Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Bayesian, and non-Bayesian, Cause-Specific Competing-Risk Analysis for Parametric and Non-Parametric Survival Functions: The R Package CFC

Alireza S. Mahani
Scientific Computing
Sentrana Inc.

Mansour T.A. Sharabiani
School of Public Health
Imperial College London

Abstract

The R package **CFC** performs cause-specific, competing-risk survival analysis by computing cumulative incidence functions from unadjusted, cause-specific survival functions. A high-level API in **CFC** enables end-to-end survival and competing-risk analysis, using a single-line function call, based on the parametric survival regression models in **survival** package. A low-level API allows users to achieve more flexibility by supplying their custom survival functions, perhaps in a Bayesian setting. Utility methods for summarizing and plotting the output allow population-average cumulative incidence functions to be calculated, visualized and compared to unadjusted survival curves. Numerical and computational optimization strategies are employed for efficient and reliable computation of the coupled integrals involved. To address potential integrable singularities caused by infinite cause-specific hazards, particularly near time-from-index of zero, integrals are transformed to remove their dependency on hazard functions, making them solely functions of cause-specific, unadjusted survival functions. This implicit variable transformation also provides for easier extensibility of **CFC** to handle custom survival models since it only requires the users to implement a maximum of one function per cause. The transformed integrals are numerically calculated using a generalization of Simpson's rule to handle the implicit change of variable from time to survival, while a generalized trapezoidal rule is used as reference for error calculation. An OpenMP-parallelized, efficient C++ implementation – using **Rcpp** and **RcppArmadillo** packages – makes the application of **CFC** in Bayesian settings practical, where a potentially large number of samples represent the posterior distribution of cause-specific survival functions.

Keywords: Newton-Cotes, adaptive quadrature, Monto Carlo Markov Chain.

1. Introduction

Motivation: Consistent propagation and calculation of uncertainty using predictive posterior distributions is a key advantage of Bayesian frameworks (Gelman and Hill 2006), particularly in survival analysis, where predicted entities such as survival probability can be highly-nonlinear, time-dependent functions of estimated model parameters. In the absence of high-performance software for Bayesian *prediction*, premature point-estimation of model parameters can only produce approximate – sometimes grossly wrong – mean values for predicted entities (Figure 1, left panel). The R package **CFC** seeks to address this void for Bayesian cause-specific competing-risk analysis.

Existing methods and tools for competing-risk procedure: In survival analysis with multiple, mutually-exclusive, end-points, competing-risk techniques must be used to properly account for interaction among causes while estimating the expected percentage of population likely to experience events of a particular cause. Several techniques exist for competing-risk analysis, some of which have been implemented as open-source R packages. Among the more established technique are the cause-specific framework (Prentice, Kalbfleisch, Peterson Jr, Flournoy, Farewell, and Breslow 1978), sub-distribution hazard (Fine and Gray 1999) (available in **cmprsk** (Gray 2014)), mixture models (Larson and Dinse 1985) (available in **NPMLEcmprsk** (Chen, Chang, and Hsiung 2015)), vertical modeling (Nicolaie, van Houwelingen, and Putter 2010), and the method of pseudo-observations (Andersen, Klein, and Rosthøj 2003) (available in **pseudo** (Maja Pohar Perme and Gerster 2012)). For an in-depth review and comparison of competing-risk methods, see Haller, Schmidt, and Ulm (2013). More recently, machine learning techniques such as random forests (Breiman 2001) and gradient boosting machines (Friedman 2001) have been extended to survival models (available via **randomForestSRC** (Ishwaran and Kogalur 2015) and **gbm** (Ridgeway 2015), respectively). The random forest survival implementation in **randomForestSRC** includes competing-risk analysis (Ishwaran, Gerds, Kogalur, Moore, Gange, and Lau 2014).

