibility discussed in Chapter 7, parameters are not re-drawn, is discussed in Section 8.5.

8.1.4 Outputs

Section 8.2.1 introduces an approach to processing the 1000 scenarios in each future time which uses 1000 tranches, each having one scenario in each future step: this leads to 1000 output files, each containing reserve values in each future time step, summed over all policies in the data set. The output is therefore 1000 files, each of 18 kB.

In order to obtain the capital requirement, a post-processing stage is needed. This straightforward stage has three phases; first, read the 1000 files and populate T arrays, each of length 1000 where T is the number of future time steps; second, perform T sorts, each on 1000 elements; finally output the 5th largest element from each of the T arrays. The array elements output by this method are the 5th largest reserve at each of the T steps, and therefore form the sequence $\{V_t'\}$ required in Algorithm 2.1.

Compared to the processing time required to obtain the 1000 results files, the time for this post-processing is insignificant and is not considered further.

8.1.5 Variability of Timings

As mentioned in Section 1.5, the run time for a brute force calculation in a commercial environment is estimated to be about 2800 CPU core years. To allow a reasonable variability in times, say ±10%, the run time would be 2800 years ± 280 years. Later sections of this chapter report run times that were achieved, on various platforms,

as a result of the work done in this project: all of the times are several orders of magnitude smaller than this estimate. The difference in magnitudes of the run times guarantees that there is no chance of the intervals overlapping and so there is no benefit in obtaining confidence intervals for the run times reported in this chapter. Therefore, to minimise the use of HPC resources, each run has been performed only once and, whilst variability in run times is acknowledged to exist, it is stressed that the times reported are representative, rather than being a mean, mode or median of the overall time.

8.2 Optimisations

The code, as presented in Section 8.1.1, has been designed and written to be particularly efficient so that optimisations at source level, such as those described in Section 5.2.2 are unlikely to have a significant effect. The majority of the optimisations discussed here are at a higher level, and are designed to take advantage of coarser parallelism.

8.2.1 Use of Tranches

A naive approach to implementing code to fulfil Solvency II regulations, as interpreted in Algorithm 2.1, is to perform 1000 simulations at each time before moving on to the next time step. Since, by assumption, the calculations for each step are independent of the calculations for any other step, such an implementation naturally leads to having each time step as the unit of parallelism. However, this approach leads to load imbalance since the earlier steps will require more processing than later time steps. Whilst such imbalance could be overcome by manually adjusting steps processed by each processor, there is an alternative approach which removes the imbalance.

This better approach, which naturally leads to load balance, and permits the possibility that the outer loop over time steps *is* sequential (as discussed in Section 2.3.3.1), is the use of *tranches*. Each tranche is implemented as a task, and each of 1000 tranches performs one simulation in each future time. This leads to the need to collate results at the end of the projection, but that is a trivial post-processing task which, as mentioned in Section 8.1.4, takes a time which is insignificant in comparison to the production of the results to be collated.

The overall loop structure for this approach is as described by Algorithm 8.1, which results from the transformations discussed in Section 4.4.2: i.e. Algorithm 8.1 is an

Algorithm 8.1 Structure of loop nest in C implementation of brute force approach to obtaining the additional capital requirement for Solvency II.

```
 1: set T = index of the maximum projection step                {per Section 3.4.1.1}
 2: for n = 1 to number of tranches do
 3:   for t = 1 to T do
 4:     draw parameters used to calculate a_{t,r,p} and V_{t,r,p}
 5:     for b = 1 to number of blocks do
 6:       for s = T to t (decreasing) do
 7:         for p = 1 to number of policies in block do
 8:           use recurrence to obtain the in-force reserve, a'_{n,s,p}
 9:           calculate the per-policy reserve V_{n,s,p} from the in-force reserve
10:         end for
11:       end for
                                 at this point s = t, so that the reserve is V_{n,t,p}
                          due to re-drawing parameters, all V_{n,s,p} : s ≠ t are discarded
12:       sum the per-policy reserves over the current block, V_{n,t,b} = Σ_{p∈b} V_{n,t,p}
13:     end for
14:     sum the per-policy reserves over all blocks of policies, V_{n,t} = Σ_b V_{n,t,b}
15:   end for
16: end for
---------------------------------------------------------------------------
17: for t = 1 to T do
18:   perform sorting to obtain V'_t, the 99.5^th percentile of the set {V_{n,t}}
19:   obtain C_t = max(V'_t − B_t, 0)
20: end for
21: discount the sequence {C_t} to obtain the Risk Margin
22: obtain the Additional Capital Requirement as fixed percentage θ of the Risk Margin
```

optimised implementation of Algorithm 7.2, where parameters are re-drawn in every future time step. Note that Lines 17 to 22 of Algorithm 8.1 have not been implemented: they form the trivial post processing stage to collate the results.

This approach has been implemented as follows:

- the n loop (over tranches) uses MPI to coordinate a task farm since all tranches may be run independently;
- the t loop (over time steps) is sequential, but it could be decomposed if there are no dependencies of the parameters on the previous t step. In that case, there is scope to increase the number of processes, while maintaining reasonable load balance, by 'folding the loop over' so that there is pairing of time steps and the steps considered by each process have similar work: since T is the maximum projection step, the pairings would be $(1, T)$, $(2, T-1)$, $(3, T-2)$, etc.;
- the b loop (over blocks) is parallelised using OpenMP;

– the s loop (over time steps for the recurrence relation) is inherently sequential due to the dependencies in Equation 3.3.1; and
– the p loop (over policies in the block) is vectorised as much as possible.

Note that the complexity of this implementation is $O(T^2)$, as it must be due to the re-drawing of parameters (per the discussion in Section 8.1.3.3).

8.2.2 Organisation of Scenario Data

Section 8.2.1 discussed the use of tranches, with each tranche performing the calculations for one scenario in each future time step. The implementation developed in this project reads separate interest rates, inflation rates, and investment expense rates for each month, for each scenario within a tranche, from the disk. The independence of the scenarios generates independence of the parameters, and so the implementation combines all assumptions for each tranche into a `.csv` file. Each value is stored in the file in 13.11 floating point notation[1], and ASCII format. There are therefore, in total, 1000 parameter files, each of 148 MB.

Combining the parameters for all time steps into one file has the added advantage of requiring fewer file handles and less file-checking housekeeping than having them all separated. The ARCHER Best Practice Guide [27] suggests that 'accessing several files from the same directory is not particularly efficient'. Therefore, this project's implementation uses 1000 directories, each with one parameter file. Although this approach does not have any clear benefits on other platforms, it is used without alteration to enhance portability of the implementation.

8.2.3 Re-calculation of Benefits

In order to calculate any reserve, which by definition is 'the amount held in order to meet future liabilities', it is necessary to know what the future cash flows are. The cash flows which constitute the benefits payable to the holder of an annuity policy form part of the $\mathbf{c}_{\mathbf{x},t}$ element of Equation 3.2.6: the remainder of $\mathbf{c}_{\mathbf{x},t}$ is made up of the expenses attributable to the administration of that policy. For every policy, the benefits are invariant over the Monte Carlo simulations, and so it would be possible to calculate benefits only once, at the outset of the projection.

[1] 13 characters, of which 11 follow the decimal point: being mortality and interest rates, the values are all $0.\cdots$ In the example scenarios used in this project, all values are positive, so there is no need to allow another character to hold the sign.

However, calculating the payments to be made to each policy for each step only once, and retaining those values, requires a significant amount of memory: for example, half a million policies, allowing a maximum projection term of 65 years, using monthly steps, requires around 3 GB. If the MPI which controls the processing of the tranches were to use a replicated data approach, then 3 GB per rank becomes significant. Alternatively, if the MPI used a distributed data approach, then the communications pattern to access the correct data at the correct times would lead to significant communication overhead. Therefore, at the start of every iteration of the b loop (in Algorithm 8.1) the benefits payable to each policy within the block are re-calculated: on modern machines, the cost of compute is small compared to the cost of moving data, so that repeated re-calculation is unlikely to be wasteful.

A simple algorithmic optimisation was to calculate the benefits only for steps `s=t:T` (where `T` is the index of the maximum projection step), rather than for `s=1:T`. However, the fact that the synthetic data used in this project reflect data from the commercial environment meant that the payment amount contained in the data relates to the *next* payment, rather than the previous payment. In order to be able to escalate the payment amount in the same way at any step (thereby removing the if-test which checks whether inflation should be applied), the data were read in and changed to remove any escalation applied: the additional calculational effort of winding back any escalation applicable to the step prior to the valuation date is minimal when offset against the saving in the overall run time.

This straightforward change led to an overall performance improvement of about 15%.

8.2.4 Length of Blocks

To allow for rapid developments in hardware, and increase portability of the code, it is desirable to allow for different CPUs having different vector lengths, and hence different platforms having different optimum block lengths. The ability to tune the block size in loops in which vectorisation is used is encapsulated by including the block length as a compile-time constant. Although this requires the code to be recompiled when the parameter is changed, recompilation time is only a few seconds and so is insignificant as a consideration, especially compared to some scientific codes which take of the order of hours to compile. Also, having the parameter as a compile-time constant, rather than a run-time argument, provides the compiler with information which it could use to perform data alignment, or other optimisations which relate to the data transfers.

The performance benefits of different block lengths are discussed in Section 8.2.5 where the parameters are tuned for each of the different machines.

8.2.5 Work Allocation: 'processes *vs.* threads'

For a fixed number of cores on a CPU, it may initially seem preferable to use OpenMP threads to MPI processes since threads do not need to hold copies of the entire address space. This reduction in memory usage means that more threads can fit into a fixed amount of overall memory when compared to the number of processes which would fit that space. Also, the larger number of threads leads to a relatively smaller cache footprint per thread, which is beneficial because of the reduction in movement of data, through the cache, required to keep the threads busy. However, the problem with increasing the number of threads is synchronisation; a larger number of threads could lead to an increase in time spent at barriers. Therefore, there is a clear argument for fewer threads and more MPI processes.

Hence, in order to determine the optimum combination of block size, number of MPI processes and number of OpenMP threads, a subset of tranches was run on each of the machines: the results of those runs are discussed in the following sub-sections.

8.2.5.1 Cray XC30

On the Cray, 96 tranches were processed using 4 nodes, i.e. 96 cores, and various combinations of block size, number of processes, and number of threads. The use of 96 cores to process 96 tranches leads to a balanced load for a range of combinations of processes and threads, thereby ensuring that load imbalance cannot cloud the conclusion about which combination leads to best performance. The times for this collection of runs are shown in Table 8.1.

For smaller block sizes, one OpenMP thread per MPI process leads to best performance. However, this has a disadvantage in terms of load imbalance and scalability: the nodes on the Cray, like many HPC machines, have a number of cores which is not a divisor of the 1000 tranches required for the approach proposed in this project. Therefore, spreading 1000 tranches over 24-core nodes (with one core per process) results in 42 nodes being required, with one not being fully populated, and this leads to two problems. Firstly, since one node is not fully populated, this use of pure MPI should be expected to lead to load imbalance: using fewer processes, each with more threads, is a simple way of overcoming such imbalance, even if that results in marginally worse

MPI Processes	OpenMP Threads	Block Size 8	16	32	64	128	256
4	24	2263	2191	2258	2210	3559	4747
8	12	1984	1887	1927	1928	3125	4425
16	6	2080	1752	1767	1804	2815	4138
24	4	1818	1700	1721	1740	2803	4109
32	3	2438	1703	1695	1715	2783	4203
48	2	1864	1673	1686	1721	2740	4067
96	1	1738	1633	1656	1926	2912	4326

Table 8.1 Run times (in seconds) to process 500,000 policies, over 96 tranches using various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 4 nodes of the Cray XC30.

performance. Secondly, and more importantly, the number of cores required by pure MPI is fixed: if more cores are available, for example because more nodes are available, then cores in excess of the number of ranks will be unused.

For all block sizes, one MPI process per node leads to worst performance: this is consistent with expectation because the 24 threads would be spread over the two NUMA regions on the node. Other than the case of one MPI process per node (which can be discarded because it has worst performance), block sizes of 16 and 32 have best performance for all combinations of processes and threads: the difference in time of these block sizes is likely to be within the variability of the run times. Also, for most block sizes, larger numbers of threads have longer run times than smaller numbers: this is likely to be due to the threads starting to compete for bandwidth when accessing Level 3 cache. However, with the exception of 8 processes and 3 threads per node, each combination of processes and threads has its minimum run time for a block size of 16: the exception reported here is likely to be noise, resulting from the variability of timings. Therefore, it seems reasonable to deduce that best performance is obtained for a block size of 16.

For a block size of 16, there is not a significant difference between the minimum time (using 1 thread) and the times using 3 or 4 threads. However, by using 3 or 4 threads per process, it is possible to utilise 3000 or 4000 cores with almost perfect efficiency. For the exploratory runs summarised in Table 8.1, using 8 processes each with 3 threads, and a block size of 16, takes only 4.5% longer than the optimal combination. Note that 3 threads per process, with 8 processes per node, has a slight advantage in that there will be no unused cores as the number of processes per node is a divisor of the number of tranches. It is also possible to use higher numbers of cores by reducing the number of processes per node with a corresponding increase in the number of threads per process:

MPI Processes	OpenMP Threads	Block Size 4	8	16	32	64	128
2	8	5379	5056	4912	4715	4831	7367
4	4	5297	4966	4810	4616	4727	6882
8	2	5284	4886	4762	4538	4673	6812
16	1	5320	4891	4910	4509	4649	5772

Table 8.2 Run times (in seconds) to process 500,000 policies over 32 tranches, using various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 1 node of the Xeon cluster.

while this leads to a slight worsening of the time to process each tranche, more tranches are processed concurrently, leading to a shorter overall 'time to solution'.

The Xeon processors in the Cray allow simultaneous multi-threading (SMT). In order to assess whether SMT benefits this code, 8 tranches were run on one node using 8 MPI processes, each with 3 OpenMP threads and SMT disabled: the time (including I/O) was 551.4 sec. When SMT was enabled and the number of threads was doubled, the time (including I/O) to process the 8 tranches was 536.5 seconds, indicating a speedup of 1.03× from SMT. The fact that the two submissions were in the same script ensured that the same node was used, thereby eliminating some forms of timing variability, e.g. nodes running at different speeds, and so it seems there was some small benefit in using SMT. This small improvement agrees to the fact that the Ivy Bridge architecture is known not to benefit too much from SMT. The fact that the benefit on the Cray is small, allied to the fact that not all CPUs implement SMT, meant that the use of SMT was not pursued further: none of the other timings in this thesis were taken with SMT enabled.

The times for the Cray to perform a complete brute force run, consisting of 1000 tranches, are discussed in Section 8.4.1.

8.2.5.2 The Xeon Cluster

In order to determine the optimal combination of processing parameters on the Xeon cluster, a modest number of tranches was processed using 1 node and various combinations of block size, number of processes, and number of threads. For each combination, all 500,000 policies in the synthetic data were processed, while the number of tranches was 32 to allow for full utilisation of the node. The times for this collection of runs are in shown Table 8.2.

Runs using 1 process with 16 threads were not performed for any block size because spreading the OpenMP across 2 NUMA regions is expected to lead to longer runs times than those where the OpenMP threads are kept within 1 NUMA region: this was the case on the Cray, as shown by the results in Table 8.1.

For all combinations of processes and threads, a block size of 32 leads to best performance: this differs to the Cray, where the optimum block size was 16. This difference is likely to be a result, primarily, of the use of different compilers on the different machines, but the characteristics of the different CPUs may have some effect.

As with the Cray, pure MPI led to best performance, but this has the same disadvantage in terms of load imbalance and scalability: the nodes on the Xeon cluster have 16 cores, and hence perfect load balance can only be obtained through the use of 2, 4 or 8 MPI processes per node. Using 4 processes per node allows up to 250 nodes to be used, whereas using pure MPI limits the number of nodes to 63. From this collection of processing parameters, the minimum run time results from using 8 processes with 2 threads each; the time of 4538 seconds is only about 0.6% longer than the shortest time, but as with the Cray, this combination of processes and threads allows greater numbers of nodes to be used than would be possible with pure MPI, i.e. up to 125 nodes may be used.

8.2.6 Vectorisation

The vectorisation of calculation intensive loops is a standard method of improving the speed of a program, and it is generally desirable to vectorise as many such loops as possible. Many loops are trivial to transform to a form which the compiler is able to vectorise. However, some loops need further manipulation in order that they may be vectorised. The calculation of the death probabilities for each policy is a clear target for vectorisation, but there are two problems with a naive implementation.

Firstly, because not all lives are the same age at the valuation date, the number of q_x's required varies on a policy-by-policy basis and the number of iterations through the loops varies accordingly. In order to overcome this, redundant calculations were added: rather than stopping the calculation process when the policyholder reaches age 120, q_x's are calculated for all policies up to the maximum time step implied by ages in the data file and setting $q_x = 1$ for $x > 120$, thereby imitating a standard mortality table.

Secondly, because mortality rates can vary by year of use, an intuitive implementation uses a two-dimensional array to hold the mortality rates, i.e. the table is indexed by age

Block Length	Upper Bound declared as Standard Variable	Macro Variable
Optimal, with `$SIMD` pragmas	4538	4493
Optimal, but no pragmas	4919	4695
1	8152	
Blocking removed	5498	
Redundant Calcs removed	5001	

Table 8.3 Run times (in seconds) to process 500,000 policies over 32 tranches using various block sizes, on 1 node of the Xeon cluster.

and calendar year. The Intel compiler could not vectorise the calculations when two-dimensional indexing was used. Therefore, a function was created to obtain a mortality rate by converting the two-dimensional indices to a one-dimensional index: by calling this function, the calculation of mortality rates could be vectorised. Also, because the function only has a small body, it was inlined by the compiler.

8.2.6.1 The Effect of Vectorisation

Vectorisation was built into the brute force code as it was developed: its effect (as measured by its removal) is shown in Table 8.3. The fully vectorised code is constructed using loops whose lengths are the block sizes which were tuned for optimality in Section 8.2.5. There are two possibilities for specifying the loops bounds, i.e.

i) use of a macro variable, e.g. `for p = 1 to BLOCK_SIZE` where `BLOCK_SIZE` has been `#define`'d, or
ii) use of a standard variable, e.g. `for p = 1 to block_len` where `block_len` is calculated for each block.

Where the optimal block size is used, the times using both possibilities for specifying the upper bound have been measured.

The code was developed using the Intel compiler: this had the advantage of allowing the production of large quantities of information about which loops were vectorised, and the reasons why others were not. The outcome is that, in the final code, as many loops as possible are vectorised. The use of Intel's compiler had a useful side effect: by using Intel's `#pragma SIMD`, it is possible to measure the effect of the pragmas to see how much benefit the guidance to the compiler was. It is apparent from Table 8.3 that using these pragmas leads to the best performing code, and that it does not matter whether the loop bounds are specified using macro variables or standard variables.

Removing the pragmas shows several things. Firstly, when the upper bound is specified with a standard variable, there is a speedup of about 1.08×, indicating that the pragmas do aid the compiler. Secondly, when the upper bound is specified with a macro variable, there is a speedup of about 1.04×: this reduction is speedup is likely to come from the compiler not being able to make optimisations which result from knowing the block length, or equivalently, the loop's trip count. Thirdly, when pragmas are not used, using a macro variable as the upper bound leads to a speedup of roughly 1.05×: this speedup is also likely to come from optimisations which the compiler can make as a result of knowing the block size.

Despite the performance advantages of using a macro variable for the upper bound, there is a clear disadvantage to this approach: the number of policies processed must be a multiple of the block size. However, in practical terms, this is not a problem because if there is a partial block then it may be padded with null data, e.g. by setting the payment amount to zero.

Using a block length of 1 led to the run time increasing by a factor of about 1.5×: this increase is likely to be partly due to the number of calls to subroutines increasing (since each call only processes one policy), and may be partly due to the overhead associated with having many trips through loops, each of which has a trip count of 1. The fact that the upper bound is specified as a standard variable means that the compiler cannot absorb the overhead, whereas it might have been possible for the compiler to optimise the loop away if the bound had been a macro variable.

The complete removal of blocking leads to the same run time as using blocks of optimal length, but without pragmas. This might indicate that vectorisation only benefits this code when the compiler is explicitly told that the loop is vectorisable, i.e. the code is more complex than the compiler's heuristics are designed to handle.

The difference in times between code without blocking and code also having redundant calculations removed is roughly 1.6%. Section 2.5.5.4 discussed the addition of redundant calculations to improve the performance of the vectorised code, but these results indicate that adding redundant calculations also improves performance of the non-vectorised version: this may be due to improved optimisation resulting from removing `if`-tests. However, the difference in times is within level of noise and is, therefore, not significant.

Overall, the difference between blocking with optimal blocks and SIMD pragmas, and a complete absence of blocks is a speedup of 1.21×. The fact that the selection of compiler switches led to vectorisation being enabled in both cases indicates that vectorisation

is not particularly beneficial to this code on this CPU architecture. However, the fact that the code is vectorised adds some degree of 'future-proofing' to the code in two ways. Firstly, the memory layout is as required to permit memory coalescing on GPUs: the possibility of porting this code to GPUs is discussed in Section 9.3.1. Secondly, Intel's Xeon Phi processors have longer vectors and so vectorisation is likely to be more beneficial there: the possibility of porting this code to Xeon Phis is discussed in Section 9.3.2. Additionally, although vectorisation and the use of the SIMD directive is a different concept to the use of blocks, the addition of blocking allows the lengths of the block to be changed in order to minimise the runtime of the program.

In Section 2.5.2.3.4 it was mentioned that the benefit of SMT on a Xeon CPU comes from the ability to fill empty instruction slots. The lack of benefit of SMT to this code when running on Xeon CPUs suggests that there are not many such empty slots, or some hardware resource, such as cache bandwidth, cache latency or cache capacity, is saturated: see Section 8.4.1.2 for a discussion of the performance of this code on the Xeons in the Cray.

