Trials, Tribulations, and eventual Triumph

Computing and Optimising Fisher Information for POSBP

Matthew P. Skerritt
from research conducted with A. Eshragh, J.V. Ross, B. Salvy, T. McCallum

CARMA Priority Research Centre
University of Newcastle, NSW, Australia

Maple Conference
November 2–6, 2020

Preliminaries

Partially Observable Simple Birth Processes

We consider a growing population whose population at time t is x_t.

There are parameters (coming from parameters of the underlying statistical model)

λ: The growth rate of the population.

p: The probability $0 \leq p \leq 1$ of observing an individual of the population.

these parameters are fixed for a given population.

We intend to sample the population at some times $t_1 \leq t_2 \leq \ldots \leq t_n$.[1]

[1] *Note that this is a simplified description of complicated statistical model. This explanation given herein is sufficient to understand the computations we present.*

Fisher Information and the Likelihood Function

Definition (Likelihood Function for a POSBP)

Given times $t_1 \leq t_2 \leq \ldots \leq t_n$, let $\mathbf{y}_n = (y_{t_1}, \ldots, y_{t_n}) \in \mathbb{N}^n$ be a vector of the number of individuals observed at those times, and let $\mathbf{x}_n = (x_{t_1}, \ldots, x_{t_n}) \in \mathbb{N}^n$ be a vector of the true population value at those times.

The likelihood of seeing exactly those numbers of individuals in our POSBP population (with parameters λ and p) is given by the *likelihood function*

$$\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p) = \sum_{1 \leq x_{t_1} \leq \cdots \leq x_{t_n}} \prod_{i=1}^{n} \eta_i(\mathbf{y}_n, \mathbf{x}_n) \tag{1}$$

The meaning of $\mathbf{Y}_n$ and $\eta_i(\mathbf{y}_n, \mathbf{x}_n)$ is outside the scope of this talk.

Fisher Information and the Likelihood Function

If we consider all possible patterns of observation, we can compute the *Fisher Information*

Definition (Fisher Information for a POSBP)

Given times $t_1 \leq t_2 \leq \ldots \leq t_n$ and $0 \leq p \leq 1$ *Fisher Information* is defined by:

$$\mathcal{FI}_{\mathbf{Y}_n}(\lambda) = \sum_{\mathbf{0} \leq \mathbf{y}_n} \frac{\left(\frac{\partial}{\partial \lambda} \mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\right)^2}{\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)} \tag{2}$$

The Problem to be Solved

Optimisation Problem

Given n, p, and λ we find $0 \leq t_1 \leq \cdots \leq t_n = 1$ to maximise $\mathcal{FI}_{\mathbf{Y}_n}(\lambda)$

We know that for maximality it must be that $t_n = 1$. We are primarily interested in the case that the initial population $x_0 = 1$.

The nested sum in eq. (1) makes the computation infeasible numerically for $n > 2$, and even for $n = 2$ the computation is slow.

We may try to alleviate this intractability is to compute the Fisher Information for finitely many values of $(t_1, \ldots, t_{n-1}, 1) \in \{0, 1\}^n$, evenly distributed over the domain.

A C programming language implementation of this technique was produced by A. Eshragh for the $n = 2$ case.

An Improvement by Bruno Salvy

Computation efficient is greatly improved by the realisation of a generating function, ϕ, for the likelihood function. This generating function is a rational polynomial with the property

$$\phi(u_0, \ldots, u_n) = \frac{P(u_0, \ldots, u_n)}{Q(u_0, \ldots, u_n)} = \sum_{y_{t_n}=0}^{\infty} \cdots \sum_{y_{t_1}=0}^{\infty} \sum_{x_0=1}^{\infty} \mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\, u_0^{x_0} u_1^{y_{t_1}} \cdots u_n^{y_{t_n}}$$

which we can rearrange to get

$$Q(u_0, \ldots, u_n) \sum_{y_{t_n}=0}^{\infty} \cdots \sum_{y_{t_1}=0}^{\infty} \sum_{x_0=1}^{\infty} \mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\, u_0^{x_0} u_1^{y_{t_1}} \cdots u_n^{y_{t_n}} = P(u_0, \ldots, u_n)$$

from which we calculate a recurrence relation for the likelihood function by equating coefficients of like terms.