Cause-specific competing-risk analysis: The cause-specific framework for competing-risk analysis (CFC) is a two-step process. First, independent survival models are constructed for each cause. In each cause-specific model, events due to alternative causes are treated as censoring. To arrive at cause-specific cumulative incidence functions, however, the population depletion due to competing risks must be taken into account, in order to avoid over-estimating the cause-specific incidence probabilities (Figure 1, right panel). This leads to a second step, where a set of coupled, first-order differential equations must be solved. A key advantage of CFC is that, by separating the two steps, it allows for full flexibility in using different survival models as input into the second step, including a combination of parametric and non-parametric model. The only requirement for each cause-specific survival model is the implementation of a function that returns the unadjusted survival probability at any given time from index. While the first step is straightforward, applying the second step - calculation of cumulative incidence functions from cause-specific survival models - is non-trivial due to numerical and computational challenges. These challenges are especially pronounced in Bayesian settings involving a large number of MCMC samples representing the posterior distribution of time-dependent functions for each subject. While a non-parametric version of the cause-specific framework is available in the **survival** package (Therneau 2015) via function `survfit`, no reliable and modular open-source software enables the application of CFC to arbitrary parametric models, Bayesian or non-Bayesian, despite the popularity and intuitive

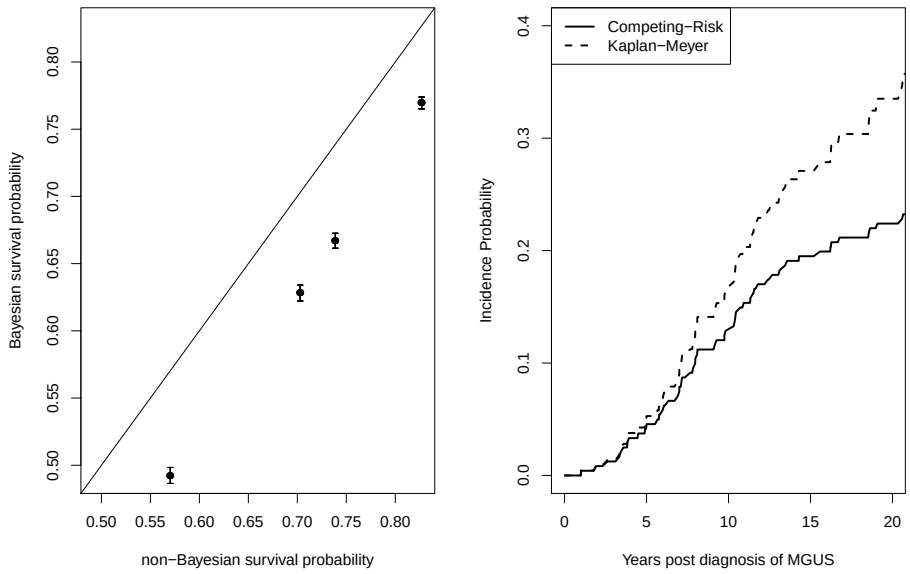

Figure 1: Motivating Bayesian cause-specific competing-risk analysis. Left panel: Estimated survival probability, based on a Bayesian Weibull regression, at 614 days from index, for 'ovarian' data set, available in R package **BSGW** (Mahani and Sharabiani 2015a). The x axis corresponds to the 'incorrect' (non-Bayesian) method of calculating the average of model coefficients, followed by calculation of survival probability using the coefficient averages. The y axis corresponds to the mean+/se values of the same survival probabilities, this time using the 'correct' (Bayesian) method of calculating the probabilities once for each sample, and then averaging the probabilities. All survival probabilities are significantly over-estimated by the non-Bayesian method. The underlying MCMC run consisted of 5000 iterations, the first 2500 of which were discarded as burn-in. Right panel: Comparison of cumulative incidence probability, with and without competing-risk correction, for the 'pcm' event in 'mgus1' data set, taken from R package **survival** (Therneau 2015). Compared to a naive Kaplan-Meyer estimate, competing-risk analysis removes cases lost to 'death', thus reducing the pool of available subjects for 'pcm', and therefore leading to a smaller estimate for cumulative incidence probability for 'pcm'.

appeal of this approach to competing-risk analysis.

Our contribution: The R package **CFC** provides – to our knowledge – the first open-source, general-purpose software for numerical calculation of cumulative incidence functions from cause-specific (unadjusted) survival functions (hereafter referred to as survival functions). Through a combination of algorithmic innovations, performance optimization techniques, and software design choices, **CFC** provides both an easy-to-use API for performing competing-risk analysis of standard parametric survival regression models (via integration with **survival** package) as well as the machinery for efficient and reliable application of CFC to arbitrary survival models, using both R and C++ interfaces. The C++ implementation and interface is based on the convenient framework of **Rcpp** (Eddelbuettel, François, Allaire, Chambers, Bates, and Ushey 2011) and **RcppArmadillo** (Eddelbuettel and Sanderson 2014) packages. As such, **CFC** should appeal to practitioners as well as package developers.