8.3 Simplifications

The changes discussed in Section 8.2 are optimisations; they improve the performance of the code without changing the results. The changes discussed in this section are approximations; they improve the performance of the code at the expense of producing marginally different results.

8.3.1 Change of Interpolation Methodology

Chapter 6 discussed the effect of changing the interpolation method used, from the commercial approach using cubic interpolation in the l_x's, to a simplification based on linear interpolation in the q_x's. There, it was shown that the effect on the monthly death probabilities, over the time range where discounting is not completely dominant, was small. The change in probabilities also led to an insignificant change in the reserves required, as discussed in Section 6.7.2.

Chapter 6 also indicated that the simplification would lead to a reduction of the number of data retrievals and arithmetic operations required to calculate the monthly death probabilities: this reduction in complexity led to a speedup of 2.4× for the interpolation routine, and 1.8× for the entire program.

8.3.2 Timing of Cash Flows

A policyholder may choose to have their benefits paid on any day of the month so that the discount factor to the time of the cash flow, from the start of the month, is v^α where $\alpha \in \left\{\frac{1}{31}, \frac{2}{31}, \cdots, 1\right\}$. The implementation in this project allows each projection step to use a different monthly interest rate, and hence the sequence $\{v^\alpha\}$ needs to be obtained in every step. Even using the method of repeated multiplication, as discussed in Section 5.2.2.1, populating the sequence takes time because obtaining the first element, $v^{\frac{1}{31}}$, requires evaluation of a power for a non-integral exponent.

It is possible to simplify the calculations by assuming that all cash flows occur either at the start of the step (so that $f = 0$ and hence $v^f = 1$), or at the end of the step (so that $f = 1$ and hence $v^f = v$). In either case, the discount factor does not require any calculation and hence the approximated result is obtained more quickly than the true result. However, both of these possibilities introduce a systematic bias to the financial results.

Therefore, to remove the bias, the assumption that all cash flows happen mid-step could be made. This has the additional benefit that the only discount factor required in any step, since it applies to all policies, is $\sqrt{v}$, and that may be computed reasonably quickly: most compilers have an efficient implementation of `sqrt()`, and performance may be improved further on platforms where an efficient hardware implementation is available.

Assuming that cash flows are mid-step has a performance advantage in terms of survival probabilities: the probability of dying before reaching the middle of a monthly step is

$$_{f}q_x = {}_{\frac{1}{24}}q_x$$

which, under the assumption of uniform distribution of deaths, is

$$\approx \frac{1}{2}\,{}_{\frac{1}{12}}q_x$$

Under this construction, the simplification in Equation 6.6.2 only needs to be evaluated once (for the full step): the mid-step probability may then be trivially obtained through a single multiplication.

The time (including I/O) for the Cray to process all 500,000 policies using 1 MPI process and 3 OpenMP threads (as discussed in Section 8.2.5.1) was 556.1 sec with cash flows assumed to be in the middle of the step, and 1284.0 sec with accurate cash

flows, indicating a speedup of $2.3\times$ from this approximation.

It is not clear how a single measure which shows the effect of this simplification should be derived: the fact that all reserves at all time steps are affected clouds the problem. However, as an indication, the total initial reserve[2] for the 300,000 single life policies increases by a relative amount of 1.88×10^{-5}, which is immaterial, and all reserves at all other time steps show a correspondingly small change.

8.4 Performance when Re-Drawing Parameters

The optimisations discussed in Section 8.2 and the approximations discussed in Section 8.3 have all had an effect on the overall run time, but each effect was mentioned in isolation. This section discusses the overall performance of the code on two platforms, a Cray supercomputer and a Unix cluster containing Xeon CPUs. The time on the Cray is taken as the benchmark: as a result, there is some discussion about the profile of the code in the Cray.

8.4.1 Cray XC30

Section 8.2.5.1 concluded that using 8 processes per node, each with 3 threads, gave near optimal performance while permitting perfect load balance and potentially allowing the use of a large number of nodes: this combination of processes and threads therefore seems preferable to other combinations. However, for some node counts, using 8 threads will not lead to an equal number of tranches per process: in these cases, the number of processes is chosen so that an equal number of tranches is processed by each process, and the number of threads is chosen to fully populate each node with one thread per core. The times for a complete run of 1000 tranches have been obtained for a range of numbers of nodes, and the results are shown in Table 8.4: these results are discussed in Section 8.4.1.1 and the performance of the Cray while producing these results is discussed in Section 8.4.1.2.

[2] The 'initial reserve' is the reserve required at the start of step 1. Note that, because the policy *is* in the data file, the probability of being in force at the start of step 1 is 1, the per-policy reserve is the same as the in-force reserve at that point.

Number of Nodes Number of Cores	5 120	10 240	25 600	50 1200	125 3000	250 6000	500 12000
MPI Processes per Node OpenMP Threads per Process	8 3	4 6	8 3	4 6	8 3	4 6	2 12
Tranches per Process	25	25	5	5	1	1	1
Wall Clock Time (sec)	13875	7285	2863	1461	571	294	155
Portion of Run Time in I/O	2.1%	4.1%	2.1%	4.0%	2.2%	4.2%	8.2%

Table 8.4 Run time (in seconds) to process 500,000 policies over 1000 tranches using a block size of 16, and various numbers of nodes on the Cray XC30.

8.4.1.1 Performance

The run using 500 nodes indicates that, given enough hardware for every tranche to be processed concurrently, the entire brute force calculation can be performed in under 3 minutes. However, the expense of purchasing and operating a 500 node Cray is unlikely to be justifiable in most life offices unless other uses can be found for its computational capabilities. Similarly, 500 nodes equates to 12,000 cores, and very few offices are likely to obtain clusters of this size. Therefore, although the calculations *can* be performed in such a short time, it seems doubtful that, in reality, they *will* be.

The run using 125 nodes indicates that the entire brute force calculation can be performed in less than about 10 minutes, i.e. within a coffee break. However, on our Cray, which is heavily used by the UK academic community, it takes significantly longer than this for the job to navigate to the top of the queue. The fact that 125 nodes equates to 3000 cores places this amount of hardware within the realms of clusters which could be contemplated by many offices, so that the waiting in the queue of a heavily used machine becomes far less of a perceived obstacle.

25 nodes equates to 600 cores: this amount of hardware is similar to that which some of the larger life offices have available, and so it is possible for those offices to perform the calculations in under 1 hour using Xeon CPUs.

10 nodes equates to 240 cores: this is similar to the quantity of hardware many life offices have available, and so it is possible for most offices to perform the calculations in about four hours on the wall clock using Xeon CPUs. In general, using a few hundred cores, most life offices could perform the required calculations using the processing scheme presented here.

5 nodes equates to 120 cores: this amount of hardware is easily affordable, and so there is no reason why all offices could not perform the calculations in, what is effectively, an

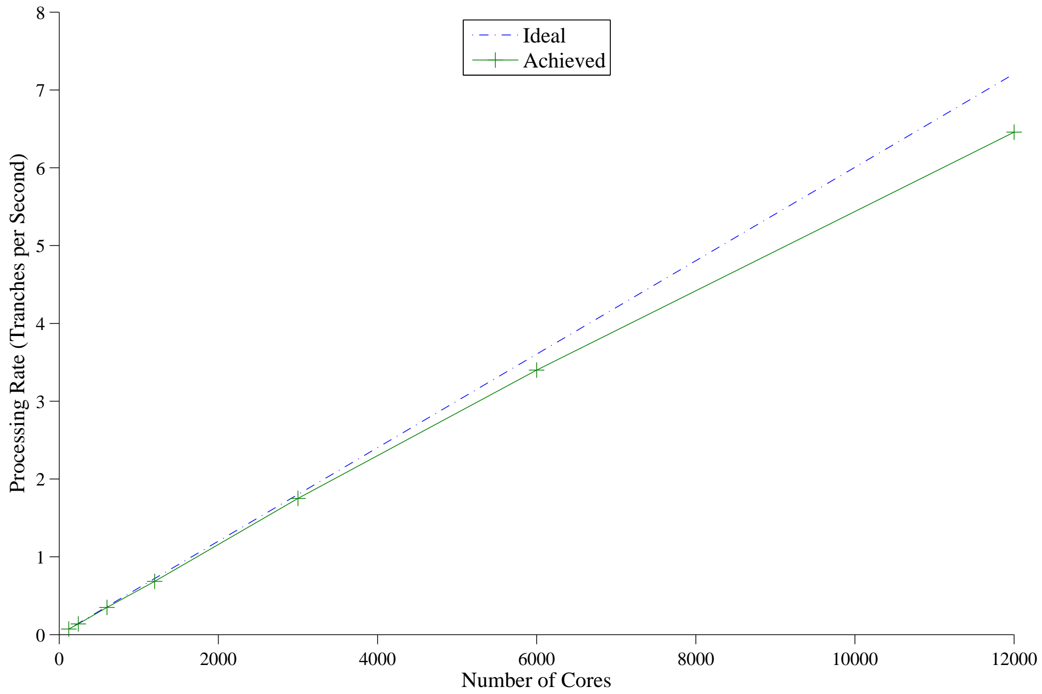


Figure 8.1 Strong scaling on the Cray XC30: the ideal processing rate is based on the time using 5 nodes (as presented in Table 8.4), i.e. including I/O, for 1000 tranches, using a block size of 16.

overnight run.

Figure 8.1 shows the scaling on the Cray: the expected performance is based on the time taken for the entire brute force calculation using 5 nodes, i.e. 120 cores. The deviation from the expected rate is larger where the number of threads is larger: this is consistent with the discussions in Section 8.2.5.1 which highlighted that run time increased with number of threads per process.

The portion of run time in I/O in Table 8.4 may be summarised as

Portion of Time in I/O	$\approx 2\%$	$\approx 4\%$	$\approx 8\%$
Threads per Process	3	6	12

This is consistent with an increasing number of threads reducing the time to perform calculations, making the time to perform the fixed amount of I/O a larger portion of the run time.

```
   Samp% |     Samp |  Imb. |  Imb. |Group
         |          |  Samp | Samp% | Function
         |          |       |       |  PE=HIDE
         |          |       |       |   Thread=HIDE

  100.0% | 29470.0 |     -- |    -- |Total
 |-----------------------------------------------------------------
 |  95.4% | 28112.2 |     -- |    -- |USER
||-----------------------------------------------------------------
||  54.6% | 16076.0 | 142.0 |  1.0% |calcQx_allSteps_block
||  13.7% |  4034.5 | 107.5 |  3.0% |annuitySL_calcReserves_block
||  10.5% |  3107.8 | 230.2 |  7.9% |annuityXX_calcBenefits_block
||   7.8% |  2286.5 |  60.5 |  2.9% |annuityRA_calcReserves_block
||   5.6% |  1658.8 |  44.2 |  3.0% |annuityLS_calcReserves_block
||   2.9% |   855.5 |  37.5 |  4.8% |annuityJL_calcReserves_block
||=================================================================
 |   3.9% |  1154.5 |     -- |    -- |ETC
||-----------------------------------------------------------------
||   1.9% |   568.0 |  27.0 |  5.2% |__isoc99_fscanf
||   1.2% |   361.8 |  35.2 | 10.1% |____strtod_l_internal
 |=================================================================
```

Figure 8.2 Summary sampling results from CrayPAT when processing 50% of the data over 8 tranches using 8 processes, each with 3 threads, and a block size of 16.

8.4.1.2 Profile

The Cray XC30 has CrayPAT, Cray's 'Performance Analysis Tools' [19], available. CrayPAT may be used as a sampling profiler, but has additional capabilities. The results from any sampling must be viewed with caution: the sampling errors mean that the information portrayed is only indicative. However, despite this caveat, the discussion below is relevant in its order of magnitude of the performance achieved by the Cray when running this code. The sampling was performed while processing 50% of the data over 8 tranches using 8 processes, each with 3 threads on a single node of the Cray. The results are shown in Figure 8.2, which shows that about 55% of the time is spent in the routine which performs interpolation to calculate the q_x's. Therefore, despite the optimisations, and the adoption of the simplification introduced in Section 6.6, the calculation of survival probabilities is still the dominant part of the program's execution.

The re-calculation of benefits, as discussed in Section 8.2.3 accounts for about 10% of the time. The fact that this routine relates to all annuity policies suggests that any time spent further optimising this routine will only benefit annuity policies: this is in contrast to the interpolation routine which will be used by other policy types as they are implemented. The remaining four dominant routines in Figure 8.2 relate to the

```
==============================================================================
Total
------------------------------------------------------------------------------
CPU_CLK_UNHALTED:THREAD_P                          850855016169
CPU_CLK_UNHALTED:REF_P                              29363425998
DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                    74827744
DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                    3582966
L1D:REPLACEMENT                                     34977227601
L2_RQSTS:ALL_DEMAND_DATA_RD                         31839808309
L2_RQSTS:DEMAND_DATA_RD_HIT                         24416064737
MEM_UOPS_RETIRED:ALL_LOADS                         572199371747
FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE                  323187260991
FP_COMP_OPS_EXE:X87                                   237961551
FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE                   57338067
SIMD_FP_256:PACKED_DOUBLE                          251029051343
PM_ENERGY:NODE                  40.046 /sec               11943 J
User time (approx)             298.224 secs        805502887165 cycles
CPU_CLK                          2.898 GHz
HW FP Ops / User time         4451.876 M/sec      1327656104048 opsS 20.6%peak(DP)
Total DP ops                  4451.876 M/sec      1327656104048 ops
Computational intensity           1.65 ops/cycle           2.32 ops/ref
MFlop/s (aggregate)           35615.01 M/sec
TLB utilization                7297.46 refs/miss         14.253 avg uses
D1 cache hit,miss ratios         93.9% hits                6.1% misses
D1 cache utilization (misses) 16.36 refs/miss             2.045 avg hits
D2 cache hit,miss ratio          78.8% hits               21.2% misses
D1+D2 cache hit,miss ratio       98.7% hits                1.3% misses
D1+D2 cache utilization          77.08 refs/miss          9.635 avg hits
D2 to D1 bandwidth            6516.404MiB/sec   2037747731776 bytes
==============================================================================
```

Figure 8.3 Statistics from CrayPAT when processing 50% of the data over 8 tranches using 8 processes, each with 3 threads, and a block size of 16.

calculation of reserves for each type of policy: these are the routines which implement the recurrence approach to calculating the reserves, as introduced in Chapter 3. The amount of time spent in each routine approximately reflects the number of each type of policy, although there is a slight effect on the two-life policies caused by requiring differing numbers of factors, depending on the policy type.

The fact that ancillary routines (such as reading data) take minimal time is expected from the fact that this project was based on the premise that the calculation was beyond contemplation, even though the I/O is achievable: the fact that the profile is dominated by calculation reinforces the suggestion that I/O is not currently a major concern.

Figure 8.3 shows that the code achieves roughly 20% of the peak Flop rate, and has a computational intensity of about 2.3 operations per reference. A survey of scientific

codes running on Cray XT4's [4] showed that many of those codes achieve in the region of 5% to 15% of peak performance: only those codes which are heavily reliant on Fourier transforms perform significantly better. While the results quoted in that survey are for Opteron CPUs, rather than the Xeons in our Cray, there is some evidence that the code from this project does perform quite reasonably.

Additionally, memory is loaded in the Cray in pages of 4 kB so that 512 double precision numbers are moved in each transfer. Therefore, if each number were to be used once, then the expected TLB utilisation would be 512 references per miss: this code achieves about 7300 references per miss, suggesting that the code has good reuse of data once it is loaded. This suggestion is reinforced by the fact that the D1 cache hit ratio is about 94%, and that the D1+D2 cache hit ratio is almost 99%. The fact that this code is particularly good at using data in cache is to be expected: it results from tuning the block sizes to the machine, allowing the optimal block size to be found.

8.4.1.3 Analysis of Performance

In order to verify the profile information in Section 8.4.1.2, further tests were performed: those tests are considered in this section.

As discussed in Section 8.2.1, MPI has been used to coordinate a task farm across tranches since they may be run independently: that independence ensures that there is no communication between processes and hence any investigation into performance need only consider a single node. Also, the two sockets on a node have independent memory hierarchies, and the runs were performed using fewer OpenMP threads than the cores in a socket, so that processes on each of the sockets are independent. Therefore, an investigation into performance need only consider a single socket.

8.4.1.3.1 Using Global Variables to Count Flops

It is possible to estimate the number of Flops performed by the program: this may be achieved by finding the average outstanding term of a policy, and working through the algorithm and estimating the number of operations per month per policy. However, it is just as straightforward (and time efficient) to add global counters to the program and increment them every time operations of the relevant type are performed: this is the approach taken by this project.

Due to the number of operations, and running time, processing the complete data set over 1000 tranches, counting all operations is excessive. It suffices to count the

Total Flops	Calls to pow()	floor()	sqrt()
3.885×10^{12}	2.242×10^{6}	9.834×10^{10}	8.760×10^{9}

Table 8.5 Count of operations and function calls while processing 50% of the data, over one tranche, running on one node of the Cray, with the frequency set to 2.7 GHz using the `--p-state` option for `aprun`.

operations for 50% of the policies for one tranche, running on one core of one node: this gives a good indicator of the performance of the Cray while running this code, and this setup corresponds to the information presented in Figure 8.3. Note that, in order for the variables not to wrap round to negative numbers, the type for the counters must be `unsigned long long` which, on the Cray, has an upper limit of $2^{64} - 1$. The results of this manual instrumentation are shown in Table 8.5.

The run time was 900.0 seconds which, given the number of operation in Table 8.5, equates to a rate of 4.316×10^9 Flops/sec. This is in close agreement to the value of 4.452×10^9 Flops/sec produced by CrayPAT, and shown in Figure 8.3. The Xeons in the Cray can perform 8 Flops per cycle: with the frequency fixed at 2.7 GHz, this leads to a peak Flops per core of 21.6×10^9. Therefore, the achieved performance for this sample is 19.98% of peak which is also in close agreement to the value of 20.6% produced by CrayPAT.

8.4.1.3.2 Validating CrayPAT using Pure MPI

In order to further understand the performance of the Cray, profiling runs were performed on a single socket with differing numbers of processes running.

The Xeon CPUs in the Cray provide 'turbo boost' when the chip is not fully utilised. However, as more cores are used, and the chips run hotter, the frequency reduces to its nominal value, thereby ensuring that the chip does not overheat. In order to remove this effect, the frequency of the chip was restricted to its nominal value of 2.7 GHz using the `aprun` parameter, `--p-state`. Pertinent metrics from these runs are shown in Table 8.6, where the information is straight from CrayPAT (albeit presented with a smaller number of significant digits): it should be noted that a) the CrayPAT documentation states that "with the exception of the aggregate Flop rate, the results for runs using more than one process are averaged over the processes" and b) that the averaging occurs within CrayPAT, before the results are output.

Using the results in Table 8.6, several observations may be made from the fact that each process is running in a single-threaded manner. Firstly, the wall clock times are

Item		Processes 1	2	3	6	12
CPU						
Frequency	GHz	2.7	2.7	2.7	2.7	2.7
Time						
Overall Wall	sec	1784	1783	1787	1793	1789
Read Data	sec	3.5	3.3	3.8	5.3	2.1
Read Parameters	sec	12.8	10.5	10.8	10.9	10.6
Operations						
Total Number	$\times 10^{12}$	8.178	8.197	8.192	8.194	8.195
Number Vectorised	$\times 10^{12}$	1.549	1.553	1.552	1.552	1.552
Number Scalar	$\times 10^{12}$	1.963	1.963	1.963	1.964	1.963
Aggregate Flop Rate	$\times 10^{9}$/s	4.59	9.20	13.77	27.51	55.13
Fraction of peak Flops		21.3%	21.4%	21.3%	21.3%	21.3%
Cache						
D2 to D1 bandwidth	GiB/s	6.246	6.255	6.240	6.234	6.246
D2 cache hit rate		95.9%	95.4%	95.7%	95.7%	95.6%
L3 cache hit rate		100.0%	99.8%	99.7%	99.6%	99.6%
L1 cache misses	$\times 10^{9}$	177.90	176.87	177.54	177.51	177.37
L2 cache misses	$\times 10^{9}$	13.72	13.62	13.34	13.53	13.58
L3 cache misses	$\times 10^{6}$	5.03	31.71	35.49	47.85	55.86

Table 8.6 Metrics from CrayPAT using single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, one thread per process, all processes on same socket, CPU frequency fixed using `--p-state=2700000`.

near-constant: this is consistent with the fact that each process-count is for a run where the specified number of processes each processes exactly one scenario. Secondly, as an immediate consequence, the aggregate Flop rate increases as an almost linear function of the number of processes. Next, the near-constant execution time indicates that there is no significant contention for shared resources such as memory bandwidth, L_3 bandwidth, or L_3 capacity. Next, the L_3 hit rate is high but slowly decreasing: the slow decrease is expected from the fact that L_3 is shared between all cores on a socket so that, as the number of processes using the cache increases, the probability of the required data being there decreases. Further, the number of L_3 misses is several orders of magnitude smaller than the number of L_1 or L_2 misses: this indicates good reuse of L_3 cache. Finally, the number of operations is roughly the sum of i) the number of scalar operations and ii) four times the number of vectorised operations: this is expected from the fact that vector units in Ivy Bridge chips can perform four double precision operations at the same time.

