A Latent-Factor Model for Loss in Heterogeneous Credit Portfolios, with GPU-Accelerated Convolution

Mark Rotchell

Thesis submitted as part of the degree of MSc Data Science

Department of Computer Science and Information Systems,
Birkbeck College, University of London, 2021

Abstract

It is shown how existing pseudo-polynomial algorithms for computing the probability distribution functions of a Poisson Binomial (PB) random variable can be extended to the Generalised Poisson Binomial (GPB) distribution. The PB distribution is that of a sum of non-identically-distributed but mutually-independent Bernoulli random variables. The GPB distribution extends this to sums of mutually-independent two-point random variables with arbitrary support. It can be used to model total default risk in heterogeneous credit portfolios, i.e. where a lender has extended loans of varying amounts to borrowers of varying creditworthiness. Algorithm performance is then improved using parallelisation and GPU-acceleration. It is further shown how the GPB distribution can be extended to relax the assumption of mutual independence by introducing the Latent Factor Generalised Poisson Binomial (LFGPB) distribution, which makes use of the Gaussian copula. Algorithms are developed for computing the probability distribution functions of LFGPB random variables. The algorithms developed to deal with the GPB and LFGPB distributions are combined into a new Python package, `pyGPB`, and a demonstration is provided of their use to model risk in a conceived heterogeneous credit portfolio.

Extra Fortunam Est

Contents

Acknowledgements v

List of Figures vi

List of Tables vii

List of Algorithms viii

Notation and Conventions ix

Glossary xi

1 Introduction 1
- 1.1 Preamble . . . 1
- 1.2 Project Preview . . . 2
- 1.3 Structure of this Report . . . 4

2 Preliminaries 5
- 2.1 The Generalised Poisson Binomial distribution . . . 5
  - 2.1.1 A Single Loan . . . 5
  - 2.1.2 A Collection of Loans . . . 5
  - 2.1.3 A Heterogeneous Collection of Loans . . . 6
- 2.2 Example of a GPB random variable and its distribution functions . . . 7
- 2.3 The Latent Factor GPB distribution . . . 9
  - 2.3.1 Local Latent Factors . . . 9
  - 2.3.2 Joint Success Probabilities: The Gaussian Copula . . . 10
  - 2.3.3 The Latent Factor GPB distribution . . . 11

3 Specification of Requirements 12
- 3.1 Extend the algorithms of Biscari et al.[4] to apply to the GPB distribution . . . 12
- 3.2 Parallelise the above algorithms . . . 12
- 3.3 Extend the GPB distribution to allow for some simple correlation structure . . . 13
- 3.4 Create a Python package that implements the above . . . 13
- 3.5 Demonstrate how the package might be used in the real world. . . . 13

4 Approaches to computing the pmf of the GPB distribution 14
- 4.1 Summary of approaches . . . 14
- 4.2 Dynamic Programming Approach, $O(nm)$ . . . 15
- 4.3 Divide and Conquer FFT approach, O(n log m log n) . . . 16
  - 4.3.1 Primer on Convolution and Omnivolution . . . 16
  - 4.3.2 Divide and Conquer . . . 17

4.3.3 Convolution via Fourier Transforms . . . 17
4.3.4 Application to the GPB distribution . . . 17
4.3.5 Application to the Toy Example . . . 18
4.3.6 Combination Approach . . . 19
4.4 Parallel Dynamic Programming . . . 19
4.5 Parallel DC-FFT and DP Combination . . . 20

5 Approaches to computing the pmf of the LFGBP Distribution 22
5.1 A Conditional-Independence Framework . . . 22
5.1.1 A Global Latent Factor . . . 22
5.1.2 Conditional Independence . . . 22
5.2 The LFGPB pmf as an Expectation . . . 23
5.2.1 Conditional Probabilities . . . 24
5.2.2 The probability mass function . . . 24
5.2.3 Numerical Integration, aka Quadrature . . . 24

6 Software Implementation 25
6.1 Language and Framework . . . 25
6.2 Key Tools . . . 25
6.3 The `pyGPB` package . . . 27
6.4 Unit Testing . . . 27

7 Accuracy Testing 28
7.1 Testing Framework . . . 28
7.1.1 Generation of Test Scenarios . . . 28
7.1.2 Choice of Accuracy Measure . . . 28
7.1.3 Choice of Benchmark . . . 28
7.2 Accuracy Testing Results . . . 29
7.3 Accuracy Testing the LFGPB Distribution . . . 30
7.3.1 Testing Against the Linear Algebra Approach . . . 30
7.3.2 Testing Using Moments . . . 32

8 Performance Results 34
8.1 Benchmarking Methodology . . . 34
8.1.1 Test Cases . . . 34
8.1.2 Speed Measurement . . . 34
8.1.3 Hardware Specification . . . 35
8.2 Scope of Benchmarking . . . 35
8.3 Algorithm Speeds . . . 35

9 Example Application 39
9.1 Case Description . . . 39
9.2 Data Gathering . . . 39
9.2.1 Default Probabilities . . . 39
9.2.2 Implying Default Probabilities from CDS Spreads . . . 40
9.2.3 Portfolio Weighting . . . 41
9.2.4 Portfolio Composition and Loss Given Default . . . 43
9.3 Application . . . 43
9.3.1 Distributions . . . 43
9.3.2 Quantiles . . . 45
9.3.3 Expected Shortfall . . . 45
9.4 Understanding Changes over Time . . . 45

9.4.1 Incremental Impacts . . . 47

10 Directions for Further Work **50**

11 Conclusions and Critical Evaluation **52**

Appendices **55**

A Additional Detail on Computing the GPB Distribution **55**
A.1 Existing Approaches to the GPB . . . 55
A.1.1 Naïve approach, $O(2^m)$ . . . 55
A.1.2 Characteristic Function Approach, $O(NM)$ . . . 55
A.2 Alternative Weight Parametrisations . . . 59
A.2.1 Non-Positive Success Weights . . . 59
A.2.2 Three Parameter Form . . . 60
A.2.3 Extension to Non-Integers . . . 60
A.3 Connection to Subset Sum . . . 60

B Additional Detail on Computing the LFGPB distribution **62**
B.1 Properties of the LFGPB distribution . . . 62
B.1.1 Mean . . . 62
B.1.2 Variance . . . 62
B.1.3 Higher Order Moments . . . 63
B.1.4 Parameter Estimation . . . 64
B.2 Linear Algebra Approach . . . 65
B.3 Gauss Kronrod Quadrature . . . 68
B.3.1 Overview of GK Quadrature . . . 68
B.3.2 Adaptive Quadrature . . . 69
B.4 Integration of the Characteristic Function . . . 69
B.5 Computation of the Gaussian Copula with an Equicorrelation Matrix . . . 70

C The `pyGPB` Python Library **71**
C.1 Package Structure . . . 71
C.2 Distribution and Installation . . . 71
C.3 A brief look at the pyGPB API . . . 72
C.4 Documentation . . . 73

D Additional Detail on Testing **75**
D.1 Test Case Generation . . . 75
D.1.1 Test Space for Number of Components, m . . . 75
D.1.2 Test Space for Total Weight, n . . . 75
D.1.3 Weights . . . 75
D.1.4 Success Probabilities . . . 77
D.2 Relative Accuracy Issues for FFT approaches . . . 77
D.3 Hardware Specification for Benchmarking . . . 78

Bibliography **80**

Acknowledgements

This project would not have been possible without the support of my wife, Dorthe, and the patience of our son Arthur.

I would also like to thank my supervisor, Vladislav Ryzhikov for his excellent guidance, and the other markers of the Project Proposal document for their useful feedback.

List of Figures

1.1 Algorithm performance for computing the GPB pmf vector for m loans totalling $n = \$10,000,000$ for three different approaches . . . 3
1.2 probability mass function for a LFGPB random variable for different underlying correlation . . . 3

2.1 Probability Tree diagram for the naïve approach to calculating the pmf of the GPB distribution . . . 8
2.2 probability mass function and cumulative distribution function for the Toy Example 8

6.1 Diagram of the key Libraries and tools used to build `pyGPB` and how they worked together . . . 26

8.1 Algorithm speed comparison for $n = 10,000$ . . . 36
8.2 Algorithm speed comparison for $n = 10,000,000$ . . . 37
8.3 Speed of Algorithms across the $n \times m$ parameter space . . . 37
8.4 Domain of m and n where each algorithm is fastest . . . 38

9.1 The World Government Bond page for sovereign CDS spreads . . . 40
9.2 The markit tool for pricing CDS and implying PDs . . . 41
9.3 Country GDP Data from Trading Economics . . . 43
9.4 cumulative distribution function for various correlations . . . 46
9.5 probability mass function for various correlations . . . 46
9.6 Change in cdf from the previous parameters to the current parameters . . . 48
9.7 Change in pmf from the previous parameters to the current parameters . . . 48

B.1 Venn diagram showing all outcome probabilities . . . 65
B.2 Venn diagram showing the outcome probabilities corresponding to the event probability p_2^e . . . 66

C.1 Homepage of the pyGPB library docs . . . 74
C.2 Documentation of the GPB class . . . 74

D.1 Histograms of Weights for two particular test cases where $m = 10,000, n = 1,000,000$. 76
D.2 Histograms of Success Probabilities for two particular test cases where $m = 10,000$. 77
D.3 pmf of a test case with $m = 100, n = 10,000$ via the DP and DC-FFT approaches . 78
D.4 Relative Accuracy Issue of FFT approaches for small probabilities . . . 78

List of Tables

2.1 Parameters for the Toy Example used throughout the report 7
2.2 probability mass function and cumulative distribution function for the toy example 9

4.1 Summary of Approaches to computing the pmf of the GPB distribution 14
4.2 Tabular layout of computing the GPB pmf via Dynamic Programming 16

7.1 Accuracy results when comparing two different implementations of the Naïve approach 29
7.2 Accuracy Results when comparing 16-byte float Naive and Dynamic Programming approaches 30
7.3 TAE Accuracy Results of DC-FFT, and GPU approaches vs DP Approach 31
7.4 MAE Accuracy Results of DC-FFT, and GPU approaches vs DP Approach 31
7.5 Results of testing the implemented LFGPB approach against the Linear Algebra approach 32
7.6 Results of testing the LFGPB implementation using moments. Values are relative errors 33

8.1 Summary of algorithms that were speed benchmarked 35
8.2 Speed Benchmarking Results for select values of n and m in milliseconds 35

9.1 Example Input Data 42
9.2 Portfolio Composition: Investment Amount and Loss Given Default 44
9.3 Quantiles of total loss for differing correlation assumptions 45
9.4 Expected shortfalls for differing correlation assumptions 47
9.5 Change in 90% Expected Shortfall allocated to individual changes 49

A.1 Layout of Calculation for Characteristic Function Approach 58

B.1 relationship between outcome and event probabilities 67
B.2 Nodes and weights for Gauss Kronrod Quadrature 69

List of Algorithms

1 GPB pmf computation with Dynamic Programming 16
2 GPB pmf computation with Divide-and-Conquer FFT Omnivolution 18
3 GPB pmf computation with a combination of divide-and-conquer FFT Omnivolution and Dynamic Programming 19
4 GPB pmf computation with Dynamic Programming in Parallel 20
5 GPB pmf computation with a combination of Dynamic Programming and batch FFT in Parallel 21

6 Naïve Approach 55
7 Naïve Approach using a Stack 56
8 Characteristic Function Approach 58

9 Adaptive Gauss-Kronrod Quadrature 69

10 Generating Collections of Weights with a Target Sum 76

Notation and Conventions

Notation in this work mostly follows that of [10]. This section provides a precise reference.

Numbers and Arrays

a	a scalar
$\boldsymbol{a}$	a vector or sequence
$\boldsymbol{A}$	a matrix
A	a scalar-valued random variable
$\mathbf{A}$	a vector-valued random variable
$\begin{bmatrix} a & b & c \end{bmatrix}$	a row vector with values a, b and c
$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ or $\begin{bmatrix} a & b & c \end{bmatrix}^\top$	a column vector with values a, b and c
$(a_0, a_1, \ldots a_n)$	a sequence of $n+1$ scalars
$\boldsymbol{0}_m$	an m-dimensional vector of zeros.
$\boldsymbol{I}_m$	the $m \times m$ identity matrix.
$\boldsymbol{a} \cdot \boldsymbol{b}$	the dot product of $\boldsymbol{a}$ and $\boldsymbol{b}$
$\boldsymbol{a} \circ \boldsymbol{b}$	the Hadamard product, (i.e. element-wise product) of $\boldsymbol{a}$ and $\boldsymbol{b}$
$\lvert a \rvert$	the absolute value of the scalar a
$\dim(\boldsymbol{a})$	the dimensionality (i.e. number of elements) of the vector $\boldsymbol{a}$ (or, more precisely, the dimensionality of the vector space of which $\boldsymbol{a}$ is an element).

Sets and Intervals

$\mathcal{A}$	a set
$\{a, b, c\}$	the set containing a, b and c
$[\![a : b]\!]$	all integers from a to b inclusive
$[a : b]$	the real interval from a to b, inclusive
$(a : b)$	the real interval from a to b, exclusive
$\mathcal{A}^n$	the n-fold Cartesian product of $\mathcal{A}$ with itself
$\mathcal{A} \backslash \mathcal{B}$	set subtraction, i.e. the set of elements that are in $\mathcal{A}$ but not in $\mathcal{B}$
$\mathsf{Pow}(\mathcal{A})$	the power set of $\mathcal{A}$, i.e. the set of all subsets of $\mathcal{A}$
$\lvert \mathcal{A} \rvert$	the cardinality of the set $\mathcal{A}$, i.e. the number of elements in $\mathcal{A}$
$\mathrm{Supp}(\mathcal{A})$	the support of the multiset $\mathcal{A}$, i.e. the set of unique elements in $\mathcal{A}$
$\mathbb{N}_+$	the natural numbers excluding zero, i.e. $\{1, 2, 3, \ldots\}$

Indexing

Unless otherwise stated, indices in this work are zero-based.

a_i	the ith element of vector $\boldsymbol{a}$, set $\mathcal{A}$, or sequence $(a_0, a_1, \ldots a_n)$
A_i	the ith element of the vector-valued random variable $\mathbf{A}$
$\mathcal{A}_i$	the ith element of the family of sets $\mathcal{A}$

Probability

$\mathbb{P}[\mathrm{A} = a]$	the probability that the random variable A takes the value a
$\mathbb{P}[\mathrm{A}]$	the probability that the outcome of random variable A is *success*
$\mathbb{E}_x[f(x)]$	the expected value of $f(x)$ over all values of x. The x may be dropped where it can be inferred from context.
$\mathbb{P}[\mathrm{A}\|\mathrm{B}]$	the probability of A conditional on B
$\mathbb{E}_x[\mathrm{A}\|\mathrm{B}]$	the expectation of A conditional on B
$\mathrm{A} \sim \mathcal{B}(c, d)$	A has the probability distribution B, parametrised by c and d
$\mathcal{N}$	the standard normal distribution
$\mathcal{N}_n$	the n-dimensional multivariate standard normal distribution
$\mathcal{U}[a, b]$	the continuous uniform distribution over the real interval $[a, b]$
$\mathcal{U}[\![a, b]\!]$	the discrete uniform distribution over the integers $[\![a, b]\!]$
$\mathrm{Var}(\mathrm{A})$	the variance of A
$\mathrm{Cov}(\mathrm{A}, \mathrm{B})$	the covariance of A and B
f_{A}	the pdf (or pmf) of the random variable A
F_{A}	the cdf of the random variable A
$\boldsymbol{R}_m$	the $m \times m$ *equicorrelation* matrix, i.e. a matrix with 1 on the main diagonal and ρ everywhere else. The m may be omitted where it can be inferred from context.

Particular Probability Density Functions

$g(x; \boldsymbol{p}, \boldsymbol{w})$ or $g(x)$	the pmf of a GPB random variable parametrised by $\boldsymbol{p}$ and $\boldsymbol{w}$
$G(x; \boldsymbol{p}, \boldsymbol{w})$ or $G(x)$	the cdf of a GPB random variable parametrised by $\boldsymbol{p}$ and $\boldsymbol{w}$
$\lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho)$ or $\lambda(x)$	the pmf of a LFGPB random variable parametrised by $\boldsymbol{p}$, $\boldsymbol{w}$ and ρ
$\Lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho)$ or $\Lambda(x)$	the cdf of a LFGPB random variable parametrised by $\boldsymbol{p}$, $\boldsymbol{w}$ and ρ
$\boldsymbol{g}, \boldsymbol{G}, \boldsymbol{\lambda}, \boldsymbol{\Lambda}$	the probability distribution functions above expressed as a vector of values over the support of the distribution
$\phi(x)$	the pdf of the standard normal distribution
$\Phi(x)$	the cdf of the standard normal distribution
$\Phi^{-1}(p)$	the quantile function of the standard normal distribution
$\Phi_{\boldsymbol{\Sigma}}$	the multivariate-normal cdf with means of zero and correlation matrix $\boldsymbol{\Sigma}$
$C^{\mathrm{Gauss}}_{\boldsymbol{\Sigma}}$	the Gaussian copula with correlation matrix $\boldsymbol{\Sigma}$

Fourier Transforms and Convolution

$\mathcal{F}\{f\}$ or $\hat{f}$	the Fourier transform of the function f
$\mathcal{F}^{-1}\{\hat{f}\}$	the Inverse Fourier transform of the function $\hat{f}$
$\mathcal{F}\{\boldsymbol{f}\}$ or $\hat{\boldsymbol{f}}$	the Discrete Fourier transform of the sequence $\boldsymbol{f}$
$\mathcal{F}^{-1}\{\hat{\boldsymbol{f}}\}$	the Inverse Discrete Fourier transform of the sequence $\hat{\boldsymbol{f}}$
$f * g$	the convolution of the functions f and g
$\boldsymbol{f} * \boldsymbol{g}$	the discrete convolution of the sequences $\boldsymbol{f}$ and $\boldsymbol{g}$
$\overset{n}{\underset{i=0}{\ast}} \boldsymbol{f}_i$	the omnivolution (i.e. iterated convolution) of the elements of the sequence of sequences $(\boldsymbol{f}_0, \boldsymbol{f}_1, \ldots, \boldsymbol{f}_n)$

Glossary

component random variable refers, in this report, to the two-point random variables that are summed to create a GPB or LFGPB random variable. In particular, it refers to elements of the vector-valued random variable $\mathbf{V}$ as defined in Section 2.1.3.

creditworthiness is the attractiveness of a borrower to a lender by virtue of the perceived likelihood that the borrower shall honour their obligations.

cumulative distribution function (cdf) is a function that provides the probabilty that a given random variable will be equal to or less than a particular value. Formally, where $\mathcal{S}$ is the support of a discrete random variable, X, then the cdf is the function $f_{\mathrm{X}} : \mathcal{S} \to [0 : 1]$ such that $f_{\mathrm{X}}(x) = \mathbb{P}[\mathrm{X} \leqslant x]$. For collections of random variables, i.e. when considering a "multivariate" distribution, then the cdf is the probability that each component is below a respective certain value.

default is the failure of a borrower to meet their obligations with respect to their debts.

divide-and-conquer is an algorithmic design technique that breaks a problem into two (or more) smaller peices that are similar to the original problem. The problem is thus broken down recursively until the sub problems are small enough to be trivially solveable.

divide-and-conquer FFT , also divide-and-conquer FFT *tree* or just **DC-FFT**, is an omnivolution algorithm where the pmfs of component random variables are convolved together in a tree-like fashion. Pairs of pmfs are convolved together using Fast Fourier Transform algorithms and the result passed up the tree where they are convolved with the result of another branch. The leaves may be the pmf of a single component random variable (which is trivial to compute), or they may be the pmf of a small (disjoint) subset of component random variables which have been computed using another method (in this work a dynamic-programming approach). See 4.3.4 for an example of this approach.

Dynamic-Programming (DP) is an algorithmic technique where problems are solved recursively using the cached solutions to smaller overlapping sub-problems. Dynamic programming contrasts with divide-and-conquer (DC) in that DC has non-overlapping subproblems.

embarrassingly parallel is a description for a problem (or an algorithm to solve such a problem) that can be easily split into multiple disjoint subtasks that can completed in parallel.

Fast Fourier Transform (FFT) a fast algorithm for computing the Discrete Fourier Transform. See Appendix A.1.2 for a primer on Fourier Transforms. Generally credited to Cooley and Tukey[7], although it had previously been described by Gauss. The inverse transform can be equivalently computed using an Inverse Fast Fourier Transform (IFFT) algorithm. Where values in the time domain are all real numbers then there are also "Real" FFT algorithms, which are faster by a factor of two. These are denoted RFFT and IRFFT respectively.

Gaussian copula A full technical defintition and explanation is beyond the scop of this work; it suffices to say that the Gaussian copula is a function $C_{\boldsymbol{\Sigma}}^{\text{Gauss}} : [0:1]^k \to [0:1]$ defined such that

$$C_{\boldsymbol{\Sigma}}^{\text{Gauss}}(\boldsymbol{q}) = \Phi_m \left(\begin{bmatrix} \Phi^{-1}(q_0) \\ \Phi^{-1}(q_1) \\ \vdots \\ \Phi^{-1}(q_{k-1}) \end{bmatrix} ; \boldsymbol{0}_k, \boldsymbol{\Sigma} \right)$$

where $\Phi_k(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the cdf of the k-variate normal distribution with means $\boldsymbol{\mu}$ and correlation matrix $\boldsymbol{\Sigma}$.

Generalised Poisson Binomial (GPB) distribution is the probability distribution of a sum of mutually-independent but not-necessarily-identical two-point random variables. See Section 2.1 for a more detailed explanation.

GPU-acceleration is increasing the speed of a computation by employing the massively-parallel processing capability of Graphics Processing Units (GPUs), which are hardware devices specifically designed for parallel computation. By contrast, most computation is normally performed by Central Processing Units (CPUs), which are generally more performant for sequential computation.

mutually-independent A collection of events are mutually independent if the outcome of all events or subset of events is indepenent of the other events or subsets of the other events.

parallel algorithm is an algorithm in which multiple operations are performed at the same time. Contrasts with a serial algorithm where operations are performed sequentially. *Parallelisation* is the adaptation of a serial algorithm to be a parallel algorithm.

Poisson Binomial (PB) distribution is a special case of the GPB distribution where all success weights are one.

probability distribution functions are the probability mass function and cumulative distribution function.

probability mass function (pmf) is a function that provides the probabilty that a given discrete random variable will take a particular value. Formally, where $\mathcal{S}$ is the support of a discrete random variable, X, then the pmf is the function $f_{\text{X}} : \mathcal{S} \to [0:1]$ such that $f_{\text{X}}(x) = \mathbb{P}[\text{X} = x]$.

Probability of Default (PD), also *default probability*, is the probability that a particular borrower will default on their debt. In the context of a portfolio of loans being modelled by a GPB random variable these probabilities are the elements of the $\boldsymbol{p}$ parameter.

pseudo-polynomial time is the time complexity of an algorithm if its running time is a polynomial in the *value* of the input, rather than the *length* (in bits) of the input. As the number of bits required to store a value is proportional to its logarithm, pseudo-polynomial algorithms are still exponential in the length of the input.

two-point random variable is a random variable that is the result of a random experiment with two outcomes: "success" and "failure". Success happens with probability p and failure with probability $1-p$. Each outcome is associated with a particular value that the random variable will realise. The simplest example is a Bernoilli random variable, which takes the value 1 on success and 0 on failure. The component random variables of the GPB distribution are two-point random variables which can realise arbitrary (but pre-defined) values for success and failure.

Chapter 1

Introduction

1.1 Preamble

Consider a lender who has extended loans of various amounts to borrowers of different creditworthiness. The lender knows the size of each loan and the probability that each borrower will default over a particular period of time; they wish to understand how much they stand to lose in total over that period due to borrowers defaulting. This total defaulted loan amount follows the Generalised Poisson Binomial (GPB) distribution, which was introduced by Zhang et al. (2018)[26]. This distribution has myriad applications including modelling:

- The combined number of votes that a candidate receives in an electoral-college voting system[1] where colleges are enfranchised with varying numbers of votes and each college has a different probability of giving its votes to that candidate.
- The total payout of an insurer who has underwritten contracts of multiple sizes against events of varying likelihoods.
- The possible expense a factory owner might bear to maintain an assembly line composed of machines of differing replacement costs and expected lifetimes.

Suppose the random variable X is the total value of all the loans that default during a given period of time[2], this random variable follows the GPB distribution, formally $\mathrm{X} \sim \mathrm{GPB}(\boldsymbol{p}, \boldsymbol{w})$, where $\boldsymbol{w}$ is vector containing the size of each loan, and $\boldsymbol{p}$ is a vector containing the associated default probabilities.

Our lender wishes to know the probability that X will take some given value, x. A function which can return this result is called the *probability mass function* (pmf), formally $g(x) = \mathbb{P}[\mathrm{X}{=}x]$, which can also be represented as a vector $\boldsymbol{g}$ where the ith element, $\boldsymbol{g}_i$, has the value $g(i)$.

Computing the value of the vector $\boldsymbol{g}$ given the portfolio parameters (i.e. $\boldsymbol{w}$ and $\boldsymbol{p}$) by brute force is exceedingly slow, taking $O(2^m)$ time where $m = \dim(\boldsymbol{p}) = \dim(\boldsymbol{w})$, however the work by Zhang et al.[26] introduced the "DFT-CF" algorithm which has pseudo-polynomial time complexity of $O(mn)$, where $n = 1 + \sum_i w_i$. Biscarri et al. (2018)[4] introduced the $O(n \log m \log n)$ "DC-FFT" algorithm for the simpler Poisson Binomial distribution[3] and suggested it could be extended to the GPB distribution, but did not explore it further as part of that work.

[1] Such systems are used for presidential elections in the USA, India, Estonia etc..

[2] for example, suppose the lender has made fours loans with sizes \$100, \$130, \$200 and \$210 and that during the period of time in question the first and last borrower defaults. The value of X would be \$100 + \$210 = \$310

[3] the Poisson Binomial distribution is a special case the Generalised Poisson Binomial distribution where all the weights are 1

The objectives of this project are to: extend the DC-FFT algorithm to the GPB distribution; increase performance further via parallelisation with GPU-acceleration; extend the GPB distribution to allow for some simple correlation between the component random variables; combine these findings into a new Python package, `pyGPB`; and to demonstrate the package's use on an example built from real-world data.

Whilst preparing this report it was discovered that Junge (2021)[13] has also extended the DC-FFT algorithm for the GPB distribution as part of the `PoissonBinomial` package for `R`, however, the details of their implementation have not been published in the literature so far as we've been able to find.

1.2 Project Preview

This project sought to address three areas regarding the recent state-of-the-art ability to compute the probability distribution functions of a GPB random variable:

Improving computational complexity: Finding any algorithm whose time complexity is polynomial in m is not viable[4]. However, we can extend the $O(n \log m \log n)$ DC-FFT algorithm from Biscari et al.(2018)[4] to the GPB. This method uses a divide-and-conquer structure with Fast Fourier Transform algorithms in combination with a fast $O(mn)$ Dynamic-Programming (DP) approach, which is employed for small m. This achieves a reasonable improvement on Zhang et al.'s[26] DFT-CF algorithm. See Figure 1.1 for a chart showing the speed up achieved.

Parallelisation with GPU acceleration: Much of the work of the algorithms discussed above can be made "embarrassingly parallel", this project seeks to take advantage of this by pushing much of the computation onto GPUs. For small n the overhead of communication between host (i.e. CPU and main memory) and device (i.e. GPU cores and GPU memory) can be prohibitive, but for larger problems there was a speed up of at least an order of magnitude. See Figure 1.1 for a chart showing the speed up achieved.

Relaxing the assumption of mutual independence: If the component random variables are not mutually independent then the GPB distribution does not apply. This limits applicability in the real world where, for example, a worsening economy impacts all borrowers and national politics affects state elections, so those borrowers and states cannot be described as independent. An extension to the GPB distribution is proposed that allows for some simple correlation structure to exist between the component random variables. We call this the "Latent Factor Generalised Poisson Binomial" (LFGPB) distribution. An example of how correlation can effect the probability distributions of a GPB random variable can be seen in Figure 1.2.

Having made an attempt to address the issues listed above, this project provides the following:

1. A software library in Python that can be used when investigating random variables of this type.
2. A demonstration of the use of the library by applying it to an example portfolio of government bonds with distribution parameters taken from real-world data.

[4] it would constitute a polynomial-time algorithm for the Subset Sum, proving P = NP, see appendix A.3 for further discussion on this point

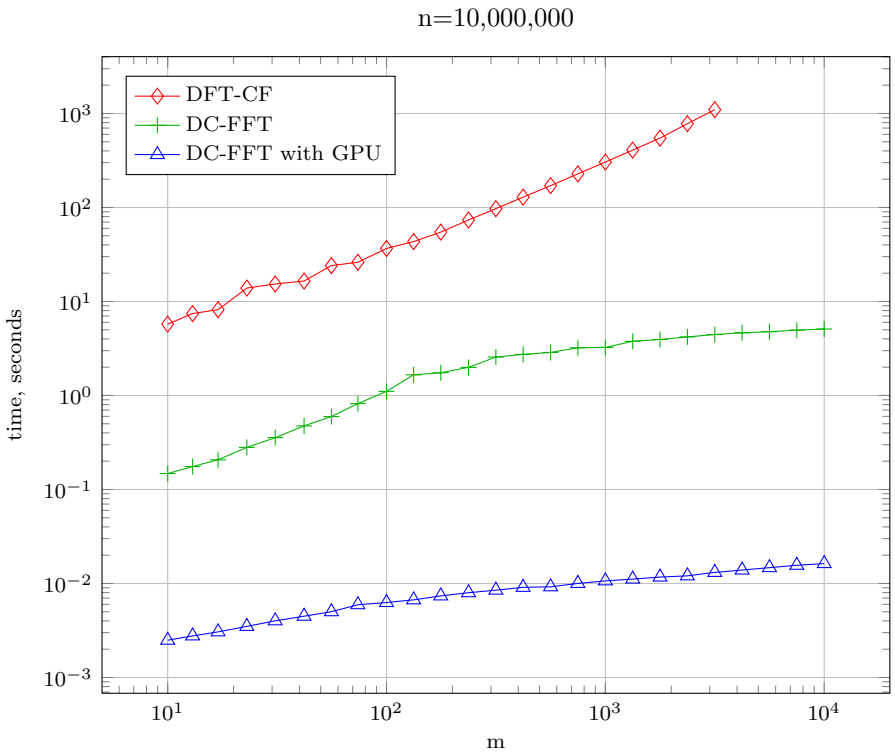

Figure 1.1: Algorithm performance for computing the GPB pmf vector for m loans totalling $n = \$10,000,000$ for three different approaches

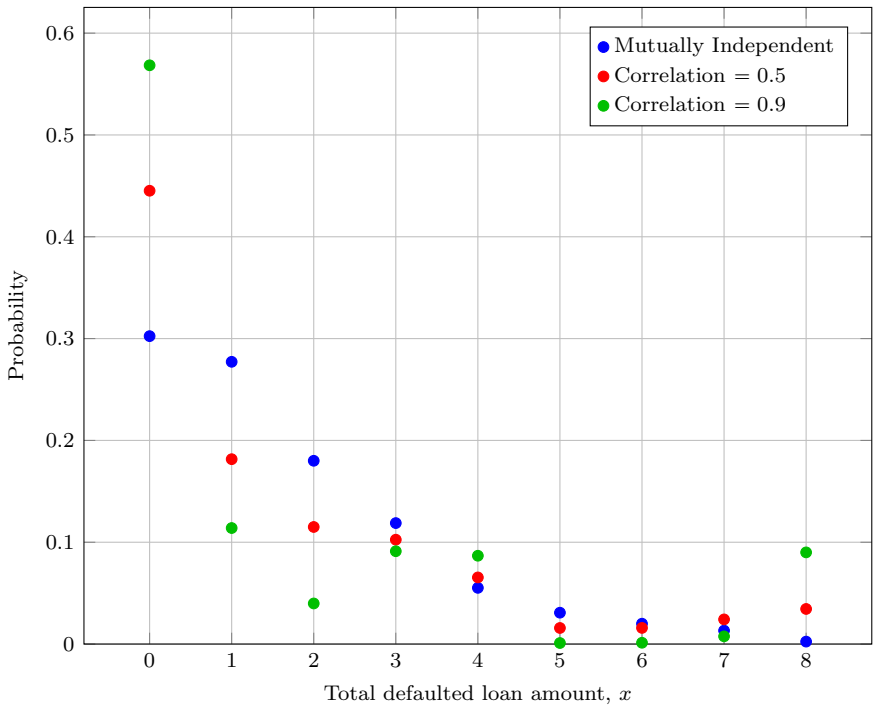

Figure 1.2: probability mass function for a LFGPB random variable for different underlying correlation

.

1.3 Structure of this Report

Chapter 2 gives an overview of the GPB distribution and provides a toy example for use throughout the report. It also introduces the Latent Factor Generalised Poisson Binomial distribution.

Chapter 3 lays out the five goals that this project set out to achieve and details the associated requirements.

Chapter 4 derives and explains the algorithms that were created or adapted for computing the pmf of the GPB distribution. Discussion of existing approaches and additional technical detail from this project regarding the GPB distribution can be found in Appendix A.