Paper outline: The rest of this paper is organized as follows. In Section 2 we discuss the challenges of – and solutions for – the numerical quadrature problem of Bayesian CFC. In Section 3, we review the **CFC** package in some detail, including a description of its three usage modes. Section 4 presents several examples illustrating these usage modes and other features of **CFC**. Section 5 concludes with a summary of our work and pointers to pontential future research and development.

2. Bayesian CFC quadrature

This section provides the mathematical framework for CFC, discusses the shortcomings of existing quadrature techniques for the numerical integration involved in Bayesian CFC, and presents our alternative quadrature algorithm, the generalized Newton-Cotes framework.

2.1. Cause-specific competing-risk survival analysis

The following set of K coupled, first-order differential equations describe the time evolution of cause-specific cumulative incidence functions, $F_k(t)$:

$$\frac{dF_k(t)}{dt} = \left(1 - \sum_{k'=1}^{K} F_{k'}(t)\right) \lambda_k(t), \tag{1}$$

where $\lambda_k(t)$ (≥ 0, $\forall t > 0$) refers to the k'th (non-negative) cause-specific hazard function. These equations can be decoupled and transformed into integrals. We do so by summing the two sides of Equation 1 over all k:

$$\sum_{k=1}^{K} \frac{dF_k(t)}{dt} = \left(1 - \sum_{k'=1}^{K} F_{k'}(t)\right) \sum_{k=1}^{K} \lambda_k(t), \tag{2a}$$

$$\implies \frac{dE(t)}{E(t)} = -\sum_{k} \lambda_k(t), \tag{2b}$$

where we have defined the event-free probability function, $E(t)$, as:

$$E(t) \equiv 1 - \sum_{k=1}^{K} F_k(t), \tag{3}$$

Solving for $E(t)$ in Equation 2b, we obtain:

$$E(t) = \prod_{k=1}^{K} S_k(t), \tag{4}$$

where $S_k(t)$ stands for the unadjusted survival function for cause k. They are defined as:

$$S_k(t) \equiv \exp\left(-\int_{t'=0}^{t} \lambda_k(t')\, dt'\right) \tag{5}$$

Since hazard functions are non-negative, we conclude that

$$0 \leq S_k(t) \leq 1, \qquad \forall\ k = 1, \ldots, K\ ,\ t \geq 0, \tag{6}$$

and also that $S_k(0) = 1$. These functions correspond to the naive, Kaplan-Meyer approach depicted in Figure 1 (left panel). Substituting back into Equation 1 and integrating the two sides, we arrive at the following K, one-dimensional integral equations:

$$F_k(t) = \int_{t'=0}^{t} \left(\prod_{k'=1}^{K} S_k(t')\right) \lambda_k(t') dt'. \tag{7}$$

The integrals of Equation 7 do not generally have closed-form solutions. For example, in a Weibull survival model, we have

$$\lambda_k(t) = \alpha_k\, \gamma_k\, t^{\alpha_k - 1}, \tag{8a}$$
$$S_k(t) = \exp(-\gamma_k t^{\alpha_k}), \tag{8b}$$

which leads to the following expression for cumulative incidence functions with two competing risks ($K = 2$):

$$F_1(t) = -\alpha_1 \gamma_1 \int_0^t u^{\alpha_1 - 1}\, \mathrm{e}^{-(\gamma_1 u^{\alpha_1} + \gamma_2 u^{\alpha_2})}\, du, \tag{9}$$

and similarly for $F_2(t)$. In the absence of an exact solution, we must resolve to numerical integration.

In most real-world problems, each of the K integrals of Equation 7 must be evaluated more than once. For example, in regression settings each observation will have a distinct set of cause-specific hazard, survival, and cumulative incidence functions that are dependent on the value of the feature vector for that observation. Furthermore, in Bayesian settings where posteriors are approximated by MCMC samples, each combination of observation and cause will have as many integrals as samples. For example, in a competing-risk analysis with 3 causes, 1000 observations, and 5000 posterior samples, $3 \times 1000 \times 1000 = 3$ million cumulative incidence functions must be calculated. As a result, computational efficiency of the quadrature algorithm is of prime importance in Bayesian CFC.