Based on the results in Table 8.6, it is possible to derive several further metrics: those metrics are obtained as follows, and shown in Table 8.7:

i) fraction of flops which were vectorised $\equiv \dfrac{4 \times \text{number of vectorised ops}}{\text{total number of ops}}$
   where 4 is the width of the Ivy Bridge vector units.

ii) flops per L_1 miss $\equiv \dfrac{\text{total number of operations}}{\text{number of } L_2 \text{ requests}}$

iii) flops per L_2 miss $\equiv \dfrac{\text{total number of operations}}{\text{number of } L_3 \text{ requests}}$

iv) a) time waiting for L_1 misses $\equiv$ number of L_1 misses $\times \dfrac{L_2 \text{ latency}}{\text{cycles per sec}}$
   where, per Appendix B, the L_2 latency is 12 cycles

   b) fraction of time waiting for data from $L_2 \equiv \dfrac{\text{time waiting for } L_1 \text{ misses}}{\text{overall wall time}}$

v) a) time waiting for L_2 misses $\equiv$ number of L_2 misses $\times \dfrac{L_3 \text{ latency}}{\text{cycles per sec}}$
   where, per Appendix B, the L_3 latency is 30 cycles

   b) fraction of time waiting for data from $L_3 \equiv \dfrac{\text{time waiting for } L_2 \text{ misses}}{\text{overall wall time}}$

A summary of the results in Tables 8.6 and 8.7 is that: the code is achieving better than 20% of peak performance; there is a good level of reuse of data once it is loaded into L_3 and L_2 cache which, as mentioned in Section 8.4.1.2, results from tuning the block size to achieve optimal performance; about 75% of operations are vectorised; about 45% of the overall time is spent waiting for data from L_2 cache; and about 8.5% of the overall time is spent waiting for data from L_3 cache. It is therefore apparent that

Processes	1	2	3	6	12
Fraction of Flops vectorised	75.77%	75.81%	75.78%	75.76%	75.75%
Flops per L_1 miss	43.77	43.85	43.83	43.84	43.84
Flops per L_2 miss	595.9	601.8	614.2	605.5	603.6
Time waiting for L_1 misses					
Wall Clock (sec)	790.7	792.2	791.0	790.1	791.3
Fraction of Overall Wall Time	44.33%	44.45%	44.26%	44.09%	44.24%
Time waiting for L_2 misses					
Wall Clock (sec)	152.5	151.3	148.2	150.4	150.8
Fraction of Overall Wall Time	8.55%	8.49%	8.29%	8.39%	8.43%

Table 8.7 Derived metrics for single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, one thread per process, all processes on same socket, CPU frequency fixed using `--p-state=2700000`

performance-limiting factor is the L_1 cache miss latency.

8.4.1.3.3 Validating CrayPAT when using OpenMP

The preceding discussion related to running the code using pure MPI, which is consistent with the results in Section 8.2.5.1, which showed that the fastest execution was obtained for pure MPI. However the results in Figure 8.3 were obtained using three OpenMP threads per process. It is therefore prudent to investigate the performance of the Cray when running in this manner. CrayPAT output when using three threads per process is shown in Table 8.8.

Noting that, with the exception of aggregate Flop rate, the values are averages over the number of used cores so that the fact that the number of operations is roughly one-third of the values in Table 8.6 is to be expected because each process uses three threads, i.e. the calculations are spread over three times the number of cores. The reduction in time to roughly one-third of the value in Table 8.6 is a direct consequence of most of the work being parallelised using OpenMP, and running with three threads.

The combination of reduction in number of operations per core, and corresponding reduction in run time, mean that the percentage of peak performance is roughly the same as for pure MPI, i.e. about 21%. The fact that the achieved peak performance is marginally lower than for pure MPI is consistent with the fact that, in Section 8.2.5.1, using pure MPI was shown to be the fastest method of running the code, but use of OpenMP allowed for use of a greater number of cores while reducing the performance marginally.

The CrayPAT output in Table 8.8 shows that the percentage of peak performance is

Item		Processes 1	2	3	4	8
CPU						
Frequency	GHz	2.7	2.7	2.7	2.7	2.7
Time						
Overall Wall	sec	614	613	615	613	613
Operations						
Total Number	$\times 10^{12}$	2.728	2.734	2.733	2.730	2.730
Number Vectorised	$\times 10^{11}$	5.166	5.183	5.179	5.171	5.171
Number Scalar	$\times 10^{11}$	6.551	6.540	6.545	6.548	6.548
Aggregate Flop Rate	$\times 10^{9}$/s	4.44	8.91	13.33	17.81	35.65
Fraction of peak Flops		20.8%	20.8%	20.8%	20.7%	20.8%
Cache						
D2 to D1 bandwidth	GiB/s	6.052	6.055	6.044	6.058	6.070
D2 cache hit rate		95.8%	95.7%	95.5%	95.7%	95.9%
L3 cache hit rate		100.0%	99.7%	99.7%	99.7%	99.7%
L1 cache misses	$\times 10^{9}$	59.24	59.37	59.04	59.36	59.31
L2 cache misses	$\times 10^{9}$	4.30	4.46	4.62	4.476	4.49
L3 cache misses	$\times 10^{6}$	1.72	1.16	1.34	1.52	1.51

Table 8.8 Metrics from CrayPAT using OpenMP-enabled code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, 3 threads per process, CPU frequency fixed using `--p-state=2700000`. Note that the figures for Aggregate Flop Rate are under-reported by a factor of three (i.e. the number of threads per process): see Section 8.4.1.3.3 for details.

Processes	1	2	3	4	8
Fraction of Flops vectorised	75.75%	75.82%	75.81%	75.75%	75.76%
Flops per L_1 miss	43.74	43.87	43.81	43.81	43.76
Flops per L_2 miss	634.23	612.8	591.69	611.39	607.57
Time waiting for L_1 misses					
Wall Clock (sec)	263.5	263.9	263.7	263.7	263.8
Fraction of Overall Wall Time	42.90%	43.02%	42.87%	42.99%	43.07%
Time waiting for L_2 misses					
Wall Clock (sec)	47.8	49.6	51.3	49.6	49.9
Fraction of Overall Wall Time	7.78%	8.08%	8.34%	8.09%	8.15%

Table 8.9 Derived metrics for single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, 3 threads per process, CPU frequency fixed using `--p-state=2700000`.

roughly 21%, and this value has been justified above: this highlights a problem with the aggregate Flop rate calculated by CrayPAT and shown in Table 8.8. The total number of operations, number of vectorised operations and number of scalar operations are all averaged over the number of 'working cores', i.e. the product of the number of processes and the number of threads per process: this may be verified by comparing the column for one process with the corresponding column in Table 8.6. Therefore, because aggregation is performed over processes rather than cores, the aggregate Flop rate is too low by a factor of the number of threads used per process: this is easily verified by noting that the implied peak Flop rate (which may be derived by dividing the aggregate Flop rate by the percentage achieved) is about one-third of the value derived from multiplying the chip's clock rate, the number of operations per cycle and the number of used cores.

Table 8.8 shows that, for runs with three threads per process, the number of L_2 cache misses is less than one-third of the number for the pure MPI run (as shown in Table 8.6): this may be due to a L_2 miss in the threaded version being satisfied by a request to the L_2 on a different core, rather than a request to the shared L_3.

Table 8.9 shows the same collection of derived metrics that were obtained for pure MPI, and presented in Table 8.7. The values of the derived metrics, shown in Table 8.9, for the code run using three-threaded OpenMP are similar to the metrics for the pure MPI version of the code. In particular, the run time when using three threads is about 3.5% longer than those for pure MPI which might be due to coherency misses causing longer L_2 miss latencies; there is a good level of reuse of data once it is loaded into cache; about 43% of the overall time is spent waiting for data from L_2 cache; and about 8% of the overall time is spent waiting for data from L_3 cache. It is apparent that, as with

Block Size	4	8	16	32	64	96	128	256
Run Time (sec)	1947	1471	1298	1232	1227	1282	1256	1727

Table 8.10 Run times (in seconds) to process 500,000 policies over 96 tranches, using single precision, vectorised code and various block sizes, but pure MPI, on 4 node of the Cray.

the pure MPI case, the performance-limiting factor is the L_1 cache miss latency. Based on these observations, there seems limited scope for further optimisation.

8.4.1.3.4 Performance of The Code when Run in Other Configurations

In order to glean further performance information, other configurations of the code have been run. As discussed in Section 8.2.6, the original code was implemented using as much vectorisation as possible so as to gain optimal performance. Also, the code uses double precision throughout in order not to lose accuracy. To allow the effect of these design decisions to be estimated, a single precision version of the vectorised code was created, and non-vectorised versions of both precision levels were created.

The performance of the code when run using OpenMP, compared to the performance using pure MPI has just been discussed: the conclusion that there was similar performance under both paradigms means that it suffices to run further analyses using pure MPI.

Sections 8.2.4 and 8.2.5 discussed the need to run the code using the optimal block size when processing 1000 tranches whilst re-drawing parameters: the need to find the optimal block size also applies to the single precision code used in these comparisons. The timings from running the code using various block lengths are shown in Table 8.10, from which it is apparent that a block size of 64 is best, although this is only marginally better than a block size of 32. This compares well with the results in Section 8.2.5.1 where the times for block lengths of 32 and 16 were not particularly different. The doubling of the optimum block size for single precision calculations might be expected from the fact that single precision requires half as many bytes per datum, so that there are twice as many items in each cache line: whilst this is a reasonable expectation, it is not guaranteed that, on all machines, best performance of the single precision code will be with twice the block size which is best for double precision. Although the scalar code was created by removing the loop over blocks of policies and the `SIMD pragma`s which guided the Intel compiler during development, the compiler switch to completely disable vectorisation was not used so that the compiler was free to create vectorised code. Also, because the non-vectorised version of the code does not contain a loop over

Item		Double Vector	Double Scalar	Single Vector
CPU				
Frequency	GHz	2.7	2.7	2.7
Time				
Overall Wall	sec	1788	3017	1327
Operations				
Total Number	$\times 10^{12}$	8.198	8.225	8.126
Number Vectorised	$\times 10^{12}$	1.552	1.004	0.769
Number Scalar	$\times 10^{12}$	1.963	4.059	1.959
Aggregate Flop Rate	$\times 10^{9}$/s	55.13	32.70	73.95
Fraction of peak Flops		21.30%	12.70%	28.50%
Cache				
D2 to D1 bandwidth	GiB/s	6.246	10.056	4.199
D2 cache hit rate		95.6%	100.0%	91.3%
L3 cache hit rate		99.6%	86.4%	99.8%
L1 cache misses	$\times 10^{9}$	177.37	508.93	83.66
L2 cache misses	$\times 10^{9}$	13.58	0.34	13.40
L3 cache misses	$\times 10^{6}$	55.86	46.72	20.42

Table 8.11 Metrics from CrayPAT using single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one socket, twelve processes per socket, one scenario per process, one thread per process, CPU frequency fixed using `--p-state=2700000`

blocks of policies, the concept of an optimal block size is redundant.

Table 8.11 contains information from CrayPAT resulting from also running single precision vectorised and double precision non-vectorised configurations of the code using pure MPI on a full socket of a single node, i.e. 12 processes on 12 cores: these results show several things, as follows.

Firstly, for all configurations, the percentage of peak Flops is based on a theoretical maximum of 259.2 GFlops, which is 8 Flops per cycle, for each of 12 cores running at 2.7 GHz. However, the 8 Flops per cycle is for double precision calculations and so the percentage of peak derived by CrayPAT for the single precision code is misleading: the theoretical maximum ought to be doubled so that achieved peak is halved, but Table 8.11 contains the information exactly as it is calculated within, and presented by, CrayPAT.

Secondly, the speedup obtained from vectorisation is about 1.69×: this differs to Section 8.2.6.1 where the overall benefit of vectorisation was 1.2×. However, the analysis in Section 8.2.6.1 was performed on the Xeon cluster, using the Intel compiler, whereas the analysis in this section was performed on the Cray, using the Cray compiler.

Next, in Tables 8.7 and 8.9 the fraction of Flops which were vectorised was calculated as

$$\frac{4 \times \text{number of vectorised operations}}{\text{total number of operations}}$$

because the Ivy Bridge vector units can perform 4 double precision operations concurrently. However, for single precision, the vector units can perform 8 operations concurrently: this means that, for single precision, the fraction of Flops which were vectorised should be calculated as

$$\frac{8 \times \text{number of vectorised operations}}{\text{total number of operations}} = 8 \times \frac{0.769 \times 10^{12}}{8.126 \times 10^{12}} \equiv 75.7\%$$

which is almost identical to the fraction vectorised in the double precision version.

Next, the fraction of Flops which were vectorised in the scalar code is

$$\frac{4 \times \text{number of vectorised operations}}{\text{total number of operations}} = 4 \times \frac{1.004 \times 10^{12}}{8.225 \times 10^{12}} \equiv 48.8\%$$

This indicates that, although loop blocking and explicit `pragma`s were removed, the compiler vectorised almost half the calculations.

Next, changing from double precision to single precision leads to approximately half the number of L_1 misses, which is consistent with each load of the cache line retrieving twice the number of variables. The fact that there is no real change in L_2 misses suggests that the missed values would not have been re-used, i.e. those accesses did not display particularly good temporal locality.

Next, for the double precision code, changing from vectorised to scalar leads to L_1 misses increasing by a factor of about $2.9\times$, and L_2 misses decreasing by a factor of about $\frac{1}{40}\times$. This is consistent with vectorisation improving spatial locality, and hence reducing L_1 misses. Conversely, however, vectorisation increases the working set size, increasing L_2 misses. Overall, the improvement in spatial locality and reduction in floating-point instruction count, but worsening of temporal locality, caused by vectorisation, has a beneficial effect, as seen by the decrease in run time.

Next, when vectorisation is removed from the double precision code the L_3 cache hit rate reduces significantly. This is simply a result of the near-zero number of L_3 misses changing only slightly, but having a large effect on the rate.

Finally, the speedup achieved by using single precision, rather than double precision, for the vectorised code is about $1.35\times$. The fact that the code does not run in half the time is another indication that the performance limitation is not memory bandwidth:

MPI Processes per Node	8
OpenMP Threads per Processes	2
Tranches per Process	25
Wall Clock Time (sec)	28357
Portion of RunTime in I/O	0.996%

Table 8.12 Run time (in seconds) to process 500,000 policies over 200 tranches using a block size of 32, on 1 node of the the Xeon cluster.

if memory bandwidth had been the limit then, for a given memory bandwidth, twice as many single precision variables as double precision variables could be transferred in a given time so that, *ceteris paribus*, the performance would have doubled.

8.4.2 The Xeon Cluster

Section 8.4.1 discussed the ability to perform the full brute force calculation on a Cray XC30. However, such machines are generally not common in life offices: although there is a possibility of renting time on external machines, the misconception that 'data security will be compromised' is likely to be a barrier to such an approach. Therefore, this section considers the ability to perform the brute force calculation on a cluster which is small enough to be operated within many life offices.

Section 8.2.5.2 concluded that a block size of 32 was optimal for the Xeon cluster. Table 8.12 shows timings for processing 200 tranches on one node: by extrapolation, the independence of the tranches suggests that it should be possible to complete a full brute force run in under 8 hours on the wall clock using 5 nodes. Since five nodes equates to 80 cores, all offices should be able to perform the calculations in an overnight run. Allowing for the differences in CPU speeds, and the numbers of cores on the nodes, this would be equivalent to 16804 sec on the Cray. The actual time on the Cray (per Table 8.4) is 13875 sec: the Xeon cluster therefore takes about 21% more time than expected and is therefore about 17.4% slower than the Cray. Given the similarity in the CPU specification of the two machines, the difference is likely to be due partly to the use of different compilers (the Cray compiler was used on the Cray, while the Intel compiler was used on the Xeon cluster), and partly to the memory management system in the Cray being particularly efficient in comparison to clusters which do not have the Cray operating system.

8.5 Performance when Not Re-Drawing Parameters

The situation where parameters are not re-drawn has only been run on the Cray: this section describes the set-up for, and results of, that processing.

8.5.1 Run Time Set-up

8.5.1.1 Algorithm and Code

Conceptually, Algorithm 2.1 (without the need to re-draw parameters in Line 11) is required: implementationally, when the recurrence approach is used, this becomes Algorithm 7.4. As mentioned in Section 8.2.1, the code used for the main part of the investigation (i.e. the results discussed in Section 8.4) implements Algorithm 8.1 which is an optimisation of Algorithm 7.2. In order to create an implementation of Algorithm 7.4, so that parameters are not re-drawn, the outer loop over time steps was removed from the implementation of Algorithm 7.2: in fact, because the in-force reserves calculated for future time steps are considered valid under this approach, they are stored in an array, rather than being discarded. Therefore, the code changes between the implementations of the two algorithms are minimal.

8.5.1.2 Data and Parameters

By not re-drawing parameters each time step, the amount of calculation per read of the policy data is greatly reduced if only one scenario is processed. A simple method of increasing the ratio of calculation to I/O is to process several scenarios for each read of the data: the data files discussed in Section 8.1.3.1 may therefore be re-used without modification. However, efficient use of this scheme requires that the parameters are stored differently; having 1000 files, each with parameters for 750 simulations, allows for up to 750,000 scenarios[3] while allowing the files to remain open, thereby reducing the amount of housekeeping required.

[3]This is an arbitrary limit to prove the concept: a commercial situation could have several orders more scenarios.

MPI Processes	OpenMP Threads	Block Size						
		4	8	16	32	64	128	256
8	12	170.2	168.5	179.0	372.1	538.0	581.1	596.2
16	6	146.9	137.6	163.0	351.0	520.9	577.2	583.0
24	4	141.1	131.5	156.6	336.8	514.4	559.3	575.5
32	3	139.4	131.0	159.6	343.4	527.3	561.1	580.1
48	2	137.0	127.4	155.0	337.2	517.4	557.5	571.2
96	1	133.5	124.3	155.9	349.3	517.0	568.6	569.8

Table 8.13 Run times (in seconds) to process 10 scenarios per read of the data for 500,000 policies, over 96 tranches using various combinations of block size, number of MPI processes and OpenMP threads per process, on 4 nodes of the Cray XC30.

8.5.2 Finding the Optimal Block Length

Section 8.2.5 considered the best combination of processes and threads to use in the situation where parameters are re-drawn from the relevant distribution in each step: in order to ensure that the best combination of processes and threads is used when parameters are not re-drawn it is necessary to perform a similar preparatory investigation. The run times resulting from processing 10 scenarios per read of the data are shown in Table 8.13.

From Table 8.13 it is apparent that a block size of 8 is optimal, irrespective of the combination of processes and threads. As the number of scenarios processed for each read of the data increases, it is reasonable to expect that, because the processing for each scenario is identical, the run time should increase as a linear function of number of processes. It should therefore be expected that the optimal block size will be 8, irrespective of the number of scenarios processed. Run times for varying numbers of scenarios per read of the data, using a block size of 8, are shown in Table 8.14: the table also shows the results of using least squares to fit the straight line

$$t_r = m \times\ n_s + c$$

where t_r is the overall time for the run and n_s is the number of scenarios per read of the data.

Table 8.14 shows several things. Firstly, pure MPI leads to the shortest run times, irrespective of how many scenarios are processed: the fact that pure MPI results in fastest processing is consistent with Section 8.2.5.1 – which is to be expected because the choice of whether or not parameters are re-drawn has no effect on the processing of each policy once the parameter values are known. Secondly, the overall processing time

MPI Processes	OpenMP Threads	Scenarios per Process				Linear Regression		
		1	10	20	50	m	c	R^2
8	12	46.35	168.5	270.1	625.0	11.69	40.76	0.99906
16	6	34.54	137.6	248.9	600.5	11.56	21.36	0.99990
24	4	28.08	131.5	245.4	588.2	11.43	16.88	1.00000
32	3	28.51	131.0	242.7	583.4	11.32	17.19	0.99999
48	2	27.04	127.4	240.8	582.1	11.34	14.68	0.99999
96	1	23.86	124.3	236.1	571.3	11.17	12.66	1.00000

Table 8.14 Run times (in seconds) to process 500,000 policies with varying number of scenarios: parameters for linear models are included. The processing was for 96 tranches using a block size of 8 and various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 4 nodes of the Cray XC30.

per scenario increases most slowly for pure MPI so that the shortest time to process larger numbers of scenarios should be expected from the use of pure MPI. Finally, for all combinations of processes and threads, the overall processing time is modelled well by a linear function of number of scenarios processed.

8.5.3 The Full Run

The calculation of the ACR involves the 99.5^{th} percentile of the distribution of aggregated per-policy reserves at each future step. Therefore, it is likely that, if the parameters are not re-drawn, then the statistics of the final ACR will not be very robust. It is possible to show that, asymptotically, the sample variance of the 99.5^{th} percentile of a sample from a standard normal distribution is roughly 24 times the sample variance of the mean of a sample from a standard normal distribution: see for example [10], which refers to [34, Sections 5.4.7 and 2.3]. It therefore seems likely that a sample of only 1000 scenarios will not be sufficient.

The result in [10] cannot be directly applied to the ACR because the parameters are not necessarily independent (e.g. a high degree of correlation between interest rates and inflation rates should be expected), the distributions from which the parameters are drawn are not necessarily normal, and the ACR is not a simple function of the parameters.

Without performing numerical experiments with realistic input parameters, it is not possible to determine the number of scenarios required to keep the variance of the ACR within a given tolerance. However, for purposes of illustrating performance, runs using 24,000 scenarios and 96,000 scenarios have been performed.

Number of Nodes Number of Processes	5 120	10 240	25 600	50 1200	125 3000	250 6000	500 12000
Scenarios per Process	200	100	40	20	8	4	2
Wall Clock Time (sec)	2322	1208	493	254	115	74	43
Portion of Time in I/O	0.20%	0.25%	0.46%	1.03%	1.81%	5.25%	11.57%

Table 8.15 Run time (in seconds) on the Cray XC30 when parameters are not re-drawn each time step: pure MPI was used to process 24,000 scenarios using a block size of 16.

Number of Nodes Number of Processes	5 120	10 240	25 600	50 1200	125 3000	250 6000	500 12000
Scenarios per Process	800	400	160	80	32	16	8
Wall Clock Time (sec)	9247	4795	1924	966	402	213	122
Portion of Time in I/O	0.11%	0.17%	0.14%	0.16%	0.32%	0.90%	1.56%

Table 8.16 Run time (in seconds) on the Cray XC30 when parameters are not re-drawn each time step: pure MPI was used to process 96,000 scenarios using a block size of 16.

Section 8.4.1.1 discussed the strong scaling of the code when re-drawing parameters and running on the Cray. Tables 8.15 and 8.16 show the timings for strong scaling tests for the code running on the Cray when not re-drawing parameters and processing 24,000 and 96,000 scenarios respectively: the performance achieved when parameters are not re-drawn is shown in Figure 8.4.