Enter the Speaker!

Reproducing Salvy's Construction

Salvy's *Maple* code generated a Fisher Information calculation function (including computing the generating function) using compiled *Maple* and some clever (but esoteric) `_Inert_ASSIGN` functions. The generation function required to be told the size (in Gigabytes) of memory to use for the computation.

The speaker set about re-implementing Salvy's approach from scratch (using his code as a reference) with the intent of better understanding:

- The derivation of the recurrence relation from the generating function.
- How the recurrence should be built up in higher dimensions.
- The recurrence relation calculation as a whole.

The effort was ultimately successful.

Problems and Refinements

Generating Function Computation

The generating function is symbolically pre-computed in *Maple* as a formal sum.

It is, in fact, a nested sum of the form

$$\sum_{y_{t_1}=0}^{\infty} \cdots \sum_{y_{t_n}=0}^{\infty} \sum_{x_0=1}^{\infty} \sum_{x_{t_1}=0}^{\infty} \cdots \sum_{x_{t_n}=0}^{\infty} f\left(y_{t_1}, \ldots, y_{t_n}, x_0, x_{t_1} \ldots, x_{t_n}\right) \, u_0^{x_0} u_1^{y_{t_1}} \cdots u_n^{y_{t_n}}$$

It must be independently computed for each value of n. The pre-computation time grows significantly with n. For values as low as $n = 4$ or $n = 5$ the computation crashed *Maple* reliably.

Generating Function Computation

Since we are mainly interested in the case that $x_0 = 1$ we rearrange the generating function to

$$\sum_{x_0=1}^{\infty} \left(\sum_{y_{t_1}=0}^{\infty} \cdots \sum_{y_{t_n}=0}^{\infty} \sum_{x_{t_1}=0}^{\infty} \cdots \sum_{x_{t_n}=0}^{\infty} f(y_{t_1}, \ldots, y_{t_n}, x_0, x_{t_1}, \ldots, x_{t_n})\; u_1^{y_{t_1}} \cdots u_n^{y_{t_n}} \right) u_0^{x_0}$$

The coefficient of u_0^1 is the generating function we are interested in, which can be computed directly as

$$\sum_{y_{t_1}=0}^{\infty} \cdots \sum_{y_{t_n}=0}^{\infty} \sum_{x_{t_1}=0}^{\infty} \cdots \sum_{x_{t_n}=0}^{\infty} f(y_{t_1}, \ldots, y_{t_n}, x_0, x_{t_1} \ldots, x_{t_n})\; u_1^{y_{t_1}} \cdots u_n^{y_{t_n}}$$

Doing so greatly reduces the required pre-computation time, although the time still grows significantly with n.

Computing the Recurrence Relation

We have computed generating functions for all values $2 \leq n \leq 5$.

We adopt the following notation

Notation

Let $\mathbf{v} = (v_1, \dots, v_n) \in \mathbb{N}^n$, and consider the generating function

$$\phi(1, u_1, \dots, u_n) = \frac{P(1, u_1, \dots, u_n)}{Q(1, u_1, \dots, u_n)}$$

We denote by $P_{\mathbf{v}}$ and $Q_{\mathbf{v}}$ the coefficient of $u_1^{v_1} \cdots u_n^{v_n}$ in P and Q respectively.

In all cases computed so far, the only non-zero coefficients observed were precisely $P_{\mathbf{c}}$ and $Q_{\mathbf{c}}$ indexed by $\mathbf{c} \in \{0, 1\}^n$.

Computing the Recurrence Relation

The recurrence relation ends up in the form

$$\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p) = \frac{1}{Q_{\mathbf{0}}} \left(P_{\mathbf{y}_n} - \sum_{\substack{\mathbf{c}_n \in \{0,1\}^n \\ \mathbf{c}_n \neq (0,\dots,0)}} \mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n - \mathbf{c}_n \mid \lambda, p) \right) \tag{3}$$

We build the recurrence up starting with $\mathcal{L}_{\mathbf{Y}_n}(\mathbf{0} \mid \lambda, p)$ using the convention that $\mathcal{L}_{\mathbf{Y}_n}(\mathbf{v} \mid \lambda, p) = 0$ if $v_i < 0$ for any $1 \leq i \leq n$.