Chapter 5 derives and explains the algorithm developed for computing the pmf of the LFGPB distribution with additional technical detail and discussion of alternative approaches found in Appendix B.

Chapter 6 details the software and hardware tools that were used, discusses why they were chosen and gives a brief overview of the components of the `pyGPB` library. Further detail on the `pyGPB` library can be found in Appendix C.

Chapters 7 and 8 demonstrate how the algorithms that were developed in Chapters 4 and 5 were thoroughly tested and benchmarked for speed. Additional technical detail on testing can be found in Appendix D.

A demonstration of the application of the `pyGPB` library to a conceived heterogeneous credit portfolio is demonstrated in Chapter 9.

Chapters 10 and 11 discuss directions for further work and evaluate the project's success.

Chapter 2

Preliminaries

2.1 The Generalised Poisson Binomial distribution

2.1.1 A Single Loan

The status of a single loan at the end of a particular period of time can be modelled using a binary random variable, B. The variable will take the value of 1 if the loan has defaulted and 0 if it has not. The lender considers the probability of the loan defaulting during the period to be p. The random variable B follows the familiar Bernoulli distribution with p as a parameter. More formally:

$$\mathrm{B} \sim \text{Bernoulli}(p) \iff \mathrm{B} = \begin{cases} 1 & \text{with probability } \ p \\ 0 & \text{with probability } \ 1-p \end{cases} \tag{2.1}$$

The probability mass function, $f_{\mathrm{B}}(b)$, of the Bernoulli random variable is therefore

$$\begin{aligned} f_{\mathrm{B}}(b) = \mathbb{P}(\mathrm{B}=b) &= \begin{cases} p & \text{for } b=1 \\ 1-p & \text{for } b=0 \end{cases} \\ &= bp + (1-b)(1-p) \end{aligned} \tag{2.2}$$

or in vector form

$$\boldsymbol{f}_{\mathrm{B}} = \begin{bmatrix} (1-p) \\ p \end{bmatrix} \tag{2.3}$$

2.1.2 A Collection of Loans

A collection of loans can be represented by a collection of binary random variables arranged in a vector. Let this vector-valued random variable, $\mathbf{B}$, be a collection of m Bernoulli random variables

$$\mathbf{B} = \begin{bmatrix} \mathrm{B}_0 & \mathrm{B}_1 & \dots & \mathrm{B}_{m-1} \end{bmatrix}^\top \in \{0,1\}^m \tag{2.4}$$

parametrised by a vector of m "success"[1] probabilities

$$\boldsymbol{p} = \begin{bmatrix} p_0 & p_1 & \dots & p_{m-1} \end{bmatrix}^\top \in (0,1)^m \tag{2.5}$$

such that

$$\mathrm{B}_i \sim \text{Bernoulli}(p_i) \tag{2.6}$$

[1] clearly, a default is not a successful outcome for the lender in the normal meaning of the word, however in the vernacular of random variables "success" means the variable realising the value 1.

If the elements of $\mathbf{B}$ are summed one gets the total *number* of loans that have defaulted within the period. This sum is a Poisson Binomial random variable:

$$\sum_{i=0}^{m-1} \mathrm{B}_i \sim \mathrm{PoissonBinomial}(\boldsymbol{p}) \tag{2.7}$$

Were all the loans to have the same probability of default then this would be a *Binomial* random variable. It becomes a *Poisson* Binomial random variable when the elements of $\boldsymbol{p}$ are allowed to take different values.

2.1.3 A Heterogeneous Collection of Loans

If all of the loans are the same size then one could simply multiply the number of defaulted loans by this common loan amount to reach the total amount one stood to lose. However, it might be desirable to consider situations where the sizes are heterogeneous. For generality these sizes will be referred to as "weights"; each weight might represent a loan amount or the number of votes that an electoral college holds etc.. Let the vector $\boldsymbol{w}$ be this collection of weights. For tractability these weights are here restricted to the positive integers, see appendix A.2 for further discussion on this restriction and how it can be relaxed.

$$\boldsymbol{w} = \begin{bmatrix} w_0 & w_1 & \dots & w_{m-1} \end{bmatrix}^\top \in \mathbb{N}_+^m \tag{2.8}$$

To include these weights in the model, let the random vector $\mathbf{V}$ be defined as $\mathbf{V} = \mathbf{B} \circ \boldsymbol{w}$, where $\circ$ is the Hadamard (or element-wise) product, such that for each element

$$\mathrm{V}_i = \mathrm{B}_i w_i = \begin{cases} w_i & \text{with probability } \; p_i \\ 0 & \text{with probability } \; 1 - p_i \end{cases} \tag{2.9}$$

and the probability mass function of each element will be

$$f_{\mathrm{V}_i}(x) = \mathbb{P}(\mathrm{V}_i = x) = \mathbb{P}(\mathrm{B}_i w_i = x) = \begin{cases} p_i & \text{for } x = w_i \\ 1 - p_i & \text{for } x = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

or in vector form

$$\boldsymbol{f}_{\mathrm{V}_i} = \begin{bmatrix} 1 - p_i & 0 & \dots & 0 & p_i \end{bmatrix}^\top \qquad \text{where} \qquad \dim(\boldsymbol{f}_{\mathrm{V}_i}) = w_i + 1 \tag{2.11}$$

A scalar random variable, X, that is the the sum of the elements of $\mathbf{V}$ is said to follow the Generalised Poisson Binomial (GPB) distribution:

$$\mathrm{X} = \sum_{i=0}^{m-1} \mathrm{V}_i \sim \mathrm{GPB}(\boldsymbol{p}, \boldsymbol{w}) \tag{2.12}$$

The sum can equivalently be written as a dot product:

$$\mathrm{X} = \mathbf{B} \cdot \boldsymbol{w} \sim \mathrm{GPB}(\boldsymbol{p}, \boldsymbol{w}) \tag{2.13}$$

Zhang et al.[26] introduced the GPB in slightly different but equivalent form. A discussion of the equivalence of these forms can be found in Appendix A.2.

2.2 Example of a GPB random variable and its distribution functions

It serves to consider a simple toy example for use in this report. This example is set out in Table 2.1.

Component, j	Weight, w_j	Success Probability p_j	Failure Probability, $1 - p_j$
0	4	0.1	0.9
1	1	0.4	0.6
2	1	0.2	0.8
3	2	0.3	0.7

Table 2.1: Parameters for the Toy Example used throughout the report

At each trial, the jth component random variable, V_j, will realised the value w_j with probability p_j and realised zero otherwise. The total value of the components, $\mathrm{X} = \mathrm{V}_0 + \mathrm{V}_1 + \mathrm{V}_2 + \mathrm{V}_3$, is a GPB random variable. There are four components, so $m = 4$ with total weight of $4+1+1+2 = 8$. The GPB random variable, X, has support over the integer interval $[\![0 : 8]\!]$, which is nine integers, so $n = 9$.

There are two key *probability distribution functions* that one might wish to calculate: the probability mass function, $g(x; \boldsymbol{p}, \boldsymbol{w})$, asks what is the probability that the random variable takes a particular value, i.e. $\mathbb{P}(\mathrm{X} = x)$, and the cumulative distribution function, $G(x; \boldsymbol{p}, \boldsymbol{w})$, which asks what is the probability that the random variable takes a value less than or equal to a particular value, i.e. $\mathbb{P}(\mathrm{X} \leqslant x)$. Where $\boldsymbol{p}$ and $\boldsymbol{w}$ can be inferred from context the simplified notations $g(x)$ and $G(x)$ will be used. Note that the cdf is the cumulative sum of the pmf, i.e. $G(x) = \sum_{z \leqslant x} g(z)$.

The simplest approach to calculate the pmf of a GPB random variable is to enumerate all possible outcomes using a *probability tree*. The tree for the toy example can be seen in Figure 2.1. Each node represents a single Bernoulli trial, with the upper branch representing success, and the lower branch representing failure. Each branch shows the weight that is added to the total by that branch, and the (marginal) probability of that branch being taken at the previous node. The combined "outcome" probability of each leaf is the product of the marginal probabilities on the path from the root to the leaf, and the realised value of X for that leaf is the sum of all the weights on the same path.

The pmf at some value, x, is found by summing the outcome probabilities of all leaves where $\mathrm{X} = x$.

$$\begin{aligned}
\mathbb{P}(\mathrm{X} = 0) &= 0.3024 \\
\mathbb{P}(\mathrm{X} = 1) &= 0.2016 + 0.0756 = 0.2772 \\
\mathbb{P}(\mathrm{X} = 2) &= 0.0504 + 0.1296 = 0.1800 \\
\mathbb{P}(\mathrm{X} = 3) &= 0.0864 + 0.0324 = 0.1188 \\
\mathbb{P}(\mathrm{X} = 4) &= 0.0336 + 0.0216 = 0.0552 \\
\mathbb{P}(\mathrm{X} = 5) &= 0.0224 + 0.0084 = 0.0308 \\
\mathbb{P}(\mathrm{X} = 6) &= 0.0056 + 0.0144 = 0.0200 \\
\mathbb{P}(\mathrm{X} = 7) &= 0.0096 + 0.0036 = 0.0132 \\
\mathbb{P}(\mathrm{X} = 8) &= 0.0024
\end{aligned} \tag{2.14}$$

These results are summarised in Table 2.2 and Figure 2.2

Whilst the probability tree approach is simple to understand, the number of leaves (and therefore the computational cost) grows exponentially with m.

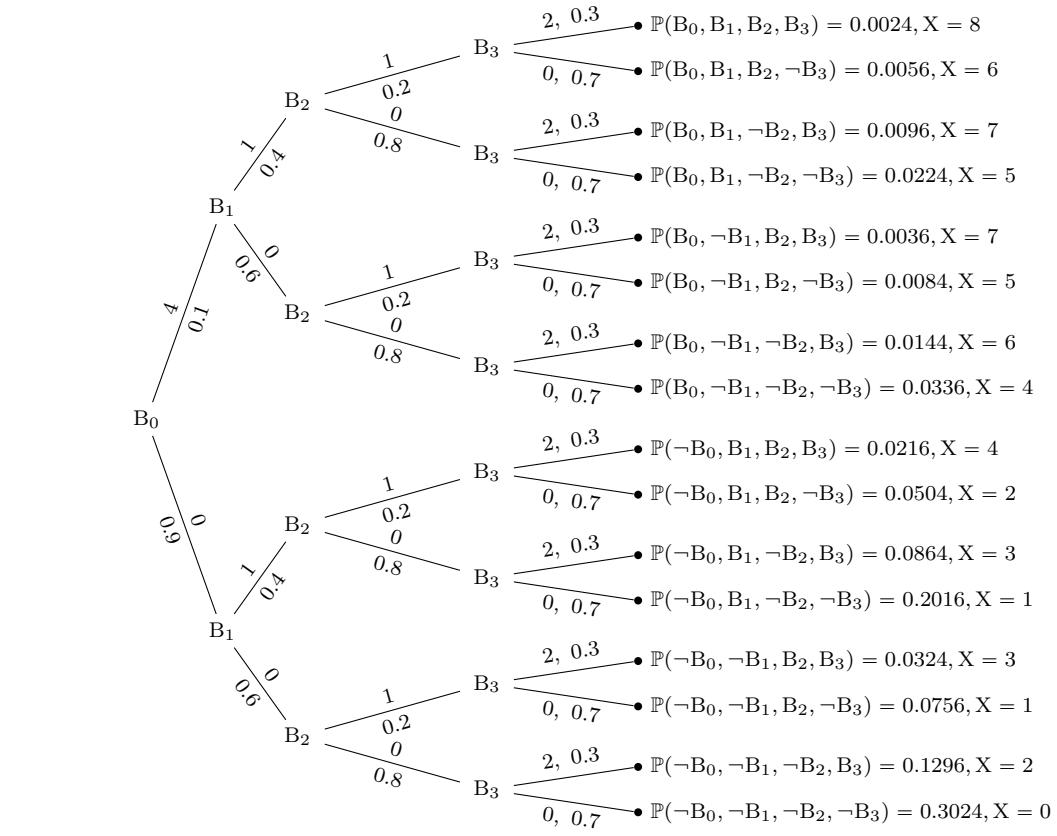

Figure 2.1: Probability Tree diagram for the naïve approach to calculating the pmf of the GPB distribution

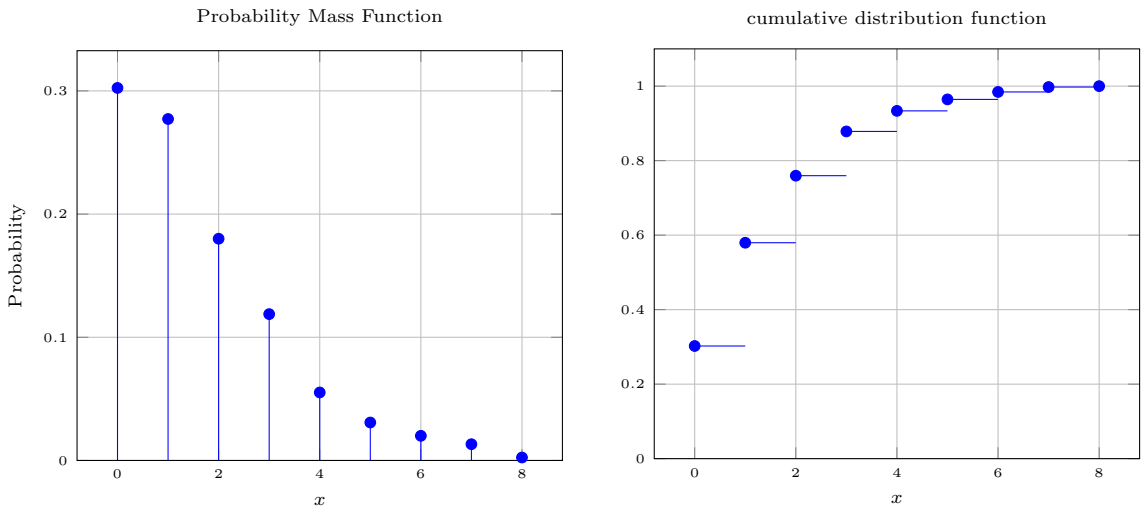

Figure 2.2: probability mass function and cumulative distribution function for the Toy Example

func. \ x	0	1	2	3	4	5	6	7	8
pmf	0.3024	0.2772	0.1800	0.1188	0.0552	0.0308	0.02	0.0132	0.0024
cdf	0.3024	0.5796	0.7596	0.8784	0.9336	0.9644	0.9844	0.9976	1

Table 2.2: probability mass function and cumulative distribution function for the toy example

When one wishes to know the value of $g(x)$ for some particular value of x, one might hope to speed things up by only visiting the leaves which result in $\mathrm{X} = x$. Unfortunately, this idea is not fruitful. The simpler decision problem of whether there even *is* a leaf which realises the value x, is equivalent to the classic subset-sum problem[2], which is known to be NP-complete. Finding *all* leaves for which $\mathrm{X} = x$ will, therefore, take (at least) exponential time.

However, as with subset sum there are pseudo-polynomial avenues that can be explored to provide faster algorithms. Existing algorithms for computing the pmf of the GPB distribution are discussed in Appendix A.1; new approaches are derived and discussed in Chapter 4.

2.3 The Latent Factor GPB distribution

2.3.1 Local Latent Factors

Suppose that underlying each binary component random variable is another "latent" random variable that is not binary. For a borrower this might be their financial health, for a US state it might be sentiment towards the candidate. These latent driving factors will vary significantly between application domains and may be multiple, however, for simplicity and generality let's encapsulate this driving latent factor for each component in a single variable, S, and assume it to be (standard) normally distributed, i.e.

$$\mathrm{S} \sim \mathcal{N}(0, 1) \tag{2.15}$$

In this framework the original Bernoulli random variable realises a 1 if S is below some threshold and 0 otherwise, e.g. if a borrower's financial health declines past a certain point then they default on their loan. This threshold can be calculated from the Bernoulli success probability, p, by using the inverse cdf (a.k.a. quantile function) of the normal distribution, $\Phi^{-1}(\cdot)$, such that

$$\mathrm{B} = \begin{cases} 1 & \text{if } \mathrm{S} \leqslant \Phi^{-1}(p) \\ 0 & \text{if } \mathrm{S} > \Phi^{-1}(p) \end{cases} \tag{2.16}$$

observing that

$$p = \mathbb{P}[\mathrm{B}{=}1] = \mathbb{P}[\mathrm{S} \leqslant \Phi^{-1}(p)] = \Phi(\Phi^{-1}(p)) \tag{2.17}$$

Each component random variable will have its own local latent factor giving the vector-valued random variable, $\mathbf{S}$, with elements

$$\mathbf{S} = \begin{bmatrix} \mathrm{S}_0 & \mathrm{S}_1 & \dots & \mathrm{S}_{m-1} \end{bmatrix}^\top \in \mathbb{R}^m \tag{2.18}$$

This random vector will therefore follow a multivariate normal distribution. For a simple model it is assumed that the correlation between the events can be described by an *equicorrelation* matrix[2], i.e. a correlation matrix with 1 on the main diagonal and the value ρ elsewhere. For an $m \times m$

[2]see Appendix A.3 for further analysis

equicorrelation matrix the following notation is adopted

$$\boldsymbol{R}_m = \begin{bmatrix} 1 & \rho & \cdots & \rho \\ \rho & 1 & \cdots & \rho \\ \vdots & \vdots & \ddots & \vdots \\ \rho & \rho & \cdots & 1 \end{bmatrix} \tag{2.19}$$

The random vector $\mathbf{S}$ then follows the m-variate normal distribution given as

$$\mathbf{S} \sim \mathcal{N}_m\left(\boldsymbol{0}_m, \boldsymbol{R}_m\right) \tag{2.20}$$

Such a simple model cannot hope to capture the full extent of the dependence structure of any real-world collection of random variables, but it does provide some extra model flexibility over the GPB, which requires mutual independence. Extension to models with other dependency structures is discussed further in section 10. Whilst understanding that all models are wrong, some are hopefully useful, and it is hoped that the this model will prove useful where the GPB distribution alone is insufficient.

2.3.2 Joint Success Probabilities: The Gaussian Copula

It will be useful to have a function which provides a probability of *joint* success of some subset of component random variables, i.e. the probability of all variables in a subset realising a success at the same time. If all joint-success probabilities are defined then all outcome probabilities can be uniquely derived (see Appendix B.2), meaning that such a function would fully describe the dependence between the component random variables.

Let $\mathcal{S} \subseteq [\![0 : m-1]\!]$ be a set of k indices with respect to the elements of $\mathbf{B}$, then the joint success probability of the component random variables whose indices are in $\mathcal{S}$ is

$$\begin{aligned} &\mathbb{P}[\mathrm{B}_i = 1 \ \forall \ i \in \mathcal{S}] \\ =&\mathbb{P}[\mathrm{S}_i \leqslant \Phi^{-1}(p_i) \ \forall \ i \in \mathcal{S}] \\ =&\Phi_k\left(\begin{bmatrix} \Phi^{-1}(p_{s_0}) \\ \Phi^{-1}(p_{s_1}) \\ \vdots \\ \Phi^{-1}(p_{s_{k-1}}) \end{bmatrix}; \boldsymbol{0}_k, \boldsymbol{R}_k\right) \\ =&C^{\text{Gauss}}_{\boldsymbol{R}_k}\left(\begin{bmatrix} p_{s_0} \\ p_{s_1} \\ \vdots \\ p_{s_{k-1}} \end{bmatrix}\right) \end{aligned} \tag{2.21}$$

where $\Phi_k(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the cdf of the k-variate normal distribution with means $\boldsymbol{\mu}$ and correlation matrix $\boldsymbol{\Sigma}$; and $C^{\text{Gauss}}_{\boldsymbol{\Sigma}}$ is the "Gaussian copula" function parametrised by correlation matrix $\boldsymbol{\Sigma}$. A full technical definition of the Gaussian copula is beyond this work, however it can be thought of as a re-parametrisation of the cdf of the standard normal distribution where the elements of the input vector are probabilities. These probabilities are transformed using the quantile function of the standard multivariate normal distribution and then the standard k-variate normal cdf applied.

There is no closed-form for the Gaussian copula function, so it is generally computed numerically[3]

[3]standard library (e.g. `SciPy`) implementations allow for arbitrary $\boldsymbol{\Sigma}$ and this generality means they were not

2.3.3 The Latent Factor GPB distribution

Let the *Latent Factor* Generalised Poisson Binomial (LFGPB) distribution be a generalisation of the Generalised Poisson Binomial distribution whereby the assumption of mutual-independence is relaxed and replaced with the assumption of joint-success probabilities given by the equicorrelation Gaussian copula. The LFGPB distribution will therefore have an extra parameter for correlation, $\rho \in [0:1)$, giving

$$\mathrm{X} \sim \mathrm{LFGPB}(\boldsymbol{p}, \boldsymbol{w}, \rho) \tag{2.22}$$

Note that the GPB distribution is a special case of the LFGPB distribution where $\rho = 0$.

When talking about a random variable that follows the LFGPB distribution the notation $\lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho)$ or simply $\lambda(x)$ will be used to refer to the pmf and $\mathit{\Lambda}(x; \boldsymbol{p}, \boldsymbol{w}, \rho)$ or simply $\mathit{\Lambda}(x)$ to refer to the cdf.

Statistical properties (mean, variance etc.) of the LFGPB distribution are derived in Appendix B.1.

sufficiently precise or quick enough for this project. Instead, a custom numerical method tailored towards equicorrelation matrices was implemented. See Appendix B.5 for details

Chapter 3

Specification of Requirements

The five objectives of the project are laid out in the 5 sections of this chapter.

3.1 Extend the algorithms of Biscari et al.[4] to apply to the GPB distribution

These algorithms will need to satisfy the following requirements:

1. Accurately calculate the value of the pmf vector of a GPB random variable for as much of the parameter space as possible, in particular they should be able to handle
   (a) Large and small numbers of component random variables, m
   (b) Large and small total weights, n
   (c) Collections of weights which follow different distributions
   (d) Collections of probabilities which follow distributions, including all close to zero and all close to one
2. Where full accuracy or precision is not achieved this should be understood
3. Have computational complexity at least as good as $O(mn)$
4. The inputs should be two vectors of length m:
   (a) a vector, $\boldsymbol{p}$, of non-trivial probabilities, i.e. real numbers from the interval $(0, 1)$ (i.e. excluding 0 and 1).
   (b) a vector $\boldsymbol{w}$ of positive integer weights, i.e. minimum value 1.
5. The result should be a vector of length $n = 1+\sum_i w_i$ with the usual properties of a probability mass function, i.e. all values are real numbers from the closed interval $[0, 1]$ and have a sum of one (allowing for some floating-number imprecision).

3.2 Parallelise the above algorithms

Any parallel algorithm should satisfy the same requirements as mentioned above for serial algorithms, but allow for computation using parallel processing. In particular, they should be disposed towards GPU-acceleration.

3.3 Extend the GPB distribution to allow for some simple correlation structure

This project will extend the GPB distribution to allow for non-independent events using the Latent Factor Generalised Poisson Binomial distribution. Methods for computing its pmf vector will be developed. In addition to the requirements listed in Section 3.1 for the GPB distribution, methods for computing the pmf vector of the LFGPB distribution should allow for an additional parameter, ρ, to represent the correlation between the component random variables.

3.4 Create a Python package that implements the above

The project should deliver a python package that can be used by people interested in dealing with GPB and LFGPB random variables. The package should satisfy the following requirements:

1. Be a full python package importable with `pip`, (though upload to PyPI will be delayed until the project has been submitted and marked, to avoid questions of plagiarism).
2. Provide classes for working with GPB and LFGPB random variables. In order to provide a familiar interface and the potential for interoperability they should, where possible, follow the example of the elements of the `scipy.stats` library, preferably subclassing `scipy.stats.rv_discrete`.
3. The library must be able to handle users with and without the availability of GPU-acceleration.
4. The user should be able to choose the algorithm and hardware (i.e. CPU or GPU) that is used to calculate the distribution functions of their random variables, choosing, for example, whether to prioritise speed over accuracy/precision when there is a trade off.
5. When the user does not specify a particular approach to computation, the fastest option should be chosen for them.
6. Unit tests should be provided for all functionality to ensure correct behaviour.
7. The library should be properly documented.
8. Code should conform to proper standards including PEP 8 and suitable docstrings.

3.5 Demonstrate how the package might be used in the real world.

The project should provide an example application of the `pyGPB` package, drawing on data from the real world, and demonstrate how insights might be drawn about the example.

Chapter 4

Approaches to computing the pmf of the GPB distribution

4.1 Summary of approaches

A summary of the different approaches to calculating the pmf of the GPB distribution can be seen in table 4.1. All of these approaches were implemented as part of this project and tested for speed and accuracy/precision. The results of those tests can be found in Chapters 7 and 8.

Algorithm	Complexity	Notes
`GPB-Naive`	$O(2^m)$	Naïve brute-force approach via probability tree, demonstrated in Figure 2.1
`GPB-CF`	$O(mn)$	The Characteristic-Function approach with a final FFT. Equivalent to the DFT-CF algorithm of Zhang et al.[26]
`GPB-DP`	$O(mn)$	The Dynamic-Programming or "Direct Convolution" (DC) algorithm from Biscari et al.[4] adapted to the GPB
`GPB-DP-DC-FFT-Combo`	$O(n \log m \log n)$	The divide-and-conquer FFT tree with Dynamic Programming for small subsets. An adaptation of the DC-FFT algorithm from Biscari et al.[4] to the GPB
`GPB-DP-Parallel`		Parallelised version of `GPB-DP`
`GPB-DP-DC-FFT-Parallel`		Parallelised version of `GPB-DP-DC-FFT-Combo`

Table 4.1: Summary of Approaches to computing the pmf of the GPB distribution

All of these approaches seek to calculate the full pmf vector, $\boldsymbol{g}$, i.e. the value of $g(x)$ for every $0 \leqslant x < n$. The possibility of even faster approaches when one only wishes to compute some smaller part of the pmf is discussed in Section 10, but were not explored in this project.

The first two approaches were implemented by Zhang et al. [26], and a summary of these approaches can be found in Appendix A.1. The remaining four algorithms are novel and were developed as an objective of this project. Their derivation and pseudocode can be found in the sections of this chapter.

The middle two algorithms were discussed by Biscari et al. [4] in their treatment of the simpler Poisson Binomial (PB) distribution, and it seems also recently adapted to the GPB by Junge

(2021)[13] in the `PoissonBinomial` package for the `R` language, though the details of that adaptation have not yet been published[1]. These approaches are very similar to classic approaches to the subset-sum problem. The similarity between this project and the subset sum problem is discussed in Appendix A.3.

The last two approaches are parallelised versions of the previous two. We have not been able to find any parallelised approaches to the GPB or PB distributions in the literature, nor implemented in any package.

4.2 Dynamic Programming Approach, $O(nm)$

A dynamic programming approach was applied to the Poisson Binomial distribution by Biscarri et al. (2018)[4]. They refer to this method as "Direct Convolution", or DC, however this report refers to it as the Dynamic Programming (DP) method to avoid confusion with "DC" also potentially standing for "divide-and-conquer". The DP approach is here extended to apply more generally to the GPB distribution.

As a notational convention, let $\mathrm{X}_{<k}$ be the GPB random variable achieved by summing the first k elements of $\mathbf{V}$, and let $g_{<k}(x)$ be the pmf of $\mathrm{X}_{<k}$ i.e.

$$\mathrm{X}_{<k} = \sum_{i=0}^{k-1} \mathrm{V}_i \sim \mathrm{GPB}([p_0, p_1, ..., p_{k-1}], [w_0, w_1, ..., w_{k-1}]) \tag{4.1}$$

Returning to the toy example set out in Section 2.2, suppose the pmf of $\mathrm{X}_{<3} \sim \mathrm{GPB}([0.1, 0.4, 0.2], [4, 1, 1])$ is already known and the pmf of $\mathrm{X}_{<4} \sim \mathrm{GPB}([0.1, 0.4, 0.2, 0.3], [4, 1, 1, 2])$ is now required.

Note that $\mathrm{X}_{<4} = \mathrm{X}_{<3} + \mathrm{V}_3$, which means $\mathrm{X}{=}x$ if and only if ($\mathrm{X}_{<3}{=}x$ and $\mathrm{V}_3{=}0$) *or* ($\mathrm{X}_{<3}{=}x{-}w_3$ and $\mathrm{V}_3{=}w_3$). This insight leads to

$$\begin{aligned} g_{<k}(x) &= \mathbb{P}[\mathrm{X}_{<k}{=}x] \\ &= \mathbb{P}[(\mathrm{X}_{<k-1}{+}\mathrm{V}_k){=}x] \\ &= \mathbb{P}[\mathrm{X}_{<k-1}{=}x,\ \mathrm{V}_k{=}0] + \mathbb{P}[\mathrm{X}_{<k-1}{=}x{-}w_k,\ \mathrm{V}_k{=}w_k] \\ &= \mathbb{P}[\mathrm{X}_{<k-1}{=}x] \cdot (1-p_k) + \mathbb{P}[\mathrm{X}_{<k-1}{=}x{-}w_k] \cdot p_k \end{aligned} \tag{4.2}$$

and finally the recurrence relation

$$g_{<k}(x) = (1-p_k) \cdot g_{<k-1}(x) + p_k \cdot g_{<k-1}(x-w_k) \tag{4.3}$$

Dynamic programming can now be used to add each component random variable to construct the combined pmf, as demonstrated in Table 4.2. In the zeroth row, column 0 takes value $(1-p)$ and column w_0 takes value p_0, thereafter the cell, $c_{i,j}$ in row i and column j is calculated as:

$$c_{i,j} = \begin{cases} (1-p) \cdot c_{i-1,j} & \text{for } \ j < w_i \\ (1-p) \cdot c_{i-1,j} + p_i \cdot c_{i-1,j-w_i} & \text{for } \ j \geqslant w_i \end{cases} \tag{4.4}$$

Algorithm 1 shows how this technique can be applied algorithmically

Complexity Analysis: This table has m rows and n columns, so $m \cdot n$ total cells. Not all cells need to be computed, but the amount that do is still $O(n{\cdot}m)$ so the time complexity of this approach

[1] or at least not as far as we can find

p_i	w_i	i \ j	0	1	2	3	4	5	6	7	8
0.1	4	0	0.9	0	0	0	0.1				
0.4	1	1	0.54	0.36	0	0	0.06	0.04			
0.2	1	2	0.432	0.396	0.072	0	0.048	0.044	0.008		
0.3	2	3	0.3024	0.2772	0.18	0.1188	0.0552	0.0308	0.02	0.0132	0.0024

Table 4.2: Tabular layout of computing the GPB pmf via Dynamic Programming

Algorithm 1: GPB pmf computation with Dynamic Programming

Function `GPB-DP`$(\boldsymbol{p}, \boldsymbol{w})$**:**

```
sort p and w so that the smallest weights come first
a ← array of zeros of length 1 + Σ_i w_i
a_0 ← 1.0
n = 1
for j = 0 to m − 1 do
    for i = n + w_j to 0 step −1 do
        if i < w_j then
            a_i ← a_i · (1−p_j)
        else
            a_i ← a_i · (1−p_j) + a_{i−w_j} · p_j
        end
    end
    n ← n + w_j
end
return a
```

is $O(n \cdot m)$, which is the same as the DFT-CF altorithm employed by Zhang et al. (2018)[26] (see Algorithm 8), however this dynamic programming solution uses only simple floating-point operations whereas the DFT-CF approach uses lots of complex-number arithmetic, slow functions like the complex exponential, and also bears the overhead of an FFT, so the Dynamic Programming algorithm executes much faster in practice (whilst still having the same time complexity).

4.3 Divide and Conquer FFT approach, O(n log m log n)

Again this approach was applied to the Poisson-Binomial distribution by Biscarri et al[4], and here it is extended to the GPB distribution.