It is not entirely obvious, but the ideal processing rates for 24,000 and 96,000 scenarios are the same: this is because the number of scenarios processed per second for five nodes are almost identical for the two regimes, and is likely to be a result of I/O not being a significant part of the overall time on a small number of cores.

It is apparent from Figure 8.4 that the efficiency for the case where parameters are not re-drawn is not as good as the efficiency for the case where parameters are re-drawn: the efficiency implied by the use of 12,000 cores is about 51% for 24,000 scenarios, and roughly 76% for 96,000 scenarios. This is a result of the fact that, for the smaller number of scenarios, the cost of reading the policy data is not amortised over as many calculations: the parameters are read for each scenario so that the cost of reading them is linear in the number of scenarios. The sole reason for not re-drawing parameters was to reduce the number of times the reserves[4] must be calculated so that the increased proportion of time spent in I/O, as the number of operations (to calculate reserves) falls, is as expected. The cost of reading the data reduces as more scenarios are processed,

[4]Both the in-force and per-policy reserves need to be obtained.

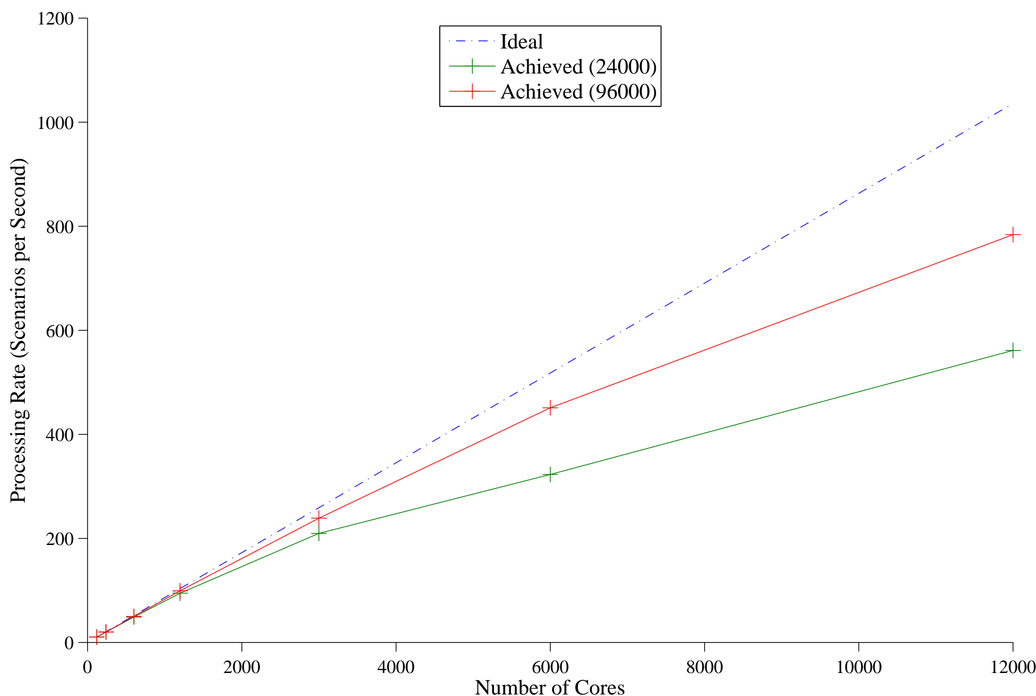


Figure 8.4 Strong scaling on the Cray XC30 when parameters are not re-drawn each time step: the ideal processing rate is based on the time using 5 nodes (as presented in Table 8.15 and 8.16), i.e. including I/O, for 24,000 and 96,000 scenarios. The code used a block size of 8 with double precision and vectorisation included.

which is also expected.

Although the code does not scale particularly well, it is possible to process 96,000 scenarios in about two minutes. This processing time requires 500 nodes of a Cray so that, as noted in Section 8.4.1.1, the expense of purchasing and running this amount of hardware is likely to prohibit life offices achieving these time scales.

8.6 Summary

This chapter has considered *ab-initio* code to perform the calculations required to fulfil Solvency II calculations using a brute force approach. By building optimisations into the code at the outset, arranging scenario data efficiently, tuning parameters relating to the machine's architecture, and making a few modest simplifications, the brute force calculations can be performed in reasonable timescales.

Section 8.2.5 indicated that, on the platforms tested, using pure MPI leads to the shortest run time for each tranche. However, in order to obtain greater scalability, a hybrid approach using a small number of OpenMP threads should be used.

Section 8.3 showed that performance was improved by a factor of more than 4$\times$ by including simplifications which changed the results by less than 0.1%.

Section 8.4.1 demonstrated that, when re-drawing parameters, the entire calculation can be performed during a coffee break using a supercomputer. Figure 8.1 shows that the implementation scales well as the number of cores is increased: the deviation from linearity is due to the combination of MPI processes and OpenMP threads changing to permit full load balance. Figure 8.2 shows that the majority of the run time is spent in routines which have been carefully considered and optimised. Figure 8.3 shows that the implementation performs well in terms of computational efficiency. An in-depth analysis of performance confirmed the computational efficiency to be reasonably high, and showed the limiting factor to be latency of transfers from L_2 cache to L_1 cache. It is, therefore, unlikely that much more performance could be extracted without further fundamental changes to the code: Section 9.1 contains some ideas on what those changes could be.

Section 8.4.2 demonstrated that it is possible to perform the entire calculation overnight using a cluster which is small enough to be within the reach of the vast majority of life assurance companies. There is therefore no reason that the approach developed in this project should not be used in the industry, throughout Europe.

Section 8.5.3 showed that, by not re-drawing parameters, the same variance of 99.5$^{\text{th}}$ percentile (as could be achieved by re-drawing) could be obtained in about 45 seconds when using 12,000 cores of the Cray: this represents a speedup of about 3.6$\times$ compared to re-drawing parameters, per Table 8.4. When compared to re-drawing parameters, the case of not re-drawing parameters runs more quickly but the scaling is far worse, as shown by Figures 8.1 and 8.4: when processing 24,000 scenarios, I/O takes a significant part of the run time because the cost of reading the policy data on each process is amortised over a smaller number of calculations than when 96,000 scenarios are processed.

Chapter 9

Future Work

Chapter 5 has shown that it is possible to estimate future profitability in far shorter times than are achievable using currently available commercial software, and Chapter 8 has shown that it is possible to perform a brute force assessment of solvency in times which are within the realms of contemplation. However, there is still scope for future work in the area investigated in this project, both in terms of improving the performance of the current code, and in applying these techniques to other situations. This chapter discusses some ideas for further investigations.

Section 9.1 suggests ideas for further work with CPUs, both in terms of further optimisations of this code, and extension to other types of policy. The derivation of the recurrence relation in Chapter 3 is independent of policy type, indicating that it can be applied to a completely arbitrary policy: Section 9.2 discusses implementation of a generalised policy. Section 9.3 considers alternative technologies, and Section 9.4 suggests other uses for a code which runs as quickly as that developed in this project. Section 9.5 considers the extension of this work to nested stochastic projections, i.e. those where the recurrence itself is based on stochastic simulation, rather than deterministic derivations. Finally, Section 9.6 contains a summary of the chapter.

9.1 Further Work with CPUs

So far, this project has concentrated on the use of CPUs: this section suggests further possible work using CPUs.

9.1.1 General Optimisation

The use of simple compiler flags means that the compiler's discretion is applied to lower level optimisations in the production of the executable file. Therefore, an investigation into optimal combinations of settings which are not already included in `-fast` might lead to a further improvement in performance.

A further serial optimisation would be to change the vectorised calculation of the q_x's to go to the maximum step implied by the ages in the current block, rather than the maximum step implied by the ages in the entire data file. However, for randomly distributed data, the maximum age in the block is not likely to differ significantly from the maximum age in the data file. Therefore, it is not currently envisaged that this would lead to much further speed improvement: any performance improvement is likely to require the data file to be sorted by date of birth so that policies within a block have similar outstanding terms, thereby removing the need for some redundant calculations. Whilst these optimisations are expected to yield small improvements in performance, any speedups resulting from these changes are not expected to be more than about $2\times$.

Table 8.3 contains statistics regarding the performance of this code on the Cray: the code achieves roughly 20% of peak performance which is quite reasonable. The statistics also show that this code achieves a ratio of Flops-per-reference of about 2.3, which is also quite reasonable. The combination of these factors suggests that there is not much more performance which could be extracted from the CPU code.

Given recent trends in hardware development, it seems reasonable to think that the trend for widening vector units within CPUs and increasing the vector instructions sets will continue for the foreseeable future. Any changes in the size of the vector unit may trivially be incorporated in this project's code by changing the block length, as discussed in Section 8.2.5. Also, the nature of the brute force approach means that the code is calculation intensive: the majority of the time is spent performing floating point arithmetic, so that it is hard to see how changes to the vector instruction set will improve performance – unless there is a fundamental shift in the way the vector units performs those calculations. Therefore, further optimisations are unlikely to lead to paradigm shifting changes in the way companies work: the changes discussed in Chapters 3 to 8 of this thesis might do.

9.1.2 Improving I/O

The Cray has a good I/O capability and, Table 8.4 shows that, within the execution of the Solvency code discussed in Section 8.4.1.1, I/O is less than 2% of the overall execution time when running on up to 125 nodes. Although this is not a significant part of the overall time, other machines have lesser I/O capabilities, and so there might be a need to improve I/O.

Reading from disk does leave some scope for improvement: for both the data and the parameters, the effect of I/O could be reduced by interleaving I/O and calculation, and the time spent in I/O could be further reduced by using binary files, rather than ASCII files.

A further enhancement would be to generate the parameter values from their underlying distributions during the run, rather than reading them from disk. Section 8.1.3.2 implied a total of 148 GB scenario data: removing the need to read this volume of data would not only improve performance, it would also allow the number of scenarios to be varied quickly, should any future change in regulations require such a variation in the underlying methodology.

9.1.3 Extension to Other Policies

The Fortran code discussed in Chapter 5 was developed only for simple annuities. An obvious candidate for further work is the extension of this code to cover estimation of profitability of other policies: in particular, two life annuities could be implemented by adding a few further routines which deal with the different payment probabilities.

Annuities were chosen for the investigation into the possibility of fulfilling Solvency II regulations because the continued existence of the companies which provide annuities is essential to the financial well-being of pensioners. However, there are a great many other policy types whose benefits are relied on by policyholders with different needs. Therefore, the extension of the code to cover other policies, such as assurances and endowments, would be beneficial. The fact that such policies have cash flows which are easily derived means that it is possible to produce highly optimised code for them.

It should also be possible to produce code for a completely arbitrary policy, and this is discussed in Section 9.2.

9.1.4 Variance of ACR as a Function of Number of Scenarios

Section 7.1 discussed whether or not parameters should be re-drawn, and Section 7.3 analysed possible algorithms which might be used to obtain the ACR depending on whether or not parameters are re-drawn. Sections 8.4.1 and 8.5.3 have shown that, irrespective of whether parameters are re-drawn, it is possible to calculate the ACR in reasonable wall clock times when using implementations of Algorithms 7.2 and 7.4 and a large enough portion of a supercomputer.

Section 1.5 contained an informal argument for using 1000 scenarios to obtain the ACR when parameters are re-drawn and Section 8.5.3 contained an justification for an increase in the number of scenarios required when parameters are not re-drawn. However, there is no rigorous justification for what the number of scenarios should be in either case.

The distribution for the ACR will depend on not only the distributions for the underlying parameters, but also the number of scenarios processed and whether the parameters are re-drawn in each time step. Actually calculating the ACR's distribution analytically will be difficult, and may not even be possible. However, it would be possible to perform many simulations of the ACR calculation in order to obtain an approximation to the distribution, and how it depends on the number of scenarios: the processing rates achieved using the advances made by this project would make such simulations possible, irrespective of whether parameters are re-drawn or not.

The first stage of such an investigation would be to replace the current method of reading parameters from files by generating them within the program: this allows a) the number of scenarios to be varied rapidly, perhaps as a command line input, and b) the distributions which model the parameters to be easily changed, e.g. by writing new functions. In order for the final distribution of the ACR to be realistic, functions which produce economic and demographic parameters would also need to be realistic: for example interest rates would need to contain some level of correlation between consecutive months, rather than being independently drawn from a particular distribution, as is the case in this project.

9.2 The General Case

The theoretical foundations for the use of the recurrence relation for a general policy were discussed in Section 4.4. An implementation for general policies would require a

framework whose only constraint is that code to generate cash flows for each new policy type could be fed into the existing structure relatively easily: it would be necessary to generate sufficient code that any policy type could be processed through the addition of minimal further code.

The most general form of the recurrence relation, as given in Section 3.2.5.3, is

$$\mathbf{r}_{\mathbf{x},t} = v_t^f \, \mathbf{W}_{\mathbf{x},t,f} \, \mathbf{c}_{\mathbf{x},t} + v_t \, \mathbf{W}_{\mathbf{x},t,1} \, \mathbf{r}_{\mathbf{x+1},t}$$

and, after removing zero reserve states, this becomes

$$\overline{\mathbf{r}}_{\mathbf{x},t} = v_t^f \, \overline{\mathbf{W}}_{\mathbf{x},t,f} \, \overline{\mathbf{c}}_{\mathbf{x},t} + v_t \, \overline{\mathbf{W}}_{\mathbf{x},t,1} \, \overline{\mathbf{r}}_{\mathbf{x+1},t}$$

Using a two-life, last survivor annuity as a specific example, nothing is payable after both lives have died meaning that there is only one state with zero cash flows. The corresponding required reserve factors, in actuarial notation, are a_{xy}, a_x, and a_y so that, after removing ZRSs, $\overline{\mathbf{W}}_{\mathbf{x},t,f}$ and $\overline{\mathbf{W}}_{\mathbf{x},t,1}$ are 3×3 matrices, and the cash flow and reserve factor vectors, $\overline{\mathbf{c}}_{\mathbf{x},t}$ and $\overline{\mathbf{r}}_{\mathbf{x},t}$, are 3×1. Therefore, implementation of matrix-vector multiplication for this contract is trivial.

In order to implement the general case it would be necessary to be able to identify zero cash flows, and there needs to be sufficient information to calculate the cash flows for each possible state for each future step. The identification of zero cash flows would necessarily be a policy-dependent step, and it is likely that a conditional expression would be needed. Additionally, $\overline{\mathbf{W}}$ would be needed in terms of q_x's: this should not be too onerous for each particular policy since only one $\mathbf{W}$ is needed for each transition diagram – the fact that there is one transition matrix for each combination of number of lives and number of states means that the structure of $\overline{\mathbf{W}}$ can be determined in advance.

In order to minimise the amount of code required for each new policy type, it could be beneficial to implement such a generalised code in an object oriented language so that its inheritance capabilities can be harnessed. The fact that the current code is C suggests that C++ is the natural choice, allowing much of the existing code to be reused. However, Fortran is a continually evolving language which now supports several aspects of object oriented design and programming. Hence, rewriting the code in Fortran 2003 would serve as well as transferring the code to C++.

It is clearly possible to perform implementational optimisations for specific contracts.

Performing such optimisations for the general case would be more difficult because the need to allow for all possible states must be admitted. It is therefore reasonable to expect that any implementation of the general case would have lower performance than code which is crafted for the specific cases. However, because the general case would be based on the recurrence relation, it could reasonably be expected to be two orders of magnitude faster than an implementation based on summation for a projection over several hundred steps. Therefore, using this approach, there would be hard-coded, highly optimised code for the most common policies (which account for the majority of liabilities), and an implementation of the general case for contracts where there are not many policies.

9.3 Alternative Technologies

Other computing technologies are used in other arenas of science and engineering. This section suggests whether those technologies could be applied to the actuarial problems investigated in this project.

The results in Section 8.2.6 showed that vectorisation of the code only led to a speedup of about 1.2× on the Xeon processors. However, the amount of vectorisation in the code provides some level of allowance for future changes to hardware: the data layout required for vectorisation is the same as that required by GPUs in order that memory coalescing may be performed.

9.3.1 GPUs

Transferring data to a GPU takes a long time in relation to CPU-related data movements. Therefore, in order to benefit from performing calculations on a GPU, there needs to be sufficient calculation to overcome the cost of transferring the data to the GPU.

As a result of changes made during the course of this project, profitability calculations can be performed sufficiently quickly that they can be completed in a few seconds on a moderately sized SMP. However, although small clusters of PCs are becoming commonplace within the industry, SMPs of the size used in this project are still relatively rare in life assurance offices in the UK. It is therefore unlikely that the approach used in Chapter 5 will be widely adopted.

In order to be able to perform the calculations on a reasonable amount of hardware, an alternative solution may be to transfer the program to GPUs. However, it is not clear that there is sufficient computation within profitability calculations to overcome the time cost involved in transferring the data to the device. An investigation into the performance of the profitability code on alternative hardware would therefore be beneficial.

The demonstration of solvency which this project considers appears to be ideally suited to implementation on GPUs because it is compute intensive: Section 3.4.1 demonstrated that, for a projection of N policies over T time steps, the amount of compute is $O(NT)$. Since the amount of policy data[1] for N policies is $O(N)$, for a projection over 50 years using monthly steps, so that $T = 600$, there is a reasonable amount of calculation on a small amount of data, and none of the calculations for any policy involve the data from any other policy.

Implementation on GPUs could be via CUDA, OpenCL, or OpenMP's recently added extensions for GPUs. A CUDA version could be used to determine the speed improvement using a highly optimised GPU code relative to the optimised CPU code developed in this project. The OpenMP version would demonstrate the performance available from the use of a programming model which is straightforward to implement.

Allowing for the discussions in Section 2.6.4, it is expected that a highly tuned CUDA version of the code could be of the order of only 10× faster than the highly optimised CPU code developed here: this expectation is consistent with the comparison of GPU hardware to CPU hardware – as mentioned in Section 2.6.4, a single GPU has roughly 11× the compute capacity and about 8× the memory bandwidth of a single CPU, so that the overall performance could be expected to lie between these limits.

Without some degree of optimisation, the performance of an OpenMP version on a GPU would not be expected to match that of a highly optimised CUDA version. However, because of the optimisations built into the code as a result of the use of the OpenMP API and its run time, the performance would not be expected to be orders of magnitude worse than CUDA. A future investigation could lead to the ability to justify a statement such as 'OpenMP leads to 50% of the performance of CUDA for 15% of the effort'.

[1] The amount of scenario data is independent of the number of policies.

9.3.2 Xeon Phi MIC

Intel's Many Integrated Core chip, known as 'Xeon Phi', has 512-bit wide vector processors, resulting in the ability to perform 8 double-precision SIMD instructions in each cycle. This, allied to the high degree of vectorisation in the code developed in this project, invites the possibility of investigating whether the Xeon Phi is a better chip than a standard CPU for performing the brute force approach. The processing strategy discussed in Section 8.2.1 harnesses the parallelism at both the scenario, and policy, levels. It is likely that this strategy will still be required in order to make best use of all cores available on the Xeon Phi, while continuing to expose plenty of parallelism at the thread level.

Code can be built to run on older versions of the Xeon Phi, code named Knights Corner, and often abbreviated as KNC, in 'native' mode or in 'offload' mode: in either case, the Intel compiler must be used. For native mode, the compiler produces code which runs entirely on the KNC, and MPI and OpenMP may be used within the code. For offload mode, the compiler inserts directives which mark regions of the code to be run on the KNC, while the remainder of the code runs on the CPU. The hybrid code which was developed in Chapter 8, and run on the Cray, should run on the KNC in native mode without alteration, although the parameters relating to block size, number of processes and number of threads per process will need to be tuned.

Another aspect of a future investigation could be modifying the code to run in offload mode on the KNC, although it is not clear that there will be much benefit to this because a new evolution of the Xeon Phi chip has been released and is expected to replace KNC chips in most systems. The new Xeon Phi, code named Knights Landing, and often abbreviated as KNL, is expected to be available mainly as CPU replacements, rather than co-processors, so that the entire code will run on the KNL, making offload mode redundant. KNL chips are being built into Cray systems where the Cray compiler suite may also be used: the choice of compilers therefore invites an exploration of the techniques employed by the different compilers.

KNL chips include an updated 512-bit vector instruction set and incorporate fused-multiply-add (FMA) instructions. It should therefore be expected that best performance on the KNL is obtained for codes which are highly vectorised and have FMA instructions. However, the code developed in this project does not have a significant proportion of instructions which can be reformed as FMAs. Therefore, although the code is highly vectorised, it is unlikely to attain a high percentage of the KNL's peak performance.

Another aspect of KNLs which could be of interest is their high bandwidth memory: this is 'multi-channel DRAM', known as MCDRAM. KNLs have 16 GB of MCDRAM which may be configured as 16 GB of cache: since this sits between the Level 2 cache for the cores and the main memory, the entire setup has become known as *hierarchical memory*, or *stacked memory*. The amount of memory required for the brute force approach is small enough to benefit from the KNL's high bandwidth memory. However, using pure MPI on the KNL would probably exhaust the 16 GB MCDRAM: one KNL chip might hold all the data required to process 16 tranches concurrently, e.g. using 16 MPI processes, each with 4 threads, on a 64-core system.

9.4 Other Uses

Until now, this project has concentrated on profitability and solvency. Whilst these are two of the major calculationally intensive areas within assurance and pensions provision businesses, there are others which could form the foundation of further investigations.

9.4.1 Pricing

The process of determining the level of premium which should be charged for a particular level of benefit, given a particular set of assumptions, is a prime concern to a business in any market. The assumptions used form the 'pricing basis', which contains a similar set of parameters to the reserving and realistic bases used in Chapters 5 and 8, although the values of those parameters are likely to differ.

Setting premium levels usually consists of several profitability runs, each using a different basis with a collection of model points. The premium for a contract is taken to be that which leads to the profit which is desired for the level of risk which is implied by the contract. The problem with this approach is the inability to test a large enough collection of bases, each with differing parameters, against a large enough collection of model points. This leads to a very coarse grained sweep of the parameter space, and hence to a not particularly well polished pricing structure.