2.2. Shortcomings of current quadrature techniques for Bayesian CFC

Most modern techniques for one-dimensional, numerical integration are based on function interpolation. Perhaps the most common set of techniques are Gaussian quadratures (Stroud

and Secrest 1966), which approximate an integral by a weighted sum of function values evaluated on a pre-specified, irregular grid, often yielding exact results for polynomials. A particular flavor called Gauss-Kronrod quadrature (Laurie 1997) allows function evaluations to be re-used in successive, adaptive iterations. The **QUADPACK** library (Piessens, de Doncker-Kapenga, Überhuber, and Kahaner 2012), ported to R via the `integrate` function in **stats** package, is based on Gauss-Kronrod, and augmented by Wynn's epsilon algorithm (Wynn 1966) to accelerate convergence for end-point singularities.

A direct application of this software to the Bayesian CFC quadrature problem, however, is inefficient due to several reasons. Firstly, and most importantly, users often expect a *dense output* for cumulative incidence functions, i.e., $F_k(t)$'s in Equation 7 must be evaluated at multiple values of t. **QUADPACK**, on the other hand, is not designed to produce dense output, and therefore would require as many calls as the number of outputs desired. This can lead to an excessive number of function evaluations, which is particularly troubling in time-consuming, Bayesian problems. Secondly, there is a significant opportunity to share computational work across the K integrals in CFC. This is due to the fact that the integrands in Equation 7 are various multiplicative permutations of cause-specific hazard and survival functions. Taking advantage of this opportunity, however, requires custom code that is designed for the particular structure of the CFC problem. Finally, direct calculation of each intergal in Equation 7 is suboptimal from a usability pespective, since it requires the user to supply two functions per cause: the hazard function, $\lambda_k(t)$, and the survival function, $S_k(t)$. While the two functions are related by Equation 5, yet their conversion requires integration or differentiation. Also, in non-parametric survival models, the survival function is usually not differentiable (e.g., piece-wise step function) and hence the hazard function is unavailable. In the best case, requiring both functions adds a burden to the user and increases the possibility of mistakes. Numeric differentiation, e.g., using **numDeriv** (Gilbert and Varadhan 2012), is an option, but it is computationally expensive.

A second class of integration algorithms are quadrature by variable transformation (Press 2007, Chapter 4), better known as double-exponential (DE) or Tanh-Sinh methods (Takahasi and Mori 1974). They combine a hyperbolic change-of-variable with trapezoidal rule to induce exponential convergence of the integral near end-point singularities. As such, they are better suited to handle such singularities compared to Gauss-Kronrod techniques. DE quadrature has recently been implemented in R via the package **deformula** (Okamura 2015). For an empirical comparison of quadrature techniques, see Bailey, Jeyabalan, and Li (2005). Of the three problems with **QUADPACK** discussed above, the last two are equally applicable to DE methods. (The trapezoidal rule used in DE methods can produce cumulative integrals.) We therefore develop a custom quadrature algorithm for Bayesian CFC that addresses the shortcomings of existing techniques.

2.3. Implicit variable transformation quadrature using generalized Newton-Cotes

We transform Equation 7 to an equivalent form that removes the hazard function, thereby freeing it from potential singularities. To do so, we use the definition of $S_k(t)$ in Equation 5 to get

$$dS_k(t) = d\left(\exp(-\int_0^t \lambda_{k'}(t')dt')\right) = -\lambda_k(t)S_k(t) \tag{10}$$

Solving for $\lambda_k(t)$, and inserting back into Equation 7, we obtain:

$$F_k(t) = -\int_0^t \left(\prod_{k' \neq k} S_{k'}(t') \right) dS_k(t'), \quad \forall\, k = 1, \ldots K. \tag{11}$$

Thanks to the conditions described in 6, the integrand is now bounded.

The transformation from Equation 7 to Equation 11 is closely related to the DE method, with two notable differences: First, while in DE we use a double-exponential change of variable such as $t = \tanh(\frac{\pi}{2}\sinh(u))$, here we use a custom transformation, $t = S_k^{-1}(u)$. Secondly, rather than applying the transformation explicitly – which would require the user to supply the inverse survival functions – we leave it implicit. This allows us to handle cases where explicit derivation of inverse survival functions is impossible; it also makes the software user-friendly.