4.3.1 Primer on Convolution and Omnivolution

Suppose there are two independent random variables, A and B, for which the pmfs are already known (f_{A} and f_{B} respectively). It is wished to know the pmf of the sum of A and B. The pmf of the sum of two independent variables is the *convolution* of their individual pmfs, where convolution, denoted by $*$, is defined as

$$\mathbb{P}[\mathrm{A}+\mathrm{B}=x] = f_{\mathrm{A}+\mathrm{B}}(x) = [f_{\mathrm{A}} * f_{\mathrm{B}}](x) = \sum_{k=-\infty}^{\infty} f_{\mathrm{A}}(k) \cdot f_{\mathrm{B}}(x-k) \tag{4.5}$$

The pmf of the sum of a collection of m random variables, $\mathrm{X} = \sum_{j=0}^{m-1} \mathrm{A}_j$, can be expressed as the iterated convolution, or *omnivolution*, of the individual pmfs, i.e.

$$f_{\mathrm{X}}(x) = [f_{\mathrm{A}_0} * f_{\mathrm{A}_1} * ... * f_{\mathrm{A}_m}](x) \equiv \left[\bigast_{j=0}^{m-1} f_{\mathrm{A}_j} \right](x) \tag{4.6}$$

4.3.2 Divide and Conquer

Omnivolution can be performed in a divide-and-conquer fashion by splitting the collection of random variables into two at the midpoint, $k = \lfloor \frac{m}{2} \rfloor$, to get

$$\mathrm{X}_{<k} = \sum_{j=0}^{k-1} \mathrm{A}_j, \qquad \mathrm{X}_{\geqslant k} = \sum_{j=k}^{m-1} \mathrm{A}_j \tag{4.7}$$

and convolving the pmfs of each half together, giving the recurrence relation:

$$f_{\mathrm{X}}(x) = [f_{\mathrm{X}_{<k}} * f_{\mathrm{X}_{\geqslant k}}](x) \tag{4.8}$$

4.3.3 Convolution via Fourier Transforms

The Convolution Theorem links convolution to the Fourier Transform[2]. Usefully, the Fourier Transform of the convolution of two functions is identical to the Hadamard (i.e. element-wise) product of the Fourier Transform of each function. Put formally:

$$f(x) = [g * h](x) \iff \mathcal{F}\{f\}(y) = \mathcal{F}\{g\}(y) \circ \mathcal{F}\{h\}(y)$$

where $\circ$ is the Hadamard product. If $\boldsymbol{g}$ and $\boldsymbol{h}$ are known then $\boldsymbol{f}$ can be computed via

$$\boldsymbol{f} = [\boldsymbol{g} * \boldsymbol{h}](x) = \mathcal{F}^{-1}\{\mathcal{F}\{\boldsymbol{g}\} \circ \mathcal{F}\{\boldsymbol{h}\}\} \tag{4.9}$$

As a practical note, for this to work the input and output vectors all must have the same number of elements, so the input vectors need to be padded on the right-hand-side with zeros until they have $(\dim(\boldsymbol{g}) + \dim(\boldsymbol{h}) - 1)$ elements, which will be the same dimensionality as $\boldsymbol{f}$.

4.3.4 Application to the GPB distribution

The divide-and-conquer approach laid out in section 4.3.2 can be combined with Fast-Fourier-Transform algorithms to compute the pmf vector of the GPB distribution recursively as laid out in Algorithm 2. This creates a divide-and-conquer FFT "tree", where the leaves are the pmfs of the individual component random variables, and each node represents the convolution of two pmfs together until the pmf of the entire distribution is achieved at the root.

[2] A primer on the Fourier Transform is provided in Appendix A.1.2

Algorithm 2: GPB pmf computation with Divide-and-Conquer FFT Omnivolution

Function `GPB-DC-FFT`$(\{(p_0, w_0), (p_1, w_1), ..., (p_{m-1}, w_{m-1})\})$**:**

  if $m = 1$ **then**

    return the vector $[(1-p), 0, \ldots, 0, p]$ of size $w_0 + 1$

  else

    $k \leftarrow \lfloor \frac{m}{2} \rfloor$

    $L \leftarrow$ `GPB-DC-FFT`$(\{(p_0, w_0), (p_1, w_1), ..., (p_{k-1}, w_{k-1})\})$

    $R \leftarrow$ `GPB-DC-FFT`$(\{(p_k, w_k), (p_{k+1}, w_{k+1}), ..., (p_{m-1}, w_{m-1})\})$

    pad L and R with zeros to size $(\dim(L) + \dim(R) - 1)$

    return `InverseFFT(FFT(`L`)` $\circ$ `FFT(`R`))`

  end

4.3.5 Application to the Toy Example

1. Begin with the initial pmf vectors for each component random variable. These will have the value $1 - p_1$ for the zeroth element, the value p_i as the w_ith element, and zeros elsewhere:

$$\begin{aligned}
\boldsymbol{g}_0 &= \begin{bmatrix} 0.9 & 0 & 0 & 0 & 0.1 \end{bmatrix}^\top \\
\boldsymbol{g}_1 &= \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}^\top \\
\boldsymbol{g}_2 &= \begin{bmatrix} 0.8 & 0.2 \end{bmatrix}^\top \\
\boldsymbol{g}_3 &= \begin{bmatrix} 0.7 & 0 & 0.3 \end{bmatrix}^\top
\end{aligned}$$

2. then in pairs, pad them to the appropriate length (one less than each pair's previous total length)

$$\begin{aligned}
\boldsymbol{g}_0 &= \begin{bmatrix} 0.9 & 0 & 0 & 0 & 0.1 & 0 \end{bmatrix}^\top \\
\boldsymbol{g}_1 &= \begin{bmatrix} 0.6 & 0.4 & 0 & 0 & 0 & 0 \end{bmatrix}^\top \\
\boldsymbol{g}_2 &= \begin{bmatrix} 0.8 & 0.2 & 0 & 0 \end{bmatrix}^\top \\
\boldsymbol{g}_3 &= \begin{bmatrix} 0.7 & 0 & 0.3 & 0 \end{bmatrix}^\top
\end{aligned}$$

3. perform a real[3] FFT on each vector

$$\begin{aligned}
\hat{\boldsymbol{g}}_0 &= \mathrm{RFFT}(\boldsymbol{g}_0) = \begin{bmatrix} 1 & 0.85{+}0.0866i & 0.85{-}0.0866i & 1 \end{bmatrix}^\top \\
\hat{\boldsymbol{g}}_1 &= \mathrm{RFFT}(\boldsymbol{g}_1) = \begin{bmatrix} 1 & 0.8{-}0.3464i & 0.4{-}0.3464i & 0.2 \end{bmatrix}^\top \\
\hat{\boldsymbol{g}}_2 &= \mathrm{RFFT}(\boldsymbol{g}_2) = \begin{bmatrix} 1 & 0.8{-}0.2i & 0.6 \end{bmatrix}^\top \\
\hat{\boldsymbol{g}}_3 &= \mathrm{RFFT}(\boldsymbol{g}_3) = \begin{bmatrix} 1 & 0.4 & 1 \end{bmatrix}^\top
\end{aligned}$$

4. multiply pairs element-wise

$$\begin{aligned}
\hat{\boldsymbol{g}}_0 \circ \hat{\boldsymbol{g}}_1 &= \begin{bmatrix} 1 & 0.71 - 0.2252i & 0.31 - 0.329 & 0.2 \end{bmatrix}^\top \\
\hat{\boldsymbol{g}}_2 \circ \hat{\boldsymbol{g}}_3 &= \begin{bmatrix} 1 & 0.32 - 0.08i & 0.6 \end{bmatrix}^\top
\end{aligned}$$

[3] see Appendix A.1.2 for the meaning of "real" in this context

5. perform an inverse real FFT on each vector

$$\boldsymbol{g}_0 * \boldsymbol{g}_1 = \text{IRFFT}(\hat{\boldsymbol{g}}_0 \circ \hat{\boldsymbol{g}}_1) = \begin{bmatrix} 0.54 & 0.36 & 0 & 0 & 0.06 & 0.04 \end{bmatrix}^\top$$
$$\boldsymbol{g}_2 * \boldsymbol{g}_3 = \text{IRFFT}(\hat{\boldsymbol{g}}_2 \circ \hat{\boldsymbol{g}}_3) = \begin{bmatrix} 0.56 & 0.14 & 0.24 & 0.06 \end{bmatrix}^\top$$

6. return to step 2 and loop until only one vector remains, finally getting

$$\boldsymbol{g}_0 * \boldsymbol{g}_1 * \boldsymbol{g}_2 * \boldsymbol{g}_2 = \begin{bmatrix} 0.3024 & 0.2772 & 0.18 & 0.1188 & 0.0552 & 0.0308 & 0.02 & 0.0132 & 0.0024 \end{bmatrix}^\top$$

Complexity Analysis: The first iteration of the loop requires FFTs of total length $n + m - 1$, going down to $n + 1$ in the final iteration. As $m < n$ this total length is always less than $2n$, which is $O(n)$. Performing FFTs of total length $O(n)$ takes as most $O(n \log n)$. Each iteration halves the number of pmfs which remain, so there will need to be $O(\log m)$ iterations in total. $O(\log m)$ iterations of $O(n \log n)$ leads to total time complexity of $O(n \log n \log m)$.

4.3.6 Combination Approach

For smaller values of m the overheads of FFTs and recursion outweigh the benefits of lower computational complexity. A combination of the FFT-tree and dynamic-programming techniques proves faster. This approach is laid out in Algorithm 3.

Algorithm 3: GPB pmf computation with a combination of divide-and-conquer FFT Omnivolution and Dynamic Programming

Parameter: μ, the maximum size of m for dynamic programming

Function `GPB-DP-DC-FFT-Combo`$(\{(p_0, w_0), (p_1, w_1), ..., (p_{m-1}, w_{m-1})\})$**:**
  if $m < \mu$ **then**
    return `GPB-DP`$([p_0, p_1, \ldots, p_{m-1}], [w_0, w_1, \ldots, w_{m-1}])$
  else
    $k \leftarrow \lfloor \frac{m}{2} \rfloor$
    $L \leftarrow$ `GPB-DP-DC-FFT-Combo`$(\{(p_0, w_0), (p_1, w_1), ..., (p_{k-1}, w_{k-1})\})$
    $R \leftarrow$ `GPB-DP-DC-FFT-Combo`$(\{(p_k, w_k), (p_{k+1}, w_{k+1}), ..., (p_{m-1}, w_{m-1})\})$
    pad L and R with zeros to length $(\dim(L) + \dim(R) - 1)$
    return `InverseFFT(FFT(`L`)` $\circ$ `FFT(`R`))`
  end

4.4 Parallel Dynamic Programming

The inner `for` loop from the dynamic programming approach from Algorithm 1 can be made "embarrassingly parallelizable" with only a small modification. If the algorithm uses two arrays, one for reading the result of the previous outer-loop iteration, and one for writing the result of the current outer-loop iteration then the iterations of the inner loop no longer have any temporal reliance on each other and can be performed in parallel. Once all parallel threads have finished their task (i.e. they have been synchronised) the pointers to the "read" array and "write" array can be swapped. The approach is outlined in Algorithm 4.

Algorithm 4: GPB pmf computation with Dynamic Programming in Parallel

Function `GPB-DP-Parallel(`$\boldsymbol{p}, \boldsymbol{w}$`):`

```
sort p and w so that the smallest weights come first
ar, aw ← arrays of zeros each of length 1 + Σ_i w_i
ar_0 ← 1.0
n ← 0
for j = 0 to m − 1 do
    for i = 0 to n + w_j in parallel do
        if i < w_j then
            aw_i ← ar_i · (1−p_j)
        else
            aw_i ← ar_i · (1−p_j) + ar_{i−w_j} · p_j
        end
    end
    synchronize threads
    swap pointers to ar and aw
    n ← n + w_j
end
return ar
```

4.5 Parallel DC-FFT and DP Combination

Fast Fourier Transforms are themselves embarrassingly parallel and libraries already exist to take account of the speedup possible. Unfortunately, the overhead for each FFT is larger than a serial FFT, and so the benefit only really comes with large arrays. One way to reduce this overhead is to do so-called "batch" FFTs where multiple transforms of the same length are performed together. As the component random variables of a GPB random variable do not (necessarily) have pmfs of the same length, they require padding with zeros to, potentially, significantly longer than otherwise. This extra length does waste some time in performing the FFT, however the reduced overhead can make it worthwhile.

As with the serial `GPB-DP-DC-FFT-Combo` approach shown in Algorithm 3, it makes sense to use Dynamic Programming first for small collections of events, and then use FFT to omnivolve the pmfs together. A collection of smaller pmfs is produced first using a parallel DP approach, arranging them as rows of a 2D array, with each row being the pmf of a disjoint subset of the original component random variables. The rows of this 2D array can then be omnivolved together iteratively using parallel batch FFT. The process is outlined in Algorithm 5

Algorithm 5: GPB pmf computation with a combination of Dynamic Programming and batch FFT in Parallel

Parameter: n, the number of disjoint subsets

Function `GPB-DP-DC-FFT-Parallel`$(\boldsymbol{p}, \boldsymbol{w})$**:**

`/* Prepare subsets of events */`

$\ell \leftarrow \left\lceil \frac{\dim(\boldsymbol{w})}{n} \right\rceil$

$\mathbf{W} = \{\boldsymbol{w}_0, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}\} \leftarrow$ partition $\boldsymbol{w}$ into n disjoint subsets of cardinality ℓ, minimizing disparity between total weight of each subset

sort the contents of each $\boldsymbol{w}_0$ in ascending order

$\mathbf{P} \leftarrow \{\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1}\} \leftarrow$ subsets of probabilities associated with the subsets and elements of $\mathbf{W}$

$m \leftarrow 1+$ largest total weight of any subset in $\mathbf{W}$

$\boldsymbol{ar}, \boldsymbol{aw} \leftarrow$ arrays of zeros each of shape n rows by m columns

$ar_{:,0} \leftarrow 1.0$

$N \leftarrow$ array of zeros of length n

`/* Use Parallel Dynamic Programming to prepare pmf of each subset */`

for $i = 0$ **to** $\ell - 1$ **do**
  for $j = 0$ **to** n ***in parallel*** **do**
    for $k = 0$ **to** $N_j + w_{j,i}$ ***in parallel*** **do**
      if $k < w_{j,i}$ **then**
        $aw_{j,k} \leftarrow ar_{j,k} \cdot (1 - p_{j,i})$
      else
        $aw_{j,k} \leftarrow ar_{j,k} \cdot (1 - p_{j,i}) + ar_{j,k-w_{j,i}} \cdot p_{j,i}$
      end
    end
  end
  synchronize threads
  swap pointers to $\boldsymbol{ar}$ and $\boldsymbol{aw}$
  for $j = 0$ **to** n ***in parallel*** **do**
    $N_j \leftarrow N_j + w_{j,i}$
  end
end

`/* Omnivolve rows together using Batch Parallel FFT */`

while $n > 1$ **do**
  $m \leftarrow 2m + 1$
  $\boldsymbol{aw} \leftarrow$ pad rows of $\boldsymbol{ar}$ to length m
  $\boldsymbol{ar} \leftarrow$ `BatchParallelRFFT` $(\boldsymbol{aw})$
  $\boldsymbol{aw} \leftarrow$ mulitiply disjoint pairs of rows of $\boldsymbol{ar}$ together elementwise, returning $\lceil n/2 \rceil$ rows
  $\boldsymbol{ar} \leftarrow$`InverseBatchParallelFFT` $(\boldsymbol{aw})$
  $n \leftarrow \lceil n/2 \rceil$
end
return $\boldsymbol{ar}_0, :$

Chapter 5

Approaches to computing the pmf of the LFGBP Distribution

This chapter sets out the method that was developed for the `pyGPB` package for computing the pmf of a LFGPB random variable. It is referred to as the Quadrature approach. A second method based on linear algebra and brute force was developed for benchmarking, but has very poor computational complexity and so was not a viable method for general use. The derivation of this approach can be found in Appendix B.2. A third method was explored but abandoned, see Appendix B.4 for details of this abandoned approach.

5.1 A Conditional-Independence Framework

5.1.1 A Global Latent Factor

Suppose that along with the local latent factors discussed in Section 2.3.1 there is also a global latent factor affecting the component random variables. For loans this may be the state of the economy, for an election it might be the quality of the national candidate. Again, the global factors will vary depending on the actual domain in question, but for simplicity and generality let's assume it is a standard normal random variable, T.

$$\mathrm{T} \sim \mathcal{N}(0, 1) \tag{5.1}$$

5.1.2 Conditional Independence

It is now assumed that events are affected by their own local situation *and* the global situation *but not* the local situations of other events. For borrowers this would mean that whilst they are all affected by a worsening economy, their own financial health is not dependent on the financial health of the strangers who happen to borrow from the same lender. In an election this would be assuming that a state's electorate is affected by their own local politics and by national politics, but not by the local politics of other states.

This means that the correlation between the local latent factors is not indicative of a direct causal link, instead they happen to be correlated because of a causal link with the global latent factor.

This idea can be formalised by saying that the local factors *conditional* on the global factor are

mutually independent, and so

$$\mathbf{S}|(\mathrm{T}{=}t) \sim \mathcal{N}_m\left(\boldsymbol{0}_m, \boldsymbol{I}_m\right) \tag{5.2}$$

If two elements, S_i and S_j, are conditionally independent as above and yet both have correlation with T of ρ_{ST}, then they will have an observed correlation of ρ_{ST}^2 with each other (see [8]), and the multivariate normal random vector $\mathbf{S}$ will have correlation matrix

$$\begin{bmatrix} 1 & \rho_{ST}^2 & \cdots & \rho_{ST}^2 \\ \rho_{ST}^2 & 1 & \cdots & \rho_{ST}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{ST}^2 & \rho_{ST}^2 & \cdots & 1 \end{bmatrix} \tag{5.3}$$

which is an equicorrelation matrix. A set of latent random variables whose causal structure is as defined above will be an LFGPB random variable with correlation parameter $\rho = \rho_{ST}^2$.

Importantly, the probability mass function of a random variable is uniquely defined by the parameters of the distribution, as such this random variable will have exactly the same pmf as all other LFGPB random variables with the same $\boldsymbol{p}$, $\boldsymbol{w}$ and with $\rho = \rho_{ST}^2$, whether or not the underlying causal reality is as describe above. In this way we can compute the the pmf of any LFGPB random variable by assuming the above conditional independence framework, with no loss of generality.

5.2 The LFGPB pmf as an Expectation

As the elements of the *conditional* random vector $\mathbf{S}|(\mathrm{T}{=}t)$ are mutually independent, so will be the elements of $\mathbf{V}|(\mathrm{T}{=}t)$. Therefore, the *conditional* random variable $\mathrm{X}|(\mathrm{T}{=}t)$ has mutually-independent component random variables and follows the GPB distribution

$$\mathrm{X}|(\mathrm{T}{=}t) \sim \mathrm{GPB}(\boldsymbol{p}|(\mathrm{T}{=}t), \boldsymbol{w}) \tag{5.4}$$

And so by the *Law of the Unconscious Statistician*

$$\begin{aligned} \lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho) &= \mathbb{E}_t\left[g(x, \boldsymbol{p}|(\mathrm{T}{=}t), \boldsymbol{w})\right] \\ &= \int_{-\infty}^{+\infty} \phi(t) \cdot g(x, \boldsymbol{p}|(\mathrm{T}{=}t), \boldsymbol{w}) \mathrm{d}t \end{aligned} \tag{5.5}$$

where $\phi(\cdot)$ is the pdf of the standard normal distribution. To make the limits finite, substitute $t = \Phi^{-1}(z)$ where $\mathrm{Z} \sim \mathcal{U}[0,1]$, and therefore $f_{\mathrm{Z}}(z) = 1$, to get

$$\lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho) = \int_0^1 g(x, \boldsymbol{p}|(\mathrm{T}{=}\Phi^{-1}(z)), \boldsymbol{w}) \mathrm{d}z \tag{5.6}$$

This provides the possibility of computing $\lambda(x; \boldsymbol{p}, \boldsymbol{w}, \rho)$ using numerical integration so long as it is possible to compute the conditional probabilities $\boldsymbol{p}|(\mathrm{T}{=}\Phi^{-1}(z))$

5.2.1 Conditional Probabilities

As each S_i and T are bivariate normally distributed with correlation $\rho_{ST} = \sqrt{\rho}$, we can use a standard result to get the distribution of S_i conditional on T

$$\begin{aligned} \mathrm{S}_i|(\mathrm{T}{=}t) \; &\sim \mathcal{N}\left(\mu_{\mathrm{S}_i} + \frac{\sigma_{\mathrm{S}_i}}{\sigma_{\mathrm{T}}}\rho_{\mathrm{ST}}(t-\mu_{\mathrm{T}}),\, (1-\rho_{\mathrm{ST}}^2)\sigma_{\mathrm{S}_i}^2\right) \\ &\sim \mathcal{N}\left(t\cdot\sqrt{\rho}, 1-\rho\right) \end{aligned} \tag{5.7}$$

The cdf of the generic Normal distribution is $F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$, so for $\mathrm{S}_i|\mathrm{T} = t$ the cdf will be

$$\mathbb{P}[\mathrm{S}_i \leqslant s \mid \mathrm{T} = t] = \Phi\left(\frac{s - t\cdot\sqrt{\rho}}{\sqrt{1-\rho}}\right) \tag{5.8}$$

Which in combination with equation 2.17, and again substituting in $t = \Phi^{-1}(z)$, gives

$$p_i|\left(\mathrm{T}{=}\Phi^{-1}(z)\right) = \mathbb{P}[\mathrm{S} \leqslant \Phi^{-1}(p_i) \mid \mathrm{T}{=}\Phi^{-1}(z)] = \Phi\left(\frac{\Phi^{-1}(p_i) - \Phi^{-1}(z)\cdot\sqrt{\rho}}{\sqrt{1-\rho}}\right) \tag{5.9}$$

With the above, it is possible to define the vector-valued function $\kappa(z;\boldsymbol{p},\rho)$ for conditional probabilities, such that

$$\kappa(z;\boldsymbol{p},\rho)_i = \left[\boldsymbol{p}|\left(T{=}\Phi^{-1}(z)\right)\right]_i = \Phi\left(\frac{\Phi^{-1}(p_i) - \Phi^{-1}(z)\cdot\sqrt{\rho}}{\sqrt{1-\rho}}\right) \tag{5.10}$$

5.2.2 The probability mass function

Finally, by combining Equations 5.6 and 5.10 we get an expression for the pmf of the LFGPB distribution as

$$\lambda(x;\boldsymbol{p},\boldsymbol{w},\rho) = \int_0^1 g(x,\kappa(z;\boldsymbol{p},\rho),\boldsymbol{w})\mathrm{d}z \tag{5.11}$$

An analytical solution to the integral seems unlikely, so a numerical approach is necessary.

5.2.3 Numerical Integration, aka Quadrature

As with the FFTs, this project has happily relied on the solutions of others for quadrature; `SciPy` provides the fast `quad_vec` function for numerical integration of vector-valued functions, however it only works on arrays in main memory, not on those stored in GPU memory. The overhead of transporting arrays from GPU to CPU is non trivial, especially when it needs to be done several thousand times in order for the integral to reach the necessary precision. Whilst the `CuPy` library has implemented much of `SciPy`'s functionality for use on GPU arrays, it has not yet implemented `quad_vec`. As such the `pyGPB` package needed its own quadrature function. Our implementation was based on the same Gauss-Kronrod (GK) approach used by `SciPy`. See Appendix B.3 for details on GK Quadrature.

Chapter 6

Software Implementation

6.1 Language and Framework

Python was chosen for the majority of this project. Python was chosen because:

- It is the language with which the author is most familiar, resulting in a faster and easier route to results.
- The easiest and cheapest way to get access to CUDA-enabled GPUs was via the Google Colab platform. The notebooks for Google Colab are designed to be used with Python.
- The original `GPB` software package produced by [26], and the recent `PoissonBinomial` package produced by [13], are for the `R` language; there is no current library in Python, so there was a "gap in the market".
- There are several Python libraries for dealing with GPU computation.

The key drawback of Python is speed, which is especially a problem for a project whose main goal is improving algorithmic performance. The author would certainly have preferred to use C++ for the whole project, but given the extent of the project there was not time to learn another language to a robust enough standard.

Whilst the majority of the project was implemented in Python, the GPU kernels *were* written in CUDA C++. It is possible to implement them in Python and use a just-in-time compiler such as `cuda.numba` or `cupyx.jit`, however, these were found to be noticeably slower than kernels written directly in CUDA C++.

As mentioned in the benefits of Python above, Google Colab was used as a platform/IDE for the majority of the work, with only the final packaging done on a local machine.

6.2 Key Tools

This project, and the `pyGPB` Library which forms part of it, made use of several external libraries and tools, in particular for: array manipulation, function compilation, Fast Fourier Transforms and quadrature. The key classes of pyGPB were also subclassed from an external library. These relationships are summarised in Figure 6.1. The key tools were:

- For the CPU approaches to computation the numerical array functionality of `numpy` was used extensively.

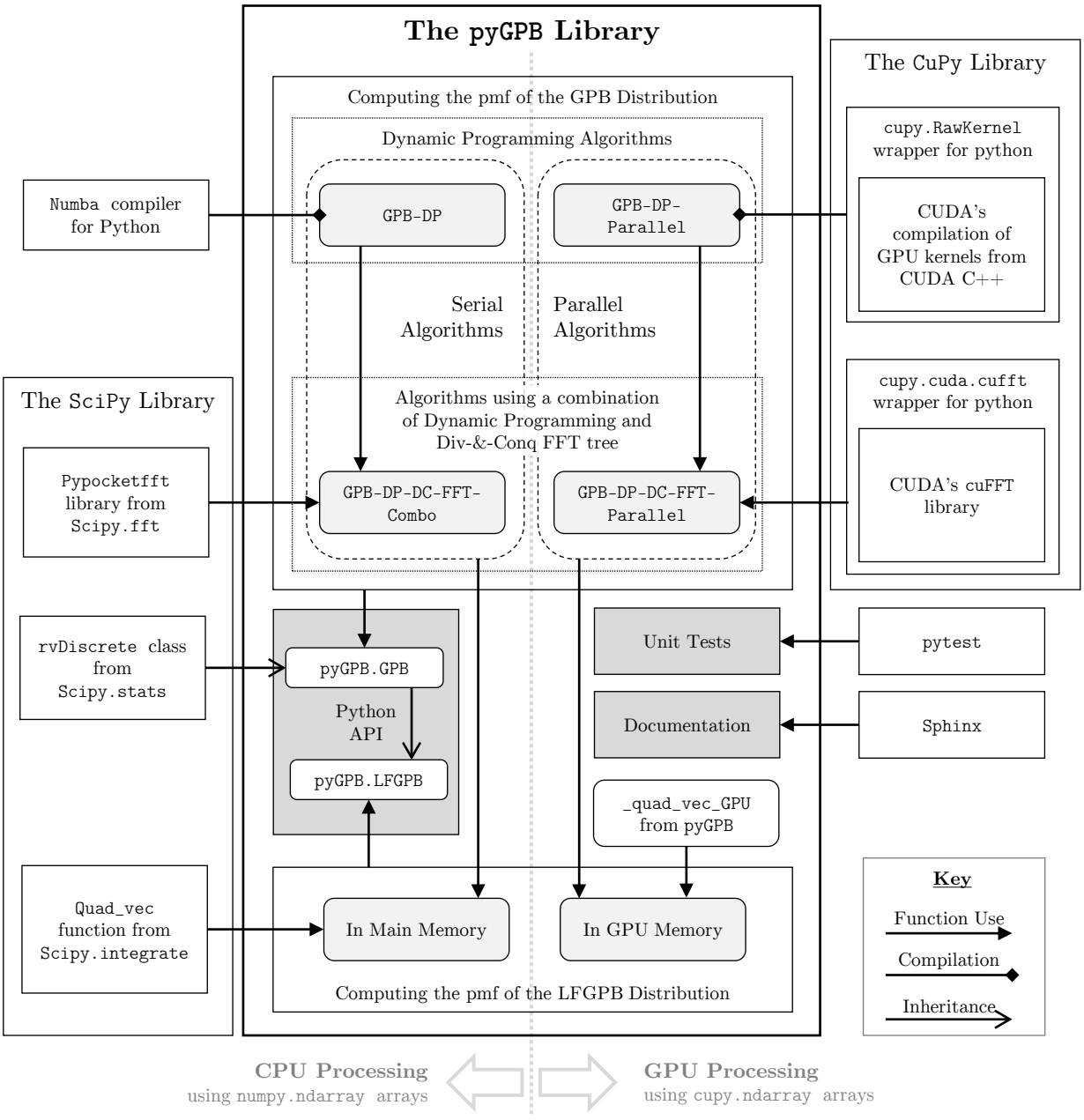

Figure 6.1: Diagram of the key Libraries and tools used to build pyGPB and how they worked together

- Where possible the just-in-time compilation of the `numba` package was used. `Numba` compiles Python functions into fast machine code and works especially well with `numpy` arrays.
- For Fast Fourier Transforms on CPU the functions from `scipy.fft._pocketfft.pypocketfft` were used. These functions are not part of the official `scipy.fft` API, but are ultimately the functions which the official API calls to perform the task after much overhead dealing with type checking etc.. In order to avoid the associated overhead the private `pypocketfft` functions were used directly.

  Unfortunately, these functions are not compilable by `numba` and it proved impossible to find any Python-accessible library which is. Attempts were made to re-implement the Cooley–Tukey[7] algorithm and compile it using `numba` but results were slower than `pypocketfft`.
- The project made extensive use of the CUDA platform. CUDA is "a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs)"[18]. The GPUs available on the Google Colab platform are all CUDA enabled.
- There are several python libraries for working with CUDA-enabled GPUs. Ultimately the `CuPy` package was chosen. `CuPy` is a "NumPy/SciPy-compatible Array Library for GPU-accelerated Computing with Python"[19] allowing for manipulation of arrays in GPU memory using an API mostly indistinguishable from `numpy`. Other options like `PyCUDA` and `numba.cuda` were experimented with but the familiar API of CuPy was the key deciding factor. `CuPy` was also used to compile GPU kernels written in CUDA C++ and to launch them on the GPU.
- For GPU FFTs the CUDA platform already provides its own GPU-optimised implementation of FFTs via the `cuFFT` library for C++. There is a Python API for cuFFT provided by `cupy.cuda.fft`.
- For quadrature on CPU the excellent `scipy.integrate.quad_vec` was used. No equivalent was found for use on arrays in GPU memory, so the project required implementing a GPU-memory version of adaptive Gauss-Kronrod quadrature, see Appendix B.3 for more details.
- Documentation was built using `Sphinx`. See Appendix C.4 for more detail on documentation.

6.3 The `pyGPB` package

Details of the pyGPB package can be found in Appendix C including the package structure, distribution and installation, documentation, and a brief look at the API. For more details consult the documentation in the project repository.

6.4 Unit Testing

The `pyGPB` library comes with a `tests` subfolder containing around 60 unit tests for the public classes, `GPB` and `LFGPB` and their associated methods. The `pytest` library was used as a framework. Tests were run on both Google Colab and a local Windows system, and all tests pass.

Chapter 7

Accuracy Testing

7.1 Testing Framework

7.1.1 Generation of Test Scenarios

To be confident that the algorithms developed in this project were suitable across the parameter space of the GPB and LFGPB distributions it was necessary to generate a large gamut of test scenarios. The full technical detail of how these cases were generated can be found in Appendix D.1.

7.1.2 Choice of Accuracy Measure

The accuracy measure used by [4] and [26] was Total Absolute Error (TAE). Also, [26] used Maximum Absolute Error (MAE). They are defined as below where $\boldsymbol{G}^*$ is the *benchmark* cdf vector and $\boldsymbol{G}$ is the cdf vector output from a putative algorithm.

$$TAE = \sum_{x=0}^{n-1} |G_x^* - G_x| \tag{7.1}$$

and

$$MAE = \max_x [|G_x^* - G_x|] \tag{7.2}$$

16 random test cases were sampled for each point of the parameter space. The mean accuracy across test these test case is shown the tables that follow, plus/minus the standard deviation.

7.1.3 Choice of Benchmark

Clearly, the Naïve approach (i.e. using a probability tree as in Section 2.2) for calculating the pmf is an obvious benchmark for accuracy. However, the exponential-time complexity means it can only be used for small problems.

Expected Accuracy with Double-Precision Floats

To get an understanding of the limits of accuracy for double-precision floating point vectors, two different implementations of the Naïve approach were compared to each other: one using integers as bitflags and one using a stack. These approaches can been seen in Algorithms 6 and 7 in Appendix A.1. A sample of the results can be seen in Table 7.1, with full results available in the `Naive vs Naive.xlsx` file in the project repository. All of these are for $n = 1,000$, but other n were tested.