However, an approach which would result in a finer grained sweep of the parameters would be to estimate the profitability within 1000 model points on each of 1000 bases. This is effectively a single profitability run for 10^6 'policies', where each policy is a model data point with a basis associated with it.

The profitability program, developed in this project could be used for this approach; at the rate of processing reported in Chapter 5, the overall processing of 10^6 policies would take only a few seconds. This would bring pricing into interactive timescales; the selection of bases could be changed at will by the operator.

Pricing using this system could lead to a clear competitive advantage in times of rapidly changing investment markets.

9.4.2 Stochastic Approach to Profitability

In addition to varying the bases used for pricing, it would be possible to implement the profitability program discussed in Chapter 5 so that, by allowing the realistic and reserving assumptions to be sampled from distributions, there could be a Monte Carlo approach to profitability: such an approach would be more realistic than the current approach where the bases are fixed, and are therefore highly reliant on actuaries' assumptions about the future.

9.4.3 Sensitivity Analysis

Knowing how both the profitability of, and reserves for, a given set of policies change as a result of changes to values of basis items is of prime importance to any life assurance company.

As with Section 9.4.1, the program developed in this project could be used with data points which allow the modelling of the derivatives of basis items. However, this requires that the values of the basis items differ by a sufficiently small margin as to approximate numeric differentiation. With a program which has high enough performance, many small changes to several parameters could be processed using a similar setup to that suggested for pricing.

An extension to sensitivity analysis is determining the point at which the company becomes insolvent, e.g. it would be possible to determine the interest rate at which the reserves required to cover the estimated liabilities exceed the available assets.

9.4.4 Asset Shares

Companies which have been trading in a with-profits market for a sufficiently long time will have funds where not all assets can be accurately apportioned to current

policyholders. For example, a whole life assurance effected in 1860 will probably have paid the benefit by 1910: this means that, for over 100 years, any shortfall in the bonus which was due to this policy, but not allocated to it, will have been accumulating in the fund. Such *orphan assets* now form a noticeable portion of the with-profits funds of some large companies, and some companies are starting to think about the best way to apportion such assets: whilst some policies may have ceased in the past few decades, so that their beneficiaries might be traceable, it is far less clear what to do with assets which relate to policies which went off the books several decades ago.

Using a program which processes policy data as quickly as that developed in this investigation, it would be possible to rebuild the actual values of all policies ever held by a company – assuming that all the relevant records could be found in order to construct the data. Whilst this does not solve the problem of what to do with the orphan assets, it would allow the calculation of the actual amount orphaned from each policy.

9.5 Nested Stochastic Projections

The work in this project was based on an interpretation of Solvency II which required 1000 simulations in each future time step, leading to 360,000 times the existing number of calculations for a projection using monthly time steps over 60 years. The results in Section 8.4.1 show that that projection can be performed in under 3 minutes on 12,000 cores of the Cray. Compared to the original naive estimate of about 2800 years on a single CPU core, this project has achieved a speedup of around 4.1×10^4.

There is another interpretation of Solvency II which suggests that, because reserves and capital must be calculated on a stochastic basis, other considerations in managing a life office (often referred to as 'management actions') need to be based on nested stochastic projections. However, in such projections, the inner stochastic loop need only be over a one year horizon: see [82] for a diagrammatic representation.

Using monthly time steps over 60 years, with 1000 inner paths for each of 1000 scenarios at valuation would require the reserves for each policy to be evaluated 720 million times: that is only 2000 times the 3.6×10^5 fold increase in calculation this project has demonstrated to be achievable in a reasonable time. Therefore, the expected run time would be roughly 3×10^5 seconds (i.e. about half a week) whilst still using 500 nodes, or it might be possible to spilt the calculations further so that it becomes possible to use more nodes.

The processing required for this alternative interpretation of Solvency II is embarrassingly parallel: it should therefore be possible to use a commercial cloud-based technology to perform the required processing. This approach would require aspects such as data security to be considered: the solutions to such perceived problems could be as simple as anonymising the data.

9.6 Summary

Several ideas for avenues of further investigation have been suggested in this chapter. They fall into three categories; implementation of the general case, implementation of other use cases, and implementation on other technologies. However, despite this categorisation, there is no requirement to pursue these in any particular order.

The variety of possible future investigations is consistent with Section 2.6 and the shortage of directly relevant literature, placing this thesis at the start of an emerging field of studying the application of HPC to actuarial calculations for life assurance policies.

Chapter 10

Conclusion

In the 150 years since Babbage's observation was published, much has been learnt about how results of computations may be arrived at more quickly: this thesis has added a little to that knowledge. Section 10.1 summarises the contribution made by each chapter, and Section 10.2 provides closing comments.

10.1 Review

Chapter 3 introduced a new algorithm for obtaining in-force reserves at a particular time step: this algorithm takes the form of a vector recurrence relation which runs backwards through time, allowing the use of boundary conditions which may be easily found (or assumed). The algorithm is completely general in that it does not require any particularly restrictive assumptions: all that is required is that cash flows and parameter values are known in advance of their use. The algorithm is also general in that it applies to many classes of non unit-linked policies, without regard to the conditions of the underlying contracts: all that is required is that cash flows must be determinable.

The algorithm reduced the computational complexity of obtaining each in-force reserve from $O(T^2)$, using the existing summation method, to $O(T)$ for T projection steps, leading to a speedup of two orders of magnitude for a projection over several hundred monthly steps. The introduction of zero reserve states further reduced the computational requirement.

Chapter 4 considered the application of the algorithm developed in Chapter 3 to

different types of policies. The matrix at the heart of Equation 3.2.6 is a stochastic matrix whose elements are all survival probabilities: it is therefore easier to think in terms of the survival states of the lives involved, than to use the traditional approach of considering individual policy types. In fact, it is possible to use the same survival matrix to model many contracts, the difference in the contracts being governed by the conditions under which the cash flows are made.

The application of the algorithm may be extended in two ways. Firstly, despite being somewhat unintuitive, the introduction of a third survival state allows many more contracts to be modelled: these additional contracts are those where the benefit is paid on transition from one state to another. An alternative interpretation of a three state model allows the recurrence relation to be applied to situations where the transitions between states could be reversible, e.g. permanent health insurances. Secondly, the algorithm may also be extended to any number of lives. Although situations where more than two or three lives are involved in a policy are rare, this algorithm is able to deal with them so long as they fulfil the same basic criteria already mentioned, i.e. the cash flows and parameter values are all known at the time they are used.

Chapter 5 discussed the optimisation of a code which performs profitability calculations on single life annuities. Although neither the code nor the data it processed are from a commercial environment, both reflect their respective entities sufficiently well to believe that the improvements made here would transfer directly to a business world, with the only requirement that hand-crafted code were used rather than using programs produced by commercially available valuation software.

Standard optimisations such as selecting appropriate compiler flags, manually optimising dominant routines, and adding parallelism, all led to efficient, scalable code. The implementation of the algorithm from Chapter 3 led to a further increase in performance: analysis of the run times showed that the time taken to calculate the reserves does in fact vary with theoretical expectation, both in terms of speedup resulting from changing from $O(T^2)$ to $O(T)$ for T projection steps, and in terms of the recurrence algorithm's linearity in the number of steps.

Overall, a speedup of around 4×10^5 was achieved: the constituents of this speedup are represented in Figure 10.1, which is based on a diagram which appeared in a course presented by the ARCHER service [28]. The observed speedup represents that actually achieved and the estimated effort required is representative of the work involved in deriving and coding the recurrence algorithm introduced in Chapter 3. This degree of speedup means that a process which currently takes a few days in an commercial

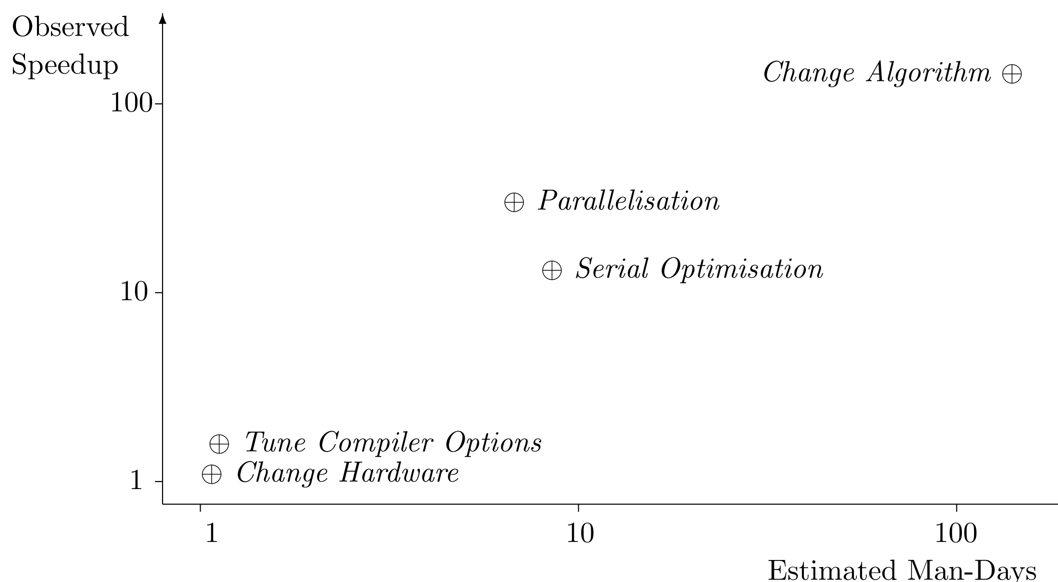


Figure 10.1 Effort and reward during the optimisation of a code which performs profitability calculations on single life annuities.

setting could be achieved in under 1 second.

Chapter 6 considered an approximation which produces monthly death probabilities at fractional ages, based on the interpolation of annual probabilities, tabulated at integral ages. In order to assess the effect of various methods of interpolation, and the approximation, a quartic polynomial was fitted to the force of mortality which underlies a particular table: this allows 'true' survival probabilities to be analytically obtained. As a further simplification, the force of mortality was only modelled over the range of ages where the effect of discounting does not entirely dominate the result of the calculation: while it would have been possible to fit a higher degree polynomial over a wider age range, this is unlikely to reveal anything new about the accuracy of the simplification.

Various methods of directly interpolating in both the l_x's and the q_x's were discussed. Desirable properties for a series of 12 consecutive monthly probabilities are i) that their cumulative effect agrees to the annual probability, and ii) that the progression is sensible with respect to actual lives. None of the direct interpolation methods display both properties, but all methods show one property or other, indicating that none of the direct methods are desirable. The chapter also considered one life assurance company's use of a convoluted method of obtaining the probabilities, based on cubic interpolation in the l_x's to derive a series of monthly q_x's. The values obtained are very close to their theoretically derived counterparts, but the time to perform the calculation is excessive in its contribution to the overall run time.

A simplification, based on a binomial expansion using the q_x's obtained by linear interpolation was considered. The simplification has far fewer arithmetical steps and therefore executes more quickly. Although this method introduces a level of approximation (in addition to that from the interpolation), the probabilities produced closely match the theoretically expected values. Additionally, the difference between using the first three terms of the expansion and only using the first two terms is negligible in terms of the probabilities produced. Furthermore, the change in reserves resulting from the change in methodology is immaterial.

Chapter 7 considered an alternative interpretation of Monte Carlo simulations for time series: that interpretation considers that re-drawing parameters in each time step is not necessary. The chapter contained a complete mathematical formulation of Solvency II and showed that if mortality and morbidity values can be obtained from the relevant life tables in constant time then obtaining the probability of being in a particular state at a particular time has complexity which is linear in the number of lives.

Algorithms for four possible implementations of calculating the additional capital requirement were presented: for a projection over T time steps, the complexity is $O(T^2)$ other than where the recurrence relation is used and parameters are not re-drawn, in which case the complexity is $O(T)$.

Chapter 8 discussed the production and performance of code which performs the calculations required for a brute force assessment of solvency for a variety of life annuity contracts. As with Chapter 5, the data were synthetic, but were generated to have the same properties as data from a commercial environment and so there is reason to believe that the code developed in this project would be useable without significant changes in the commercial setting.

Standard optimisations, such as those used in Chapter 5, were built into the code, and the optimisations considered in Chapter 8 were at a different level. Those optimisations included: the use of tranches to overcome imbalance in the scheduling of the processing of scenarios; placing the files with scenario information into separate directories to assist the file management system; re-calculating the benefits at each step to reduce memory requirements and subsequent data transfers; tuning the length of blocks; and finding the best combination of processes and threads. Although vectorisation was built in to the code wherever possible, it led to an improvement in processing speed of roughly 20% which, in terms of the overall speedup achieved in this project, is not significant.

In addition to the simplification of the interpolation discussed in Chapter 6, the timings of the cash flows were approximated. Rather than using the actual fraction through the

month, all cash flows were assumed to happen in the middle of the month, simplifying both the discounting and the survival calculations: this simple approximation had a significant effect on the performance, but a negligible effect on the financial results.

On a Cray XC30 supercomputer, the entire brute force calculation for the representative portfolio may be performed in under three minutes if sufficient cores are used. There is a lower bound on the run time and that bound is determined by the I/O requirement: it might be possible to reduce the overall time by a factor of say 5, so that the overall time is about 30 seconds, but this would not lead to any further significant changes in working practices. The use of 12,000 cores of a supercomputer is likely to be outside the standard practice of most life assurance companies and so timings were also obtained for smaller numbers of cores: the code scales well, with the deviation from linearity being due to the use of combinations of threads and processes which are not optimal but allow larger numbers of nodes to be used. On a modest cluster, the entire brute force calculation may be performed in under eight hours using only 80 cores. This amount of hardware is well within reach of the majority of life assurance companies: there is therefore no reason why any portfolio of policies of this size should not be processed in a way which meets the requirements of Solvency II.

Analysis of the code on the Cray showed that the routines which dominated the run time were those which have already been highly optimised. The code achieves around 20% of the peak Flop rate, has a high cache hit rate, and spends in excess of 40% of the overall time waiting for data from L_2 cache. These observations indicate that possibilities for extracting further performance are limited: it is likely that greater performance may only be achieved by splitting the tranches into chunks, each of which processes only some of the time steps.

If the interpretation that Monte Carlo simulations for time series do not require parameters to be re-drawn is adopted, then a smaller amount of processing is required for each scenario, but more scenarios are likely to be required. These conflicting factors lead to a reduction in run time when parameters are not re-drawn: there is a speedup of roughly 6× on 5 nodes and roughly 3.6× on 500 nodes Although not re-drawing parameters leads to the processing being done more quickly than when parameters are re-drawn, the scaling is far worse: this is a result of the time to read the policy data not being spread over as many calculations as the number of nodes increases.

Chapter 9 suggested independent areas for future investigations based on the work in this project. Although Chapter 8 concluded that there is no real scope for further significant performance improvements, it may be possible to slightly improve performance

through several minor changes in compiler flags, or through more significant changes to the way in which I/O is performed.

There is scope for extending this code to policies other than annuities. Because the recurrence relation developed in Chapter 3 is completely general, it applies to a wide range of policies. It should therefore be possible to produce highly optimised routines for each of the more common policy types, allowing the policies which make up the majority of liabilities to be valued. Also, the generality of the recurrence suggests the possibility of producing generic code which may be used for less common policy types. As discussed in Section 9.2, such code should be sufficiently general that it requires the coding of the sequence of cash flows as the only additional effort when further contract types are added.

The possibility of transferring this code to two other technologies exists. Firstly, GPUs have a large number of cores, and their use is beneficial to codes which have a large amount of computation with only a small amount of data. The problem in this project meets those criteria and so the code is a good candidate for porting to GPUs. Secondly, Intel's Xeon Phi chips have large numbers of cores and allow hyper-threading with a small number of threads. The results in Tables 8.1 and 8.2 indicate that, when running on CPUs, the code developed in this project runs more quickly with smaller thread counts. It seems reasonable to think that it might perform well on the Xeon Phi when running several tranches at the same time, using a small number of threads per tranche.

Finally, several uses for codes which run this quickly were discussed. These uses are all based on the management's ability to run the business, rather than on statutory reporting. Therefore, although these uses may initially seem unimportant, they will benefit the company by improving the quality of the management information available.

10.2 Summary

The correct fulfilment of the requirements of the Solvency II regulations in the UK life assurance and pensions markets requires a fundamentally different approach to performing the necessary calculations than is currently used within the industry. This project has demonstrated that, using a combination of algorithmic improvements, serial optimisations, vectorisation, parallelisation using MPI and OpenMP, straightforward approximations, and high performance computers it is possible to obtain a significant increase in performance over current commercial software. In Section 1.5, it was estimated that a full brute force Solvency II calculation, with parameters being re-

drawn in each time step, would take around 2810 CPU core years. However, that naive estimate was based on the approach of re-calculating reserves using summation at each step.

From Table 8.4, given a large enough portion of a supercomputer, and an improved algorithm, the calculations can be done in roughly 155 sec (on the wall clock): this is around 1.86×10^9 faster than the naive algorithm, implemented using old compilers and running on out-of-date technology. However, from Figure 5.2, changing to the recurrence algorithm leads to a performance gain of about $140\times$ for a projection over 60 years using monthly steps; Appendix A.2 shows that technological improvements leads to this type of software running about $3.25\times$ faster; and, the 155 seconds in Table 8.4 is based on the use of 12,000 cores of a Cray. The combination of these three factors means that a performance gain of 5.46×10^6 could have been expected from using the new algorithm on a modern platform. Therefore, the work in this project represents a serial speedup of about $340\times$ in addition to gains from improvements in technology, parallelisation of the code, and the new algorithm.

For uses where only one scenario is used (e.g. estimation of profitability), changing the method of calculating in-force reserves from a summation approach to a recurrence approach leads to a reduction in the complexity for the overall calculation from $O(T^2)$ to $O(T)$. Overall, the optimisation of the code used in the single scenario case led to a speedup of about 4×10^5 when using OpenMP parallelisation with 48 threads on 48 cores of the Opteron cluster. This suggests that using 16 threads on 16 cores of a reasonable Xeon CPU could lead to a speedup of at least 10^5, indicating that a profitability calculation, which currently takes a few tens of hours, could be run in a few seconds on entirely affordable hardware. This would allow all life offices to perform profitability calculations more frequently, and on a wide range of parameter values, enabling management decisions which could allow the office to remain profitable in difficult economic climates: e.g. changing the mix of assets in which the office is invested, withdrawing contracts from sale, or changing the level of premiums for various products.

For uses which require multiple scenarios (e.g. Solvency II), whether there is a reduction in the overall complexity as a result of chaging to the new recurrence algorithm depends on whether parameters are re-drawn at every time step: if parameters are re-drawn, then complexity remains $O(T^2)$ (since an $O(T)$ algorithm is used at each future step) but the overall calculation is faster because the volume of power calculations is greatly reduced; if parameters are not re-drawn, then complexity reduces from $O(T^2)$ to $O(T)$.

The work in this thesis makes a full Monte Carlo simulation of solvency a realistic possibility, and therefore should be taken as the start of a migration of assurance companies away from code generated by valuation packages, and towards code which is targeted at up-to-date HPC hardware. Bespoke code should be considered as a means of overcoming the perceived inability to perform the Monte Carlo simulations required for Solvency II. This project has shown that it is possible to perform the full brute force calculation, for a representative portfolio of policies, within time scales which are well within the realms of contemplation.

A large number of people rely on pensions bought from these companies for their financial well-being: as that number increases, there will be a corresponding increase in the importance of the companies remaining solvent, and hence continue paying the pensions. The work in this thesis provides a means by which pension providers can fulfil the Solvency II regulations correctly, ensuring that their assessments of solvency may be performed quickly and accurately, thereby contributing to the continuing financial security of an increasing proportion of the population.

Bibliography

[1] Babbage, C., *Passages from the Life of a Philosopher*, Ch 8., London, U.K., Longman, 1864.

---

[2] AMD, *Software Optimization Guide for AMD64 Processors*, Revision 3.06, 2005. http://support.amd.com/TechDocs/25112.PDF. Accessed 27/09/2016

[3] AMD, *Software Optimization Guide for AMD Family 15h Processors*, Revision 3.08, 2014. http://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf Accessed 27/09/2016

[4] Antypas, K., Shalf, J., and Wasserman, H., *NERSC6 Workload Analysis and Benchmark Selection Process.*, 2009. National Energy Research Scientific Computing Center Division, Berkeley, CA 94720. www.nersc.gov/assets/pubs_presos/NERSCWorkload.pdf. Accessed 12/11/2016

[5] Bacon, D.F., Graham, S.L. and Sharp O.J., *Compiler Transformations for High Performance Computing*, ACM Computing Surveys, Dec 1994.

[6] Benjamin, B., and Pollard, J.H., *The Analysis of Mortality and Other Actuarial Statistics.* UK, Institute Of Actuaries & Faculty Of Actuaries, 1993.

[7] Bethel, E.W., Leinweber, D., Rubel, O., and Wu, K., *Federal Market Information Technology in the Post Flash Crash Era: Roles for Supercomputing*, in Proceedings of Fourth Workshop on High Performance Computational Finance, 2011.

[8] Bik, A., *The Software Vectorisation Handbook*, USA, Intel Press, 2006.

[9] Bodie, Z., Kane, A. and Marcus, A., *Essentials of Investments*, McGraw-Hill, 6th edition, 2007.

[10] Brown, M.B. and Wolfe, R.A., *Estimation of the variance of percentile estimates*, in Computational Statistics and Data Analysis 1, 1983, 167-174.

[11] Cairns, A. J. G., Blake, D. and Dowd, K. *Modelling and Management of Mortality Risk: A Review.* Scandinavian Actuarial Journal, Volume 108, 2008, No. 2, Pages 79-113.

[12] Chang, D.J., Desoky, A.H., Ouyang, M., Rouchka, E.C., *Compute Pairwise Manhattan Distance And Pearson Correlation Coefficient Of Data Points With GPU*. SNPD '09 Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Pages 501-506.