Similar to the DE method, we apply Newton-Cotes techniques to the integrals of Equation 11. However, in exchange for removing the singularity, the new form – with variable transformation left implicit – imposes limits such as inability to cheaply find interval midpoints on the scale of transformed variable, thus rendering the classical Newton-Cotes expressions invalid (with the exception of trapezoidal rule). We have thus developed a generalized Simpson's rule that applies to integrals with an implicit change of variable, such as in Equation 11. Recall the standard Simpson's rule which approximates the integral of $f(t)$ in the interval $[a, b]$ via a quadratic approximation of the function:

$$\int_a^b f(t)\, dt \cong I_{ss}(f;\, a, b) \equiv \frac{b-a}{6} \left\{ f(a) + 4f(\frac{a+b}{2}) + f(b) \right\}. \tag{12}$$

In *generalized Simpson*, we have:

$$\begin{aligned} &\int_a^b f(t)\, dg(t) \cong I_{gs}(f, g;\, a, b) \\ &\equiv \frac{g(b)-g(a)}{6} \frac{f(a) + 4f(\frac{a+b}{2}) + f(b) + 2r(f(a)-f(b)) - 3r^2(f(a)+f(b))}{1-r^2}, \qquad \text{(13a)} \\ &r \equiv \frac{2g(\frac{a+b}{2}) - g(a) - g(b)}{g(b)-g(a)}. \qquad \text{(13b)} \end{aligned}$$

Proof is given in Appendix A, which relies on a quadratic expansion of f in terms of g over the interval $[a, b]$. Note that if $g(t)$ is linear in t, then $r = 0$ and the generalized equation in 13a reduces to the standard form in Equation 12. Also, in the corner cases where $g(a) = g((a+b)/2)$ or $g((a+b)/2)) = g(b)$ (leading to $r^2 = 1$), the above equation must be overridden in favor of a single trapezoidal step over the other half of the interval $[a, b]$. The implementation in **CFC** contains this protective measure, which is particularly useful for handling non-parametric survival functions.

While it is possible to use the inequalities of 6 to derive firm upper bounds for integration error, such upper bounds are too pessimistic since they assume piece-wise constant survival curves. Instead, we opt for a common approach in adaptive quadrature literature, i.e., using the difference between our method, and the output of a less accurate technique as a proxy for integration error. In our case, we use the trapezoidal rule as reference, which is easily

generalized to the implicit variable transformation method:

$$\int_a^b f(t)\, dg(t) \cong I_{gt}(f, g;\, a, b) = \frac{g(b) - g(a)}{2}(f(a) + f(b)). \tag{14}$$

The advantage of using a generalized Newton-Cotes framework is that 1) it permits an adaptive subdivision scheme which reuses all previous integrand evaluations (Press 2007, Chapter 4), and 2) in the process, we obtain dense output for the integral, i.e., at all subdivision boundaries. To provide dense output at arbitrary points requested by the user, we use interpolation. Pseudo-code for the CFC quadrature algorithm is listed below. (For simplicity, we focus on a single integration task, which is wrapped in a `for` loop, corresponding to observations and/or Bayesian samples.)

Algorithm 1
Input: $\{S_k(.)\},\ k = 1, \ldots, K$ (cause-specific survival functions)
Input: $\mathbf{t}_{out}$ (dense output time vector)
Input: N_{max} (maximum number of adaptive subdivisions)
Input: ϵ (relative error tolerance for integration)
Output: $\{\mathbf{I}_k\},\ k = 1, \ldots, K$ (cause-specific CI functions, each one evaluated at $\mathbf{t}_{out}$)
1. $t_{max} \leftarrow \max(\mathbf{t}_{out})$
2. $\mathbf{t} \leftarrow \begin{bmatrix} 0 & t_{max} \end{bmatrix}$ (integration time grid)
3. Apply generalized Simpson and trapezoidal to $\mathbf{t}$ to initialize CI estimates and errors.
4. $e \leftarrow$ Maximum integration error across all causes at t_{max}
5. $N \leftarrow 1$
6. **while** $(e > \epsilon)$ and $(N < N_{max})$
7. Identify time interval with maximum contribution towards integration error.
8. Split the the identified interval in half.
9. Update generalized Simpson and trapezoidal CI estimates and errors.
10. $e \leftarrow$ Maximum integration error across all causes at t_{max}
11. $N \leftarrow N + 1$
12. Interpolate generalized Simpson CI estimates over $\mathbf{t}$ to produce $\{\mathbf{I}_k\}$ for $\mathbf{t}_{out}$.
13. **return** $\{\mathbf{I}_k\}$

In implementing the above quadrature algorithm in **CFC**, we have used several performance optimization strategies, which are discussed next.