The columns w_dist and p_dist show the distributions from which the weights and probabilities for each test case were sampled. See Appendix D.1 for more details.

m	w_dist	p_dist	TAE (mean ± std)	MAE (mean ± std)
10	uniform	uniform	$8.1\times10^{-15} \pm 2.7\times10^{-14}$	$2.1\times10^{-17} \pm 5.8\times10^{-17}$
10	uniform	near_0	$1.2\times10^{-13} \pm 1.1\times10^{-13}$	$1.6\times10^{-16} \pm 1.3\times10^{-16}$
10	uniform	near_1	$2.5\times10^{-16} \pm 5.8\times10^{-16}$	$1.2\times10^{-17} \pm 2.9\times10^{-17}$
10	poisson	uniform	$2.4\times10^{-14} \pm 3.9\times10^{-14}$	$4.9\times10^{-17} \pm 6.5\times10^{-17}$
10	poisson	near_0	$1.2\times10^{-13} \pm 1.0\times10^{-13}$	$1.8\times10^{-16} \pm 1.1\times10^{-16}$
10	poisson	near_1	$1.2\times10^{-15} \pm 4.3\times10^{-15}$	$1.9\times10^{-17} \pm 3.5\times10^{-17}$
10	geometric	uniform	$2.2\times10^{-15} \pm 8.5\times10^{-15}$	$7.4\times10^{-18} \pm 2.7\times10^{-17}$
10	geometric	near_0	$8.9\times10^{-14} \pm 1.1\times10^{-13}$	$1.3\times10^{-16} \pm 1.2\times10^{-16}$
10	geometric	near_1	$2.0\times10^{-15} \pm 6.4\times10^{-15}$	$3.1\times10^{-17} \pm 7.3\times10^{-17}$
25	uniform	uniform	$3.2\times10^{-13} \pm 1.2\times10^{-13}$	$9.1\times10^{-16} \pm 1.6\times10^{-16}$
25	uniform	near_0	$1.6\times10^{-13} \pm 1.4\times10^{-13}$	$2.6\times10^{-16} \pm 1.5\times10^{-16}$
25	uniform	near_1	$3.6\times10^{-14} \pm 8.2\times10^{-14}$	$1.9\times10^{-16} \pm 2.5\times10^{-16}$
25	poisson	uniform	$4.6\times10^{-13} \pm 4.6\times10^{-13}$	$1.2\times10^{-15} \pm 8.1\times10^{-16}$
25	poisson	near_0	$1.4\times10^{-13} \pm 9.5\times10^{-14}$	$2.3\times10^{-16} \pm 1.3\times10^{-16}$
25	poisson	near_1	$1.4\times10^{-14} \pm 1.6\times10^{-14}$	$1.3\times10^{-16} \pm 1.0\times10^{-16}$
25	geometric	uniform	$3.4\times10^{-13} \pm 2.0\times10^{-13}$	$9.9\times10^{-16} \pm 4.4\times10^{-16}$
25	geometric	near_0	$1.2\times10^{-12} \pm 1.1\times10^{-12}$	$1.7\times10^{-15} \pm 1.7\times10^{-15}$
25	geometric	near_1	$4.1\times10^{-13} \pm 8.3\times10^{-13}$	$1.4\times10^{-15} \pm 2.6\times10^{-15}$

Table 7.1: Accuracy results when comparing two different implementations of the Naïve approach

The MAEs are generally of the order 10^{-16} agreeing with the errors one would expect to see for IEEE 754 standard double-precision floating point numbers. For $n = 25$ there are around 34 million leaves on the probability tree, which get aggregated up to $n = 1,000$ vector elements, so that on average each element is the sum of 34-thousand probabilities, with the central values being the sum of significantly more, so some imprecision is to be expected.

Benchmark for larger problems

For larger problems (i.e. the set of test scenarios for larger m and n described in Appendix D.1) the Naïve approaches are too slow. Instead, the Dynamic Programming algorithm, i.e. `GPB-DP`, is a good candidate to use as benchmark against the other approaches. To confirm that the `GPB-DP` algorithm is sufficiently accurate it was compared to the *stacked* Naïve approach used above (see Algorithm 7). In both cases the approaches were re-built for quadruple-precision (i.e 16-byte) floating-point numbers to ensure a robust comparison. A summary of those results can be seen in table 7.2. Again, all of these are for $n = 1,000$, but other n were used. Full results can be found in the file `Naive Precise vs DP Precise.xlsx` in the project repository.

From Table 7.2 it seems clear that the implementation of the `GPB-DP` algorithm is accurate, allowing for some floating point imprecision, and so can be used as a benchmark for the other approaches.

7.2 Accuracy Testing Results

A sample of TAE and MAE results for the implementations of the other algorithms compared to `GPB-DP` can be seen in tables 7.3 and 7.3 respectively. These tables suggest that all of these approaches have very good accuracy when absolute error is the key criteria. Full results can be found in the file `DP vs FFT and GPU.xlsx` in the project repository.

Whilst the absolute accuracy is clearly very good, there are some issues with the FFT-based

m	w_dist	p_dist	TAE (mean ± std)	MAE (mean ± std)
10	uniform	uniform	$1.6\times10^{-17} \pm 2.0\times10^{-17}$	$5.4\times10^{-20} \pm 4.3\times10^{-20}$
10	uniform	near_0	$8.9\times10^{-14} \pm 6.1\times10^{-14}$	$9.8\times10^{-17} \pm 6.5\times10^{-17}$
10	uniform	near_1	$1.1\times10^{-15} \pm 3.1\times10^{-15}$	$7.0\times10^{-18} \pm 1.8\times10^{-17}$
10	poisson	uniform	$2.3\times10^{-17} \pm 2.0\times10^{-17}$	$5.2\times10^{-20} \pm 3.2\times10^{-20}$
10	poisson	near_0	$8.9\times10^{-14} \pm 6.2\times10^{-14}$	$9.7\times10^{-17} \pm 6.6\times10^{-17}$
10	poisson	near_1	$1.0\times10^{-15} \pm 2.7\times10^{-15}$	$7.0\times10^{-18} \pm 1.8\times10^{-17}$
10	geometric	uniform	$1.4\times10^{-17} \pm 1.4\times10^{-17}$	$5.0\times10^{-20} \pm 3.5\times10^{-20}$
10	geometric	near_0	$8.9\times10^{-14} \pm 6.0\times10^{-14}$	$9.8\times10^{-17} \pm 6.5\times10^{-17}$
10	geometric	near_1	$1.8\times10^{-15} \pm 5.2\times10^{-15}$	$7.0\times10^{-18} \pm 1.8\times10^{-17}$
25	uniform	uniform	$4.8\times10^{-17} \pm 3.0\times10^{-17}$	$1.4\times10^{-19} \pm 4.6\times10^{-20}$
25	uniform	near_0	$7.3\times10^{-14} \pm 5.5\times10^{-14}$	$7.9\times10^{-17} \pm 6.0\times10^{-17}$
25	uniform	near_1	$4.7\times10^{-16} \pm 1.8\times10^{-15}$	$3.5\times10^{-18} \pm 1.3\times10^{-17}$
25	poisson	uniform	$8.0\times10^{-17} \pm 7.2\times10^{-17}$	$1.9\times10^{-19} \pm 1.2\times10^{-19}$
25	poisson	near_0	$7.3\times10^{-14} \pm 5.6\times10^{-14}$	$7.9\times10^{-17} \pm 6.0\times10^{-17}$
25	poisson	near_1	$4.1\times10^{-16} \pm 1.6\times10^{-15}$	$3.5\times10^{-18} \pm 1.3\times10^{-17}$
25	geometric	uniform	$6.5\times10^{-17} \pm 4.6\times10^{-17}$	$1.9\times10^{-19} \pm 8.6\times10^{-20}$
25	geometric	near_0	$7.3\times10^{-14} \pm 5.5\times10^{-14}$	$7.9\times10^{-17} \pm 6.0\times10^{-17}$
25	geometric	near_1	$2.3\times10^{-16} \pm 8.6\times10^{-16}$	$3.6\times10^{-18} \pm 1.3\times10^{-17}$

Table 7.2: Accuracy Results when comparing 16-byte float Naive and Dynamic Programming approaches

approaches regarding relative accuracy in the tails. For particular parts of the parameter space the probabilities in the tails of the distribution can be much smaller than the level of absolute accuracy leading to very large *relative* errors. These absolute errors also have the potential to lead to negative values, which for probabilities is clearly nonsensical. This is explored in more detail in Appendix D.2.

7.3 Accuracy Testing the LFGPB Distribution

Two approaches were taken for testing the accuracy of the Quadrature approach discussed in Section 5.2. The first is to compare it against the Linear Algebra approach discussed in Appendix B.2. This latter approach has exponential time complexity so can only be used for smaller examples. For larger examples we can look at the moments of the distribution (i.e. mean, variance, skewness and kurtosis) as calculated from the putative pmf, and compare these to the values one would expect from first principals for a sum of random variables.

7.3.1 Testing Against the Linear Algebra Approach

Due to the computational complexity of the Linear Algebra approach it was only possible to use it for small cases, i.e. where $2 \leqslant m \leqslant 10$. For each value of m in that range, 32 test cases were generated with parameters sampled as follows:

- $w_i \sim \mathcal{U}[\![1, 19]\!]$
- $p_i \sim \mathcal{U}(0, 1)$
- $\rho \sim \mathcal{U}(0, 1)$

The test results can be seen in table 7.5, with mean TAE and MAE shown plus/minus the standard deviation. Results seem to be very good, with precision deteriorating slightly as m increases. This could be due either to error in the Quadrature approach or in the Linear Algebra approach. The number of outcomes which are aggregated in the Linear Algebra approach increases exponentially,

m	n	p_dist	DC FFT TAE	DP GPU TAE	DC FFT GPU TAE
10	1,000	near_0	$4.3\times10^{-13} \pm 2.6\times10^{-13}$	$4.0\times10^{-13} \pm 2.2\times10^{-13}$	$4.7\times10^{-13} \pm 2.2\times10^{-13}$
10	1,000	near_1	$1.4\times10^{-13} \pm 6.1\times10^{-14}$	$1.7\times10^{-14} \pm 1.0\times10^{-14}$	$1.4\times10^{-13} \pm 7.6\times10^{-14}$
10	1,000	uniform	$1.5\times10^{-13} \pm 9.1\times10^{-14}$	$2.1\times10^{-13} \pm 9.1\times10^{-14}$	$2.4\times10^{-13} \pm 9.4\times10^{-14}$
10	10,000	near_0	$3.8\times10^{-12} \pm 2.5\times10^{-12}$	$4.2\times10^{-12} \pm 2.3\times10^{-12}$	$4.5\times10^{-12} \pm 2.4\times10^{-12}$
10	10,000	near_1	$1.7\times10^{-12} \pm 6.1\times10^{-13}$	$2.1\times10^{-13} \pm 1.9\times10^{-13}$	$2.0\times10^{-12} \pm 9.4\times10^{-13}$
10	10,000	uniform	$4.1\times10^{-12} \pm 1.8\times10^{-12}$	$2.7\times10^{-12} \pm 2.4\times10^{-12}$	$3.1\times10^{-12} \pm 2.5\times10^{-12}$
10	100,000	near_0	$2.5\times10^{-11} \pm 2.2\times10^{-11}$	$5.1\times10^{-11} \pm 2.1\times10^{-11}$	$4.8\times10^{-11} \pm 2.0\times10^{-11}$
10	100,000	near_1	$2.1\times10^{-11} \pm 9.8\times10^{-12}$	$2.3\times10^{-12} \pm 2.4\times10^{-12}$	$1.9\times10^{-11} \pm 6.0\times10^{-12}$
10	100,000	uniform	$4.1\times10^{-11} \pm 2.1\times10^{-11}$	$3.7\times10^{-11} \pm 2.9\times10^{-11}$	$4.0\times10^{-11} \pm 2.9\times10^{-11}$
100	1,000	near_0	$1.8\times10^{-12} \pm 8.3\times10^{-13}$	$4.7\times10^{-13} \pm 3.5\times10^{-13}$	$4.8\times10^{-13} \pm 3.6\times10^{-13}$
100	1,000	near_1	$3.1\times10^{-13} \pm 7.7\times10^{-14}$	$2.1\times10^{-14} \pm 1.2\times10^{-14}$	$2.4\times10^{-13} \pm 8.3\times10^{-14}$
100	1,000	uniform	$1.4\times10^{-12} \pm 3.4\times10^{-13}$	$2.3\times10^{-13} \pm 1.6\times10^{-13}$	$2.5\times10^{-13} \pm 1.4\times10^{-13}$
100	10,000	near_0	$1.1\times10^{-11} \pm 7.4\times10^{-12}$	$1.7\times10^{-11} \pm 1.2\times10^{-11}$	$1.5\times10^{-11} \pm 1.1\times10^{-11}$
100	10,000	near_1	$2.9\times10^{-12} \pm 7.4\times10^{-13}$	$6.6\times10^{-13} \pm 3.4\times10^{-13}$	$1.8\times10^{-12} \pm 4.2\times10^{-13}$
100	10,000	uniform	$4.7\times10^{-12} \pm 4.9\times10^{-12}$	$6.7\times10^{-12} \pm 3.0\times10^{-12}$	$6.7\times10^{-12} \pm 2.6\times10^{-12}$
100	100,000	near_0	$7.5\times10^{-11} \pm 8.4\times10^{-11}$	$1.4\times10^{-09} \pm 3.2\times10^{-10}$	$1.4\times10^{-09} \pm 3.2\times10^{-10}$
100	100,000	near_1	$3.0\times10^{-11} \pm 1.4\times10^{-11}$	$1.8\times10^{-11} \pm 8.8\times10^{-12}$	$4.0\times10^{-11} \pm 1.3\times10^{-11}$
100	100,000	uniform	$5.5\times10^{-11} \pm 2.8\times10^{-11}$	$4.5\times10^{-10} \pm 1.6\times10^{-10}$	$4.5\times10^{-10} \pm 1.6\times10^{-10}$
1,000	10,000	near_0	$1.7\times10^{-10} \pm 2.4\times10^{-11}$	$9.7\times10^{-12} \pm 6.3\times10^{-12}$	$1.2\times10^{-11} \pm 5.9\times10^{-12}$
1,000	10,000	near_1	$2.1\times10^{-11} \pm 4.4\times10^{-12}$	$5.4\times10^{-13} \pm 3.6\times10^{-13}$	$3.4\times10^{-12} \pm 1.2\times10^{-12}$
1,000	10,000	uniform	$1.0\times10^{-10} \pm 1.4\times10^{-11}$	$3.6\times10^{-12} \pm 2.6\times10^{-12}$	$8.0\times10^{-12} \pm 3.9\times10^{-12}$
1,000	100,000	near_0	$7.3\times10^{-10} \pm 4.2\times10^{-10}$	$4.8\times10^{-10} \pm 2.5\times10^{-10}$	$3.4\times10^{-10} \pm 2.7\times10^{-10}$
1,000	100,000	near_1	$1.4\times10^{-10} \pm 4.3\times10^{-11}$	$1.7\times10^{-11} \pm 8.2\times10^{-12}$	$4.4\times10^{-11} \pm 1.0\times10^{-11}$
1,000	100,000	uniform	$6.0\times10^{-10} \pm 1.6\times10^{-10}$	$3.2\times10^{-10} \pm 1.0\times10^{-10}$	$2.8\times10^{-10} \pm 1.0\times10^{-10}$

Table 7.3: TAE Accuracy Results of DC-FFT, and GPU approaches vs DP Approach

m	n	p_dist	DC FFT MAE	DP GPU MAE	DC FFT GPU MAE
10	1,000	near_0	$7.8\times10^{-16} \pm 3.5\times10^{-16}$	$9.5\times10^{-16} \pm 4.4\times10^{-16}$	$1.1\times10^{-15} \pm 4.2\times10^{-16}$
10	1,000	near_1	$7.4\times10^{-16} \pm 3.0\times10^{-16}$	$1.6\times10^{-16} \pm 9.8\times10^{-17}$	$3.2\times10^{-16} \pm 9.7\times10^{-17}$
10	1,000	uniform	$4.2\times10^{-16} \pm 2.3\times10^{-16}$	$7.7\times10^{-16} \pm 2.5\times10^{-16}$	$8.3\times10^{-16} \pm 2.5\times10^{-16}$
10	10,000	near_0	$6.4\times10^{-16} \pm 3.1\times10^{-16}$	$1.1\times10^{-15} \pm 3.2\times10^{-16}$	$1.1\times10^{-15} \pm 3.9\times10^{-16}$
10	10,000	near_1	$9.5\times10^{-16} \pm 3.1\times10^{-16}$	$1.8\times10^{-16} \pm 9.7\times10^{-17}$	$4.4\times10^{-16} \pm 1.4\times10^{-16}$
10	10,000	uniform	$9.1\times10^{-16} \pm 3.3\times10^{-16}$	$8.8\times10^{-16} \pm 4.6\times10^{-16}$	$9.0\times10^{-16} \pm 4.6\times10^{-16}$
10	100,000	near_0	$3.9\times10^{-16} \pm 2.6\times10^{-16}$	$1.3\times10^{-15} \pm 4.1\times10^{-16}$	$1.3\times10^{-15} \pm 3.8\times10^{-16}$
10	100,000	near_1	$9.4\times10^{-16} \pm 3.7\times10^{-16}$	$2.0\times10^{-16} \pm 1.3\times10^{-16}$	$5.4\times10^{-16} \pm 1.9\times10^{-16}$
10	100,000	uniform	$9.3\times10^{-16} \pm 4.5\times10^{-16}$	$1.1\times10^{-15} \pm 6.2\times10^{-16}$	$1.1\times10^{-15} \pm 6.1\times10^{-16}$
100	1,000	near_0	$2.2\times10^{-15} \pm 8.8\times10^{-16}$	$9.0\times10^{-16} \pm 4.0\times10^{-16}$	$9.4\times10^{-16} \pm 3.5\times10^{-16}$
100	1,000	near_1	$2.8\times10^{-15} \pm 8.0\times10^{-16}$	$4.0\times10^{-16} \pm 1.9\times10^{-16}$	$9.8\times10^{-16} \pm 2.7\times10^{-16}$
100	1,000	uniform	$2.8\times10^{-15} \pm 6.9\times10^{-16}$	$8.5\times10^{-16} \pm 3.7\times10^{-16}$	$8.5\times10^{-16} \pm 2.9\times10^{-16}$
100	10,000	near_0	$1.4\times10^{-15} \pm 8.6\times10^{-16}$	$2.8\times10^{-15} \pm 1.5\times10^{-15}$	$2.7\times10^{-15} \pm 1.3\times10^{-15}$
100	10,000	near_1	$1.9\times10^{-15} \pm 7.6\times10^{-16}$	$1.5\times10^{-15} \pm 5.2\times10^{-16}$	$1.6\times10^{-15} \pm 4.7\times10^{-16}$
100	10,000	uniform	$1.0\times10^{-15} \pm 9.7\times10^{-16}$	$2.8\times10^{-15} \pm 8.8\times10^{-16}$	$2.8\times10^{-15} \pm 7.7\times10^{-16}$
100	100,000	near_0	$9.9\times10^{-16} \pm 9.4\times10^{-16}$	$2.2\times10^{-14} \pm 4.8\times10^{-15}$	$2.2\times10^{-14} \pm 4.7\times10^{-15}$
100	100,000	near_1	$2.3\times10^{-15} \pm 1.3\times10^{-15}$	$3.8\times10^{-15} \pm 1.5\times10^{-15}$	$4.3\times10^{-15} \pm 1.6\times10^{-15}$
100	100,000	uniform	$1.1\times10^{-15} \pm 5.9\times10^{-16}$	$2.1\times10^{-14} \pm 5.4\times10^{-15}$	$2.1\times10^{-14} \pm 5.5\times10^{-15}$
1,000	10,000	near_0	$2.0\times10^{-14} \pm 2.8\times10^{-15}$	$1.4\times10^{-15} \pm 6.2\times10^{-16}$	$2.7\times10^{-15} \pm 1.1\times10^{-15}$
1,000	10,000	near_1	$2.2\times10^{-14} \pm 4.5\times10^{-15}$	$1.1\times10^{-15} \pm 5.5\times10^{-16}$	$2.9\times10^{-15} \pm 1.2\times10^{-15}$
1,000	10,000	uniform	$2.1\times10^{-14} \pm 2.8\times10^{-15}$	$1.2\times10^{-15} \pm 4.3\times10^{-16}$	$2.8\times10^{-15} \pm 7.6\times10^{-16}$
1,000	100,000	near_0	$8.1\times10^{-15} \pm 4.6\times10^{-15}$	$7.0\times10^{-15} \pm 1.9\times10^{-15}$	$6.9\times10^{-15} \pm 2.2\times10^{-15}$
1,000	100,000	near_1	$1.5\times10^{-14} \pm 4.8\times10^{-15}$	$4.8\times10^{-15} \pm 1.6\times10^{-15}$	$5.1\times10^{-15} \pm 1.6\times10^{-15}$
1,000	100,000	uniform	$1.2\times10^{-14} \pm 3.3\times10^{-15}$	$8.5\times10^{-15} \pm 2.2\times10^{-15}$	$7.5\times10^{-15} \pm 2.3\times10^{-15}$

Table 7.4: MAE Accuracy Results of DC-FFT, and GPU approaches vs DP Approach

so it could be that the error here is from accumulated floating-point errors in the Linear Algebra approach, rather than an issue with the Quadrature approach. Either way, this shows that the solution has been implemented accurately with *potentially* less precision for higher m.

m	MAE	TAE
2	$6.2\times10^{-14} \pm 7.4\times10^{-14}$	$9.2\times10^{-13} \pm 8.2\times10^{-13}$
3	$4.5\times10^{-13} \pm 1.6\times10^{-12}$	$2.1\times10^{-12} \pm 4.8\times10^{-12}$
4	$7.5\times10^{-14} \pm 1.3\times10^{-13}$	$1.0\times10^{-12} \pm 1.2\times10^{-12}$
5	$1.2\times10^{-13} \pm 1.7\times10^{-13}$	$1.3\times10^{-12} \pm 1.0\times10^{-12}$
6	$6.2\times10^{-13} \pm 2.7\times10^{-12}$	$5.6\times10^{-12} \pm 2.1\times10^{-11}$
7	$3.4\times10^{-09} \pm 1.9\times10^{-08}$	$7.4\times10^{-08} \pm 4.1\times10^{-07}$
8	$4.3\times10^{-11} \pm 2.4\times10^{-10}$	$7.9\times10^{-10} \pm 4.4\times10^{-09}$
9	$4.2\times10^{-10} \pm 2.3\times10^{-09}$	$6.7\times10^{-09} \pm 3.7\times10^{-08}$
10	$1.7\times10^{-08} \pm 9.4\times10^{-08}$	$4.9\times10^{-07} \pm 2.7\times10^{-06}$

Table 7.5: Results of testing the implemented LFGPB approach against the Linear Algebra approach

7.3.2 Testing Using Moments

The accuracy and precision of `pyGPB`'s implementation of the LFGPB distribution has been confirmed for smaller m. It seems reasonable to assume that the implementation will still be correct for larger m, however, precision may be an issue. To check larger values of m, the moments of the distribution are used. The moments can be calculated in two ways: from the pmf vector computed by the putative algorithm, and directly from the parameters using the methods discussed in Appendix B.1.3. A comparison of the two values will provide confidence around how precision deteriorates (or not) for larger m.

Previously, MAE and TAE have been used as measures of accuracy. Such measures of absolute error are reasonable for testing accuracy of computed probabilities, which all lie in $[0, 1]$, however moments are not so restricted. Consequently, the *relative* error was deemed more appropriate for this test. See Equation D.2 for a definition of relative error.

Although one can calculate moments for much higher m than the Linear Algebra technique could be applied, there is still some limitation caused by the time complexity of computing the higher-order moments from the parameters, especially skewness and kurtosis. These higher moments were tested where feasible, but the mean and variance were relied on for the largest values of m.

In each part of the parameter space there were eight test cases generated using the same distributions as for the Linear Algebra testing (see Section 7.3.1), with values for m spaced logarithmically up to $1,000$. The results of mean relative error plus/minus standard deviation can be seen in table 7.6. These results suggest that `pyGPB` has accurately implemented a correct algorithm for computing the pmf of the LFGPB distribution.

m	Mean	Variance	Skewness	Kurtosis
4	$4.9\times10^{-14} \pm 8.0\times10^{-14}$	$3.4\times10^{-13} \pm 2.6\times10^{-13}$	$7.7\times10^{-12} \pm 1.2\times10^{-12}$	$1.2\times10^{-13} \pm 6.1\times10^{-13}$
5	$4.1\times10^{-14} \pm 4.6\times10^{-14}$	$3.7\times10^{-13} \pm 3.3\times10^{-13}$	$9.0\times10^{-12} \pm 1.0\times10^{-12}$	$1.0\times10^{-13} \pm 4.5\times10^{-13}$
6	$8.4\times10^{-13} \pm 1.1\times10^{-13}$	$5.5\times10^{-13} \pm 4.3\times10^{-13}$	$6.2\times10^{-12} \pm 5.1\times10^{-12}$	$5.1\times10^{-13} \pm 4.0\times10^{-13}$
7	$1.7\times10^{-14} \pm 1.9\times10^{-14}$	$1.2\times10^{-13} \pm 1.1\times10^{-13}$	$1.9\times10^{-13} \pm 1.6\times10^{-13}$	$1.6\times10^{-13} \pm 1.2\times10^{-13}$
8	$1.7\times10^{-14} \pm 1.8\times10^{-14}$	$1.9\times10^{-13} \pm 1.2\times10^{-13}$	$1.1\times10^{-12} \pm 2.0\times10^{-12}$	$2.0\times10^{-13} \pm 3.6\times10^{-13}$
9	$3.6\times10^{-14} \pm 7.2\times10^{-14}$	$4.8\times10^{-13} \pm 7.8\times10^{-13}$	$5.6\times10^{-12} \pm 8.5\times10^{-12}$	$8.5\times10^{-13} \pm 7.1\times10^{-13}$
10	$7.5\times10^{-14} \pm 1.1\times10^{-14}$	$1.3\times10^{-14} \pm 7.2\times10^{-14}$	$5.8\times10^{-13} \pm 9.5\times10^{-13}$	$9.5\times10^{-13} \pm 1.8\times10^{-13}$
12	$1.4\times10^{-13} \pm 3.7\times10^{-13}$	$8.1\times10^{-12} \pm 1.9\times10^{-12}$	$2.9\times10^{-11} \pm 7.0\times10^{-11}$	$7.0\times10^{-13} \pm 6.6\times10^{-13}$
14	$3.7\times10^{-14} \pm 6.6\times10^{-14}$	$1.1\times10^{-12} \pm 1.9\times10^{-12}$	$2.3\times10^{-11} \pm 4.3\times10^{-11}$	$4.3\times10^{-12} \pm 3.3\times10^{-12}$
16	$1.2\times10^{-15} \pm 7.1\times10^{-15}$	$7.2\times10^{-13} \pm 9.4\times10^{-13}$	$1.1\times10^{-11} \pm 1.4\times10^{-11}$	$1.4\times10^{-10} \pm 1.0\times10^{-10}$
18	$9.2\times10^{-15} \pm 8.1\times10^{-15}$	$2.8\times10^{-13} \pm 2.6\times10^{-13}$	$7.3\times10^{-10} \pm 1.8\times10^{-10}$	$1.8\times10^{-10} \pm 3.9\times10^{-10}$
21	$7.5\times10^{-15} \pm 6.1\times10^{-15}$	$1.0\times10^{-14} \pm 6.3\times10^{-14}$	$6.3\times10^{-13} \pm 6.3\times10^{-13}$	$6.3\times10^{-12} \pm 7.6\times10^{-12}$
25	$3.9\times10^{-15} \pm 4.3\times10^{-15}$	$7.4\times10^{-14} \pm 1.6\times10^{-14}$	$9.3\times10^{-12} \pm 1.1\times10^{-12}$	$1.1\times10^{-10} \pm 5.8\times10^{-10}$
29	$4.8\times10^{-14} \pm 9.8\times10^{-14}$	$3.3\times10^{-12} \pm 8.3\times10^{-12}$	$6.9\times10^{-05} \pm 1.8\times10^{-05}$	$1.8\times10^{-07} \pm 1.7\times10^{-07}$
33	$4.9\times10^{-13} \pm 1.2\times10^{-13}$	$4.7\times10^{-12} \pm 1.0\times10^{-12}$	$5.2\times10^{-10} \pm 1.3\times10^{-10}$	
38	$4.2\times10^{-15} \pm 5.0\times10^{-15}$	$2.2\times10^{-13} \pm 4.2\times10^{-13}$	$4.2\times10^{-12} \pm 5.9\times10^{-12}$	
44	$1.1\times10^{-15} \pm 1.3\times10^{-15}$	$1.1\times10^{-13} \pm 1.2\times10^{-13}$	$1.2\times10^{-06} \pm 2.9\times10^{-06}$	
51	$1.3\times10^{-15} \pm 1.8\times10^{-15}$	$2.5\times10^{-09} \pm 6.6\times10^{-09}$	$1.4\times10^{-07} \pm 3.2\times10^{-07}$	
58	$3.9\times10^{-14} \pm 1.0\times10^{-13}$	$4.7\times10^{-11} \pm 1.2\times10^{-11}$	$1.6\times10^{-08} \pm 2.8\times10^{-08}$	
67	$5.1\times10^{-13} \pm 1.1\times10^{-13}$	$1.8\times10^{-10} \pm 4.7\times10^{-10}$		
78	$9.7\times10^{-14} \pm 2.4\times10^{-14}$	$3.8\times10^{-12} \pm 9.9\times10^{-12}$		
90	$7.2\times10^{-13} \pm 1.8\times10^{-13}$	$5.8\times10^{-08} \pm 1.5\times10^{-08}$		
103	$2.3\times10^{-15} \pm 3.7\times10^{-15}$	$1.1\times10^{-11} \pm 2.9\times10^{-11}$		
119	$4.8\times10^{-15} \pm 7.7\times10^{-15}$	$2.2\times10^{-09} \pm 5.8\times10^{-09}$		
137	$3.0\times10^{-16} \pm 3.6\times10^{-16}$	$2.3\times10^{-14} \pm 4.6\times10^{-14}$		
158	$1.1\times10^{-15} \pm 2.2\times10^{-15}$	$2.3\times10^{-10} \pm 5.8\times10^{-10}$		
182	$2.0\times10^{-14} \pm 2.9\times10^{-14}$	$1.8\times10^{-08} \pm 4.7\times10^{-08}$		
210	$2.1\times10^{-15} \pm 3.0\times10^{-15}$	$1.6\times10^{-13} \pm 2.2\times10^{-13}$		
242	$3.8\times10^{-14} \pm 9.3\times10^{-14}$	$1.1\times10^{-09} \pm 3.0\times10^{-09}$		
279	$4.9\times10^{-13} \pm 1.2\times10^{-13}$	$7.0\times10^{-11} \pm 7.8\times10^{-11}$		
322	$3.2\times10^{-16} \pm 3.5\times10^{-16}$	$5.9\times10^{-11} \pm 9.8\times10^{-11}$		
371	$2.0\times10^{-14} \pm 3.1\times10^{-14}$	$2.2\times10^{-08} \pm 4.1\times10^{-08}$		
427	$3.6\times10^{-14} \pm 8.9\times10^{-14}$	$4.2\times10^{-09} \pm 7.3\times10^{-09}$		
492	$8.4\times10^{-13} \pm 2.1\times10^{-13}$	$1.1\times10^{-08} \pm 1.8\times10^{-08}$		
567	$5.6\times10^{-14} \pm 8.8\times10^{-14}$	$2.1\times10^{-08} \pm 2.2\times10^{-08}$		
653	$4.3\times10^{-14} \pm 9.1\times10^{-14}$	$8.8\times10^{-08} \pm 1.1\times10^{-08}$		
753	$4.8\times10^{-16} \pm 7.6\times10^{-16}$	$1.2\times10^{-10} \pm 2.8\times10^{-10}$		
867	$9.5\times10^{-13} \pm 1.8\times10^{-13}$	$8.2\times10^{-08} \pm 1.3\times10^{-08}$		
1000	$2.6\times10^{-14} \pm 6.1\times10^{-14}$	$5.3\times10^{-09} \pm 9.4\times10^{-09}$		

Table 7.6: Results of testing the LFGPB implementation using moments. Values are relative errors

Chapter 8

Performance Results

This project set out to improve on the speed with which one could compute the pmf vector of a GPB random variable. The previous section on testing showed that these implementations are accurate. Here they are benchmarked for speed.

8.1 Benchmarking Methodology

8.1.1 Test Cases

It was desirable to test performance against a range of possible m and n values; these are the variables used in the expressions for the computational complexity of all of the algorithms discussed. To this end, the same test scenarios that were used for *accuracy* benchmarking are here used again for *speed* benchmarking. See Appendix D.1 for their definition. To reduce the amount of time needed to benchmark all cases, the scope was narrowed slightly via the following concessions:

- Speed was only benchmarked for weights and probabilities taken from uniform distributions.
- For each point in the parameter space there were 16 examples in the data set, however for the larger examples and slower algorithms this would have taken an exceedingly long time to process. Where necessary, the number of cases was reduced, but only where the total runtime for each case would take longer than 1 second.
- For the Characteristic Function algorithm, `GBP-CF`, only values of n which were a power of 10 were included.

8.1.2 Speed Measurement

Care must be taken to benchmark algorithms which make use of a GPU. To quote [20]

> Because GPU executions run asynchronously with respect to CPU executions, a common pitfall in GPU programming is to mistakenly measure the elapsed time using CPU timing utilities (such as `time.perf_counter()` from the Python Standard Library or the `%timeit` magic from IPython), which have no knowledge in the GPU runtime.

The author fell into this pitfall when presenting preliminary results in the project proposal document [23]. To avoid it here, the `cupyx.time.repeat` tool from the `cupy` library was used: it accurately measures both elapsed CPU and GPU time.

8.1.3 Hardware Specification

Benchmarking was done in the Cloud using Google Colab Pro. An detailed hardware specification can be found in Appendix D.3.

8.2 Scope of Benchmarking

Six algorithms were benchmarked for speed. The first was the Characteristic Function approach presented by Zhang et al.[26], the other five are those presented in this work. A summary can be found in table 8.1 along with links to the algorithm's pseudocode in this report.