We also observe that the Fisher Information is a sum of likelihoods over all possible vectors $\mathbf{y}_n$, and exploit the recurrence relation of likelihoods to avoid needless computation.

Weak Compositions and Slices

Definition (Weak composition)

Let n be a non-negative integer. A sequence of k non-negative integers $a_1, \ldots, a_k$ is said to be a *weak composition of n into k parts* if $n = \sum_{i=1}^{k} a_i$.
Similarly, a vector $(a_1, \ldots, a_k)$ of non-negative integers may also be said to be a weak composition of n into k parts.

Definition (Set of Weak Compositions of a Fixed Value)

For an integer $S \geq 0$, we denote the set of all weak compositions of S into n parts by

$$WC_n(S) := \{\mathbf{a}_n \in \mathbb{Z}^n \mid \mathbf{a}_n \text{ is a weak composition of } S\}.$$

Weak Compositions and Slices

Observe that for the vector $\mathbf{y}_n = (y_{t_1}, \ldots, y_{t_n})$ if we let $S = \sum_{i=1}^{n} y_{t_i}$ it is the case that $\mathbf{y}_n$ is a weak composition of S into n parts.

In general, for any vector $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{N}^n$, that vector will be a weak composition of of $S = \|\mathbf{v}\|_1 = \sum_{i=1}^{n} v_i$ into n parts.

We further observe from eq. (3) that for any given $\mathbf{y}_n$, the recursive computation of $\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)$ uses values of $\mathcal{L}_{\mathbf{Y}_n}(\mathbf{v} \mid \lambda, p)$ for vectors $v \in WC_n(S)$ for S strictly smaller than $\|\mathbf{y}_n\|_1$.

These observations are exploited for an effective computation

Weak Compositions and Slices

Definition (Slice)

Let $S > 0$ be an integer. We define the S^{th} *slice* of the computation of $\mathcal{FI}_{\mathbf{Y}_n}(\lambda)$ by

$$\mathcal{SL}_{S,\mathbf{Y}_n}(\lambda) := \sum_{\mathbf{y}_n \in WC_n(S)} \frac{\left(\frac{\partial}{\partial\lambda}\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\right)^2}{\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)}$$

From which we re-write eq. (1) as:

$$\mathcal{FI}_{\mathbf{Y}_n}(\lambda) = \sum_{\mathbf{0} \leq \mathbf{y}_n} \frac{\left(\frac{\partial}{\partial\lambda}\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\right)^2}{\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)} = \sum_{S=0}^{\infty} \sum_{\mathbf{y}_n \in WC_n(S)} \frac{\left(\frac{\partial}{\partial\lambda}\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)\right)^2}{\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)} = \sum_{S=0}^{\infty} \mathcal{SL}_{S,\mathbf{Y}_n}(\lambda)$$

Weak Compositions and Slices

Definition (Slice Order)

Two slices $\mathcal{SL}_{S,\mathbf{Y}_n}(\lambda)$, and $\mathcal{SL}_{S',\mathbf{Y}_n}(\lambda)$ are ordered by the numeric value of S and S'.

We observe:

- The individual likelihood function values in each slice can be computed in parallel once the Likelihood values for all previous slices are calculated.
- Each slice needs only the likelihood function values for the $n-1$ previous slices

So the approach we take for calculation is to build each slice up one at a time, starting at $S=0$. We store the individual likelihood function values to avoid needless re-computation.

Maple Limitations

We would prefer to

- Use only the memory for the n slices we need at any point in the computation.
- Compute each slice in parallel.

However, *Maple* provides no real memory management (as compared to a language like C or C++). In particular it has no (apparent) means of discarding used memory. We either:

- Pre-allocate a all the memory for all the $\mathcal{L}_{\mathbf{Y}_n}(\mathbf{y}_n \mid \lambda, p)$ values we need for the entire computation.
- Create tables for each slice (and rely on garbage collection).
- Do something clever with re-using arrays and overwriting unneeded values.