2.4. Performance optimization strategies in CFC

Since a key target application of **CFC** is Bayesian survival models, performance is an important consideration. We have adopted three performance optimization strategies in **CFC** for executing the quadrature algorithm described in Section 2.3: 1) C++ implementation, 2) work sharing, and 3) parallel execution. Below we discuss each strategy.

C++ implementation: This includes two interconnected but distinct concepts, 1) implementation of the CFC quadrature algorithm in C++, and 2) API definition for user-supplied survival functions in C++. While it is possible to accept and call an R implementation of survival functions inside the C++-based quadrature algorithm, the data marshalling overhead of repeated calls from C++ to R can more than nullify the benefits of porting the quadrature

algorithm to C++ (Eddelbuettel 2013). In other words, if the survival function must be executed in R, it is best for the quadrature algorithm to also run in R. We rely on the framework provided by the **Rcpp** package (Eddelbuettel *et al.* 2011) for our C++ implementation, which facilitates the development and maintenance of **CFC** and also encourages more efficient C++ implementations of survival functions by users and package developers. Details regarding the C++ components of **CFC** are provided in Section 3.

Work sharing: In most cases, evaluating the integrand is where a significant fraction, if not the majority, of time is spent in a quadrature algorithm. Therefore, minimizing the number of such function evaluations can be a rewarding performance optimization technique. In generalized Simpson (Equation 13a) and trapezoidal (Equation 14) steps applied to the CFC problem, we need to evaluate two types of entities: 1) individual survival functions (S_k's), and 2) their leave-one-out products ($\prod_{k' \neq k} S_{k'}$). We use two types of optimizations to minimize unnecessary calculations: 1) calculation of S_k's is shared across all causes, and 2) the full product $\prod_k S_k$ is calculated once, and divided by individual S_k terms to arrive at leave-one-out terms.

Parallelization: Calculation of cumulative incidence functions for different observations and/or samples is clearly an independent set of tasks, which can be executed in parallel. We use OpenMP in C++, and **doParallel** (Analytics and Weston 2015a) and **foreach** (Analytics and Weston 2015b) packages in R, for this task parallelization. As long as the span of the iterator, which is typically the product of observation count and number of samples per observation (in Bayesian models), is sufficiently large, this parallelization scales well with the number of threads used (up to the physical/logical core count on the system). For an illustration of the impact of parallelization on performance, see Section 4.3.

3. CFC implementation and features

This section introduces the **CFC** software and its components, including the three usage modes for cause-specific competing-risk analysis. Examples illustrating each usage mode will be provided in Section 4. As usual, package documentation is the ultimate reference for all details.

3.1. Overview of CFC

CFC functionalities can be categorized into three groups: 1) core functionality, 2) utilities, and 3) legacy code. The core functionality in **CFC** is numerical calculation of cause-specific, competing-risk differential equations in Eq. 1, using the implicit variable transformation in Eq.11, and based on the generalized Newton-Cotes method outlined in Eqs. 13a and 14. There are two implementations of Algorithm 1 in **CFC**: the R-based function `cscr.samples.R`, and the C++-based function `cscr_samples_Cpp`. Both these functions are private, and exposed through a common interface, `cfc`, which dispatches the right method by inspecting the first argument. The `cfc` API provides the users and package developers with the capability to perform cause-specific, competing-risk analysis using their custom-built survival functions, as long as they conform to a pre-specified but flexible interface, which is described in Section 3.2. These functions can be Bayesian or non-Bayesian, parametric or non-parametric.

The utility methods in **CFC**, on the other hand, expose a convenient API for users to perform end-to-end survival and competing-risk analysis with minimal effort. The tradeoff is that the

set of survival functions available through this API is limited to (non-Bayesian) parametric survival regression models of package **survival**. This still represents a core set of popular models, and should address the needs of many practitioners. The functions in this API are `cfc.survreg`, `summary.cfc.survreg`, `plot.summary.cfc.survreg`, `cfc.survreg.survprob` and `cfc.prepdata`. The last two functions are described in Section 3.2, and all functions are illustrated via examples in Section 4.