Short Name	Algorithm	Function name in `pyGPB`
CF	GPB-CF	`_GPB_CF`
DP	GPB-DP	`_GPB_DP`
DC-FFT	GPB-DC-FFT	`_GPB_DC_FFT`
Combo	GPB-DP-DC-FFT-Combo	`_GPB_DP_DC_FFT_Combo`
DP-GPU	GPB-DP-Parallel	`_GPB_DP_CUDA`
Combo-GPU	GPB-DP-DC-FFT-Parallel	`_GPB_DP_DC_FFT_CUDA`

Table 8.1: Summary of algorithms that were speed benchmarked

8.3 Algorithm Speeds

The results of speed benchmarking can be seen in Table 8.2. There is no algorithm which is universally fastest. Generally, CPU methods are faster for smaller problems, reflecting the overhead involved in communication between device (CPU and main memory) and host (GPU and on-board GPU memory). For larger problems the GPU methodologies show their worth. Full results can be found in the file `Speed Results.xlsx` in the project repository.

n	m	CF	DP	DC-FFT	Combo	DP-GPU	Combo-GPU
1,000	10	0.40	0.01	0.13	0.01	0.22	2.34
1,000	100	2.96	0.07	0.82	0.07	0.39	5.55
10,000	10	3.56	0.05	0.93	0.05	0.23	2.44
10,000	100	30.31	0.43	2.29	0.43	0.40	6.06
10,000	1,000	289.50	6.33	9.22	1.59	2.42	9.83
100,000	10	40.08	0.67	13.64	0.67	0.23	2.46
100,000	100	314.63	5.90	22.49	5.92	0.47	5.93
100,000	1,000	3,099.34	76.22	35.64	16.56	3.31	10.09
100,000	10,000	30,901.27	953.72	105.35	28.68	31.29	15.20
1,000,000	10	476.05	9.52	188.12	9.89	0.33	2.49
1,000,000	100	3,290.42	68.86	330.56	68.86	1.37	6.27
1,000,000	1,000	30,458.90	845.35	437.60	238.47	11.76	10.66
1,000,000	10,000	321,074.38	9,865.31	571.66	363.18	113.35	16.28
10,000,000	10	5,741.84	141.70	3,173.94	147.79	2.04	13.57
10,000,000	100	36,649.94	1,103.60	4,625.54	1,106.14	13.98	24.98
10,000,000	1,000	304,491.85	11,631.37	6,066.36	3,245.84	127.16	51.71
10,000,000	10,000	DNF	128,686.38	6,955.52	5,094.16	1,276.21	114.99

Table 8.2: Speed Benchmarking Results for select values of n and m in milliseconds

Charts 8.1 and 8.2 show the performance of the algorithms as m varies for $n = 10,000$ and $n = 10,000,000$ respectively. The slope of a line in a log-log plot is indicative of the order of the underlying polynomial being plotted, e.g. a quadratic function will have gradient of two. One would

therefore expect the $O(mn)$ approaches to have gradients around unity, and the $O(n \log n \log m)$ approaches to be much flatter, which is indeed what is seen.

The charts also show how the time-complexity benefit of FFT approaches, and the throughput benefit of GPU approaches, does not mean they dominate the entire parameter space. For smaller n the CPU methods are faster, and for smaller m the Dynamic Programming methods are faster.

Chart 8.3 shows how the three fastest algorithms (Combo on the CPU and the GPU approaches) vary across the entire parameter space. At different parts of the space each algorithm has dominance. Chart 8.4 shows which algorithm is dominant across the parameter space. The exact boundaries will depend on system hardware, but these results perhaps suggest a general pattern of CPU-methods being fastest for smaller n and Dynamic Programming methods being faster for smaller m. For overall best performance, therefore, a combination of all approaches must be used depending on the point in the parameter space.

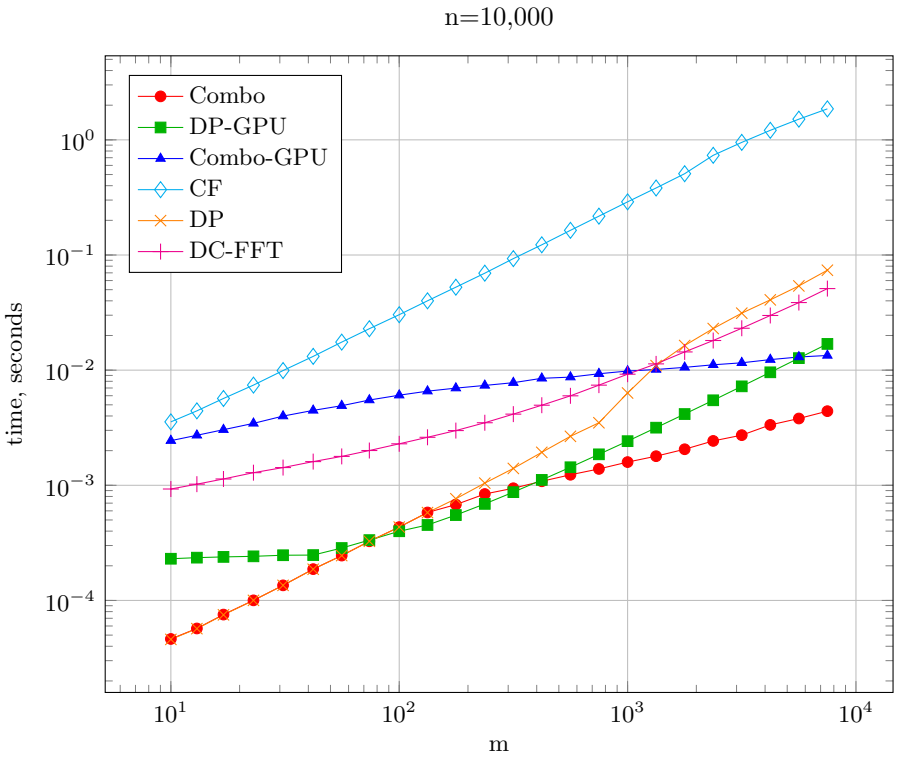

Figure 8.1: Algorithm speed comparison for $n = 10,000$

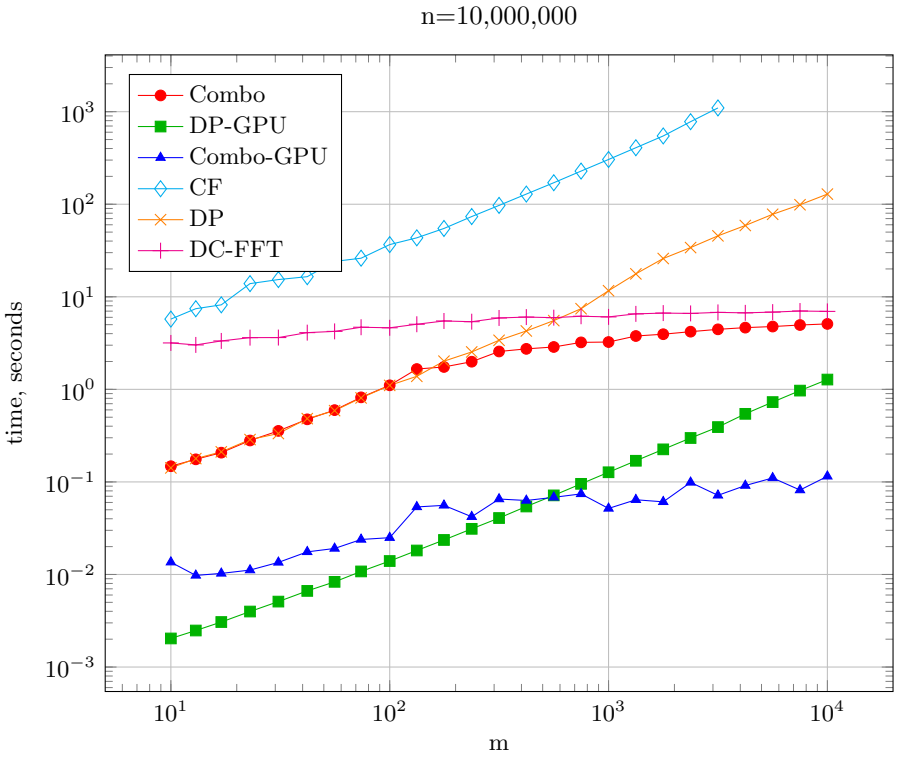

Figure 8.2: Algorithm speed comparison for $n = 10,000,000$

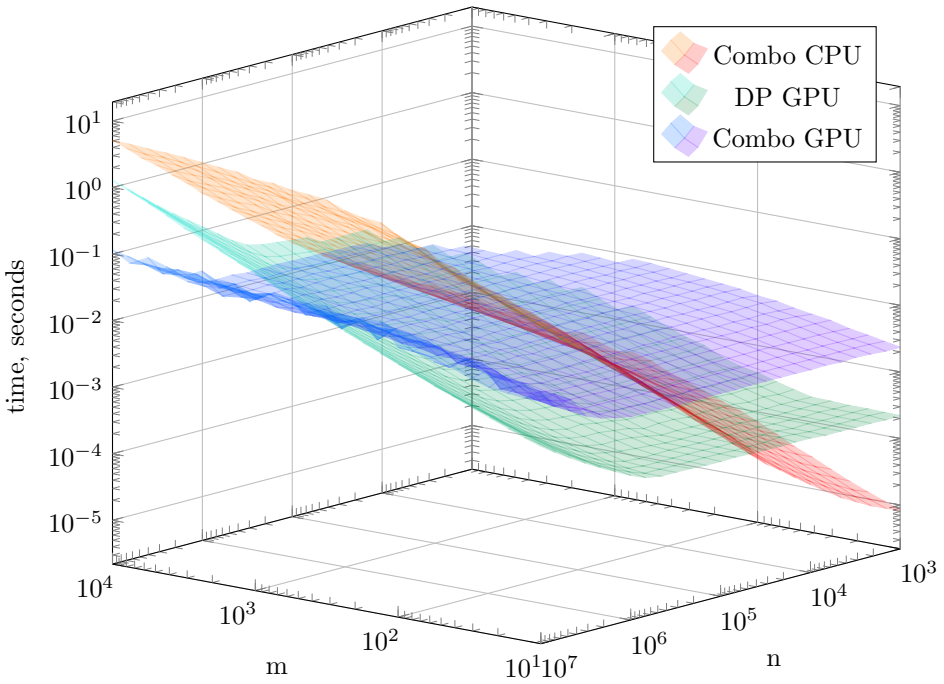

Figure 8.3: Speed of Algorithms across the $n \times m$ parameter space

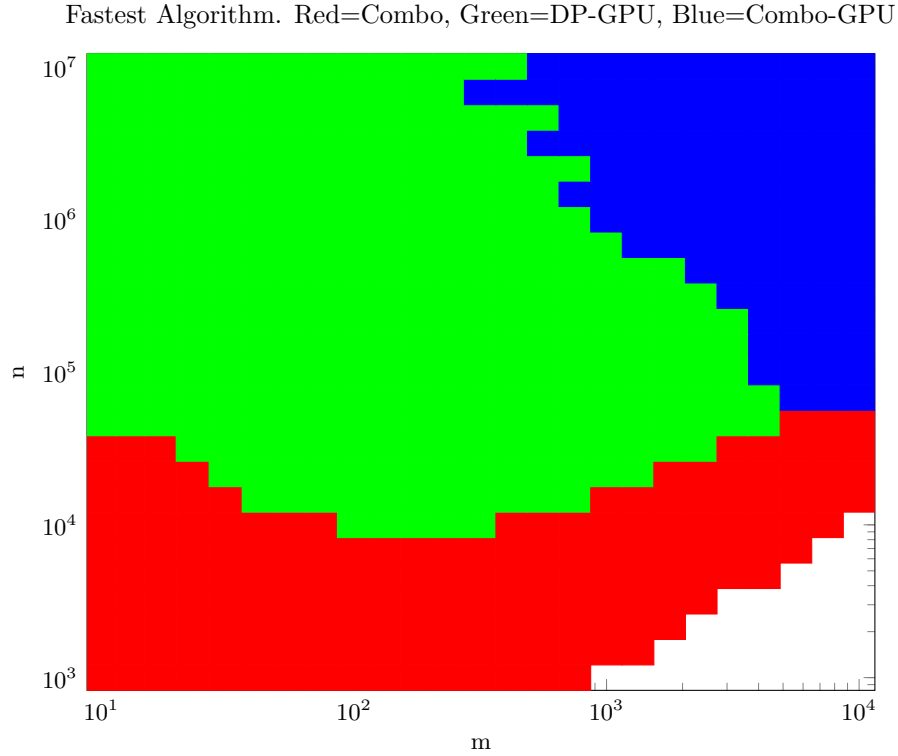

Figure 8.4: Domain of m and n where each algorithm is fastest

Chapter 9

Example Application

The tools developed above have been implemented into the pyGPB library for general use. Details of the API, documentation etc. can be found in Appendix C. This chapter demonstrates how these tools might be applied on an example heterogeneous credit portfolio.

9.1 Case Description

Lenders are, understandably, reticent to publicly share details of their current loan portfolios. Instead we will use an example portfolio of government bonds based on publicly-available market- and economic data.

Government bonds are a form of debt issued by governments to raise funds. In substance the purchaser of a bond is lending money to the government that issued it. At an agreed "maturity" date the bonds "redeem" and the issuer returns the funds to the bondholder. During the interim there is interest paid.

Consider an investor with a portfolio of government bonds. They wish to understand the following about the risk that they bare regarding sovereign default, i.e. events of governments not honouring their commitment to repay their debts:

- What is the distribution of potential losses in the next 5 years?
- How is that distribution affected if we include some correlation between sovereign default events?
- What is the risk associated with "tail events", i.e. the worst end of the probability distribution?
- As the portfolio changes over time, can the aggregate change in risk be explained by changes in the individual components?

An example portfolio is built below based on real-world data and answers to the above questions are sought.

9.2 Data Gathering

9.2.1 Default Probabilities

The limiting factor in specifying an example portfolio is availability of data regarding probabilities of default. Lenders will have proprietary models for estimating such probabilities and they are

not made publicly available. Credit Rating agencies, such as S&P and Moody's, do publish credit ratings for sovereigns. However, the agencies are not willing to publicly associate these ratings with a numerical probability of default.

Instead, we can turn to the Credit Default Swap (CDS) market. CDS are contracts somewhat akin to credit insurance: in return for a fee the "protection seller" agrees to pay a certain amount to the "protection buyer" if some particular "reference entity" (in this case a government) defaults on their debt. Given the current market price for a CDS contract it is possible to imply a default probability.

We took current sovereign CDS prices from the website http://www.worldgovernmentbonds.com/[25], which appears to be the only publicly available source. There were 28 governments included in this data source, defining the scope of our example portfolio. Prices were only available for a 5Y term (i.e. insurance lasting for five years), so this determines our use of five years as the example period over which to understand our risk. A screenshot of this data source can be see in figure 9.1. Usefully, the change in the price over the last six-months is available, giving us "before and after" values to demonstrate how one might explain changes in the portfolio risk over time.

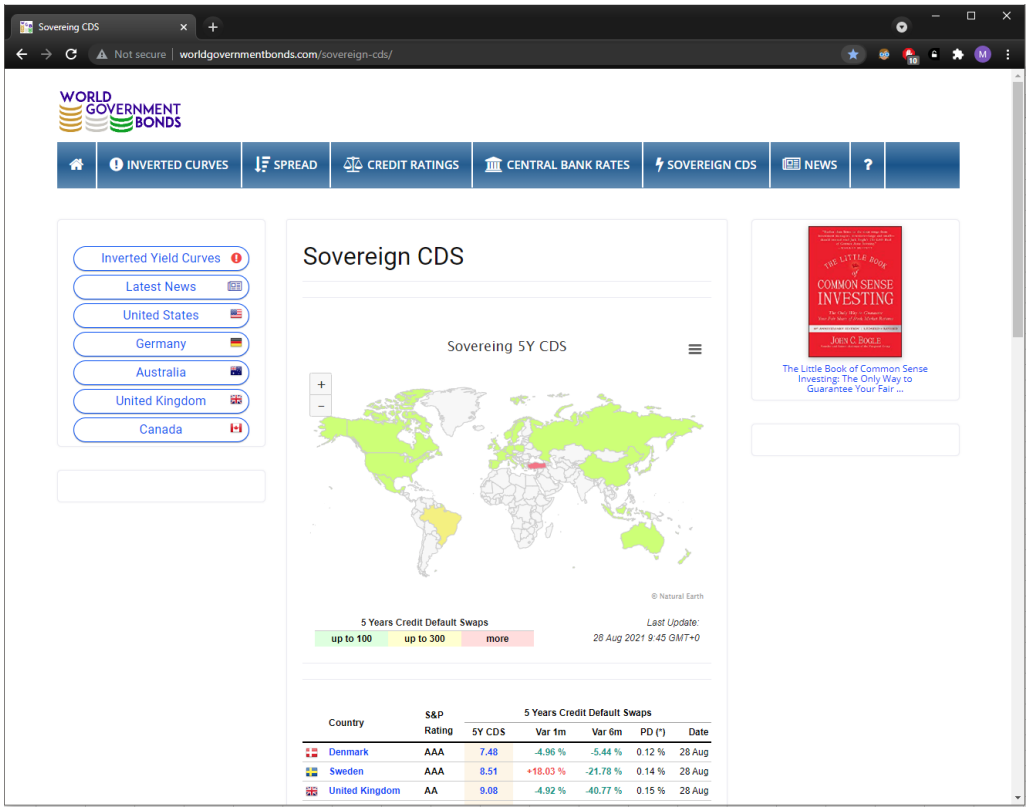

Figure 9.1: The World Government Bond page for sovereign CDS spreads

9.2.2 Implying Default Probabilities from CDS Spreads

To imply Probabilities of Default (PDs) we employed the tool provided by IHS Markit[12]. A screenshot of this tool can be seen in Figure 9.2. Given the 5Y CDS price (normally called a "spread") one can imply the PD[1]. One key input is the "recovery rate". This represents the reality

[1]strictly speaking this is a "risk neutral" default probability, not a "real world" default probability, but it serves for this purpose

that the debt of a defaulted borrower is not necessarily a lost cause. Generally, it will retain some value representing the possibility of either late payment or partial payment. If, for example, a defaulted bond of $100 can be sold for $30 then we say the recovery rate of this defaulted debt is 30%. The standard CDS-market assumption of expected recovery rate is 40% for developed-market sovereigns and 25% for emerging-market sovereigns. We follow that assumption here. In general, lenders will have their own models for recovery rates.

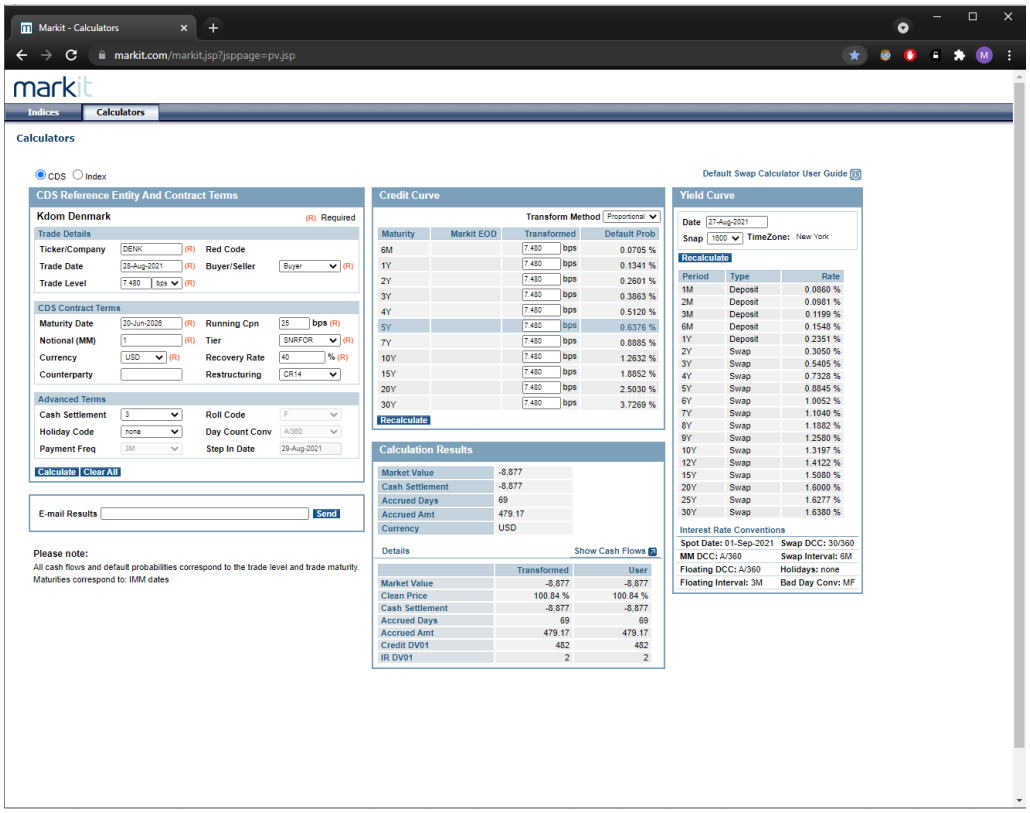

Figure 9.2: The markit tool for pricing CDS and implying PDs

9.2.3 Portfolio Weighting

The availability of CDS data from [25] has forced our hand with regards to the universe of governments we might include in our example portfolio. Attempts were made to find a third-party example portfolio (for example an investment fund) comprised only of these governments' bonds, but the available subsets were all too small to make a satisfying example. Instead, we choose to use *all* 28 of the governments for which we are able to imply a PD. To decide on the *weighting* of each government within the portfolio we use GDP as a benchmark: countries with larger GDPs should constitute a larger part of our example portfolio.

We sourced country GDPs from the website https://tradingeconomics.com/[24]; a screenshot of the relevant page can be seen in Figure 9.3. Again, it provides current values and previous values, allowing us to explore change over time. The "before-and-after" dates for the GDP and CDS-spread data do not align, but for the sake of building an example we ignore this.

The final input data for our imagined example portfolio can be seen in Table 9.1

Country	Rating	GDP	Prev. GDP	CDS Spread	Prev. CDS Spread	Implied PD	Prev. Implied PD	RR
Australia	AAA	1,331	1,397	15.54	14.57	1.320%	1.238%	40.0%
Austria	AA+	429	445	9.71	9.83	0.827%	0.837%	40.0%
Belgium	AA	515	533	12.20	12.10	1.038%	1.029%	40.0%
Brazil	BB-	1,445	1,878	176.00	194.30	11.343%	12.446%	25.0%
Canada	AAA	1,643	1,742	37.80	36.60	3.181%	3.081%	40.0%
China	A+	14,723	14,280	33.81	32.88	2.850%	2.772%	40.0%
Denmark	AAA	355	350	7.48	7.91	0.638%	0.674%	40.0%
France	AA	2,603	2,716	20.80	15.90	1.763%	1.350%	40.0%
Germany	AAA	3,806	3,861	10.20	10.20	0.868%	0.868%	40.0%
Greece	BB	189	205	74.00	78.10	6.132%	6.460%	40.0%
Hong Kong	AA+	347	366	32.20	29.70	2.716%	2.508%	40.0%
Indonesia	BBB	1,058	1,119	71.10	79.42	5.899%	6.566%	40.0%
Ireland	AA-	419	399	14.50	14.40	1.232%	1.224%	40.0%
Italy	BBB	1,886	2,005	72.80	74.70	6.035%	6.188%	40.0%
Japan	A+	5,065	4,955	17.80	15.20	1.511%	1.291%	40.0%
Mexico	BBB	1,076	1,269	90.15	98.02	5.981%	6.485%	25.0%
Netherlands	AAA	912	907	10.30	10.30	0.877%	0.877%	40.0%
New Zealand	AA+	212	209	15.70	15.40	1.334%	1.308%	40.0%
Norway	AAA	362	406	10.40	10.00	0.885%	0.851%	40.0%
Poland	A-	594	596	50.50	53.50	3.396%	3.594%	25.0%
Portugal	BBB	231	240	28.80	31.40	2.433%	2.649%	40.0%
Russia	BBB-	1,484	1,687	83.21	87.18	5.533%	5.789%	25.0%
South Korea	AA	1,631	1,647	17.64	24.31	1.497%	2.057%	40.0%
Spain	A	1,281	1,393	29.60	35.50	2.499%	2.990%	40.0%
Sweden	AAA	538	531	8.51	10.88	0.725%	0.926%	40.0%
Turkey	B+	720	761	362.91	296.52	21.986%	18.360%	25.0%
UK	AA	2,708	2,831	9.08	15.33	0.773%	1.302%	40.0%
USA	AA+	20,937	21,433	10.00	10.10	0.837%	0.860%	40.0%

Table 9.1: Example Input Data

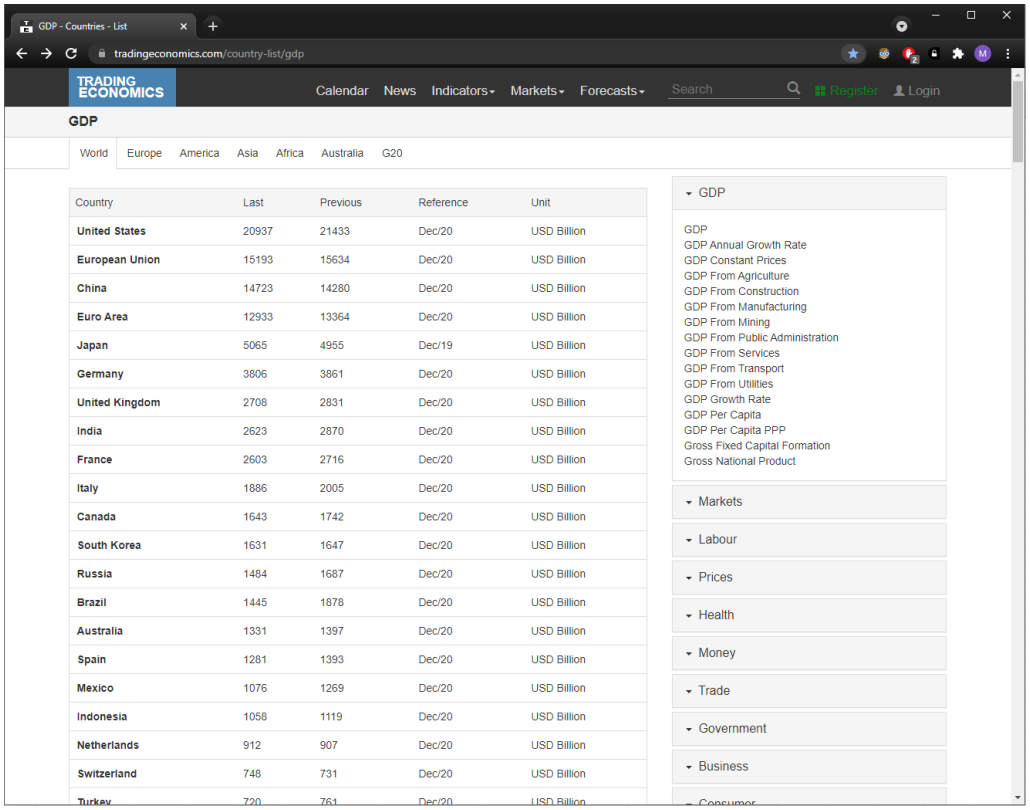

Figure 9.3: Country GDP Data from Trading Economics

9.2.4 Portfolio Composition and Loss Given Default

We suppose that our investor has a million dollars to invest. As discussed above the allocation of funds will be in proportion to each government's GDP. This gives the investment values shown in Table 9.2. Given these investment amounts we can use the standard recovery rates discussed in Section 9.2.2 to estimate a *loss-given-default* (LGD), i.e. an investment of \$100 with a recovery rate of 40% would have a loss-given-default of \$60. It is these LGDs that will be the weights of our GPB random variable. Necessarily, our weights must be integers. We could have multiplied all values by 100 so that we are expressing amounts in cents, but rounding to the nearest dollar seemed sufficiently precise. The LGDs can also be seen in Table 9.2.

9.3 Application

Given the portfolio described above we can model the amount that the investor might lose over a five-year period as a GPB random variable with probabilities equal to the probabilities of default and weights equal to the LGDs. Doing so with the `pyGPB` package is very easy and, with the addition of GPU-acceleration, very quick.

9.3.1 Distributions

The distributions of potential losses can be seen in Figures 9.4 and 9.5. The distributions were calculated for various underlying correlations, showing the effect of non-independence on potential loss. With higher correlation we would expect to have a higher likelihood of there being no losses, but also a higher likelihood of very high losses. These expectations can be seen as borne out in the distributions. The left-hand side of the cdfs are higher for higher correlations, whilst the right-hand

Country	Investment	Prev. Investment	LGD	Prev LGD
Australia	19,430.66	19,911.35	11,658	11,947
Austria	6,262.77	6,342.55	3,758	3,806
Belgium	7,518.25	7,596.81	4,511	4,558
Brazil	21,094.89	26,767.01	15,821	20,075
Canada	23,985.40	24,828.61	14,391	14,897
China	214,934.31	203,531.88	128,961	122,119
Denmark	5,182.48	4,988.53	3,109	2,993
France	38,000.00	38,710.96	22,800	23,227
Germany	55,562.04	55,030.57	33,337	33,018
Greece	2,759.12	2,921.85	1,655	1,753
Hong Kong	5,065.69	5,216.57	3,039	3,130
Indonesia	15,445.26	15,949.03	9,267	9,569
Ireland	6,116.79	5,686.92	3,670	3,412
Italy	27,532.85	28,577.13	16,520	17,146
Japan	73,941.61	70,623.28	44,365	42,374
Mexico	15,708.03	18,086.97	11,781	13,565
Netherlands	13,313.87	12,927.41	7,988	7,756
New Zealand	3,094.89	2,978.86	1,857	1,787
Norway	5,284.67	5,786.69	3,171	3,472
Poland	8,671.53	8,494.75	6,504	6,371
Portugal	3,372.26	3,420.70	2,023	2,052
Russia	21,664.23	24,044.70	16,248	18,034
South Korea	23,810.22	23,474.58	14,286	14,085
Spain	18,700.73	19,854.34	11,220	11,913
Sweden	7,854.01	7,568.31	4,712	4,541
Turkey	10,510.95	10,846.48	7,883	8,135
UK	39,532.85	40,350.05	23,720	24,210
USA	305,649.64	305,483.10	183,390	183,290
Total	1,000,000.00	1,000,000.00	611,645	613,235

Table 9.2: Portfolio Composition: Investment Amount and Loss Given Default

sides approach 1 more slowly.

9.3.2 Quantiles

Quantiles for differing correlations can be seen in Table 9.3. Again, the same pattern can be observed of correlation increasing the probability of extreme (i.e. high and low) outcomes.

p	$\rho = 0$	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.4$	$\rho = 0.5$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
0.05	-	-	-	-	-	-	-	-	-
0.10	-	-	-	-	-	-	-	-	-
0.15	-	-	-	-	-	-	-	-	-
0.20	-	-	-	-	-	-	-	-	-
0.25	-	-	-	-	-	-	-	-	-
0.30	-	-	-	-	-	-	-	-	-
0.35	-	-	-	-	-	-	-	-	-
0.40	2,023	-	-	-	-	-	-	-	-
0.45	7,751	1,857	-	-	-	-	-	-	-
0.50	7,883	7,883	1,655	-	-	-	-	-	-
0.55	7,883	7,883	7,883	1,655	-	-	-	-	-
0.60	11,220	9,267	7,883	7,883	3,039	-	-	-	-
0.65	14,286	11,781	9,538	7,883	7,883	7,883	-	-	-
0.70	15,821	15,821	14,391	11,781	9,267	7,883	-	-	-
0.75	16,520	16,292	16,248	15,821	14,890	11,220	7,883	7,883	-
0.80	19,664	19,664	19,629	18,175	16,520	16,248	9,538	7,883	7,883
0.85	24,131	24,403	24,679	24,403	24,131	23,704	18,285	15,821	7,883
0.90	30,806	32,971	34,924	35,899	36,142	35,501	32,341	27,602	23,704
0.95	48,589	55,141	60,967	67,263	71,581	74,805	75,004	70,985	65,739
0.99	161,302	176,174	183,390	193,515	207,469	224,971	273,899	306,663	345,500
0.999	209,457	230,444	268,201	331,343	378,343	432,151	545,515	588,784	611,645
1.00	611,645	611,645	611,645	611,645	611,645	611,645	611,645	611,645	611,645

Table 9.3: Quantiles of total loss for differing correlation assumptions

9.3.3 Expected Shortfall

Although quantiles are useful for understanding the value of the boundary between the body of a distribution and some percentile tail, they don't tell us anything about the values *within* the tail. For example, the 90th percentile tells us that in the worst one-in-ten period we would expect to lose *at least this much*, however it doesn't give a clear idea of what the level of loss could. Instead, we can turn to "expected shortfall" which gives the *expected value* for that worst one-in-ten period, i.e. for the 90% expected shortfall we would be calculating $\mathbb{E}[\mathrm{X}|\mathrm{X} \geqslant G^{-1}(0.9)]$. The expected shortfall of a distribution of outcomes is a common risk measure used in finance, it was introduced by Acerbi et al[1] in 2001. The expected shortfalls for our portfolio for different correlations can be seen in Table 9.4.