[13] Chen, L. (1996). *Stochastic Mean and Stochastic Volatility - A Three-Factor Model of the Term Structure of Interest Rates and Its Application to the Pricing of Interest Rate Derivatives*. Interest Rate Dynamics, Derivatives Pricing, and Risk Management. Lecture Notes in Economics and Mathematical Systems, 435. Springer. ISBN 978-3-540-60814-1.

[14] Commission of the European Communities. Solvency II (2008): *Amended Proposal for a Directive of the European Parliament and the Council on the Taking-up and Pursuit of the Business of Insurance and Reinsurance [SOLVENCY II].*, COM/2008/0119 final/2 - COD 2007/0143. Brussels, 2008.

[15] Conn, G., *Private Communications within Aegon U.K.*, Nov 2011 - Feb 2012.

[16] Corrigan, A., Camelli, F., Lohner, R. and Wallin, J., *Running Unstructured Grid Based CFD Solvers On Modern Graphics Hardware*, 19th American Institute of Aeronautics and Astronautics: Computational Fluid Dynamics, 2009.

[17] Cox, J.C., Ingersoll, J.E. and Ross, S.A., *A Theory of the Term Structure of Interest Rates.*, Econometrica 53, 1985, Pages 385-407.

[18] Cray, *Cray XC30 Architecture Overview*
`www.archer.ac.uk/training/courses/craytools/pdf/architecture-overview.pdf`
Accessed 07/11/2016

[19] Cray, *Performance Measurement and Analysis Tools*, Revision S-2376-63, 2015.
`http://docs.cray.com/books/S-2376-63/S-2376-63.pdf`
Accessed 27/10/2016

[20] Cvetanoska, V. and Stojanovski, T., *Using High Performance Computing And Monte Carlo Simulation For Pricing American Options*, European University, Skopje.
`arxiv.org/ftp/arxiv/papers/1205/1205.0106.pdf`
Accessed 12/11/2016

[21] Daly, D., Ryu, K.D. and Moreira, J.E., *Multi-variate Finance Kernels in the Blue Gene Supercomputer*, in Proceedings of First Workshop on High Performance Computational Finance, 2008.

[22] Dickson, D.C.M. Hardy, M.R., and Waters, H.R., *Actuarial Mathematics for Life Contingent Risks.* 2nd Edition, Cambridge University Press, 2013.

[23] Dixon, M., Chong, J. and Keutzer, K., *Acceleration of Market Value-at-Risk Estimation*, in Proceedings of Second Workshop on High Performance Computational Finance, 2009.

[24] Doan, V.D., Gaikwad, A., Baude, F. and Bossy, M, *"Gridifying" Classification-Monte Carlo algorithm for pricing high-dimensional Bermudan-American options*, in Proceedings of First Workshop on High Performance Computational Finance, 2008.

[25] Dozsa, G., Eleftheriou, M., Inglett, T.A., King, A.J., Musta, T.E., Sexton, J. and Wisniewski, R.W., *Stream Processing Performance for Blue Gene/P Supercomputer*, in Proceedings of First Workshop on High Performance Computational Finance, 2008.

[26] Durham University, *Institute for Advanced Research Computing* CUDA Centre at Durham University.
`www.dur.ac.uk/iarc/NVIDIAcuda`
Accessed 12/11/2016

[27] EPSRC, *ARCHER Best Practice.*
`http://www.archer.ac.uk/documentation/data-management/`
Accessed 12/11/2016

[28] EPSRC and NERC, *ARCHER: Single Node Optimisation*, Course Notes, Session 1.
`www.archer.ac.uk/training/course-material/2014/12/SglNode_Camb/L01-perfopt-intro.pdf`
Accessed 12/11/2016

[29] EPSRC and NERC, *ARCHER: Single Node Optimisation*, Course Notes, Session 2.
`www.archer.ac.uk/training/course-material/2014/12/SglNode_Camb/L02-nodearch.pdf`
Accessed 12/11/2016

[30] EPSRC, *ARCHER Technical Specifications.*
`www.archer.ac.uk/about-archer/hardware/`
Accessed 12/11/2016

[31] FIS Insurance, *Prophet Financial Modeller.*
`www.prophet-web.com/prophet-products/`
Accessed 20/09/2016

[32] Gerber, R., Bik, A., Smith, K. and Tian, Z., *The Software Optimisation Cookbook*, USA, Intel Press, 2006.

[33] Giles, M.B., Laszlo, E., Reguly, I., Appleyard, J. and Demouth, J., *GPU implementation of finite difference solvers.*, In proceedings of *7th Workshop on High Performance Computational Finance, SuperComputing, 2014*. (Published in *High Performance Computing, Networking, Storage and Analysis 2014 (SC Companion)*).

[34] Gross, A.M. and Clark, V., *Survival Distributions: Reliability Applications in the Biomedical Sciences.*, John Wiley, New York, 1975.

[35] Harris, C., Haines, K. and Staveley-Smith, L., *GPU Accelerated Radio Astronomy Signal Convolution*, Experimental Astronomy, (DOI: 10.1007/s10686-008-9114-9), 2008.

[36] Horn, R.A., and Johnson, C.R., *Topics in Matrix Analysis*, Cambridge Press, 1991.

[37] Hull, J,C., *Options, Futures and Other Derivatives*, Prentice Hall, 3rd edition, 1997.

[38] IBM, *Algo Financial Modeller.*
www-03.ibm.com/software/products/en/algofinamode
Accessed 20/09/2016

[39] Institute Of Actuaries and Faculty Of Actuaries, *Formulae and Tables for Actuarial Examinations*, UK, Institute Of Actuaries & Faculty Of Actuaries, 1980.

[40] Institute Of Actuaries and Faculty Of Actuaries: Continuous Mortality Investigation (Life Office Mortality Committee), *CMI Working Paper 25 (Revised): Stochastic projection methodologies: Lee-Carter model features, example results and implications.* Nov, 2007.

[41] Institute Of Actuaries and Faculty Of Actuaries, *'00' Series Mortality Tables: Permanent Assurances, Males, Smokers*, 2010.
https://www.actuaries.org.uk/learn-and-develop/continuous-mortality-investigation/cmi-mortality-and-morbidity-tables/00-series
Accessed 26/09/2016

[42] Institute Of Actuaries and Faculty Of Actuaries, *'00' Series Mortality Tables: Pensioners, Male, Normal, Lives*, 2010.
https://www.actuaries.org.uk/learn-and-develop/continuous-mortality-investigation/cmi-mortality-and-morbidity-tables/00-series
Accessed 26/09/2016

[43] Intel, Developer Zone (Development, Tools, Resources, RANDOM_NUMBER).
https://software.intel.com/en-us/node/526691.
Accessed 09/09/2016

[44] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2011.
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
Accessed 14/11/2016

[45] Intel, *Intel Fortran Compiler User and Reference Guides*, 2011.
Available as pdf file with compiler distribution.
Replacement at https://software.intel.com/en-us/intel-fortran-compiler-17.0-user-and-reference-guide
Accessed 14/11/2016

[46] Intel, *Intel Xeon Processor data sheet.*
ark.intel.com/products/family/93797/Intel-Xeon-Processor-E7-v4-Family#@Server
Accessed 29/09/2016

[47] International Accounting Standards Board, *The Conceptual Framework for Financial Reporting*, 2010.
www.ifrs.org/News/Press-Releases/Documents/ConceptualFW2010vb.pdf
Accessed 28/09/2016

[48] Irturk, A., Benson, B., Laptev, N. and Kastner, R., *FPGA Acceleration of Mean Variance Framework for Optimal Asset Allocation*, in Proceedings of First Workshop on High Performance Computational Finance, 2008.

[49] Jauvion, G. and Nguyen, T., *Parallelized Trinomial Option Pricing Model On GPU With CUDA*.
`https://www.scribd.com/document/74126721/Parallelized-Trinomial-Option-Pricing-Model-on-GPU-with-CUDA`
Accessed 14/11/2016

[50] Joshi, M.S., *Graphical Asian Options*, Wilmott Journal, 2:2, John Wiley & Sons, Ltd. 2010.

[51] Khronos Group, *The open standard for parallel programming of heterogeneous systems*.
`https://www.khronos.org/opencl/`
Accessed 29/10/2016

[52] Lahabar, S. and Narayanan, P.J., *Singular Value Decomposition On GPU Using CUDA*, IEEE International Parallel Distributed Processing Symposium, 2009.

[53] Lanier, J., *Virtually There*, Scientific American, April 2001.

[54] Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, AdD., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P., *Debunking The* 100× *GPU vs. CPU Myth: An Evaluation Of Throughput Computing On CPU And GPU*, ISCA10, Saint-Malo, France. 2010.

[55] Lindeman, A.J., *Opportunities for Shared Memory Parallelism in Financial Modeling*, in Proceedings of Third Workshop on High Performance Computational Finance, 2010.

[56] Lindley, D.V., and Scott, W.F., *New Cambridge Elementary Statistical Tables*, Cambridge University Press, 1984.

[57] Liu, L., Zhang, Y., Liu, L., Yang, G. and Zheng, W., *Efficient Monte Carlo-Based Options Pricing On Graphics Processors And Its Optimizations*, Science China Information Sciences, Vol 5, Pages 1703-1712, 2010.

[58] Liu, W., Schmidt, B., Voss, G. and Muller-Wittig, W., *Molecular Dynamics Simulations On Commodity GPUs With CUDA*, LNCS 4873, Pages 185-195, 2007.

[59] McCutcheon, J.J. and Scott, W.F., *An Introduction To The Mathematics Of Finance.*, Institute Of Actuaries & Faculty Of Actuaries, 1991.

[60] Microsoft, *High Performance Computing and Insurance Actuarial ModellingFinal*, November 2008.
`download.microsoft.com/documents/uk/business/finance/White-Paper-High-Performance-Computing-and-Insurance-Actuarial-Modelling-Final.pdf`
Accessed 23/11/2017

[61] Milliman, *Actuarial High Performance Computing Increasingly Accessible And Indispensable*, May 2008.
`www.milliman.com/insight/Articles/Actuarial-high-performance-`

computing-Increasingly-accessible-and-indispensable/
Accessed 23/11/2017

[62] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*, September 2009.
www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
Accessed 14/11/2016

[63] Molka, D., Hackenberg, D., Robert Schöne, R. and Matthias S. Müller, M.S., *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System*, in Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques. 2009.

[64] Moore, G.E., *Cramming More Components onto Integrated Circuits*, Electronics, pp. 114-117, April 19, 1965.

[65] Neill, A., *Life Contingencies*, Institute Of Actuaries and Faculty Of Actuaries, 1989.

[66] Nichols, B., Buttlar, D., and Proulx Farrell, J., *Pthreads Programming*, USA, O'Reilly, 1996.

[67] NVIDIA, *What is GPU Computing?*.
http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html
Accessed 29/10/2016

[68] NVIDIA, *NVIDIA Tesla K80*.
http://www.nvidia.com/object/tesla-k80.html
Accessed 10/11/2016

[69] NVIDIA, *Tesla K80 GPU Accelerator*.
https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf
Accessed 10/11/2016

[70] Office for National Statistics, *English Life Tables, No 17*.
https://www.ons.gov.uk/file?uri=/peoplepopulationandcommunity/birthsdeathsandmarriages/lifeexpectancies/datasets/1englishlifetables/current/elt17_tcm77-414349.xls
Accessed 26/09/2016

[71] OpenACC-standard.org, *OpenACC Standard for Parallel Computing*, Nov 2011.
http://www.openacc.org/node/93
Accessed 02/10/2016

[72] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 3.1*, July 2011.
www.openmp.org/mp-documents/OpenMP3.1.pdf
Accessed 02/10/2016

[73] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 4.0*, July 2013.
www.openmp.org/mp-documents/OpenMP4.0.0.pdf
Accessed 02/10/2016

[74] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 4.5*, November 2015.
www.openmp.org/mp-documents/openmp-4.5.pdf
Accessed 02/10/2016

[75] Oxford Actuarial Consultants, *Mo.net*.
www.oac-mo.net
Accessed 20/09/2016

[76] Pakin, S., *Ten Ways To Fool The Masses When Giving Performance Results On GPUs*, HPCwire, Dec 2011.
www.hpcwire.com/hpcwire/2011-12-13/ten_ways_to_fool_the_masses_when_giving_performance_results_on_gpus.html
Accessed 14/11/2016

[77] RNA Analytics, R^3S.
www.rnaanalytics.com
Accessed 23/11/2017

[78] Ross, S.M., *A Course in Simulation.*, USA, Macmillan Publishing Co, 1990.

[79] The Royal Society of Edinburgh, *Computation in Finance & Insurance, post-Napier*, April 2014.
www.rse.org.uk/wp-content/uploads/2016/10/Computation_-in_Finance_and_Insurance-post-Napier.pdf
Accessed 23/11/2017

[80] SciComp, *SciFinance*.
www.scicomp.com
Accessed 23/11/2017

[81] SciComp, *Rapid Development Model for Valuation of Exotic Option Instruments with SciFinance and Integration into a 3rd Party Vendor System*.
http://www.scicomp.com/company/case_study/request_CBA
Accessed 23/11/2017

[82] Society of Actuaries, *The Actuary's HPC Challenge*, CompAct, Apr 2008.

[83] Society of Actuaries, *Actuarial Software Wins Microsoft Innovation Award*, CompAct, Oct 2011.

[84] Smith, A., *Private Communications*, May 2018.

[85] Tian, Y., Zhu, Z., Klebaner, F.C. and Hamza, K., *Pricing Barrier And American Options Under The SABR Model On The GPU*, Concurrency Computat.: Pract Exper, 2011.

[86] Tucker, M., *The Optimisation of a Financial Modelling Code.*, Dissertation for MSc in High Performance Computing. The University of Edinburgh, 2009.
static.ph.ed.ac.uk/dissertations/hpc-msc/2008-2009/Mark_Tucker.pdf
Accessed 14/11/2016

[87] Tucker, M. and Bull, J.M., *The Application of High Performance Computing to Solvency and Profitability Calculations for Life Assurance Contracts.*, In

proceedings of 5th *Workshop on High Performance Computational Finance, SuperComputing, 2012.* (Published in *High Performance Computing, Networking, Storage and Analysis 2012 (SC Companion)*).

[88] Tucker, M. and Bull, J.M., *An efficient algorithm for the calculation of reserves for non unit-linked life policies.*, in Algorithmic Finance (2014), 3:3-4, 143-161.

[89] Tucker, M. and Bull, J.M., *Fulfilling Solvency II Regulations using High Performance Computing.*, In proceedings of 8th *Workshop on High Performance Computational Finance, SuperComputing, 2015.* (Published in *High Performance Computing, Networking, Storage and Analysis 2015 (SC Companion)*).

[90] Ufimtsev, I. and Martinez, T.J., *Quantum Chemistry On Graphical Processing Units. 3. Analytical Energy Gradients, Geometry Optimization, And First Principles Molecular Dynamics*, J. Chem. Theory Comput, Volume 5, Pages 2619-2628, 2009.

[91] Vasicek, O. (1977). *An Equilibrium Characterisation of the Term Structure.* Journal of Financial Economics 5 (2): pp.177-188.

[92] wccftech (online). *Intel Launches Its 14nm Broadwell-EX Platform*, May 2016. `http://wccftech.com/intel-broadwell-ex-xeon-e7-8890-24-cores/` Accessed 26/10/2016

[93] Williams, T.J., *Distributed Calculations on Fixed-Income Securities*, in Proceedings of Second Workshop on High Performance Computational Finance, 2009.

[94] WillisTowersWatson, *MoSes.* `https://www.towerswatson.com/en-GB/Services/Tools/moSes` Accessed 14/11/2016

[95] Wolfram, *Wolfram Mathworld: Stochastic Matrix.* `https://mathworld.wolfram.com/StochasticMatrix.html` Accessed 17/08/2017

[96] Zitzlsberger, G., *Intel Architecture for HPC Developers.* `https://indico.cern.ch/event/403113/contributions/1847268/attachments/1123555/1603259/01_Intel_Architecture_for_HPC_Developers.pdf` Accessed 11/11/2016

[97] 7-cpu, *Intel Architecture for HPC Developers.* `www.z-cpu.com/cpu/IvyBridge.thml` Accessed 11/11/2016

List of Figures

(2.1) Representation of a typical *Variable Input Screen*, as used in a standard actuarial valuation software package. 20

(2.2) Schematic overview of a standard actuarial valuation software package. . 21

(2.3) Traversing the 'policy *vs.* time step' iteration space. 48

(4.1) State transition diagram for one life and two states. 79

(4.2) State transition diagram for one life and three states. 82

(4.3) State transition diagram for one life and three states, where one transition is reversible. 84

(4.4) State transition diagram for two lives and two states. 86

(4.5) State transition diagram for two lives and three states. 88

(5.1) Time (in milliseconds) to calculate reserves for single life annuities using naive summation and recurrence algorithm: times were obtained using a single core of the Cray, and are the average over 5000 policies. 103

(5.2) Speedup (for Single Life Annuities) resulting from change from naive summation to recurrence relation: times were obtained using a single core of the Cray, and are the average over 5000 policies. 104

(5.3) Processing times (in microseconds) using recurrence relation for different annuity contracts: times were obtained using a single core of the Cray, and are the average over 100,000 policies. 105

(5.4) Performance using OpenMP on the Opteron cluster: for each thread count, the ideal processing rate is derived from the time taken to process 4×10^5 single life annuity policies on one thread. 107

(6.1) Polynomials of low degree fitted to $-\ln(p_x)$ where the q_x's are from PMA92 Ultimate. 111

(6.2) Annual death probabilities as tabulated in PMA92 Ultimate, and derived from fitting a quartic polynomial to the force of mortality. 114

(6.3) Progression of probability of death within the next month, for a life initially aged $65\frac{1}{3}$, increasing by $\frac{1}{12}^{\text{th}}$ thereafter, using linear interpolation in the l_x's. 116

(6.4) Required age deduction to ensure equivalence of average monthly mortality and average yearly mortality. 122

(6.5) Deviation from known formulaic value of monthly probabilities of death, using different interpolation methods, for a life initially aged $65\frac{1}{3}$, with the age increasing by $\frac{1}{12}^{\text{th}}$ thereafter, to a maximum of age 70. 124

(6.6) Deviation from known formulaic value of monthly probabilities of death, using different interpolation methods, for a life initially aged $65\frac{1}{3}$, with the age increasing by $\frac{1}{12}^{\text{th}}$ thereafter, to a maximum of age 95. 125

(8.1) Strong scaling on the Cray XC30: the ideal processing rate is based on the time using 5 nodes (as presented in Table 8.4), i.e. including I/O, for 1000 tranches, using a block size of 16. 163

(8.2) Summary sampling results from CrayPAT when processing 50% of the data over 8 tranches using 8 processes, each with 3 threads, and a block size of 16. 164

(8.3) Statistics from CrayPAT when processing 50% of the data over 8 tranches using 8 processes, each with 3 threads, and a block size of 16. 165

(8.4) Strong scaling on the Cray XC30 when parameters are not re-drawn each time step: the ideal processing rate is based on the time using 5 nodes (as presented in Table 8.15 and 8.16), i.e. including I/O, for 24,000 and 96,000 scenarios. The code used a block size of 8 with double precision and vectorisation included. 181

(10.1) Effort and reward during the optimisation of a code which performs profitability calculations on single life annuities. 197

List of Tables

(5.1) Performance rate (policies per second) for profitability calculations for single life annuities resulting from optimisations in MSc project: the speedup relates to the individual stage, but the rate is cumulative. . . . 101

(5.2) Performance rate (policies per second) as optimisation of profitability code progressed. 106

(6.1) Closeness of interpolation to expected values for monthly probability of death. 126

(6.2) Initial reserves for representative portfolio of 500,000 policies: the 'commercial' implementation uses cubic interpolation in the l_x's (as described in Section 6.4.2), and the 'approximation' uses the algorithm in Section 6.6. 126

(8.1) Run times (in seconds) to process 500,000 policies, over 96 tranches using various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 4 nodes of the Cray XC30. 154

(8.2) Run times (in seconds) to process 500,000 policies over 32 tranches, using various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 1 node of the Xeon cluster. 155

(8.3) Run times (in seconds) to process 500,000 policies over 32 tranches using various block sizes, on 1 node of the Xeon cluster. 157

(8.4) Run time (in seconds) to process 500,000 policies over 1000 tranches using a block size of 16, and various numbers of nodes on the Cray XC30. 162

(8.5) Count of operations and function calls while processing 50% of the data, over one tranche, running on one node of the Cray, with the frequency set to 2.7 GHz using the `--p-state` option for `aprun`. 167

(8.6) Metrics from CrayPAT using single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, one thread per process, all processes on same socket, CPU frequency fixed using `--p-state=2700000`. 168

(8.7) Derived metrics for single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, one thread per process, all processes on same socket, CPU frequency fixed using `--p-state=2700000` 170

(8.8) Metrics from CrayPAT using OpenMP-enabled code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, 3 threads per process, CPU frequency fixed using `--p-state=2700000`. Note that the figures for Aggregate Flop Rate are under-reported by a factor of three (i.e. the number of threads per process): see Section 8.4.1.3.3 for details. 171

(8.9) Derived metrics for single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one scenario per process, 3 threads per process, CPU frequency fixed using `--p-state=2700000`. 172

(8.10) Run times (in seconds) to process 500,000 policies over 96 tranches, using single precision, vectorised code and various block sizes, but pure MPI, on 4 node of the Cray. 173

(8.11) Metrics from CrayPAT using single-threaded code on 1 node when parameters are re-drawn: 500,000 policies, one socket, twelve processes per socket, one scenario per process, one thread per process, CPU frequency fixed using `--p-state=2700000` 174

(8.12) Run time (in seconds) to process 500,000 policies over 200 tranches using a block size of 32, on 1 node of the the Xeon cluster. 176

(8.13) Run times (in seconds) to process 10 scenarios per read of the data for 500,000 policies, over 96 tranches using various combinations of block size, number of MPI processes and OpenMP threads per process, on 4 nodes of the Cray XC30. 178

(8.14) Run times (in seconds) to process 500,000 policies with varying number of scenarios: parameters for linear models are included. The processing was for 96 tranches using a block size of 8 and various combinations of block size, number of MPI processes and number of OpenMP threads per process, on 4 nodes of the Cray XC30. 179

(8.15) Run time (in seconds) on the Cray XC30 when parameters are not re-drawn each time step: pure MPI was used to process 24,000 scenarios using a block size of 16. 180

(8.16) Run time (in seconds) on the Cray XC30 when parameters are not re-drawn each time step: pure MPI was used to process 96,000 scenarios using a block size of 16. 180

(A.1) Run times (in seconds) to process 1000 policies of each type using the initial code from the MSc after compiling with the Cray compiler on the Cray: since the code is single threaded, only one core was used. 220

(B.1) Selected details of the E5-2697 2.7 GHz Ivy Bridge processors in the Cray XC30. Note that 518 GFlops per node results from having two 12-core chips per node. 221

(B.2) Summary of nodes in the Xeon cluster. 223

(D.1) Estimation of portion of time in $O\left(T^2\right)$ calculations. 229

List of Algorithms

(2.1) Additional capital requirement for single life annuities using a brute force approach. 29

(3.1) Calculation of in-force reserves using a summation approach. 73

(3.2) Calculation of in-force reserves using the recurrence approach. 74

(7.1) Calculation of ACR using summation and re-drawing parameters. 140

(7.2) Calculation of ACR using recurrence relation and re-drawing parameters. 141

(7.3) Calculation of ACR using summation but not re-drawing parameters. . 142

(7.4) Calculation of ACR using recurrence relation but not re-drawing parameters. 143

(8.1) Structure of loop nest in C implementation of brute force approach to obtaining the additional capital requirement for Solvency II. 150

Appendix A

Estimation of Run Time for Brute Force Solvency II in the Commercial Environment

In order to obtain the best accuracy, the projection should be performed with the smallest time steps which reasonably represent reality, i.e. monthly steps. In a commercial environment, using industry standard software in a Windows environment on desktop PCs, profitability calculations take about 35 CPU hours for 129,000 single life annuity policies, and about $22\frac{1}{2}$ CPU hours for 32,000 reversionary annuity policies. Since the programs are single threaded, in addition to being CPU core hours, these measurements are also times on the wall clock. In terms of meaningful metrics, this translates to average times to process each policy of roughly 1.0 seconds per policy for single life annuities, and 2.5 seconds per policy for reversionary annuities.