These use too much memory and/or require knowing a priori the number of slices.

Maple Limitations

In addition to the memory troubles, *Maple*'s threading capabilities did not significantly effect the computation time at all (but did significantly effect the complexity of the code). Several strategies were attempted.

The computations required computing many slices before a reliable value for Fisher Information was found. Due to *Maple* being an interpreted language, the computation times were slow. Compiled *Maple* can (and did) help.

Nonetheless, the problems proved too troublesome, and *Maple* was abandoned for computing Fisher Information.

C++ to the Rescue

An implementation was written in C++, which avoided the above problems. We were able to leverage the standard library data structures and parallelisation capabilities to great effect.

We note that the computation time grows with the values of all inputs and parameters (n, t_i, λ, and p). The growth is particularly significant in the case of n primarily and λ secondarily.

The computation is fast enough that it is feasible to optimise the Fisher Information in a more fine grained manner than was previously possible. To this end, the C++ code was compiled to a shared library which exported functions for computing Fisher Information for $n = 2, n = 3, n = 4$, and $n = 5$ in both parallel and single-threaded variants.

Back to *Maple*

The shared library from the C++ code was read by *Maple* and the functions imported. This was *remarkably* easy.[2]

We used *Maple*'s `Optimization` package to optimise the Fisher Information.

- Wrapper functions (for both the optimisation, and the Fisher Information) with the `remember` option was created to cache results.
- The `plot` function was used to produce graphs of optimal values of t_i (one graph per t_i) against $0 \leq p \leq 1$ for fixed n and λ. This let us leverage the adaptive plotting feature.

[2]The speaker has not yet been able to successfully import these shared library functions into either *Mathematica*, nor *MATLAB*.

Back to *Maple*

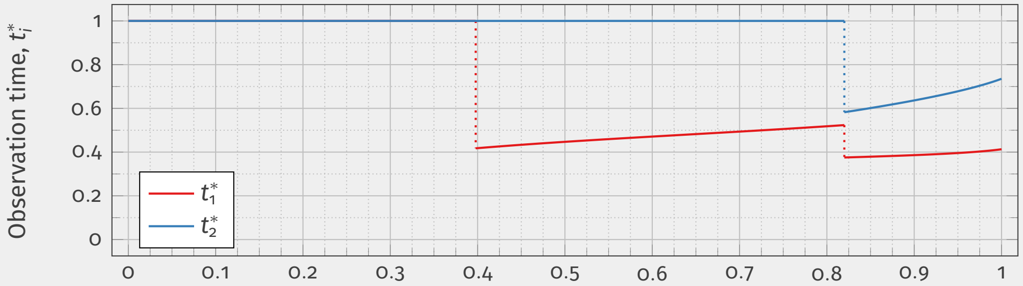

Figure: Optimal observation times for $n = 3, \lambda = 2$

Back to *Maple*

Optimisation exacerbates the execution time grown with n as each subsequent n requires optimisation over a higher dimensional search space, and consequently more lower-dimensional boundary domains.

- We were able to optimise for cases up to $n = 4$.
- As the value of n increased, the feasible values for λ decreased. On a 36 core machine:
  - For $n = 2$ the total optimisation time varied from about 1 second for $\lambda = 0.5$ to about 7 and a quarter hours for $\lambda = 5$.
  - For $n = 3$ the total optimisation time varied from about 10 minutes for $\lambda = 0.5$ to approximately 3 months for $\lambda = 4$.
  - For $n = 4$ the total optimisation time varied from about 16 and a quarter hours for $\lambda = 0.5$ to about 5 and a half days for $\lambda = 1$.

Closing

Summary

We used a dynamic programming technique to compute the likelihood function for a Partially Observable Simple Birth Process using a recurrence relation extracted from a generating function. From this we computed and optimised Fisher Information.

- *Maple* was essential to symbolically pre-calculate the generating functions.
- *Maple* was not up to the task of computing the Fisher Information directly, and a C++ program was written instead.
- The C++ program was compiled to a shared library, which *Maple* was able to easily make use of.
- The imported shared library functions were thus able to be used with the *Maple* `Optimization` package.

Thank You!