9.4 Understanding Changes over Time

Our investor might want to monitor the portfolio over time and see how the changes in weighting and PDs affect their risk. The change in the cdf and pmf between the *previous* LGDs and PDs, and the *current* values can be seen in Figures 9.6 and 9.7. Whilst these images do give some sense of how things have changed, it is often useful to choose some particular "key risk indicators" (KRIs) to monitor over time. There are many measures our investor could choose as an indicator, but let's suppose they have chosen the 90% expected shortfall with zero assumed correlation. The

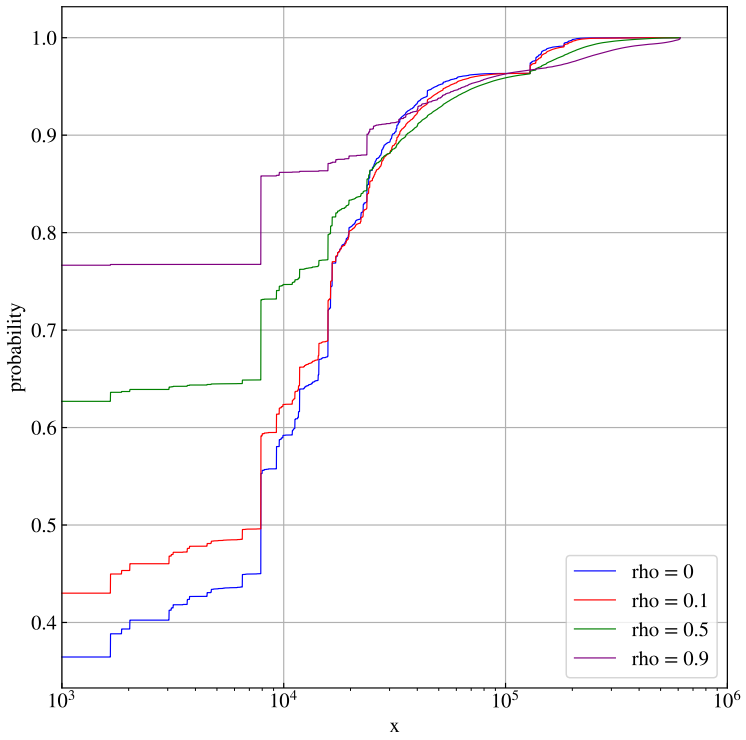

Figure 9.4: cumulative distribution function for various correlations

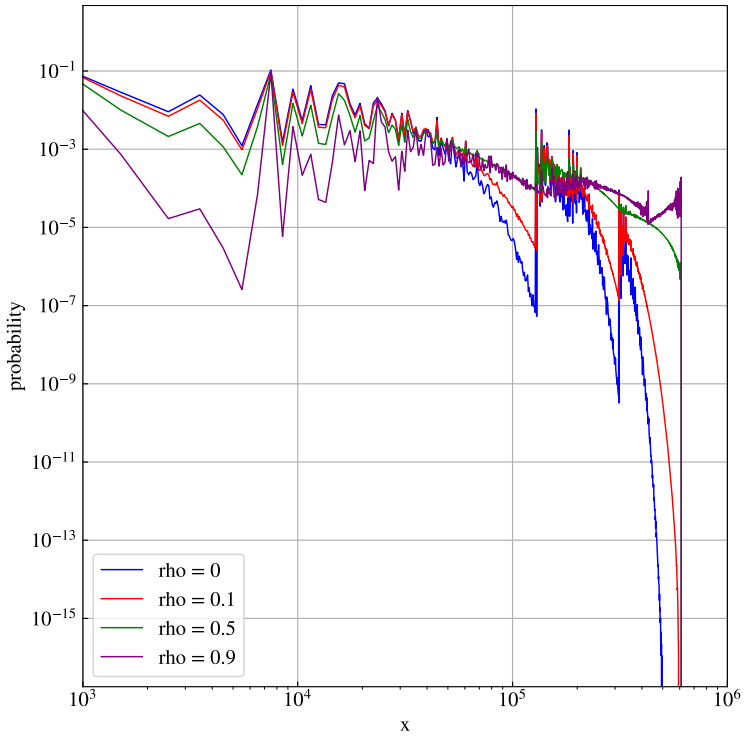

Figure 9.5: probability mass function for various correlations

p	$\rho = 0$	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.3$	$\rho = 0.4$	$\rho = 0.5$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
0.05	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.1	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.15	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.2	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.25	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.3	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.35	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290	15,290
0.4	25,122	25,122	25,122	25,122	25,122	25,122	25,122	25,122	25,122
0.45	27,285	27,285	27,285	27,285	27,285	27,285	27,285	27,285	27,285
0.5	27,289	27,289	27,289	27,289	27,289	27,289	27,289	27,289	27,289
0.55	27,289	27,289	27,289	27,289	27,289	27,289	27,289	27,289	27,289
0.6	34,312	34,312	34,312	34,312	34,312	34,312	34,312	34,312	34,312
0.65	37,474	37,474	37,474	37,474	37,474	37,474	37,474	37,474	37,474
0.7	39,219	39,219	39,219	39,219	39,219	39,219	39,219	39,219	39,219
0.75	45,779	45,779	45,779	45,779	45,779	45,779	45,779	45,779	45,779
0.8	53,244	53,244	53,244	53,244	53,244	53,244	53,244	53,244	53,244
0.85	63,757	63,757	63,757	63,757	63,757	63,757	63,757	63,757	63,757
0.9	82,308	82,308	82,308	82,308	82,308	82,308	82,308	82,308	82,308
0.95	127,130	127,130	127,130	127,130	127,130	127,130	127,130	127,130	127,130
0.99	193,077	193,077	193,077	193,077	193,077	193,077	193,077	193,077	193,077
0.999	245,526	245,526	245,526	245,526	245,526	245,526	245,526	245,526	245,526
1	611,645	611,645	611,645	611,645	611,645	611,645	611,645	611,645	611,645

Table 9.4: Expected shortfalls for differing correlation assumptions

current value is $82,307.63, as can be seen from Table 9.4. Previously it was $80,836.62, so there has been an increase of $1,471.01 in our risk indicator. How might we explain this increase in terms of changes in the underlying portfolio, i.e. in terms of the changing investment amounts and PDs? Due to so-called "portfolio effects" the impacts of each individual change is not additive: the combined impact of all the parameters changing together is different to the sum of the impact of each parameter changing on its own. Instead we will turn to *incremental* impacts.

9.4.1 Incremental Impacts

To allocate the total change in our KRI to individual governments we can consider the incremental impacts of changing the portfolio piecemeal from one state to the other, i.e. we change one country at a time from its previous parameters to the new parameters (keeping all previous changes as we go) and record the impact in KPI that results from each incremental change. In this way the sum of the allocated impacts should equal the combined impact.

There are two key choices to make: the *direction* in which we proceed (i.e. whether we start with the current state and progress by making incremental changes until we reach the previous state, or vice versa) and the *order* in which we change the elements. For the direction we decided to compute both forward impacts (i.e. starting with the previous state) and also the backwards impacts, then take the mean. For the order we use a *greedy* approach: choosing the government with the largest marginal impact at each step. This requires computing the impact of each putative change at each step and choosing the biggest, so it could be slow, however with the speed provided by GPU-acceleration the whole process took only a few seconds.

Each element's incremental impact can be broken down further, i.e. the impact of changing just the PD, just the weighting and the "cross effect" from changing both together that is not otherwise captured by the individual changes. The results can be seen in Table 9.5

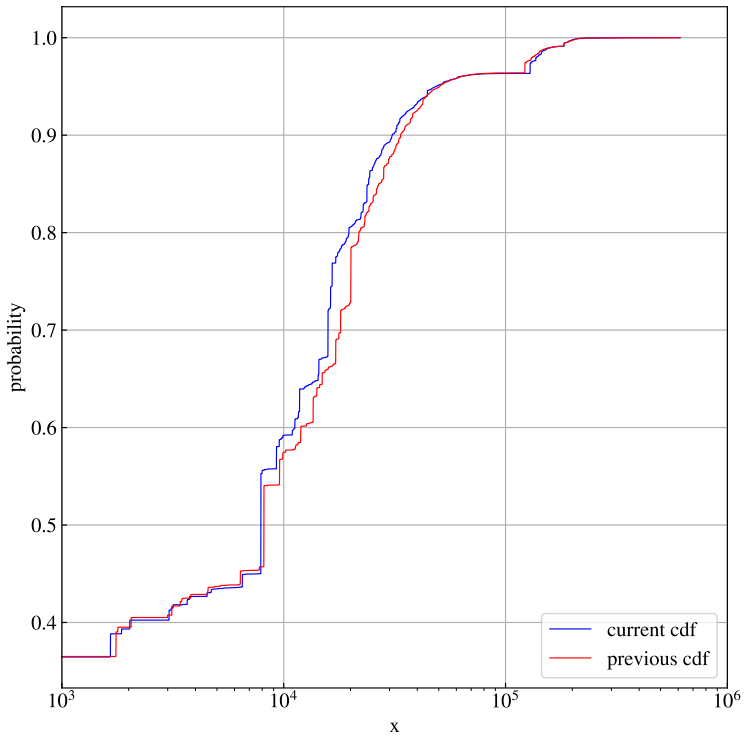

Figure 9.6: Change in cdf from the previous parameters to the current parameters

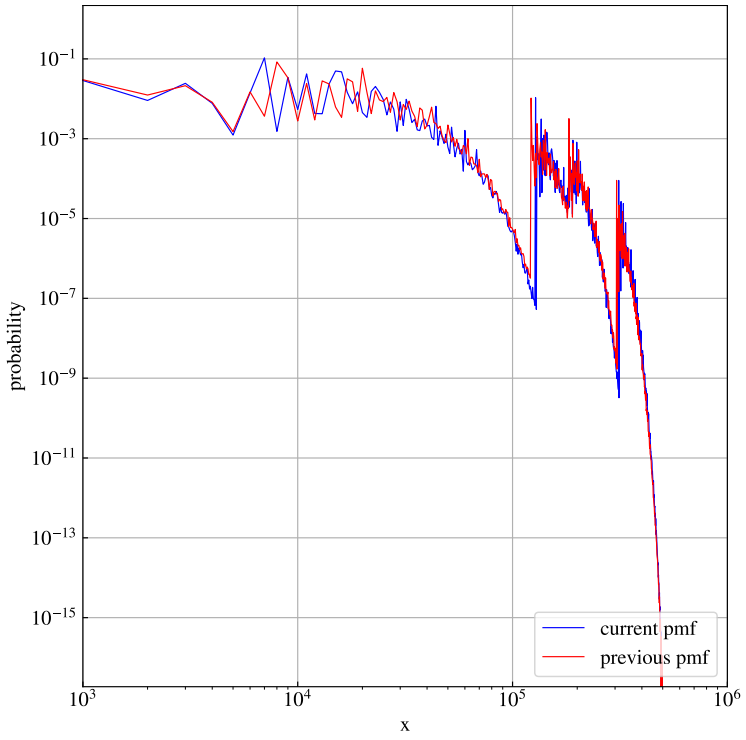

Figure 9.7: Change in pmf from the previous parameters to the current parameters

					forward incremental impacts				backward incremental impacts				mean incremental impacts			
country	pd	pd Δ	invest.	Invest. Δ	PD	LGD	Cross	Total	PD	LGD	Cross	Total	PD	LGD	Cross	Total
Australia	1.32%	0.08%	19,431	-481	177	24	-16	186	-22	60	-4	34	78	42	-10	110
Austria	0.83%	-0.01%	6,263	-80	0	-179	1	-178	1	-0	-0	1	0	-90	0	-89
Belgium	1.04%	0.01%	7,518	-79	-1	-1	-0	-1	-1	1	-0	1	-1	0	-0	-0
Brazil	11.34%	-1.10%	21,095	-5,672	-1,182	-1,369	1,039	-1,512	-755	-2,091	891	-1,955	-968	-1,730	965	-1,733
Canada	3.18%	0.10%	23,985	-843	-55	-131	-1	-187	-57	-38	1	-94	-56	-85	0	-140
China	2.85%	0.08%	214,934	11,402	415	1,849	38	2,303	1,509	1,943	-78	3,374	962	1,896	-20	2,839
Denmark	0.64%	-0.04%	5,182	194	1	-2	0	-0	2	0	0	2	2	-1	0	1
France	1.76%	0.41%	38,000	-711	509	-25	-21	463	1,221	-35	27	1,213	865	-30	3	838
Germany	0.87%	0.00%	55,562	531	0	28	0	28	0	17	0	17	0	22	0	22
Greece	6.13%	-0.33%	2,759	-163	-465	-9	4	-470	12	-4	-0	8	-227	-6	2	-231
Hong Kong	2.72%	0.21%	5,066	-151	470	5	-3	472	-11	-0	0	-11	229	2	-1	230
Indonesia	5.90%	-0.67%	15,445	-504	204	114	-10	307	120	126	13	260	162	120	1	283
Ireland	1.23%	0.01%	6,117	430	-1	184	1	184	-0	0	-0	-0	-0	92	0	92
Italy	6.04%	-0.15%	27,533	-1,044	-111	-412	239	-284	103	-29	-1	73	-4	-220	119	-106
Japan	1.51%	0.22%	73,942	3,318	1,834	249	51	2,133	1,063	300	-47	1,317	1,449	275	2	1,725
Mexico	5.98%	-0.50%	15,708	-2,379	-209	-389	-47	-646	-550	499	-702	-752	-379	55	-374	-699
Netherlands	0.88%	0.00%	13,314	386	0	416	0	416	0	3	0	3	0	209	0	209
New Zealand	1.33%	0.03%	3,095	116	-1	1	0	0	-1	0	-0	-1	-1	0	0	-0
Norway	0.89%	0.03%	5,285	-502	3	-178	-5	-180	-2	-1	0	-3	1	-90	-2	-91
Poland	3.40%	-0.20%	8,672	177	22	85	-7	100	27	3	0	30	24	44	-3	65
Portugal	2.43%	-0.22%	3,372	-48	-186	-5	6	-185	7	18	2	27	-89	6	4	-79
Russia	5.53%	-0.26%	21,664	-2,380	-0	-357	47	-311	144	517	-1,581	-920	72	80	-767	-615
South Korea	1.50%	-0.56%	23,810	336	-44	9	-20	-55	-1,299	3	3	-1,293	-671	6	-9	-674
Spain	2.50%	-0.49%	18,701	-1,154	-358	-23	37	-344	129	62	11	203	-114	19	24	-71
Sweden	0.73%	-0.20%	7,854	286	-473	-4	4	-473	16	-3	-1	11	-229	-4	1	-231
Turkey	21.99%	3.63%	10,511	-336	1,778	-7	-17	1,755	1,367	12	1	1,380	1,573	3	-8	1,568
U.K.	0.77%	-0.53%	39,533	-817	-1,780	-11	117	-1,674	-1,207	69	-56	-1,194	-1,494	29	30	-1,434
U.S.A.	0.84%	-0.02%	305,650	167	-384	9	-0	-376	-266	8	0	-258	-325	8	-0	-317
Total			1,000,000	0	165	-130	1,437	1,471	1,550	1,442	-1,521	1,471	857	656	-42	1,471

Table 9.5: Change in 90% Expected Shortfall allocated to individual changes

Chapter 10

Directions for Further Work

There are many ways in which this topic could be extended. This chapter lists some of the directions that the author would have liked to explore in this project, but time constraints did not permit.

Calculating only a tail

Every algorithm for computing the pmf of the GPB distribution that is presented in this project sought to compute the entire distribution, i.e. the value of the function at every point in its support. Where one only wishes to know part of the distribution, in particular the tail, this is potentially quite wasteful. We believe the work by Koiliaris and Xu[16] on the subset-sum problem could likely be adapted to computing the tail of the distribution of a GPB random variable and produce impressively quick results.

Implement in C/C++ and port to Python

One of the key weaknesses in the implementation of the `pyGPB` library is its use of Python. For true performance it would be preferable to write the underlying algorithms in a compiled language like C++ and then port them for use in Python. We chose not to do this as part of this project due to time constraints, however, it would be a very worthwhile direction for further work.

Comparison to Junge's Work

The existence of the `PoissonBinomial`[13] package for R, developed by Florian Junge in the last year or so, was discovered very late in the timeline of this project - after all testing and benchmarking had been completed and with the drafting of this document well underway. There was not time to benchmark our performance against this package, however this would clearly be a very beneficial endeavour.

Extension of the LFGPB distribution beyond Gaussian equicorrelation

For simplicity and speed of development this project restricted its consideration of correlation structures to the Gaussian copula with an equicorrelation matrix. Clearly, there is scope to extend this in future work. In particular:

- It should be relatively easy to extend the current implementation to a Gaussian Copula model where the non-diagonal elements of the correlation matrix can be reduced to $\rho_{ij} = r_i \cdot r_j$ where $\boldsymbol{r}$ is an m-dimensional vector. This should require only minor modification to Equation 5.10.

- It should be possible to extend the LFGPB with the Gaussian copula to any valid correlation matrix by using multiple integration, however computational complexity will be an issue so Monte-Carlo integration is likely the best approach.
- The dependence structure between component random variables could be expressed using other copulas (Student-t, Clayton etc.).

Maximum-Likelihood Parameter Estimation

As discussed in Appendix B.1.4, this paper provides an analytical formula for the population mean and variance of a LFGPB random variable which might be used to imply the correlation parameter from a sample mean and variance if $\boldsymbol{p}$ and $\boldsymbol{w}$ are known. This approach, known as the "method of moments", often shows bias so the method of "Maximum Likelihood" is usually preferred. Developing the necessary *likelihood* function for a LFGPB random variable is left for future work.

More Robust Heuristics for Algorithm Choice

The `pyGPB` package can automatically choose a fastest algorithm for the user by using an heuristic based on the benchmarking results from chapter 8. Clearly, this is biased towards the particular hardware that was used for that analysis. A more robust heuristic for algorithm choice would be preferable.

Measure How the Number of Processors Affects Parallelised Speed

The computational complexities in Table 4.1 were left blank for the parallel algorithms as we did not feel particularly confident in making any such claim. It seems reasonable to assume that the complexity of these approaches will be not just a function of n and m, but also the number of available (multi) processors. Presumably, time complexity of these algorithms might be inversely proportional to the number of processors, and it would be interesting to understand if this presumption is correct by experimenting with different numbers of processors.

Chapter 11

Conclusions and Critical Evaluation

Overall, we feel that the goals of the project have been achieved and that it has been a successful endeavour. The five goals that this project set out to achieve were:

Extend the algorithms of Biscari et al.[4] to apply to the GPB distribution

Algorithms 1, and 3 are successful extensions of the "DC" and "DC-FFT" algorithms from [4] to the GPB distribution. The testing in Chapter 7 shows that they produce results with excellent absolute accuracy, although the "DC-FFT" approach does suffer from problems of relative accuracy in the tails. The hoped-for improvements in computational complexity were achieved, as can be seen in Chapter 8, though the project was no-doubt prevented from achieving truly excellent speed by the use of Python and not C/C++.

Make use of GPU acceleration by parallelising the above algorithms

Parallelised versions of the above techniques were developed and can be seen in Algorithms 4 and 5. These algorithms were implemented in a mixture of Python and CUDA C++ for running on CUDA GPUs. The results show the same accuracy as their serial counterparts (see Chapter 7) and provide an order-of-magnitude speed up for parts of the parameter domain (see Chapter 8). Again, a pure C++ approach would have been preferable, but these results are still very encouraging.

Extend the GPB distribution to allow for some simple correlation structure

The LFGPB distribution presented in Section 2.3 presents one potential way to explore the effects of correlation on GPB random variables. A correlation parameter is added to the distribution by using a Gaussian copula, an idea commonly used in the financial sector. The Gaussian copula was chosen due to its simplicity and generality but other choices could have been made.

This model cannot, of course, hope to capture the full dependence structure of real world systems, but one might hope to use the LFGPB to model them more accurately than the GPB on its own, whilst being aware of its imperfections.

In Chapter 5 we present a method for computing the pmf of a LFGPB random variable involving numerical integration, and testing in Chapter 7 shows our implementation to be accurate.

Create a Python package that implements the above

Chapter 6 discusses the creation and use of the `pyGPB` package for Python with much more detail in Appendix C. The package provides classes for working with GPB and LFGPB random variables: computing their distributions, generating random variates, calculating moments etc.

Users are able to specify particular algorithms for computation if they wish, but otherwise a simple heuristic is used to choose the fastest method. Where a CUDA GPU is available the package will make use of it, but it still works on systems without such hardware.

The package comes with a full suite of unit tests to ensure it works correctly and testing in Chapter 7 shows that the included implementations are accurate.

`pyGPB` comes with documentation in the "Read the Docs" format that will be familiar to many Python users.

Demonstrate how the package might be used in the real world.

Chapter 9 imagines an example portfolio of government bonds and uses real-world data to generate the probability and weight parameters for a GPB random variable.

Although the parameters were chosen using real world data, the portfolio itself was not a real-world portfolio. With more time it would have been preferred to find a better example that reflected a complete real-world dataset, however as a demonstration of what can be done with the `pyGPB` package we believe this part of the project still has merit.

The `pyGPB` package is used to understand the distribution of potential losses on the portfolio due to sovereign-default events. The effect of correlation on the distribution is demonstrated and is shown to agree with intuition. The 90% expected shortfall is chosen as an example Key Risk Indicator and its change over time is explained by decomposing the total into amounts attributable to changes in the underlying components.

Appendices

Appendix A

Additional Detail on Computing the GPB Distribution

A.1 Existing Approaches to the GPB

A.1.1 Naïve approach, $O(2^m)$

There are multiple possible ways to implement a traversal of a probability tree as seen in Figure 2.1. The fastest we found was the "bit-hacky" approach shown in Algorithm 6, which sought to avoid branching and the overhead of the more-obvious recursive algorithm. Another possibility is to use a stack, as shown in Algorithm 7.

Algorithm 6: Naïve Approach

Function `GPB-Naive`$([p_0, ..., p_{m-1}], [w_0, ..., w_{m-1}])$`:`

$n \leftarrow 1 + \sum_i w_i$
$\boldsymbol{a} \leftarrow$ array of zeros of length n
for $i = 0$ **to** $2^m - 1$ **do**
  $P \leftarrow 1$
  $W \leftarrow 0$
  for $j = 0$ **to** $m - 1$ **do**
    $flag \leftarrow (i$ `>>` $j)$ `&` 1
    `/* where ">>" is right bitshift and "&" is bitwise AND */`
    $W \leftarrow W + flag \cdot w_j$
    $P \leftarrow P \cdot \text{abs}(1 - p_j - flag)$
  end
  $\boldsymbol{a}[W] \leftarrow \boldsymbol{a}[W] + P$
end
return $\boldsymbol{a}$

Complexity Analysis: For a given number of events, m, the number of leaves on the tree is 2^m. This means the complexity for calculating the pmf this way is at least $O(2^m)$ and better approaches are desirable.

A.1.2 Characteristic Function Approach, $O(NM)$

This section looks at the original approach used by Zhang et al.[26]. They rely on computing the characteristic function of a GPB random variable rather than the pmf. The latter can be

Algorithm 7: Naïve Approach using a Stack

Function `GPB-Naive-Stack`$([p_0, ..., p_{m-1}], [w_0, ..., w_{m-1}])$**:**

$n \leftarrow 1 + \sum_i w_i$
$\boldsymbol{a} \leftarrow$ array of zeros of length n
$S \leftarrow$ empty stack
S.push$((0, 1.0, 0))$
while S *is not empty* **do**
  $(t, p, w) \leftarrow$ S.pop()
  if $t = m - 1$ **then**
    $a_{w+w_t} \leftarrow a_{w+w_t} + p \cdot t$
    $a_w \leftarrow a_w + p \cdot (1 - p_t)$
  else
    S.push$((t + 1, p \cdot p_t, w + w_t))$
    S.push$((t + 1, p \cdot (1 - p_t), w))$
  end
end
return $\boldsymbol{a}$

extracted from the former by using the Inverse Discrete Fourier Transform (IDFT). The details of how they performed their complex-number calculations were not followed exactly, instead we relied on the complex-number abilities of Python. Also they used a slightly different parametrisation (see Appendix A.2), however the overall substance is the same as their approach.

Primer on the Discrete Fourier Transform (DFT):

Given a sequence of complex numbers $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ the DFT provides another sequence of complex numbers $\hat{\mathbf{y}} = \mathcal{F}\{\mathbf{y}\} = (\hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{n-1})$ defined by

$$\hat{y}_k = \mathcal{F}\{\mathbf{y}\}_k = \sum_{j=0}^{n-1} \exp\left(-2i\pi \tfrac{kj}{n}\right) \cdot y_j$$

or equivalently for a *function* of the form $y : [\![0 : n-1]\!] \to \mathbb{C}$, this can be written using the notation

$$\hat{y}(k) = \mathcal{F}\{y\}(k) = \sum_{j=0}^{n-1} \exp\left(-2i\pi \tfrac{kj}{n}\right) \cdot y(j)$$

These notations will be used interchangeably. Usefully, there is an Inverse Discrete Fourier Transform (IDFT) which allows one to obtain $y(n)$ given $\hat{y}(k)$, given by

$$y(j) = \mathcal{F}^{-1}\{\hat{y}\}(j) = \frac{1}{n} \sum_{k=0}^{n-1} \exp\left(2i\pi \tfrac{kj}{n}\right) \cdot \hat{y}(k)$$

Given any sequence of n complex numbers, the DFT (or the IDFT) of that sequence can be computed in $O(n \log n)$ time using algorithms known as Fast Fourier Transform (FFT) algorithms, such as those discovered by Cooley and Tukey[7]. This project does not delve further into the detail of FFT algorithms, and happily uses solutions already implemented by others.

Application to the GPB distribution:

The Characteristic Function of a random variable completely describes its distribution in a similar way to the pmf and cdf. Working with the characteristic function often proves more tractable. For the random variable X the characteristic function is given by

$$\psi(\tau) = \mathbb{E}\left[\exp(i\tau \mathrm{X})\right]$$

For the GPB distribution we can approach the characteristic function in two key ways. Firstly:

$$\begin{aligned}\psi(t) &= \mathbb{E}\left[\exp(i\tau \mathrm{X})\right] \\ &= \sum_{x=0}^{n-1} \exp(i\tau n) \cdot \mathbb{P}[\mathrm{X}{=}x] \\ &= \sum_{x=0}^{n-1} \exp(i\tau n) \cdot g(x)\end{aligned}$$

substituting in $\tau = \frac{-2\pi k}{n}$ we can obtain a function with the form of the DFT

$$\hat{g}(k) = \psi(\tfrac{-2\pi k}{n}) = \sum_{x=0}^{n-1} \exp\left(-2i\pi \tfrac{kx}{n}\right) \cdot g(x)$$

meaning that by applying the IDFT we can get $g(x) = \mathcal{F}^{-1}\{\psi(\frac{-2\pi k}{n})\}(x)$, however as our current expression for ψ involves $g(x)$ we don't yet have anything useable. We need to find another expression for ψ that doesn't involve g:

$$\begin{aligned}\psi(\tau) = \mathbb{E}\left[\exp(i\tau \mathrm{X})\right] &= \mathbb{E}\left[\exp\left(i\tau \sum_{j=0}^{m-1} \mathrm{V}_j\right)\right] \\ &= \mathbb{E}\left[\prod_{j=0}^{m-1} \exp\left(i\tau \mathrm{V}_j\right)\right] \\ &= \prod_{j=0}^{m-1} \mathbb{E}\left[\exp\left(i\tau \mathrm{V}_j\right)\right] \\ &= \prod_{j=0}^{m-1} \mathbb{P}[\mathrm{V}_j{=}0] \cdot \exp(it \cdot 0) + \mathbb{P}[\mathrm{V}_j{=}w_j] \cdot \exp(i\tau w_j) \\ &= \prod_{j=0}^{m-1} 1 - p_j + p_j \cdot \exp(i\tau w_j)\end{aligned}$$

Again, substituting in $\tau = \frac{-2\pi k}{n}$ we get

$$\psi(\tfrac{-2\pi k}{n}) = \prod_{j=0}^{m-1} 1 - p_j + p_j \cdot \exp\left(-2i\pi \tfrac{kw_j}{n}\right)$$

which means

$$g(x) = \mathcal{F}^{-1}\left\{\prod_{j=0}^{m-1} 1 - p_j + p_j \cdot \exp\left(-2i\pi \tfrac{kw_j}{n}\right)\right\}(x) \tag{A.1}$$

or to expand fully to get a closed-form expression for $g(x)$:

$$g(x) = \frac{1}{n} \sum_{k=0}^{n-1} \exp\left(2i\pi \tfrac{kx}{n}\right) \prod_{j=0}^{m-1} 1 - p_j + p_j \cdot \exp\left(-2i\pi \tfrac{kw_j}{n}\right) \tag{A.2}$$

To apply this in practice, let the vector $\hat{\boldsymbol{g}} = \begin{bmatrix}\hat{g}_0 & \hat{g}_1 & \dots & \hat{g}_{n-1}\end{bmatrix}^\top$ be defined such that

$$\hat{g}_k = \prod_{j=0}^{m-1} 1 - p_j + p_j \cdot \exp\left(-2i\pi\frac{kw_j}{n}\right) \tag{A.3}$$

We then employ an FFT algorithm to compute the IDFT of $\hat{\boldsymbol{g}}$ to get

$$\boldsymbol{g} = \mathcal{F}^{-1}\{\hat{\boldsymbol{g}}\} = \begin{bmatrix}g(0) & g(1) & \dots & g(n-1)\end{bmatrix}^\top$$

Returning to our running toy example we can lay out the calculation as in Table A.1 where each cell has the value $1 - p_j + p_j \cdot \exp\left(-2i\pi\frac{kw_j}{n}\right)$ for the relevant k, j and the final column shows the product of the row giving $\hat{g}_k$. Note that $\hat{g}_{n-k}$ is the complex conjugate of $\hat{g}_k$, which is true whenever $\boldsymbol{g}$ are all real. To take advantage of this, FFT libraries include functions for Real Fast Fourier Transforms and the associated inverse (IRFFT) where one only needs to specify the first $\lfloor\frac{n}{2}\rfloor + 1$ elements of $\hat{\boldsymbol{g}}$, along with n as a parameter, to compute $\boldsymbol{g}$, providing a speed up by roughly a factor of two.

p_j	0.1	0.4	0.2	0.3	
w_j	4	1	1	2	
k \ j	0	1	2	3	$\hat{g}_k$
0	$1.000 + 0.000i$	$1.000 + 0.000i$	$1.000 + 0.000i$	$1.000 + 0.000i$	$1.000 + 0.000i$
1	$0.806 - 0.034i$	$0.906 - 0.257i$	$0.953 - 0.129i$	$0.752 - 0.295i$	$0.400 - 0.435i$
2	$0.977 + 0.064i$	$0.669 - 0.394i$	$0.835 - 0.197i$	$0.418 - 0.103i$	$0.166 - 0.226i$
3	$0.850 - 0.087i$	$0.400 - 0.346i$	$0.700 - 0.173i$	$0.550 + 0.260i$	$0.162 - 0.115i$
4	$0.917 + 0.098i$	$0.224 - 0.137i$	$0.612 - 0.068i$	$0.930 + 0.193i$	$0.133 - 0.048i$
5	$0.917 - 0.098i$	$0.224 + 0.137i$	$0.612 + 0.068i$	$0.930 - 0.193i$	$0.133 + 0.048i$
6	$0.850 + 0.087i$	$0.400 + 0.346i$	$0.700 + 0.173i$	$0.550 - 0.260i$	$0.162 + 0.115i$
7	$0.977 - 0.064i$	$0.669 + 0.394i$	$0.835 + 0.197i$	$0.418 + 0.103i$	$0.166 + 0.226i$
8	$0.806 + 0.034i$	$0.906 + 0.257i$	$0.953 + 0.129i$	$0.752 + 0.295i$	$0.400 + 0.435i$

Table A.1: Layout of Calculation for Characteristic Function Approach

So that

$$\begin{aligned}\boldsymbol{g} &= \text{IRFFT}([1, 0.4 - 0.435i, 0.166 - 0.226i, 0.162 - 0.115i, 0.133 - 0.048i], 9)\\ &= [0.3024, 0.2772, 0.18, 0.1188, 0.0552, 0.0308, 0.02, 0.0132, 0.0024]\end{aligned}$$

Finally giving us our pmf. This approach is laid out in Algorithm 8

Algorithm 8: Characteristic Function Approach

Function `GPB-CF`$(\{(p_0, w_0), (p_1, w_1), ..., (p_{m-1}, w_{m-1})\})$**:**

$n \leftarrow 1 + \sum_{i=0}^{m-1} w_i$

for $k = 0$ **to** $\lfloor n/2 \rfloor + 1$ **do**

$y_k \leftarrow \prod_{j=0}^{m-1} \left(1 + p_j \left[\exp\left(-2i\pi\frac{kw_j}{n}\right) - 1\right]\right)$

$y_{N-k} \leftarrow y_k^*$ `/* where "*" indicates complex conjugate */`

end

return `InverseFFT`$([y_0, y_1, ..., y_{n-1}])$

Complexity Analysis: Each element $\hat{g}_k$ is the product of m complex numbers so will take $O(m)$ time to compute. We need to calculate n values of $\hat{g}_k$, giving complexity $O(n \cdot m)$ and then perform the IRFFT which has complexity $O(n \log n)$, so overall complexity is $O(n \cdot m + n \log n) = O(n \cdot m)$.

A.2 Alternative Weight Parametrisations

The parametrisation of the GPB defined in Section 2.1.3 and used throughout this report requires a vector of success weights $\boldsymbol{w}$ which are positive integers, i.e. recall Equation 2.8

$$\boldsymbol{w} = \begin{bmatrix} w_0 & w_1 & \dots & w_{m-1} \end{bmatrix}^\top \in \mathbb{N}_+^m \tag{2.8}$$

This is the simplest and most computationally-tractable way to parametrise the GPB, so it has been adopted for this project, however there are other options. This appendix considers three ways in which this constraint might be relaxed and how they can be transformed into the form above.