A.1 Initial Estimate

A.1.1 Derivation based on Time Steps

Let the average outstanding term of a policy be T months. Then the performance statistics of interest are $1.0/T$ seconds per policy per monthly step for single life annuities, and $2.5/T$ seconds per policy per monthly step for reversionary annuities.

The number of time steps that the average policy is involved in the Solvency II

projection is

$$
\begin{aligned}
T \text{ (for the `best estimate')} \quad & + 1000 \times (T-1) \quad \text{(for scenarios at step 1)} \\
& + 1000 \times (T-2) \quad \text{(for scenarios at step 2)} \\
& + \cdots \\
& + 1000 \quad \text{(for scenarios in the last step)}
\end{aligned}
$$

which is

$$\approx 500\, T^2$$

The representative portfolio of policies used in this project also contains joint life and last survivor policies. The times to process these policies in a commercial environment are not available but, since they are two-life annuities, it is assumed that their processing times will be similar to those for reversionary annuities. Hence, the time to process the representative portfolio is estimated as

$$
500\, T^2 \times \begin{pmatrix}
 & 300000 \times \frac{1.0}{T} & \text{(for the single life annuities)} \\
+ & 100000 \times \frac{2.5}{T} & \text{(for the reversionary annuities)} \\
+ & 50000 \times \frac{2.5}{T} & \text{(for the joint life annuities)} \\
+ & 50000 \times \frac{2.5}{T} & \text{(for the last survivor annuities)}
\end{pmatrix}
$$

which is

$$500\, T^2 \times \left(\frac{300000}{T} + \frac{2.5 \times 200000}{T} \right) = 4.0 \times 10^8\, T$$

It is common actuarial practice to allow for future improvements in mortality by considering lives as being younger than they actually are. Suppose that the average age of the policyholder at the valuation date is 65: then, to allow for the reduction in age, consider the average age the valuation date to be 60. Allowing for the boundary condition, ${}_tp_x = 0$ for $x > 120$, the average outstanding term is 60 years, i.e. $T = 720$ months. Hence, the estimated overall run time is

$$\left(4.0 \times 10^8\right) \times 720 = 2.88 \times 10^{11} \text{ CPU core seconds} \tag{A.1}$$

i.e. around 9130 CPU core years.

A.1.2 Derivation based on Number of Calculations

Suppose the number of operations in a profitability calculation is M, and that the maximum step number is T. Then, for the brute force calculation of solvency:

> the number operations in the first step is 1000 M;
> the number operations in the second step is $1000\,M\,\dfrac{T-1}{T}$;
> the number operations in the third step is $1000\,M\,\dfrac{T-2}{T}$;
> and so on until the end of the projection.

Hence, the total number of operations is

$$\begin{aligned}\sum_{t=1}^{T}\left(1000\,M\,\frac{T-t}{T}\right) &= \frac{1000\,M}{T}\sum_{t=1}^{T}(T-t)\\ &= \frac{1000\,M}{T}\sum_{t=1}^{T-1}t \qquad\qquad \approx 500\,MT\end{aligned}$$

For a projection over 60 years, using monthly steps, $T = 720$ so that the total number of operations is $3.6 \times 10^5 M$. Hence, the volume of operations increases by a factor of $360,000\times$

Therefore, for the portfolio of 300,000 single life policies and 200,000 two life policies, the expected run time (assuming that all two life annuities take the same time per policy as reversionary annuities) is

$$\begin{aligned}360,000 \times &\begin{pmatrix}\frac{35\text{ hrs}}{129000} \times 300000 \quad \text{(for the single life annuities)}\\ + \\ \frac{22.5\text{ hrs}}{32000} \times 200000 \quad \text{(for the two life annuities)}\end{pmatrix}\\ &= 8.0 \times 10^7 \text{ hrs}\\ &\equiv 2.89 \times 10^{11} \text{ seconds}\end{aligned}$$

which corroborates the value in Equation A.1.

Annuity Type	Single Life	Reversionary
Run Time (sec)	306.8	774.9
Seconds per Policy	0.3068	0.7749
Improvement	3.259	3.226

Table A.1 Run times (in seconds) to process 1000 policies of each type using the initial code from the MSc after compiling with the Cray compiler on the Cray: since the code is single threaded, only one core was used.

A.2 Allowing for Technological Advances

The estimates in Section A.1 are based on software produced by a commercial valuation package, running on desktop PCs in one particular life office at the time this project was first considered, i.e. 2009, when the MSc dissertation [86] was completed. The code produced for the MSc originally ran at the same rate as the commercial software when run in a similar environment [86, Section 1.3]. Therefore, in order to allow for advances in technology and improvements in compilers during the lifetime of this project, the initial code from the MSc has been re-run on the Cray under identical compilation conditions, i.e. single threaded, with no optimisation and full debugging info enabled: the run times and resultant improvement in performance (in relation to the life office's PCs) are shown in Table A.1.

From the results in Table A.1, it is prudent to consider that a speedup of about 3.25× results from changing from the Compaq compiler and PC-based CPUs from 2009 to the Cray compiler and Xeon CPUs in our Cray in 2016. Therefore, the estimate in Equation A.1 should be reduced by this factor, so that the estimated overall run time is

$$\frac{2.88 \times 10^{11} \text{ CPU core seconds}}{3.25} \equiv 8.86 \times 10^{10} \text{ CPU core seconds}$$

i.e. around 2810 CPU core years.

Appendix B

Hardware Specifications

Sections 2.5.2.2, 2.5.2.3 and 2.7.3 quote figures relating to hardware specifications: those details are summarised in this Appendix.

B.1 Features of a Xeon CPU

The nodes on the Cray XC30 have two Intel Xeon E5-2697 2.7 GHz Ivy Bridge processors on each node [30]: the features of these Xeons are summarised by Cray [18], and have been repeated in Table B.1.

Cores per die	12
Dies per node	2
Each core has:	
User threads	2
Function group	1 AVX (vector)
bits wide	256 bits wide
functional units	1 add and 1 multiply
L1 cache (per core)	32 kB
L2 cache (per core)	256 kB
L3 cache (per die)	30 MB
Cache BW Per core (GB/s)	
L1 / L2 / L3	100 / 40 / 23
Peak DP Flops per core	8 Flops/clk
Peak DP Flops per node	518 GFlops

Table B.1 Selected details of the E5-2697 2.7 GHz Ivy Bridge processors in the Cray XC30. Note that 518 GFlops per node results from having two 12-core chips per node.

Other features of Intel Xeon Ivy Bridge processors [96] may be summarised as:

- ⊛ cache latencies:

Cache Level	Latency (cycles)
L1 D	4
L2 (unified)	12
L3	26-31
L2 and L1 D-Cache in other cores	43 or 60

- ⊛ maximum memory bandwidth = 59.7 GB/s
- ⊛ instruction set extension = Intel AVX

Pertinent times for Intel Xeon Ivy Bridge processors [97] may be summarised as:

Item	Value
RAM Latency	30 cycles + 53×10^{-9} sec
Branch misprediction	14 cycles

Converting these values to times to retrieve data into a location from the next layer away from the registers, on a 2.7 GHz processor, gives:

Location	How Long	Time (nanoseconds)
L1 Cache	4 cycles	1.5
L2 Cache	12 cycles	4.4
L3 Cache	30 cycles	11.1
RAM	30 cycles + 53 nanoseconds	64.1

The cumulative sums of these values are the times to retrieve data from a certain location into a register on a 2.7 GHz processor: these are the values quoted in Section 2.5.2.2.

B.2 Nodes on the Xeon Cluster

The Xeon cluster has a large number of different types of nodes: these are enumerated at the ECDF's website[1], but that is only accessible from within the Edinburgh University domain and so the information is reproduced in Table B.2. Despite this variety of available nodes, the code in this project does not require vast amounts of memory per core. Therefore, all runs were done on "Standard 64G" nodes: these are the nodes described in Section 2.7.3.

[1] `https://www.wiki.ed.ac.uk/display/ResearchServices/Memory+Specification`

Description	Cores	RAM	Nodes	Xeon Processor
Standard 64G	16	64 GB	71	E5-2630 v3 (2.4 GHz)
Standard 128G	16	128 GB	22	E5-2630 v3 (2.4 GHz)
Intermediate 192G	16	192 GB	63	E5-2630 v3 (2.4 GHz)
Intermediate 256G	16	256 GB	12	E5-2630 v3 (2.4 GHz)
Large 512G	16	512 GB	10	E5-2630 v3 (2.4 GHz)
Large 768G	16	768 GB	6	E5-2630 v3 (2.4 GHz)
Large 2T	32	2 TB	2	E7-4820 v2 (2.0 GHz)
IGMM Standard	16	128 GB	107	E5-2630 v3 (2.4 GHz)
IGMM Large	16	768 GB	6	E5-2630 v3 (2.4 GHz)
Roslin Standard 256G	16	256 GB	16	E5-2630 v3 (2.4 GHz)
Roslin Intermediate 512G	16	512 GB	12	E5-2630 v3 (2.4 GHz)
Roslin Intermediate 768G	16	768 GB	8	E5-2630 v3 (2.4 GHz)
Roslin Large 2T	40	2 TB	2	E7-4820 v3 (1.9 GHz)
Roslin Large 3T	40	3 TB	2	E7-4820 v3 (1.9 GHz)

Table B.2 Summary of nodes in the Xeon cluster.

B.3 Features of an NVIDIA GPU

In addition to the CPU nodes in the Xeon cluster (summarised in Section B.2), the cluster also has 2 nodes which each have one Xeon E5-2630 v3, 2.4 GHz CPU and one NVIDIA Tesla K80 GPU [68, 69]. The relevant features of a Tesla K80 may be summarised as:

- there are 4992 NVIDIA CUDA cores with a dual-GPU design, i.e. there are two GK210 GPUs, each having 2496 cores;
- the base clock speed is 560 MHz, although this may be boosted to 562–875 MHz;
- double-precision performance is up to 2.91 TFlop/s (with boost enabled), and single-precision performance is up to 8.73 TFlop/s (with boost enabled);
- memory bandwidth is 480 GB/s, i.e. 240 GB/s for each of the two GK210 GPUs;
- there is 24 GB GDDR5 memory, i.e. 12 GB for each of the two GK210 GPUs.

B.4 Comparisons

From the data in Sections B.1 and B.3:

i) the Flop rate of a GPU is 2.91 TFlop/s whereas the tabulated value for the Flop rate of CPUs is 518 GFlop/s per CPU node which, allowing for two CPUs per node, means that a GPU has about 11.2× the Flop rate of a CPU;

ii) the memory bandwidth of a GPU is 480 GB/s whereas the memory bandwidth of a CPU is 59.7 GB/s so that a GPU has about 8$\times$ the memory bandwidth of a CPU.

Appendix C

Number of States for Assurances

Using only two states in the vector form of the recurrence relation, as derived in Chapter 3, cannot work for Assurances.

Proof by Contradiction

For the simplest, single life, whole life assurance, where a benefit of 1 is paid at the end of the step in which death occurs, the relevant relation, in non-vector form, using standard actuarial notation, is

$$A_x = v\, q_x\, 1 + v\, p_x\, A_{x+1} \tag{C.1}$$

Suppose that it is possible to produce a vector form of the recurrence using only two states. Then, because the benefit is payable at the end of the step, the two transition matrices in Equation 3.2.6 are the same and, because 'payment at the end of the step' relates to $f = 1$, the possible transitions dictate that the matrix is

$$\begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix}$$

Using the approach with only two states, state 0 refers to 'alive' so that the required reserve factor is A_x and state 1 refers to 'dead' so that the required reserve is 0. Hence, the vector for the reserves required is

$$\begin{pmatrix} A_x \\ 0 \end{pmatrix}$$

Therefore, the full vector equation becomes

$$\begin{pmatrix} A_x \\ 0 \end{pmatrix} = v \begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \theta \\ \phi \end{pmatrix} + v \begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} A_{x+1} \\ 0 \end{pmatrix} \tag{C.2}$$

where θ and ϕ are the, as yet undetermined, cash flows which depend on the state occupied at the end of the step.

For state 0, the top row of Equation C.2 gives

$$A_x = v\, p_x\, \theta + v\, q_x\, \phi + v\, p_x\, A_{x+1}$$

and, to match the non-vector form in Equation C.1, the required cash flow values are

$$\begin{cases} \theta = 0 \\ \phi = 1 \end{cases} \tag{C.3}$$

For state 1, the bottom row of Equation C.2 gives

$$0 = v\, \phi$$

and, using the value of ϕ from state 0 (per Equation C.3), this gives

$$0 = v$$

which is a contradiction ... unless the interest rate is $i = \infty$

The absurdity of an infinite interest rate is highlighted by the interpretation that, by investing the smallest monetary unit possible, the interest earnt on that investment, in the shortest time possible, will be infinite: such interest will cover any payments which could ever be made, and hence there is never any need to hold any reserves.

Appendix D

Regression Analyses

D.1 Summation *vs.* Recurrence Relation

D.1.1 Individual Regression Curves

Let T be the number of time steps in the projection, and let t_s be the average time per policy using the summation approach, and t_r be the average time per policy using the recurrence approach. Then fitting polynomials to the data represented by Figure 5.1 gives

$$t_s = 1.0910 \times 10^{-8}\, T^2 + 1.7092 \times 10^{-7}\, T + 4.2454 \times 10^{-5}$$

and

$$t_r = 6.0110 \times 10^{-8}\, T - 5.6484 \times 10^{-7}$$

Let R_s and R_r be correlation coefficients for the summation and recurrence approaches, respectively. Then $R_s^2 = 0.999998$ and $R_r^2 = 0.999949$. The high degree of correlation indicates how well the times for the summation and recurrence approached are represented by quadratic and linear curves, respectively: this reinforces the derivation of the complexities in Section 3.4.1.

D.1.2 Speedup

The speedup resulting from the change of algorithm is represented in Figure 5.2: using T for the number of time steps in the projection, and σ for the speedup, the line of best-fit has equation

$$\sigma = 0.17924\,T + 7.3649$$

with a correlation coefficient, R_σ, given by $R_\sigma^2 = 0.999913$. The high degree of correlation suggests that the speedup is linear, and that is expected from the fact that the algorithm's complexity has changed from $O(T^2)$ to $O(T)$ for T time steps.

D.1.3 Portion of Time in $O(T^2)$ Calculations

The portion of time in $O(T^2)$ calculations may be approximated using an approach similar to Amdahl's law.

Let λ be the portion of time spent in $O(T^2)$ calculations while calculating the reserve, so that $1-\lambda$ is the portion of time spent in $O(T)$ calculations: note that this assumes that the portion of $O(1)$ calculations is zero – which may or may not be true. Then, under this construction, the total time for T steps using the summation approach is t, which may be decomposed as $\lambda t + (1-\lambda)\,t$. Hence, the expected time for T steps using the recurrence approach is $\dfrac{\lambda t}{T} + (1-\lambda)\,t$ and the expected speedup, Γ, is therefore

$$\Gamma = \frac{\lambda t + (1-\lambda)\,t}{\frac{\lambda}{T}t + (1-\lambda)\,t} = \frac{T}{\lambda + (1-\lambda)\,T}$$

Let T_s be the measured time to calculate the reserves using the summation algorithm, and let T_r be the measured time using the recurrence algorithm. Then the measured speedup is $\sigma = \dfrac{T_s}{T_r}$. Equating the measured speedup to the expected speedup, so that the measured speedup is an estimator for the expected value, gives

$$\sigma = \frac{T}{T-(T-1)\,\lambda}$$

from which

$$\lambda = \frac{1-\frac{1}{\sigma}}{1-\frac{1}{T}} \tag{D.1}$$

T	300	360	480	540	600	660	720	840	960	1020
σ	61.6	72.3	93.0	103.7	114.4	125.4	136.5	157.6	179.5	190.9
λ	0.987	0.989	0.991	0.992	0.993	0.994	0.994	0.995	0.995	0.996

Table D.1 Estimation of portion of time in $O\left(T^2\right)$ calculations.

The times T_s and T_r, which were used to create Figure 5.2, are the times purely for the reserve calculation – all ancillary calculations (such as obtaining cash flow amounts and q_x's) are performed prior to the reserve calculation. Table D.1 contains the estimates (obtained using Equation D.1) of proportion of time spent in $O(T^2)$ calculations while calculating the reserve. From Table D.1, it is apparent that around 99% of the time calculating reserves is spent in $O(T^2)$ calculations.

D.2 Single Life and Reversionary Annuities

Let T be the number of time steps in the projection, and let t_{sl} be the average time per policy to perform profitability calculations, using the recurrence approach, for single life annuities, and t_{ra} be the average time per policy to perform profitability calculations, using the recurrence approach, for reversionary annuities. Then fitting straight lines to the data represented by Figure 5.3 gives

$$t_{sl} = 1.5978 \times 10^{-8}\, T - 1.5144 \times 10^{-7}$$

and

$$t_{ra} = 2.5815 \times 10^{-8}\, T - 2.5360 \times 10^{-7}$$

Let R_{sl} and R_{ra} be correlation coefficients for the single life and reversionary annuities, respectively. Then $R_{sl}^2 = 0.9999995$ and $R_{ra}^2 = 0.9999999$

Appendix E

Probability of Transition before Time t

It is not possible to move between states in zero time so, for the start step of 1, i.e. time $t = 0$,

$$\mathbf{H}_{x,0} = \mathbf{I} = \mathbf{H}_{xy,0}$$

For the start step of 2, i.e. time $t = 1$, there is only one transition possible (i.e. that which may have occurred during the interval from time $t = 0$ to time $t = 1$), and so the matrices $\mathbf{H}_{x,2}$ and $\mathbf{H}_{xy,2}$ are simply the relevant transition matrices from Chapter 4. For larger values of t, $\mathbf{H}_{x,t}$ and $\mathbf{H}_{xy,t}$ must be obtained by multiplying the matrix for $\mathbf{H}_{x,t-1}$ or $\mathbf{H}_{xy,t-1}$ with the matrix for probability of transition during the interval from time $t-1$ to time t. Each case is considered separately in the remainder of this chapter.

E.1 Single Life Policies

E.1.1 Two States

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp_x & {}_tq_x \\ 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Proof

$$\begin{aligned}
\mathbf{H}_{x,1} &= \begin{pmatrix} p_x & q_x \\ 0 & 1 \end{pmatrix} \\
&\equiv \begin{pmatrix} {}_1p_x & {}_1q_x \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_tp_x & {}_tq_x \\ 0 & 1 \end{pmatrix} \qquad \text{where } t = 1
\end{aligned}$$

Hence the result holds for $t = 1$.

Suppose the result is true for $t = k$ where $k \geq 1$, i.e. $\mathbf{H}_{x,k} = \begin{pmatrix} {}_kp_x & {}_kq_x \\ 0 & 1 \end{pmatrix}$

Then

$$\begin{aligned}
\mathbf{H}_{x,k+1} &= \mathbf{H}_{x,k} \begin{pmatrix} p_{x+k} & q_{x+k} \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_kp_x & {}_kq_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p_{x+k} & q_{x+k} \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_kp_x \, p_{x+k} & {}_kp_x \, q_{x+k} + {}_kq_x \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_{k+1}p_x & {}_{k|}q_x + {}_kq_x \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_{k+1}p_x & {}_{k+1}q_x \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_tp_x & {}_tq_x \\ 0 & 1 \end{pmatrix} \qquad \text{where t=k+1}
\end{aligned}$$

Hence the result is true for $t = k + 1$ whenever it is true for $t = k$, and the result is true for $t = 1$. Therefore, by induction, the result is true for all $t \geq 1$, i.e.