A.2.1 Non-Positive Success Weights

The case of a zero weight, i.e. $w_i = 0$, is trivial in that regardless of whether $\mathrm{B}_i = 1$ or $\mathrm{B}_i = 0$ it will always be the case that $\mathrm{V}_i = 0$, and so this element will have no bearing on the pmf.

Negative weights can be allowed for by applying an appropriate offset and "flipping" the associated probability. Suppose the GPB random variable X is parametrised by the usual $\boldsymbol{p}$ and $\boldsymbol{w}$ but now a weight can be any non-zero integer. Let there be a second random variable, X′ which has weights defined by $w'_i = |w_i|$ and probabilities defined by

$$p'_i = \begin{cases} p_i & \text{if } \; w_i > 0 \\ 1 - p_i & \text{if } \; w_i < 0 \end{cases} \tag{A.4}$$

Let $\mathscr{M}_+$ and $\mathscr{M}_-$ be the indices of the positively-weighted and negatively-weighted component random variables of X. We can now connect X to X′ via the following:

$$\begin{aligned} \mathrm{X} &= \sum_{i=0}^{m-1} \mathrm{B}_i w_i \\ &= \sum_{i \in \mathscr{M}_+} \mathrm{B}_i w_i + \sum_{i \in \mathscr{M}_-} w_i + \sum_{i \in \mathscr{M}_-} -(1 - \mathrm{B}_i) w_i \\ &= \sum_{i \in \mathscr{M}_-} w_i + \sum_{i \in \mathscr{M}_+} \mathrm{B}_i w_i + \sum_{i \in \mathscr{M}_-} +\mathrm{B}'_i w'_i \\ &= \mathrm{X}' + \sum_{i \in \mathscr{M}_-} w_i \end{aligned} \tag{A.5}$$

As all the weights of X′ are positive, its pmf can be computed using all of the algorithms discussed in this report, and we can extract the pmf of X by simply shifting it to the left by the absolute total value of the negative weights.

Allowing negative weights for a LFGPB random variable is slightly more tricky: the success probabilities cannot just be flipped and a new LFGPB variable used under the pretence that failure is success. Instead, the true success probabilities should be used in calculating the conditional probabilities and *then* flipped.

A.2.2 Three Parameter Form

The GPB distribution was introduced by Zhang et al.[26] in a three-parameter form $\overline{\mathrm{GPB}}(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b})$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ are the weights associated with failure and success respectively, i.e. they considered a random variable $\overline{\mathrm{X}}$ as the sum of the elements of the random vector $\overline{\mathbf{V}}$ with elements taking values as

$$\overline{\mathrm{V}}_i = a_i(1 - \mathrm{B}_i) + b_i\mathrm{B}_i = \begin{cases} b_i & \text{with probability } \; p_i \\ a_i & \text{with probability } \; 1 - p_i \end{cases} \tag{A.6}$$

Consider a second random variable $\overline{\mathrm{X}}' = \overline{\mathrm{X}} - \sum_{i=0}^{m} a_i$, this will be the sum of the elements of $\overline{\mathbf{V}}'$ where

$$\begin{aligned} \overline{\mathrm{V}}' &= a_i(1 - \mathrm{B}_i) + b_i\mathrm{B}_i - a_i \\ &= (b_i - a_i)\mathrm{B}_i \end{aligned} \tag{A.7}$$

so that $\overline{\mathrm{X}}'$ has zero failure weights for all elements and can be modelled using the algorithms in this report with $w_i = (b_i - a_i)$. If any $b_i < a_i$ then the weight will be negative and can be dealt with using the method in Section A.2.1.

As such, any three-parameter GBP random variable can converted into a two-parameter random variable by using the difference between the $\boldsymbol{b}$ and $\boldsymbol{a}$ as weights and shifting the support by a constant. The two-parameter version is much easier to work with algorithmically than the three-parameter version, so this project focused on the two-parameter version.

A.2.3 Extension to Non-Integers

For arbitrary real weights the Naïve solutions of building a probability tree are the only available approaches. However, one can use the techniques in this project for a collection of real weights that share a common divisor. By dividing the weights by their common divisor they become integers which can than be used as weights. Any results then need to be suitably rescaled. For example, if one has weights in pounds and pence then one can compute the pmf in pence and then scale the support accordingly.

A.3 Connection to Subset Sum

For the pmf of a GPB random variable to be non-zero at a given point, say x, there must be some subset of the weights which can be added together to reach a total of x. For example, if $\boldsymbol{w} = \begin{bmatrix} 1 & 2 & 5 \end{bmatrix}^\top$ then the pmf must be zero at $x = 4$ regardless of the values in $\boldsymbol{p}$: there is no subset of $\{1, 2, 5\}$ which sums to 4.

In this way the GPB distribution is connected to the classic subset-sum decision problem. This problem asks if, given a multiset of integers $\mathcal{S}$ and a target value t, there is some subset of $\mathcal{S}$ whose total is t[15]. The subset sum problem has complexity $O(2^m)$ for m integers and is one of the original NP-complete problems from Karp's seminal 1972 paper[14]. Calculating the pmf of a GPB random variable must be at least as difficult as answering the subset sum problem for the associated weights, so there is no hope of a polynomial (in m) time algorithm unless P = NP.

The Dynamic Programming and Divide-and-Conquer-FFT algorithms in this work can be considered as modifications of existing ideas from the theory of subset sum. The Dynamic Programming approach is a modification of the classic dynamic-programming subset-sum algorithm by Bellman (1966)[3].

The GPB DC-FFT algorithm is essentially a modification of a standard divide-and-conquer-FFT algorithm for subset sum going back to at least Eppstein (1997)[9], with a general divide-and-conquer approach for omnivolution without FFT dating back to at least Horowitz & Sahni (1974)[11].

If pseudo-polynomial approaches to computing GPB pmfs seem to be linked to subset sum algorithms, then we might hope to adapt the current state-of-the-art for subset sum to the GPB. When computing the full pmf of the GPB (equivalent to finding *all* subset sums) the state of the art is still $O(n \log n \log m)$ [16]. However, if one wishes only to compute all sums up to given target, t, then the state of the art is $O(t \log^{5/2} t\sqrt{m})$ [16]. Potentially, this method could be used to compute the tail of a GPB significantly quicker than the algorithms presenting in this report - we leave exploring this idea further for future work.

Appendix B

Additional Detail on Computing the LFGPB distribution

B.1 Properties of the LFGPB distribution

B.1.1 Mean

The mean of a sum of random variables is the sum of their means. This holds even when the variables are taken from different distributions and/or are not mutually independent. The mean of a LFGPB r.v. can therefore be calculated reasonably simply as

$$\mu = \sum_{j=0}^{m-1} p_j w_j = \boldsymbol{p} \cdot \boldsymbol{w} \tag{B.1}$$

B.1.2 Variance

The variance of a sum of not-necessarily-independent random variables is the sum of the elements of their covariance matrix, i.e.

$$\sigma^2 = \sum_{j=0}^{m-1} \sum_{k=0}^{m-1} \mathrm{Cov}(\mathrm{V}_j, \mathrm{V}_k) \tag{B.2}$$

which can be split out into the main diagonal (where $j = k$), and twice the upper (or lower) triangular matrix:

$$\sigma^2 = \sum_{j=0}^{m-1} \mathrm{Var}(\mathrm{V}_j) + 2 \sum_{0 \leqslant j < k}^{m-1} \mathrm{Cov}(\mathrm{V}_j, \mathrm{V}_k) \tag{B.3}$$

For the terms in the first sum

$$\begin{aligned} \mathrm{Var}(\mathrm{V}_j) &= \mathbb{E}[\mathrm{V}_j^2] - \mathbb{E}[\mathrm{V}_j]^2 \\ &= p_j \cdot w_j^2 - (p_j \cdot w_j)^2 \\ &= w_j^2 \cdot (p_j - p_j^2) \end{aligned} \tag{B.4}$$

For computing the covariance terms it is necessary to evaluate $\mathbb{E}[\mathrm{V}_j\mathrm{V}_k]$, which for an LFGPB r.v. will be zero if either component j or component k are failures; and $w_j w_k$ otherwise, so that

$$\mathbb{E}[\mathrm{V}_j\mathrm{V}_k] = w_j w_k \mathbb{P}[\mathrm{S}_j \leqslant \Phi^{-1}(p_j), \mathrm{S}_k \leqslant \Phi^{-1}(p_k)] = w_j w_k C^{\mathrm{Gauss}}_{\boldsymbol{R}_2}\left(\begin{bmatrix} p_j \\ p_k \end{bmatrix}\right) \tag{B.5}$$

Giving

$$\begin{aligned} \mathrm{Cov}(\mathrm{V}_j, \mathrm{V}_k) &= \mathbb{E}[\mathrm{V}_j\mathrm{V}_k] - \mathbb{E}[\mathrm{V}_j]\mathbb{E}[\mathrm{V}_k] \\ &= w_j w_k C^{\mathrm{Gauss}}_{\boldsymbol{R}_2}\left(\begin{bmatrix} p_j \\ p_k \end{bmatrix}\right) - p_j w_j p_k w_k \\ &= w_j w_k \left[C^{\mathrm{Gauss}}_{\boldsymbol{R}_2}\left(\begin{bmatrix} p_j \\ p_k \end{bmatrix}\right) - p_j p_k\right] \end{aligned} \tag{B.6}$$

Putting Equations B.3, B.4 and B.6 together to finally get

$$\sigma^2 = \sum_{j=0}^{m-1} w_j^2 \cdot (p_j - p_j^2) + 2 \sum_{0 \leqslant j < k}^{m-1} w_j w_k \left[C^{\mathrm{Gauss}}_{\boldsymbol{R}_2}\left(\begin{bmatrix} p_j \\ p_k \end{bmatrix}\right) - p_j p_k\right] \tag{B.7}$$

B.1.3 Higher Order Moments

A similar approach to Equation B.2 can be taken to compute higher-order moments. The skewness can be calculated from the elements of the co-skewness cube 3-tensor of $\mathbf{V}$, and the kurtosis from the equivalent co-kurtosis tesseract 4-tensor.

Recall that variance is the order-2 central moment of a random variable, X, defined by

$$\mathrm{Var}(\mathrm{X}) = \mathbb{E}\left[(\mathrm{X} - \mu_l)^2\right] \tag{B.8}$$

and covariance is the order-2 central *mixed* moment between random variables, defined by

$$\mathrm{Cov}(\mathrm{X}, \mathrm{Y}) = \mathbb{E}\left[(\mathrm{X} - \mu_x)(\mathrm{Y} - \mu_y)\right] \tag{B.9}$$

We can extend this to 3 and 4 random variables via

$$\begin{aligned} \mu_3(\mathrm{X}) &= \mathbb{E}\left[(\mathrm{X} - \mu_x)^3\right] \\ \mu_4(\mathrm{X}) &= \mathbb{E}\left[(\mathrm{X} - \mu_x)^4\right] \end{aligned} \tag{B.10}$$

for the central moments and the Co-Central (or mixed) moments as:

$$\begin{aligned} \mathrm{CoC3}(\mathrm{X}, \mathrm{Y}, \mathrm{Z}) &= \mathbb{E}\left[(\mathrm{X} - \mu_x)(\mathrm{Y} - \mu_y)(\mathrm{Z} - \mu_z)\right] \\ \mathrm{CoC4}(\mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}) &= \mathbb{E}\left[(\mathrm{W} - \mu_w)(\mathrm{X} - \mu_x)(\mathrm{Y} - \mu_y)(\mathrm{Z} - \mu_z)\right] \end{aligned} \tag{B.11}$$

Usefully, as we saw in equation B.2 the variance of the sum of not-necessarily independent random variables in the sum of the elements of their covariance matrix

$$\sigma^2 = \sum_{j=0}^{m-1} \sum_{k=0}^{m-1} \mathrm{Cov}(\mathrm{V}_j, \mathrm{V}_k)$$

This idea also extends to the 3rd order so that

$$\mu_3 = \sum_{j=0}^{m-1}\sum_{k=0}^{m-1}\sum_{\ell=0}^{m-1} \mathrm{Co3}(\mathrm{V}_j, \mathrm{V}_k, \mathrm{V}_\ell) \tag{B.12}$$

And indeed in general to the kth order

$$\mu_n = \sum_{\boldsymbol{s}\in[\![0:m-1]\!]^k} \mathrm{CoCk}(\mathrm{V}_{s_1}, \mathrm{V}_{s_2}, \ldots, \mathrm{V}_{s_k}) \tag{B.13}$$

We can also generalise the CoCk function by looking at the expanding form, for example for CoC2 (i.e. variance) and CoC3 they expand to:

$$\begin{aligned}\mathrm{CoC2}(\mathrm{X},\mathrm{Y}) &= \mathbb{E}\left[(\mathrm{X}-\mu_x)(\mathrm{Y}-\mu_y)\right] \\ &= \mathbb{E}[\mathrm{XY}] - \mathbb{E}[\mathrm{X}]\mu_y - \mathbb{E}[\mathrm{Y}]\mu_x + \mu_x\mu_y\end{aligned} \tag{B.14}$$

and

$$\begin{aligned}\mathrm{CoC3}(\mathrm{X},\mathrm{Y},\mathrm{Z}) =& \mathbb{E}\left[(\mathrm{X}-\mu_x)(\mathrm{Y}-\mu_y)(\mathrm{Z}-\mu_z)\right] \\ =& \mathbb{E}[\mathrm{XYZ}] - \mathbb{E}[\mathrm{XY}]\mu_z - \mathbb{E}[\mathrm{XZ}]\mu_y - \mathbb{E}[\mathrm{YZ}]\mu_x \\ &+ \mathbb{E}[\mathrm{X}]\mu_y\mu_z + \mathbb{E}[\mathrm{Y}]\mu_x\mu_z + \mathbb{E}[\mathrm{Z}]\mu_x\mu_y - \mu_x\mu_y\mu_z\end{aligned} \tag{B.15}$$

And so in general where $\mathscr{V}_{\boldsymbol{s}} = \{\mathrm{V}_{s_1}, \mathrm{V}_{s_2}, \ldots, \mathrm{V}_{s_k}\}$ is a multiset[1] of elements of $\mathbf{V}$ we have

$$\mathrm{CoC}k(\mathscr{V}_{\boldsymbol{s}}) = \sum_{\mathscr{U}\in\mathsf{Pow}(\mathscr{V}_{\boldsymbol{s}})} \left((-1)^{k-|\mathscr{U}|} \cdot \mathbb{E}\left[\prod_{\mathrm{U}_i\in\mathscr{U}} \mathrm{U}_i\right]\left[\prod_{\mathrm{Y}_i\in\mathscr{V}_{\boldsymbol{s}}\setminus\mathscr{U}} \mathbb{E}[\mathrm{Y}_i]\right]\right) \tag{B.16}$$

Let $\mathscr{P}_{\boldsymbol{s}} = \{p_{s_1}, p_{s_2}, \ldots, p_{s_k}\}$ be a multiset of probabilities taken from $\boldsymbol{p}$. For the LFGPB the equation for CoCk becomes

$$\mathrm{CoC}k(\mathscr{V}_{\boldsymbol{s}}) = \left[\prod_{i\in\boldsymbol{s}} w_i\right] \sum_{\mathbb{Q}\in\mathsf{Pow}(\mathscr{P}_{\boldsymbol{s}})} \left[(-1)^{n-|\mathbb{Q}|} \cdot C_{\boldsymbol{R}}\left(\mathrm{Supp}(\mathbb{Q})\right) \prod_{p_{s_j}\in\mathscr{P}\setminus\mathbb{Q}} p_{s_j}\right] \tag{B.17}$$

Where $\mathrm{Supp}(\mathbb{Q})$ is the support of $\mathbb{Q}$[2]. Giving the full equation of the kth central moment of the LFGPB as

$$\mu_k = \sum_{\boldsymbol{s}\in[\![0:m-1]\!]^k} \left(\left[\prod_{i\in\boldsymbol{s}} w_i\right] \sum_{\mathbb{Q}\in\mathsf{Pow}(\mathscr{P}_{\boldsymbol{s}})} \left[(-1)^{k-|\mathbb{Q}|} \cdot C_{\boldsymbol{R}}(\mathrm{Supp}(\mathbb{Q})) \prod_{p_{s_j}\in\mathscr{P}\setminus\mathbb{Q}} p_{s_j}\right]\right) \tag{B.18}$$

B.1.4 Parameter Estimation

With a way to compute the variance from a known correlation, the "method of moments" can be used as a simple way to estimate the correlation of a sample from its variance, i.e. numerically solve for the value of ρ such that the expected variance equals the sample variance. Parameters estimated by this method do generally show some bias and a Maximum-Likelihood approach would be preferable. However, such an approach was not derived as part of this project.

[1] i.e. a set where the elements can occur more than once

[2] i.e. the set of all distinct elements in $\mathbb{Q}$. For this purpose we consider two indexed items of $\boldsymbol{p}$ to be distinct even if they have the same value; they will both appear in $\mathrm{Supp}(\mathbb{Q})$

B.2 Linear Algebra Approach

The probability tree method from Figure 2.1 cannot be used directly for the LFGPB because multiplying together success probabilities to get a joint-success probability only works when the component random variables are mutually independent. However, if it is possible to calculate the outcome probabilities in some other fashion then the technique can still be used. This Linear Algebra Approach sets out one way that the outcome probabilities might be computed.

Outcome Probabilities

When talking about outcome probabilities, notation like p^o_{12} will be used to denote the outcome that elements 1 and 2 are a success and all other elements are failures, e.g. for an example where $m = 3$

$$\begin{aligned} p^o_{12} &= \mathbb{P}[\mathrm{B}_0{=}0, \mathrm{B}_1{=}1, \mathrm{B}_2{=}1] \\ &= \mathbb{P}[\mathrm{S}_0 > \Phi^{-1}(p_0), \mathrm{S}_1 \leqslant \Phi^{-1}(p_1), \mathrm{S}_2 \leqslant \Phi^{-1}(p_2)] \end{aligned} \tag{B.19}$$

For a GPB or LFGPB random variable with $m = 3$ there are $8 = 2^3$ possible outcomes. They can be visualised as in the Venn diagram in Figure B.1. Generally, software packages do not provide

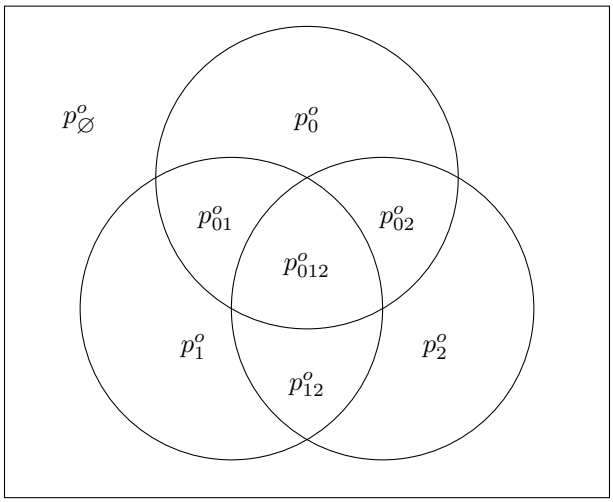

Figure B.1: Venn diagram showing all outcome probabilities

functions for computing probabilities like that seen in Equation B.19 where one wishes to know the probability that some variables are above a value and others are below a value. Instead the joint-cdf is provided which considers only situations where all variables are below an associated value. In the case of a LFGPB random variable the joint-cdf provides joint-success probabilities for some subset of the component random variables, i.e. a joint-success "event".

Event Probabilities

For short we will refer to these joint-success-event probabilities as *event* probabilities. For example, p^e_2 is the event probability that element 2 will be a success, whether or not elements 1 and 3 are also a success. This event is shown as the shaded area in Figure B.2, from which it can be seen that $p^e_2 = p^o_{012} + p^o_{12} + p^o_{02} + p^o_2$. Similarly, p^e_{12} is the event probability that elements 1 and 2 are a success, this means $p^e_{12} = p^o_{012} + p^o_{12}$.

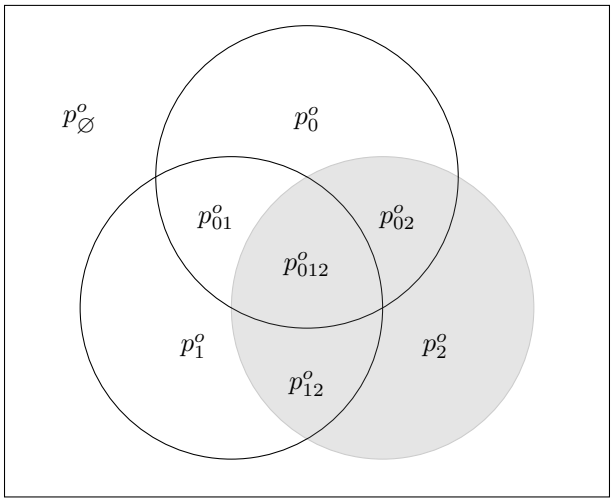

Figure B.2: Venn diagram showing the outcome probabilities corresponding to the event probability p^e_2

Computing Event Probabilities

These event probabilities can be computed using the Gaussian copula, for example

$$\begin{aligned} p^e_{12} &= \mathbb{P}[\mathrm{B}_1{=}1, \mathrm{B}_2{=}1] \\ &= C^{\text{Gauss}}_{\boldsymbol{R}_2}\left(\begin{bmatrix} p_1 \\ p_2 \end{bmatrix}\right) \end{aligned} \tag{B.20}$$

As the LFGPB relies on an equicorrelation matrix the Gaussian copula can be computed somewhat more efficiently and accurately than standard generic library implementations provide (`SciPy`'s multivariate-normal cdf implementation seemingly uses Monte-Carlo methods). This faster and more accurate method was implemented for use in this project, see Appendix B.5 for details.

Consider the joint success probability, $p^e_{\mathcal{S}}$ of some subset of indices, $\mathcal{S} \subseteq [\![0 : m-1]\!]$. Let the associated marginal (i.e. single component random variable) success probabilities $\boldsymbol{p}^{\mathcal{S}}$ be defined such that $p^{\mathcal{S}}_i = p_{s_i}$.

For example, if a LFGPB random variable has probability parameter $\boldsymbol{p} = \begin{bmatrix} 0.1 & 0.3 & 0.4 & 0.2 \end{bmatrix}^\top$, it has indices $[\![0 : 3]\!] = \{0, 1, 2, 3\}$. Suppose that $\mathcal{S} = \{1, 3\}$ which is a subset of these indices. Then $\boldsymbol{p}^{\mathcal{S}} = \begin{bmatrix} 0.3 & 0.2 \end{bmatrix}^\top$

The event probability of joint-success for the component random variables whose indices are in $\mathcal{S}$ will be

$$\begin{aligned} p^e_{\mathcal{S}} &= \mathbb{P}[\mathrm{B}_i = 1 \ \forall \ i \in \mathcal{S}] \\ &= C^{\text{Gauss}}_{\boldsymbol{R}_{|\mathcal{S}|}}\left(\boldsymbol{p}^{\mathcal{S}}\right) \end{aligned} \tag{B.21}$$

The relationship between Outcome and Event probabilities

All possible outcomes (and also all possible joint-success events) can be enumerated as the elements of the powerset of $[\![0 : m-1]\!]$. A powerset of a set, $\mathsf{Pow}(\mathcal{M})$, is the set of all subsets of $\mathcal{M}$. For example,

$$\mathsf{Pow}(\{0, 1, 2\}) = \{\varnothing, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\} \tag{B.22}$$

which corresponds to the set of all index combinations of the probabilities in Figures B.1 and B.2.

To provide an order to these sets of indices, let the binary operator $\lhd$ be an arbitrary strict total order[3] over the powerset of integers, allowing the definition of the sequence $(\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{2^m-1})$ where

$$\mathcal{S}_i \in \mathsf{Pow}(\llbracket 0 : m-1 \rrbracket), \qquad 0 \leqslant i < 2^m \tag{B.23}$$

and

$$\mathcal{S}_i \lhd \mathcal{S}_{i+1} \lhd \mathcal{S}_{i+2} \ldots \tag{B.24}$$

With a way to order the event and outcome probabilities it is possible to lay their relationship out in a table, as in table B.1, where, the ith row corresponds to $p^e_{\mathcal{S}_i}$ and the jth column corresponds to $p^o_{\mathcal{S}_j}$. The entries show which outcomes are elements of which event, for example because $p^e_{12} = p^o_{012} + p^o_{12}$ there is a 1 in the cells where p^e_{12} intersect with p^o_{012} and p^o_{12}.

	$p^o_\varnothing$	p^o_0	p^o_1	p^o_2	p^o_{01}	p^o_{02}	p^o_{12}	p^o_{012}
$p^e_\varnothing$	1	1	1	1	1	1	1	1
p^e_0	0	1	0	0	1	1	0	1
p^e_1	0	0	1	0	1	0	1	1
p^e_2	0	0	0	1	0	1	1	1
p^e_{01}	0	0	0	0	1	0	0	1
p^e_{02}	0	0	0	0	0	1	0	1
p^e_{12}	0	0	0	0	0	0	1	1
p^e_{012}	0	0	0	0	0	0	0	1

Table B.1: relationship between outcome and event probabilities

The value of cell $a_{i,j}$ in row i and column j in table B.1 is

$$a_{i,j} = \begin{cases} 1 & \text{if } \mathcal{S}_i \subseteq \mathcal{S}_j \\ 0 & \text{otherwise} \end{cases} \tag{B.25}$$

Table B.1 and the relationship between outcome probabilities and event probabilities can be expressed using the language of linear algebra with an upper-triangular matrix as

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ & & 1 & 0 & 1 & 0 & 1 & 1 \\ & & & 1 & 0 & 1 & 1 & 1 \\ & & & & 1 & 0 & 0 & 1 \\ & & & & & 1 & 0 & 1 \\ & & & & & & 1 & 1 \\ & & & & & & & 1 \end{bmatrix} \begin{bmatrix} p^o_\varnothing \\ p^o_0 \\ p^o_1 \\ p^o_2 \\ p^o_{01} \\ p^o_{02} \\ p^o_{12} \\ p^o_{012} \end{bmatrix} = \begin{bmatrix} p^e_\varnothing \\ p^e_0 \\ p^e_1 \\ p^e_2 \\ p^e_{01} \\ p^e_{02} \\ p^e_{12} \\ p^e_{012} \end{bmatrix} \tag{B.26}$$

or more compactly

$$\mathbf{A}\boldsymbol{p^o} = \boldsymbol{p^e} \tag{B.27}$$

which means the values of $\boldsymbol{p^o}$ can easily be computed from $\boldsymbol{p^e}$ using standard linear algebra techniques via

$$\boldsymbol{p^o} = \mathbf{A}^{-1}\boldsymbol{p^e} \tag{B.28}$$

[3] a strict total order is a binary relation that allows one to compare any two elements of a set and specify which one "comes first". Although any arbitrary order will suffice (so long as it is consistently applied to order both outcome and event probabilities), for this example an ordering based on cardinality and nth-smallest element has been used.

Similarly, an associated vector of outcome total weights can be expressed as

$$\begin{bmatrix} w^o_{\varnothing} \\ w^o_0 \\ w^o_1 \\ w^o_2 \\ w^o_{01} \\ w^o_{02} \\ w^o_{12} \\ w^o_{012} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \tag{B.29}$$

or more compactly

$$\boldsymbol{w}^o = \boldsymbol{\Omega}\boldsymbol{w} \tag{B.30}$$

where the value $\omega_{i,j}$ in row i and column j of matrix $\boldsymbol{\Omega}$ is given by

$$\omega_{i,j} = \begin{cases} 1 & \text{if } j \in \mathcal{S}_i \\ 0 & \text{otherwise} \end{cases} \tag{B.31}$$

By aggregating the elements of $\boldsymbol{w}^o$ and $\boldsymbol{p}^o$ as in Equation 2.14 we get the pmf of the LFGPB distribution, i.e.

$$\lambda(x) = \sum_{\{p^o_i \in \boldsymbol{p}^o | w^o_i = x\}} p^o_i \tag{B.32}$$

Complexity Analysis: As mentioned in section 2.2, there are 2^m outcomes for a problem with m elements. This requires computing 2^m event probabilities first, i.e. the vector $\boldsymbol{p^e}$. The triangular matrix $\mathbf{A}$ is $2^m \times 2^m$. Generally a linear-algebraic system involving a $k \times k$ triangular matrix can be solved in $O(k^2)$, so overall, complexity is $O(2^m) + O(2^{m^2}) = O(2^{m^2})$, and so quickly becomes untenable, but the method is possible for small m, i.e. no more than 10.

B.3 Gauss Kronrod Quadrature

B.3.1 Overview of GK Quadrature

The idea of GK quadrature is to approximate a definite integral of a function as a weighted sum of the value of the function at particular points [6], traditionally expressed over the interval $[-1, 1]$

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=1}^{n} \omega_i f(x_i) \tag{B.33}$$

and more generically over the interval $[a, b]$

$$\int_{a}^{b} f(x)\mathrm{d}x \approx \frac{b-a}{2} \sum_{i=1}^{n} \omega_i f\left(\frac{b-a}{2}x_i + \frac{b+a}{2}\right) \tag{B.34}$$

The points x_i and weights ω_i are generic for use with any function; they can be retrieved from online calculators such as [6]. In the version of GK approach we implemented there are 15 nodes, $\mathcal{X}^{15}$ with 15 associated "Kronrod" weights, $\mathcal{W}^{15}$. The *odd* nodes of $\mathcal{X}^{15}$ form a subset of 7 nodes, $\mathcal{X}^7$, which also have "Gauss" weights, $\mathcal{W}^7$. This gives two approximations to the integral, $K15$ and $G7$, and an estimate of the current error in the integral is given by $|K15 - G7|$. The nodes and weights can be seen in Table B.2 (n.b. more precision than shown was used).

i	Nodes	Kronrod Weights	Gauss Weights
0	-0.9914553711	0.0229353220	
1	-0.9491079123	0.0630920926	0.1294849662
2	-0.8648644234	0.1047900103	
3	-0.7415311856	0.1406532597	0.2797053915
4	-0.5860872355	0.1690047266	
5	-0.4058451514	0.1903505781	0.3818300505
6	-0.2077849550	0.2044329401	
7	0.0000000000	0.2094821411	0.4179591837
8	0.2077849550	0.2044329401	
9	0.4058451514	0.1903505781	0.3818300505
10	0.5860872355	0.1690047266	
11	0.7415311856	0.1406532597	0.2797053915
12	0.8648644234	0.1047900103	
13	0.9491079123	0.0630920926	0.1294849662
14	0.9914553711	0.0229353220	

Table B.2: Nodes and weights for Gauss Kronrod Quadrature

B.3.2 Adaptive Quadrature

To get an approximation of the integral to sufficient precision we needed to apply the above approximation recursively as shown in Algorithm 9

Algorithm 9: Adaptive Gauss-Kronrod Quadrature

Function `GK-Quad-Vec`$(\mathbf{f}, a, b, \varepsilon)$**:**

$\boldsymbol{I}_K \leftarrow \frac{b-a}{2} \sum_{i=0}^{14} \mathcal{W}_i^{15} \mathbf{f}\left(\frac{b-a}{2}\mathcal{X}_i^{15} + \frac{b+a}{2}\right)$

$\boldsymbol{I}_G \leftarrow \frac{b-a}{2} \sum_{i=0}^{6} \mathcal{W}_i^{7} \mathbf{f}\left(\frac{b-a}{2}\mathcal{X}_i^{7} + \frac{b+a}{2}\right)$

if $\max(\text{abs}(\boldsymbol{I}_K - \boldsymbol{I}_G)) \leqslant \varepsilon$ **then**

  return $\boldsymbol{I}_K$

else

  return `GK-Quad-Vec`$(\mathbf{f}, a, \frac{b+a}{2}, \frac{\varepsilon}{2})$ + `GK-Quad-Vec`$(\mathbf{f}, \frac{b+a}{2}, b, \frac{\varepsilon}{2})$

end

B.4 Integration of the Characteristic Function

We briefly mention here an approach to computing the LFGPB pmf that was explored but abandoned. The approach that Zhang et al.[26] took to the GPB was via the characteristic function. See Appendix A.1 for a explanation of this approach. It leads to the following closed-form expression for the GPB (equivalent to Equation A.2)

$$g(x) = \frac{1}{n} \sum_{k=0}^{n-1} \exp\left(2i\pi \tfrac{kx}{n}\right) \prod_{j=0}^{m-1} 1 + p_j \cdot \left[\exp\left(-2i\pi \tfrac{kw_j}{n}\right) - 1\right] \tag{B.35}$$

By combining with Equation 5.11 we get the following closed-form expression for the pmf of a LFGPB random variable:

$$\lambda(x;\boldsymbol{p},\boldsymbol{w},\rho) = \int_0^1 \frac{1}{n}\sum_{k=0}^{n-1} \exp\left(2i\pi\tfrac{kx}{n}\right) \prod_{j=0}^{m-1} 1+\kappa(z;\boldsymbol{p},\rho)_j \cdot \left[\exp\left(-2i\pi\tfrac{kw_j}{n}\right)-1\right] \mathrm{d}z \tag{B.36}$$

which, via the linearity of the Fourier Transform, can also be expressed as

$$\lambda(x;\boldsymbol{p},\boldsymbol{w},\rho) = \frac{1}{n}\sum_{k=0}^{n-1} \exp\left(2i\pi\tfrac{kx}{n}\right) \int_0^1 \prod_{j=0}^{m-1} 1+\kappa(z;\boldsymbol{p},\rho)_j \cdot \left[\exp\left(-2i\pi\tfrac{kw_j}{n}\right)-1\right] \mathrm{d}z \tag{B.37}$$

leading to the possibility of an approach involving a numerical contour integral over the complex plane. Experiments were done to see if this was feasible, however convergence was exceedingly slow. As such, this line of attack was abandoned and not implemented in the final project files.

B.5 Computation of the Gaussian Copula with an Equicorrelation Matrix

Generally there is not an easy way to compute the cdf of the multivariate normal distribution. The implementation in `SciPy` produces stochastic results, so presumably uses Monte-Carlo methods for integration. The standard level of accuracy from SciPy is around 1×10^{-6} with significant degradation in speed when further precision is required. However, according to [17] there is a simpler form of the cdf that can be used whenever the correlation matrix has the form $\rho_{ij} = b_i b_j (i \neq j)$ with $\boldsymbol{b} \in [-1,1]^m$. When considering our LFGPB random variables we do have this form with $b_i = \sqrt{\rho},\ \forall\ 0 \leqslant i < m$. Where $\boldsymbol{q} \subseteq \boldsymbol{p}$ is a sequence of probabilities corresponding to the successful events, we have:

$$C^{\text{Gauss}}_{\boldsymbol{R}_{\dim(\boldsymbol{q})}}(\boldsymbol{q}) = \int_{-\infty}^{\infty} \phi(u) \prod_{q_i \in \boldsymbol{q}} \Phi\left(\frac{\Phi^{-1}(q_i) - u\sqrt{\rho}}{\sqrt{1-\rho}}\right) \mathrm{d}u \tag{B.38}$$

This technique meant that the Linear-Algebra approach to the LFGPB from section B.2 was both significantly faster and more accurate than otherwise. Similarly for calculating LFGPB moments without first computing the whole pmf.

Appendix C

The pyGPB Python Library

The algorithms discussed in Chapters 4 and 5 have been implemented and provided in the pyGPB library for Python.

C.1 Package Structure

The package is put together as a standard python package. The key files are below.

```
base-pyGPB                  --> Base Folder
|-- pyGPB                     --> Actual Module
  |-- tests                     --> Unit Tests
    |-- test_cpu_methods.py       --> Testing for CPU algorithms
    |-- test_GPB.py               --> Testing for GPB class
    |-- test_gpu_methods.py       --> Testing for GPU algorithms
    |-- test_LFGPB.py             --> Testing for LFGPB class
  |-- __init__.py.py            --> Speficies the classes available for pyGPB
  |-- _cpu_methods.py           --> CPU algorithms
  |-- _gpu_methods.py           --> GPU algorithms, only loaded where user has a CUDA GPU
  |-- _tools.py                 --> Functions needed by both CPU and GPU algorithsm
  |-- pyGPU.py                  --> GPB and LFGPB classes
|-- docs                      --> Documentation
  |-- various                   --> Documentation files
```

C.2 Distribution and Installation

This package will be made available for others to use on The Python Package Index (PyPI), which "is a repository of software for the Python programming language"[21]. The upload has been kept on hold until the completion of this project lest there be any chance of confusion around plagiarism. Once uploaded the package will be installable with the following command in the terminal

```
pip install pyGPB
```

Currently, the package can be installed from either the tarball (`pyGPB-0.0.1.tar.gz`), or wheels (`pyGPB-0.0.1-py3-none-any.whl`) files in the `dist` folder in the project repository by downloading the file, navigating to its location and running the following in the terminal (for e.g. the wheels)

```
pip install pyGPB-0.0.1-py3-none-any.whl
```

C.3 A brief look at the pyGPB API

The package exposes two classes: `GPB` and `LFGPB`. Both of these are sub-classed from `SciPy`'s `rv_discrete`, which is the Python standard for representing discrete random variables. The classes take distribution parameters (e.g. an array of probabilities and an array of weights) in the constructor and provides objects which exposure various methods for computing with the random variables. The package allows for alternative parametrisation of weights, including negative weights and the specification of *failure* weights along with success weights. See Appendix A.2 for a discussion of how this is done. Listing 1 gives a demonstration of how the package might be used.

```
>>> from pyGPB import GPB, LFGPB
>>> x = GPB(probs=[0.1,0.2,0.3], weights=[1,2,3])
>>> # See the probability mass function as a vector
>>> x.pmf_vec
array([0.504, 0.056, 0.126, 0.23 , 0.024, 0.054, 0.006])

>>> # See the cumulative distribution function as a vector
>>> x.cdf_vec
array([0.504, 0.56 , 0.686, 0.916, 0.94 , 0.994, 1.   ])

>>> # Generate some random variates, sampled from the distribution
>>> x.rvs(size=(3,6))
array([[1, 2, 0, 0, 5, 0],
       [0, 2, 3, 0, 0, 0],
       [3, 0, 3, 3, 0, 4]])

>>> # Calculate the Entropy of the distribution
>>> x.entropy()
array(1.3835984)

>>> # Get the pmf and cdf at specific points
>>> x.pmf([0,2,4]), x.pmf([1,3,5])
(array([0.504, 0.126, 0.024]), array([0.056, 0.23 , 0.054]))

>>> # Get the mean, variance, skewness and kurtosis
>>> x.stats(moments='mvsk')
(array(1.4), array(2.62), array(0.73287359), array(-0.62333197))

>>> # Check the support
>>> x.support()
(0, 6)

>>> # Compute the expected value of some function of x
>>> x.expect(lambda x: x**3-x**2)
12.276000000000002

>>> # Generate a variable with both success and failure weights (even negative)
>>> x = GPB(probs=[0.1,0.2,0.3], weights=[-1,2,-3], failure_weights=[0,4,2])
>>> # Check support and pmf
>>> x.support(), x.pmf_vec
((-2, 6),
 array([0.006, 0.054, 0.024, 0.216, 0.   , 0.014, 0.126, 0.056, 0.504]))

>>> # Generate a variable which enforces CPU use only
>>> x = GPB(probs=[0.1,0.2,0.3], weights=[1,2,3], allow_GPU=False)
```

```
>>> # Generate a variable which prioritises accuracy over speed (i.e. avoids FFT)
>>> x = GPB(probs=[0.1,0.2,0.3], weights=[1,2,3], prefer_speed=False)

>>> # Allow for correlation in a variable
>>> x = LFGPB(probs=[0.1,0.2,0.3], weights=[1,2,3], rho=0.5)
>>> x.pmf_vec
array([0.59292935, 0.02231788, 0.07241397, 0.17090654, 0.02618503,
       0.07608894, 0.03915829])
```

Listing 1: Example of use for the pyGPB package

C.4 Documentation

The package is documented using `Sphinx`, an open-source documentation generator[5] with a theme provided by Read the Docs, an open-source documentation hosting platform[22]. This enables generation of documentation directly from the docstrings included in each class, method etc.. For example, the start of the docstring for the `GPB` class is given in Listing 2. Note the `:param name:` and `:type name:` tags which tell Sphinx details about the class at the same time as providing documentation to those reading the code. Similar doc strings are used for all public classes and their members.

```
class GPB(rv_sample):
    """A Generalised Poisson Binomial random variable.

    Distribution paramters can be supplied in either 2-parameter (p, w) or 3-parameter (p, w_s, w_f) form:

    **2-Parameter Form**:

    Requires probs and weights. Weights should be non-zero. An event failure is assumed to have zero weight.

    **3-Parameter Form**:

    Requires probs, weights and failure_weights. Success and Failure weights for each event must be different.

    :param probs: Success probabilities. Must all be between 0 and 1 (exclusive)
    :type probs: 1D :class:`numpy.ndarray` (or other array-like) of float
    :param weights: Success weights. If failure_weights is not provided then must all be non-zero
    :type weights: 1D :class:`numpy.ndarray` (or other array-like) of int
    :param failure_weights: Failure weights. If provided then must be different to weights (elementwise)
    :type failure_weights: 1D :class:`numpy.ndarray` (or other array-like) of int, optional
```

Listing 2: Beginning of docstring for the GPB class

Using this framework, it is possible to generate very beautiful documentation for the `pyGPB` package using very minimal markup in the reStructuredText language as show in Listing 3. Impressively, Sphinx is able to include methods inherited from classes in external libraries, so they can include the methods inherited from `SciPy` by reference.

```
.. autoclass:: pyGPB.GPB
   :members:
   :show-inheritance:
   :inherited-members:
```

Listing 3: Markup for generating docs for the GPB class in the reStructuredText language using Sphinx

Screenshots of the documentation can be seen in Figures C.1 and C.2. The html files are included in the project repository.

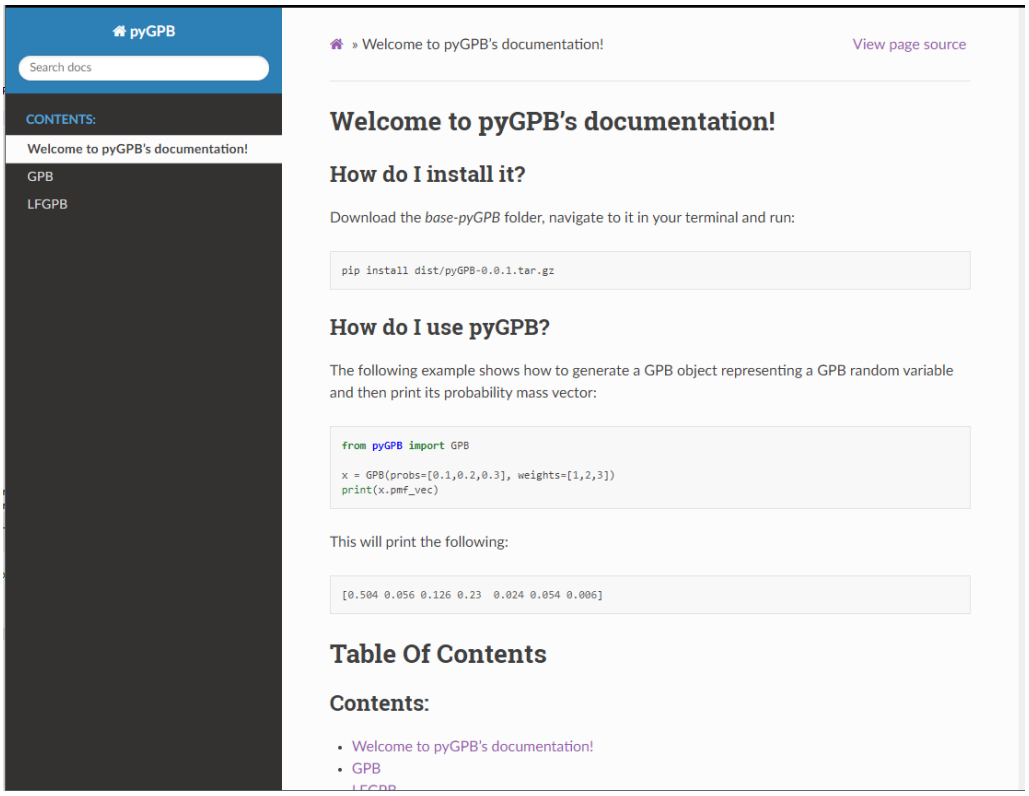

Figure C.1: Homepage of the pyGPB library docs

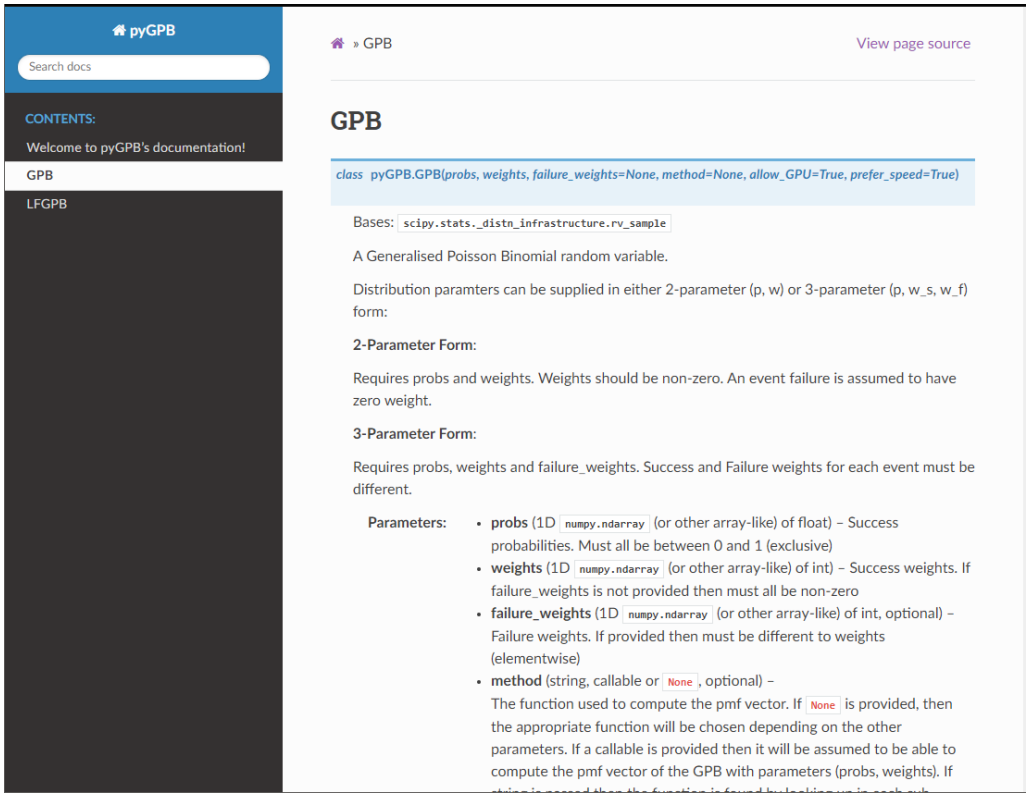

Figure C.2: Documentation of the GPB class

Appendix D

Additional Detail on Testing

D.1 Test Case Generation

The parameter space for the pmf-calculating algorithms is very large, so a gamut of test scenarios was required. Ultimately, two sets of test scenarios were generated: one over a large range of n and m, and one with smaller values of m and n that could be used for benchmarking against the Naive $O(2^m)$ approach. These test dimensions are laid out in the following subsections

D.1.1 Test Space for Number of Components, m

For the large test set there were 25 values used between 10 and 10,000, they were: 10, 13, 17, 23, 31, 42, 56, 74, 100, 133, 177, 237, 316, 421, 562, 749, 1,000, 1,333, 1,778, 2,371, 3,162, 4,216, 5,623, 7,498 and 10,000. Logarithmic spacing was chosen so that they would appear evenly spaced on log-log plots and provide a better understanding of behaviour over several orders of magnitude.

For the smaller test set there were five values used: 5, 10, 15, 20, 25

D.1.2 Test Space for Total Weight, n

For the large test set there were 25 values used between 1,000 and 10,000,000 they were: 1,000, 1,467, 2,154, 3,162, 4,641, 6,812, 10,000, 14,677, 21,544, 31,622, 46,415, 68,129, 100,000, 146,779, 215,443, 316,227, 464,158, 681,292, 1,000,000, 1,467,799, 2,154,434, 3,162,277, 4,641,588, 6,812,920 and 10,000,000. Again these are logarithmically evenly spaced.

For the smaller test set there were seven values used: 100, 200, 500, 1,000, 2,000, 5,000, 10,000

D.1.3 Weights

Collections of weights of the relevant cardinality, m, and total, n, were generated from three distributions corresponding to three scenarios:

- Weights distributed roughly uniform: Uniform Distribution
- Weights clustered close to 1: Geometric Distribution
- Weights clustered around the mean: Poisson Distribution

A direct method of generating collections of m integers from a distribution with a predefined sum, n, was not found. If the relevant distribution is parametrised with the appropriate mean then it is likely that the sum n will be close, but not necessarily close enough for these purposes. Matters were complicated further by the need for there to be a minimum weight of 1.

To solve this problem, a brute-force approach was employed: continuously generating vectors of weights sampled from the given distribution. Vectors were kept if the total weight was within a given tolerance of n, otherwise they were discarded. Vector generation continued until enough vectors had been generated. For all combinations of m, n and distribution there were 16 test cases generated. This process is outlined in algorithm 10

Algorithm 10: Generating Collections of Weights with a Target Sum

Parameter: ε, the tolerance for accepting a putative vector's total weight

Function `GenerateWeightVectorTestCases`$(m, n, \mathcal{D})$**:**

$\mathcal{T} \leftarrow \varnothing$ `/* set of test cases */`

while $|\mathcal{T}| < 16$ **do**

$\boldsymbol{w} \leftarrow$ sample m weights from distribution $\mathcal{D}$ with mean $\frac{n}{m}$

if abs $\left(\frac{\sum_i w_i}{n} - 1\right) < \varepsilon$ **then**

$\mathcal{T} \leftarrow \mathcal{T} \cup \{\boldsymbol{w}\}$

end

end

return $\mathcal{T}$

Figure D.1 gives an idea of the distribution of weights for the Poisson and Geometric scenarios.

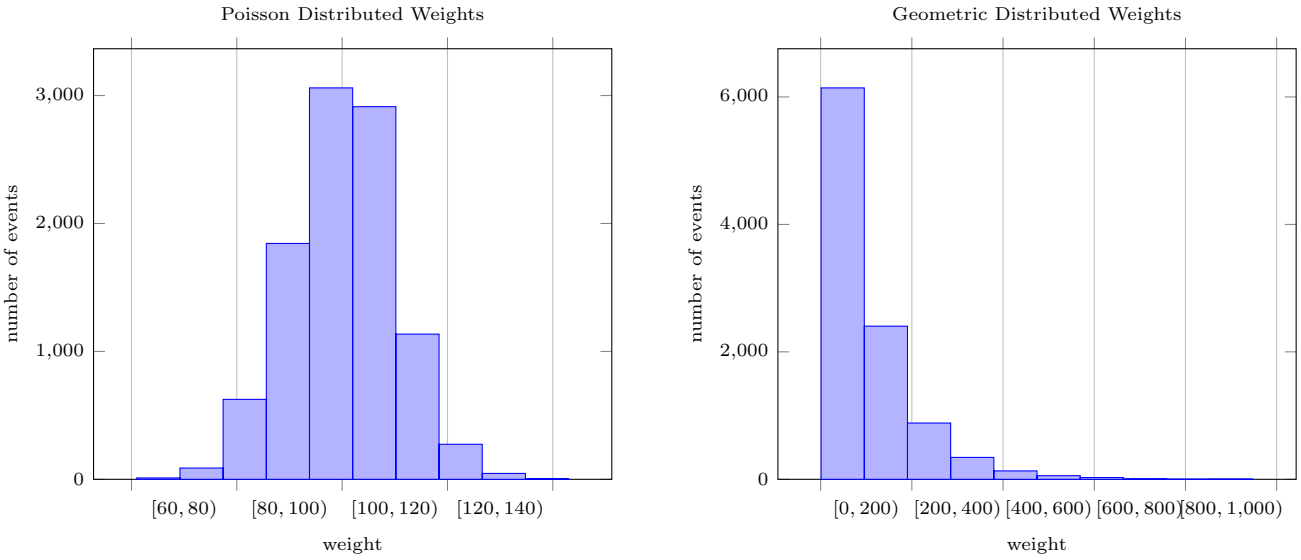

Figure D.1: Histograms of Weights for two particular test cases where $m = 10,000, n = 1,000,000$.

Note on sampling from the Uniform Distribution

In general it was necessary to have mean weight close to $\mu = \frac{n}{m}$ with a minimum weight of 1, and therefore a maximum weight of $2\mu - 1$. When sampling integers from the *discrete* uniform distribution the upper bound must be specified as an integer, requiring a choice of either $\lceil 2\mu - 1 \rceil$ or $\lfloor 2\mu - 1 \rfloor$, both of which produce total weights far from n when $\frac{n}{m} \approx 1$. Instead, sampling was performed such that for weight w_i

$$w_i \sim \begin{cases} \mathcal{U}[\![1, \lfloor 2\mu - 1 \rfloor]\!] & \text{if} \quad i \geqslant m \cdot (2\mu - 1 - \lfloor 2\mu - 1 \rfloor) \\ \mathcal{U}[\![1, \lceil 2\mu - 1 \rceil]\!] & \text{if} \quad i < m \cdot (2\mu - 1 - \lfloor 2\mu - 1 \rfloor) \end{cases} \tag{D.1}$$

Note on sampling from the Poisson Distribution

Sampling with $w_i \sim \text{Poisson}(n/m)$ can produce weights with the value zero, which was to be avoided, instead they were sampled such that $w_i - 1 \sim \text{Poisson}((n-m)/m)$.

D.1.4 Success Probabilities

Collections of success probabilities of the relevant size, m, were generated from three distributions corresponding to three scenarios:

- Probabilities distributed roughly uniform: $p_i \sim \mathcal{U}(0, 1)$
- Probabilities clustered close to 0: $p_i \sim \text{Beta}(0.5, 5)$
- Probabilities clustered around the 1: $p_i \sim \text{Beta}(5, 0.5)$

To avoid any probability having the exact values 0 or 1, a similar approach was used to Algorithm 10. Figure D.2 gives an idea of the distribution of probabilities for the "clustered close to " 0 and 1 scenarios.

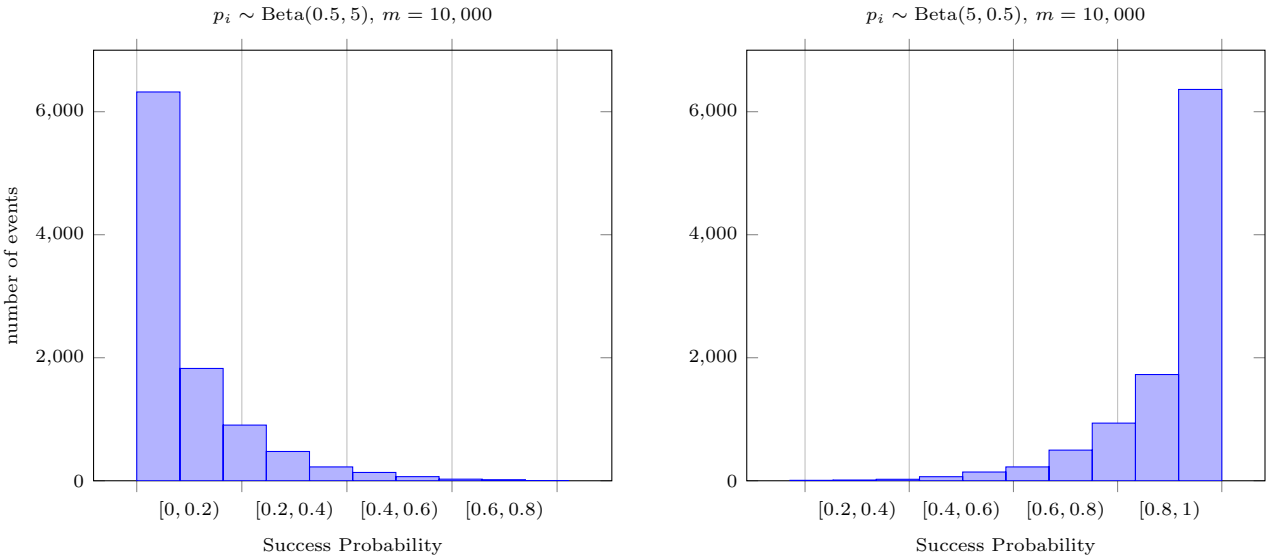

Figure D.2: Histograms of Success Probabilities for two particular test cases where $m = 10,000$.

D.2 Relative Accuracy Issues for FFT approaches

Absolute accuracy testing shows that the GPB pmf vector values calculated by the `pyGPB` libarary tend to be within $\varepsilon \approx 10^{-15}$ or so of the correct value, roughly representing floating point precision of values close to 1. Unfortunately, this level of absolute accuracy is not sufficient for much smaller probabilities. The two FFT approaches (on and off GPU) suffer from relative accuracy issues for small probabilities, generally those at the start and end of pmfs of GPB variables with large m. This effect can be seen in figure D.3, which shows the pmf of a test case with $m = 100, n = 10,000$ computed via the DP and DC-FFT approaches. The DC-FFT pmf is roughly horizontal in the tails with values around 10^{-20}.

This will be an issue with any Fourier-Transform based approach using floating-point arithmetic. Each value in the "time" domain is a function of every value in the "frequency" domain and so the maximum precision will be relative to the largest value across the whole pmf. An absolute precision of around 10^{-20} is roughly what one might expect for 16-byte (i.e. quadruple-precision)

floats. In this case we are using 16-btye complex floats (comprised of two 8-byte double-precision floats), so it is not surprising that the absolute error is roughly the same.

Unfortunately, absolute errors around 10^{-20} on probabilities as small as 10^{-50} or smaller lead to very large relative errors, where relative error in this case is defined as:

$$\varepsilon_{rel}(x) = \frac{|g_x^* - g_x|}{g^*} \tag{D.2}$$

Where, as in Equations 7.1 and 7.2 the asterisk indicates the benchmark. The relative error for the test case from Figure D.3 can be seen in Figure D.4.

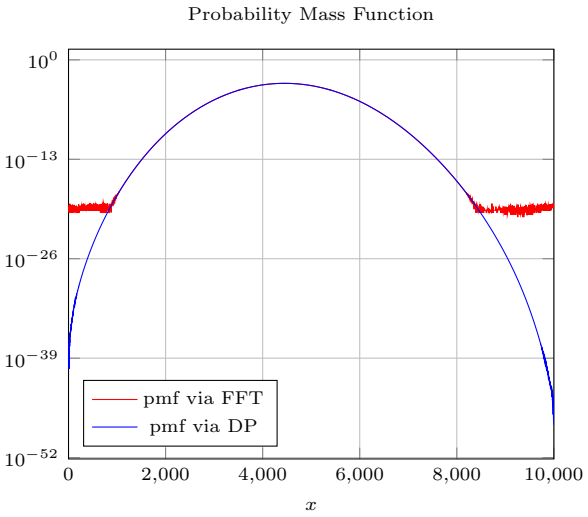

Figure D.3: pmf of a test case with $m = 100, n = 10,000$ via the DP and DC-FFT approaches

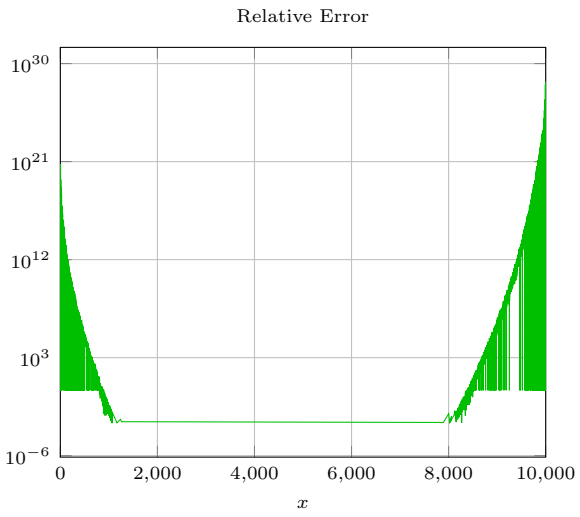

Figure D.4: Relative Accuracy Issue of FFT approaches for small probabilities

Such large relative error in the tails of the probability distribution for larger m may or may not be a problem depending on the application. The decision of whether to sacrifice complete accuracy for speed is left as a choice for the user. A related issue that emerges from the imprecision of FFT approaches in the tails is that the error may result in *negative* values for some points. Clearly negative probabilities are nonsensical. When implementing `pyGPB` this problem was dealt with by setting all negative values to zero.

In all of the approaches tested there was generally an issue for large n where the total of the pmf vector would not exactly sum to one. This is unavoidable when adding so many floating point numbers. Thankfully, the implementation of the `scipy.stats.rv_discrete` class which was subclassed in this work allowed for some small leeway in the total.

D.3 Hardware Specification for Benchmarking

For performance benchmarking, the following hardware specification was employed:

CPU Specification:

Model Name: Intel(R) Xeon(R) CPU @ 2.30GHz

CPU count: 2

Clock Rate: 2300 MHz

Threads per core: 2

L1d Cache Size: 32KB

L1i Cache Size: 32KB

L2 Cache Size: 256KB

L3 Cache Size: 46,080KB

Memory 13Gb

GPU Specification:

Model Name: Tesla P100-PCIE-16GB

Clock Rate: 1329 MHz

Streaming Multiprocessor Count: 56

Max Threads per Streaming Multiprocessor: 2048

L1 Cache Size: 24KB per Streaming Multiprocessor

L2 Cache Size: 4 MB

Memory Size: 16 GB

Driver Version: 460.32.03

CUDA Version: 11.2

Bibliography

[1] Carlo Acerbi and Dirk Tasche. Expected shortfall: a natural coherent alternative to value at risk. *Economic notes*, 31(2):379–388, 2002. doi:https://doi.org/10.1111/1468-0300.00091.

[2] J.P. Basu and P.L. Odell. Effect of intraclass correlation among training samples on the misclassification probabilities of bayes procedure. *Pattern Recognition*, 6(1):13–16, 1974. doi:https://doi.org/10.1016/0031-3203(74)90004-1.

[3] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. doi:https://doi.org/10.1126/science.153.3731.34.

[4] William Biscarri, Sihai Dave Zhao, and Robert J Brunner. A simple and fast method for computing the poisson binomial distribution function. *Computational Statistics & Data Analysis*, 122:92–100, 2018. doi:https://doi.org/10.1016/j.csda.2018.01.007.

[5] Georg Brandl. Sphinx Python Documentation Generator, 2021. visited 2021-08-25. URL: https://www.sphinx-doc.org/en/master/.

[6] CASIO COMPUTER CO., LTD. Nodes and weights of gauss-kronrod calculator, 2021. visited 2021-08-09. URL: https://keisan.casio.com/exec/system/1289382036.

[7] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. doi:https://doi.org/10.2307/2003354.

[8] Encyclopedia of Mathematics. Partial correlation coefficient, 2012. visited 2021-08-11. URL: http://encyclopediaofmath.org/index.php?title=Partial_correlation_coefficient&oldid=24254.

[9] David Eppstein. Minimum range balanced cuts via dynamic subset sums. *Journal of Algorithms*, 23(2):375–385, 1997. doi:https://doi.org/10.1006/jagm.1996.0841.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[11] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974. doi:https://doi.org/10.1145/321812.321823.

[12] IHS Markit. Calculators, 2021. visited 2021-08-28. URL: https://www.markit.com/markit.jsp?jsppage=pv.jsp.

[13] Florian Junge. *PoissonBinomial: Efficient Computation of Ordinary and Generalized Poisson Binomial Distributions*, 1.2.4 edition, 2021. Published 2021-07-27. URL: https://ftp.uni-sofia.bg/CRAN/web/packages/PoissonBinomial/.

[14] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972. doi:10.1007/978-1-4684-2001-2_9.

[15] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[16] Konstantinos Koiliaris and Chao Xu. Faster pseudopolynomial time algorithms for subset sum. *ACM Transactions on Algorithms (TALG)*, 15(3):1–20, 2019. doi:https://doi.org/10.1145/3329863.

[17] R.E. Melchers and A.T. Beck. *Structural Reliability Analysis and Prediction*. Wiley, 2017. URL: https://books.google.co.uk/books?id=SWQ6DwAAQBAJ.

[18] NVIDIA Corporation. Cuda zone, 2021. URL: https://developer.nvidia.com/cuda-zone.

[19] Preferred Networks, Inc. & Preferred Infrastructure, Inc. . Cupy, 2021. URL: https://cupy.dev/.

[20] Preferred Networks, Inc. & Preferred Infrastructure, Inc. . Cupy, performance best practices, 2021. visited 2021-08-09. URL: https://docs.cupy.dev/en/stable/user_guide/performance.html.

[21] Python Software Foundation. Python package index - pypi, 2021. visited 2021-08-25. URL: https://pypi.org/.

[22] Read the Docs, Inc and contributors. Create, host, and browse documentation., 2021. visited 2021-08-25. URL: https://readthedocs.org/.

[23] Mark Rotchell. A latent-factor model for loss in heterogeneous credit portfolios, with gpu-accelerated convolution and quadrature; msc data science project proposal. 2021.

[24] Trading Economics. Trading economics, gdp, world, 2021. visited 2021-08-28. URL: https://tradingeconomics.com/country-list/gdp.

[25] World Government Bonds. Sovereign cds, 2021. visited 2021-08-28. URL: http://www.worldgovernmentbonds.com/sovereign-cds/.

[26] Man Zhang, Yili Hong, and Narayanaswamy Balakrishnan. The generalized poisson-binomial distribution and the computation of its distribution function. *Journal of Statistical Computation and Simulation*, 88(8):1515–1527, 2018. doi:https://doi.org/10.1080/00949655.2018.1440294.