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp_x & {}_tq_x \\ 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Q.E.D.

E.1.2 Three States

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Proof

$$\begin{aligned} \mathbf{H}_{x,1} &= \begin{pmatrix} p_x & q_x & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &\equiv \begin{pmatrix} {}_1p_x & {}_{0|}q_x & {}_0q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{where } t = 1 \end{aligned}$$

Hence the result holds for $t = 1$.

Suppose the result is true for $t = k$ where $k \geq 1$, i.e. $\mathbf{H}_{x,k} = \begin{pmatrix} {}_kp_x & {}_{k-1|}q_x & {}_{k-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

Then

$$\begin{aligned} \mathbf{H}_{x,k+1} = \mathbf{H}_{x,k} &\begin{pmatrix} p_{x+k} & q_{x+k} & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} {}_kp_x & {}_{k-1|}q_x & {}_{k-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_{x+k} & q_{x+k} & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} {}_kp_x\, p_{x+k} & {}_kp_x\, q_{x+k} & {}_{k-1|}q_x + {}_{k-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} {}_{k+1}p_x & {}_{k|}q_x & {}_kq_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{where } t = k+1$$

Hence the result is true for $t = k + 1$ whenever it is true for $t = k$, and the result is true for $t = 1$. Therefore, by induction, the result is true for all $t \geq 1$, i.e.

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Q.E.D.

E.1.3 Three States with One Reversible Transition

$$\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp_x^{00} & {}_tp_x^{01} & {}_tp_x^{02} \\ {}_tp_x^{10} & {}_tp_x^{11} & {}_tp_x^{12} \\ 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Proof

$$\begin{aligned} \mathbf{H}_{x,1} &= \begin{pmatrix} p_x^{00} & p_x^{01} & p_x^{02} \\ p_x^{10} & p_x^{11} & p_x^{12} \\ 0 & 0 & 1 \end{pmatrix} \\ &\equiv \begin{pmatrix} {}_1p_x^{00} & {}_1p_x^{01} & {}_1p_x^{02} \\ {}_1p_x^{10} & {}_1p_x^{11} & {}_1p_x^{12} \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} {}_tp_x^{00} & {}_tp_x^{01} & {}_tp_x^{02} \\ {}_tp_x^{10} & {}_tp_x^{11} & {}_tp_x^{12} \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{where } t = 1 \end{aligned}$$

Hence the result holds for $t = 1$.

Suppose the result is true for $t = k$ where $k \geq 1$, i.e. $\mathbf{H}_{x,k} = \begin{pmatrix} {}_kp_x^{00} & {}_kp_x^{01} & {}_kp_x^{02} \\ {}_kp_x^{10} & {}_kp_x^{11} & {}_kp_x^{12} \\ 0 & 0 & 1 \end{pmatrix}$

Then

$$
\begin{aligned}
\mathbf{H}_{x,k+1} &= \mathbf{H}_{x,k} \begin{pmatrix} p^{00}_{x+k} & p^{01}_{x+k} & p^{02}_{x+k} \\ p^{10}_{x+k} & p^{11}_{x+k} & p^{12}_{x+k} \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_kp^{00}_x & {}_kp^{01}_x & {}_kp^{02}_x \\ {}_kp^{10}_x & {}_kp^{11}_x & {}_kp^{12}_x \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p^{00}_{x+k} & p^{01}_{x+k} & p^{02}_{x+k} \\ p^{10}_{x+k} & p^{11}_{x+k} & p^{12}_{x+k} \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \left\{ \begin{matrix} {}_kp^{00}_x\, p^{00}_{x+k} \\ + \\ {}_kp^{01}_x\, p^{10}_{x+k} \end{matrix} \right\} & \left\{ \begin{matrix} {}_kp^{00}_x\, p^{01}_{x+k} \\ + \\ {}_kp^{01}_x\, p^{11}_{x+k} \end{matrix} \right\} & \left\{ \begin{matrix} {}_kp^{00}_x\, p^{02}_{x+k} \\ + \\ {}_kp^{01}_x\, p^{12}_{x+k} \\ + \\ {}_kp^{02}_x\, 1 \end{matrix} \right\} \\ \left\{ \begin{matrix} {}_kp^{10}_x\, p^{00}_{x+k} \\ + \\ {}_kp^{11}_x\, p^{10}_{x+k} \end{matrix} \right\} & \left\{ \begin{matrix} {}_kp^{10}_x\, p^{01}_{x+k} \\ + \\ {}_kp^{11}_x\, p^{11}_{x+k} \end{matrix} \right\} & \left\{ \begin{matrix} {}_kp^{10}_x\, p^{02}_{x+k} \\ + \\ {}_kp^{11}_x\, p^{12}_{x+k} \\ + \\ {}_kp^{12}_x\, 1 \end{matrix} \right\} \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_{k+1}p^{00}_x & {}_{k+1}p^{01}_x & {}_{k+1}p^{02}_x \\ {}_{k+1}p^{10}_x & {}_{k+1}p^{11}_x & {}_{k+1}p^{12}_x \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} {}_tp^{00}_x & {}_tp^{01}_x & {}_tp^{02}_x \\ {}_tp^{10}_x & {}_tp^{11}_x & {}_tp^{12}_x \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{where } t = k+1
\end{aligned}
$$

Hence the result is true for $t = k + 1$ whenever it is true for $t = k$, and the result is true for $t = 1$. Therefore, by induction, the result is true for all $t \geq 1$, i.e.

$$
\mathbf{H}_{x,t} = \begin{pmatrix} {}_tp^{00}_x & {}_tp^{01}_x & {}_tp^{02}_x \\ {}_tp^{10}_x & {}_tp^{11}_x & {}_tp^{12}_x \\ 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}
$$

Q.E.D.

E.2 Two-Life Policies

E.2.1 Two States

$$\mathbf{H}_{xy,t} = \begin{pmatrix} {}_tp_x \ {}_tp_y & {}_tp_x \ {}_tq_y & {}_tq_x \ {}_tp_y & {}_tq_x \ {}_tq_y \\ 0 & {}_tp_x & 0 & {}_tq_x \\ 0 & 0 & {}_tp_y & {}_tq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Proof

$$\begin{aligned} \mathbf{H}_{xy,1} &= \begin{pmatrix} p_x \ p_y & p_x \ q_y & q_x \ p_y & q_x \ q_y \\ 0 & p_x & 0 & q_x \\ 0 & 0 & p_y & q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &\equiv \begin{pmatrix} {}_1p_x \ {}_1p_y & {}_1p_x \ {}_1q_y & {}_1q_x \ {}_1p_y & {}_1q_x \ {}_1q_y \\ 0 & {}_1p_x & 0 & {}_1q_x \\ 0 & 0 & {}_1p_y & {}_1q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} {}_tp_x\, {}_tp_y & {}_tp_x\, {}_tq_y & {}_tq_x\, {}_tp_y & {}_tq_x\, {}_tq_y \\ 0 & {}_tp_x & 0 & {}_tq_x \\ 0 & 0 & {}_tp_y & {}_tq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{where } t = 1$$

Hence the result holds for $t = 1$.

Suppose the result is true for $t = k$ where $k \geq 1$, i.e. $\mathbf{H}_{xy,k} = \begin{pmatrix} {}_kp_x\, {}_kp_y & {}_kp_x\, {}_kq_y & {}_kq_x\, {}_kp_y & {}_kq_x\, {}_kq_y \\ 0 & {}_kp_x & 0 & {}_kq_x \\ 0 & 0 & {}_kp_y & {}_kq_y \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Then

$$\mathbf{H}_{xy,k+1} = \mathbf{H}_{xy,k} \begin{pmatrix} p_{x+k}\, p_{y+k} & p_{x+k}\, q_{y+k} & q_{x+k}\, p_{y+k} & q_{x+k}\, q_{y+k} \\ 0 & p_{x+k} & 0 & q_{x+k} \\ 0 & 0 & p_{y+k} & q_{y+k} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$
= \begin{pmatrix}
\left\{\begin{matrix} \left({}_kp_x\ {}_kp_y\right) \\ \left(p_{x+k}\ p_{y+k}\right) \end{matrix}\right\} &
\left\{\begin{matrix} \left({}_kp_x\ {}_kp_y\right)\ \left(p_{x+k}\ q_{y+k}\right) \\ + \\ \left({}_kp_x\ {}_kq_y\right)\ \left(p_{x+k}\right) \end{matrix}\right\} &
\left\{\begin{matrix} \left({}_kp_x\ {}_kp_y\right)\ \left(q_{x+k}\ p_{y+k}\right) \\ + \\ \left({}_kq_x\ {}_kp_y\right)\ \left(p_{y+k}\right) \end{matrix}\right\} &
\left\{\begin{matrix} \left({}_kp_x\ {}_kp_y\right)\ \left(q_{x+k}\ q_{y+k}\right) \\ + \\ \left({}_kp_x\ {}_kq_y\right)\ \left(q_{x+k}\right) \\ + \\ \left({}_kq_x\ {}_kp_y\right)\ \left(q_{y+k}\right) \\ + \\ \left({}_kq_x\ {}_kq_y\right)\cdot 1 \end{matrix}\right\} \\
0 & \left({}_kp_x\right)\ \left(p_{x+k}\right) & 0 &
\left\{\begin{matrix} \left({}_kp_x\right)\ \left(q_{x+k}\right) \\ + \\ \left({}_kq_x\right)\cdot 1 \end{matrix}\right\} \\
0 & 0 & \left({}_kp_y\right)\ \left(p_{y+k}\right) &
\left\{\begin{matrix} \left({}_kp_y\right)\ \left(q_{y+k}\right) \\ + \\ \left({}_kq_y\right)\cdot 1 \end{matrix}\right\} \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

$$= \begin{pmatrix} {}_{k+1}p_x\,{}_{k+1}p_y & \begin{pmatrix} {}_{k+1}p_x\,{}_{k|}q_y \\ + \\ {}_{k+1}p_x\,{}_{k}q_y \end{pmatrix} & \begin{pmatrix} {}_{k|}q_x\,{}_{k+1}p_y \\ + \\ {}_{k}q_x\,{}_{k+1}p_y \end{pmatrix} & \begin{pmatrix} {}_{k|}q_x\,{}_{k|}q_y \\ + \\ {}_{k|}q_x\,{}_{k}q_y \\ + \\ {}_{k}q_x\,{}_{k|}q_y \\ + \\ {}_{k}q_x\,{}_{k}q_y \end{pmatrix} \\ 0 & {}_{k+1}p_x & 0 & {}_{k|}q_x + {}_{k}q_x \\ 0 & 0 & {}_{k+1}p_y & {}_{k|}q_y + {}_{k}q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} {}_{k+1}p_x\,{}_{k+1}p_y & {}_{k+1}p_x\,{}_{k+1}q_y & {}_{k+1}q_x\,{}_{k+1}p_y & {}_{k+1}q_x\,{}_{k+1}q_y \\ 0 & {}_{k+1}p_x & 0 & {}_{k+1}q_x \\ 0 & 0 & {}_{k+1}p_y & {}_{k+1}q_y \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} {}_{t}p_x\,{}_{t}p_y & {}_{t}p_x\,{}_{t}q_y & {}_{t}q_x\,{}_{t}p_y & {}_{t}q_x\,{}_{t}q_y \\ 0 & {}_{t}p_x & 0 & {}_{t}q_x \\ 0 & 0 & {}_{t}p_y & {}_{t}q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{where t=k+1}$$

Hence the result is true for $t = k + 1$ whenever it is true for $t = k$, and the result is true for $t = 1$. Therefore, by induction, the result is true

for all $t \geq 1$, i.e.

$$\mathbf{H}_{xy,t} = \begin{pmatrix} {}_{t}p_x \, {}_{t}p_y & {}_{t}p_x \, {}_{t}q_y & {}_{t}q_x \, {}_{t}p_y & {}_{t}q_x \, {}_{t}q_y \\ 0 & {}_{t}p_x & 0 & {}_{t}q_x \\ 0 & 0 & {}_{t}p_y & {}_{t}q_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Q.E.D.

E.2.2 Three States

$$\mathbf{H}_{xy,t} = \begin{pmatrix} {}_{t}p_x \, {}_{t}p_y & {}_{t}p_x \, {}_{t-1|}q_y & {}_{t}p_x \, {}_{t-1}q_y & {}_{t-1|}q_x \, {}_{t}p_y & {}_{t-1|}q_x \, {}_{t-1|}q_y & {}_{t-1|}q_x \, {}_{t-1}q_y & {}_{t-1}q_x \, {}_{t}p_y & {}_{t-1}q_x \, {}_{t-1|}q_y & {}_{t-1}q_x \, {}_{t-1}q_y \\ 0 & 0 & {}_{t}p_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & {}_{t}p_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{t}p_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{t}p_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

$$= \begin{pmatrix} {}_{t}p_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} {}_{t}p_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Proof

$$
\mathbf{H}_{xy,1} = \begin{pmatrix}
p_x\, p_y & p_x\, q_y & 0 & q_x\, p_y & q_x\, q_y & 0 & 0 & 0 & 0 \\
0 & 0 & p_x & 0 & 0 & q_x & 0 & 0 & 0 \\
0 & 0 & p_x & 0 & 0 & q_x & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & p_y & q_y & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & p_y & q_y & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$$
= \begin{pmatrix}
{}_1p_x\, {}_1p_y & {}_1p_x\, {}_{0|}q_y & {}_1p_x\, {}_0q_y & {}_{0|}q_x\, {}_1p_y & {}_{0|}q_x\, {}_{0|}q_y & {}_{0|}q_x\, {}_0q_y & {}_0q_x\, {}_1p_y & {}_0q_x\, {}_{0|}q_y & {}_0q_x\, q_y \\
0 & 0 & {}_1p_x & 0 & 0 & {}_{0|}q_x & 0 & 0 & {}_0q_x \\
0 & 0 & {}_1p_x & 0 & 0 & {}_{0|}q_x & 0 & 0 & {}_0q_x \\
0 & 0 & 0 & 0 & 0 & 0 & {}_1p_y & {}_{0|}q_y & {}_0q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & {}_1p_y & {}_{0|}q_y & {}_0q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$$
= \begin{pmatrix}
{}_tp_x\,{}_tp_y & {}_tp_x\,{}_{t-1|}q_y & {}_tp_x\,{}_{t-1}q_y & {}_{t-1|}q_x\,{}_tp_y & {}_{t-1|}q_x\,{}_{t-1|}q_y & {}_{t-1|}q_x\,{}_{t-1}q_y & {}_{t-1}q_x\,{}_tp_y & {}_{t-1}q_x\,{}_{t-1|}q_y & {}_{t-1}q_x\,{}_{t-1}q_y \\
0 & 0 & {}_tp_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\
0 & 0 & {}_tp_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\
0 & 0 & 0 & 0 & 0 & 0 & {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \qquad \text{where } t = 1
$$

$$
= \begin{pmatrix}
{}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\
0 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix} \otimes \begin{pmatrix}
{}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\
0 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix}
$$

Hence the result holds for $t = 1$.

Suppose the result is true for $t = k$ where $k \geq 1$, i.e.

$$\mathbf{H}_{xy,k} = \begin{pmatrix} {}_kp_x\,{}_kp_y & {}_kp_x\,{}_{k-1|}q_y & {}_kp_x\,{}_{k-1}q_y & {}_{k-1|}q_x\,{}_kp_y & {}_{k-1|}q_x\,{}_{k-1|}q_y & {}_{k-1|}q_x\,{}_{k-1}q_y & {}_{k-1}q_x\,{}_kp_y & {}_{k-1}q_x\,{}_{k-1|}q_y & {}_{k-1}q_x\,{}_{k-1}q_y \\ 0 & 0 & {}_kp_x & 0 & 0 & {}_{k-1|}q_x & 0 & 0 & {}_{k-1}q_x \\ 0 & 0 & {}_kp_x & 0 & 0 & {}_{k-1|}q_x & 0 & 0 & {}_{k-1}q_x \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_kp_y & {}_{k-1|}q_y & {}_{k-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_kp_y & {}_{k-1|}q_y & {}_{k-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Then

$$\mathbf{H}_{xy,k+1} = \mathbf{H}_{xy,k} \begin{pmatrix} p_{x+k}\,p_{y+k} & p_{x+k}\,q_{y+k} & 0 & q_{x+k}\,p_{y+k} & q_{x+k}\,q_{y+k} & 0 & 0 & 0 & 0 \\ 0 & 0 & p_{x+k} & 0 & 0 & q_{x+k} & 0 & 0 & 0 \\ 0 & 0 & p_{x+k} & 0 & 0 & q_{x+k} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_{y+k} & q_{y+k} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_{y+k} & q_{y+k} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$
= \begin{pmatrix}
{}_{k+1}p_x\,{}_{k+1}p_y & {}_{k+1}p_x\,{}_{k|}q_y & \begin{pmatrix} {}_{k+1}p_x\,{}_{k-1|}q_y \\ + \\ {}_{k+1}p_x\,{}_{k-1}q_y \end{pmatrix} & {}_{k|}q_x\,{}_{k+1}p_y & {}_{k|}q_x\,{}_{k|}q_y & \begin{pmatrix} {}_{k|}q_x\,{}_{k-1|}q_y \\ + \\ {}_{k|}q_x\,{}_{k-1}q_y \end{pmatrix} & \begin{pmatrix} {}_{k-1|}q_x\,{}_{k+1}p_y \\ + \\ {}_{k-1}q_x\,{}_{k+1}p_y \end{pmatrix} & \begin{pmatrix} {}_{k-1}q_x\,{}_{k+1}p_y \\ + \\ {}_{k-1}q_x\,{}_{k+1}p_y \end{pmatrix} & \begin{pmatrix} {}_{k-1|}q_x\,{}_{k-1|}q_y \\ + \\ {}_{k-1|}q_x\,{}_{k-1}q_y \\ + \\ {}_{k-1}q_x\,{}_{k-1|}q_y \\ + \\ {}_{k-1}q_x\,{}_{k-1}q_y \end{pmatrix} \\
0 & 0 & {}_{k}p_x\,p_{x+k} & 0 & 0 & {}_{k}p_x\,q_{x+k} & 0 & 0 & {}_{k-1|}q_x + {}_{k-1}q_x \\
0 & 0 & {}_{k}p_x\,p_{x+k} & 0 & 0 & {}_{k}p_x\,q_{x+k} & 0 & 0 & {}_{k-1|}q_x + {}_{k-1}q_x \\
0 & 0 & 0 & 0 & 0 & 0 & {}_{k+1}p_y & {}_{k|}q_y & {}_{k-1|}q_y + {}_{k-1}q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & {}_{k+1}p_y & {}_{k|}q_y & {}_{k-1|}q_y + {}_{k-1}q_y \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$$= \begin{pmatrix} {}_{k+1}p_x\,{}_{k+1}p_y & {}_{k+1}p_x\,{}_{k|}q_y & {}_{k+1}p_x\,{}_{k}q_y & {}_{k|}q_x\,{}_{k+1}p_y & {}_{k|}q_x\,{}_{k|}q_y & {}_{k|}q_x\,{}_{k}q_y & {}_{k}q_x\,{}_{k+1}p_y & {}_{k}q_x\,{}_{k}p_y & {}_{k}q_x\,{}_{k}q_y \\ 0 & 0 & {}_{k+1}p_x & 0 & 0 & {}_{k|}q_x & 0 & 0 & {}_{k}q_x \\ 0 & 0 & {}_{k+1}p_x & 0 & 0 & {}_{k|}q_x & 0 & 0 & {}_{k}q_x \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{k+1}p_y & {}_{k|}q_y & {}_{k}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{k+1}p_y & {}_{k|}q_y & {}_{k}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} {}_{t}p_x\,{}_{t}p_y & {}_{t}p_x\,{}_{t-1|}q_y & {}_{t}p_x\,{}_{t-1}q_y & {}_{t-1|}q_x\,{}_{t}p_y & {}_{t-1|}q_x\,{}_{t-1|}q_y & {}_{t-1|}q_x\,{}_{t-1}q_y & {}_{t-1}q_x\,{}_{t}p_y & {}_{t-1}q_x\,{}_{t-1|}q_y & {}_{t-1}q_x\,{}_{t-1}q_y \\ 0 & 0 & {}_{t}p_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & {}_{t}p_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{t}p_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{t}p_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{where t=k+1}$$

Hence the result is true for $t = k + 1$ whenever it is true for $t = k$, and the result is true for $t = 1$. Therefore, by induction, the result is true

for all $t \geq 1$, i.e.

$$\mathbf{H}_{xy,k} = \begin{pmatrix} {}_tp_x\,{}_tp_y & {}_tp_x\,{}_{t-1|}q_y & {}_tp_x\,{}_{t-1}q_y & {}_{t-1|}q_x\,{}_tp_y & {}_{t-1|}q_x\,{}_{t-1|}q_y & {}_{t-1|}q_x\,{}_{t-1}q_y & {}_{t-1}q_x\,{}_tp_y & {}_{t-1}q_x\,{}_{t-1|}q_y & {}_{t-1}q_x\,{}_{t-1}q_y \\ 0 & 0 & {}_tp_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & {}_tp_x & 0 & 0 & {}_{t-1|}q_x & 0 & 0 & {}_{t-1}q_x \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

$$= \begin{pmatrix} {}_tp_x & {}_{t-1|}q_x & {}_{t-1}q_x \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} {}_tp_y & {}_{t-1|}q_y & {}_{t-1}q_y \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad t \in \{1, 2, 3, \dots\}$$

Q.E.D.

E.3 Continuous Time

The proofs in Sections E.1 and E.2 have relied on the upper limit of the time interval being an integer (to align with the start of a projection step, where all steps are of unit length). It is possible to show that similar matrices can be derived for continuous time intervals, i.e. non-integral end-points: such proofs would require integration of the force of mortality. However, because those proofs are not directly required here, they have been neither provided nor rigorously derived.