Legacy functions in **CFC** (`cfc.tbasis` and `cfc.pbasis` and their associated `S3` methods) apply a composite trapezoidal rule to user-supplied survival *curves*, i.e., evaluated at predetermined time points. Here the choice of time intervals is not optimized, and no error analysis is performed. The use of legacy **CFC** is deprecated in favor of the new machinery described in this paper. Interested readers can refer to package documentation for further details on how to use the legacy code.

3.2. CFC usage modes

In order to achieve the dual objectives of user-friendliness *and* flexibility, **CFC** offers three usage modes: 1) end-to-end survival and competing-risk analysis, using `cfc.survreg`, 2) competing-risk analysis of user-supplied survival functions, written in R, and 3) competing-risk analysis of user-supplied functions, written in C++. The last two usage modes are exposed through a common interface, i.e., the public function `cfc`. We review each usage mode below, with examples to follow in Section 4.

Parametric survival regression and competing-risk analysis: This is the most convenient usage mode, in which a one-line call to `cfc.survreg` performs both cause-specific survival regression and competing-risk analysis. More specifically, `cfc.survreg` executes three steps: i) parsing the survival formula to create cause-specific status columns and formulas; ii) calling the `survreg` function from **survival** package (Therneau 2013) for each cause; iii) calling the internal **CFC** function, `cscr.samples.R`, to perform competing-risk analysis. To perform the first step, we use the (public) utility function `cfc.prepdata`. To perform the last step, the (unadjusted) survival function associated with the `survreg` models is needed, which we have implemented in `cfc.survreg.survprob`. This survival function is made public, so as to allow users to easily combine `survreg` models with other types of survival models. It contains a custom implementation of the survival functions needed by `cscr.samples.R`, using the `"dist"` field of the returned object from `survreg`, along with the definition of distributions made available via `survreg.distributions`. The implementation is a verbatim translation of the definition of location-scale family (Datta 2005). This usage mode trades off flexibility for user-friendliness, as it is restricted to the survival regression models covered in the **survival** package, as well as user-defined survival distributions following the conventions of that package. An example is provided in Section 4.1.

CFC for user-supplied survival functions implemented in R: If the `f.list` argument passed to `cfc` is a list of functions - implemented in R - then `cfc` dispatches `cscr.samples.R`. This is more flexible than using `cfc.survreg` since the user can supply any arbitrary survival function, as long as it conforms to the prototype expected by `cscr.samples.R`. However, it requires the user to implement one or more survival functions. These functions must follow this prototype:

```
func(t, args, n)
```

where `t` is a vector of time-from-index (non-negative) values, `args` is a list of argument needed by the function, and `n` is an iterator that spans the observations and/or samples. It is the responsibility of function implementation to consistently interpret `n`, and map it from one dimension to multiple dimensions, e.g., to obtain observation and sample indices. Of course, we could have chosen to hide `n` inside `args`, but decided to make it explicit to draw attention to its special meaning. In Section 4 we will see several example implementations of survival functions conforming to the above prototype.

CFC for custom models - C++ mode: This usage mode offers the same functionality as the previous one, but requires the user to supply the survival functions in C++. This often leads to significant speedup; however, implementation is also more involved as it requires at least three functions per distinct cause-specific model: initializer, survival function, and resource de-allocator. An example is provided in Section 4.3, illustrating the impact of transition from C++ to R implementation (of survival functions) on performance. To facilitate both package development and maintenance as well as survival-function implementation by users, we have adopted the framework of **Rcpp** (for data exchange with R) and **RcppArmadillo** (for linear algebra):

```
typedef arma::vec (*func)(arma::vec x, void* arg, int n);
typedef void* (*initfunc)(Rcpp::List arg);
typedef void (*freefunc)(void *arg);
```

Note the use of `void*` pointer in function prototypes. This allows for a uniform API across all survival functions, leaving the casting and interpretation of this pointer to each implementation. See Example 3 in Section 4.3.

4. Using CFC

As discussed in Section 3.2, **CFC** can be used in three modes, which progressively become more flexible but also require more significant programming effort. Examples 1-3 illustrate how each mode can be used. The final example illustrates a key advantage of the CFC framework, namely the logical separation of cause-specific survival analysis from the competing-risk analysis, which in turns allows for combining survival models of different kind in the same competing-risk analysis.

4.1. Example 1: end-to-end competing-risk analysis using Weibull regression

In our first example, we illustrate the easiest usage mode in **CFC**, i.e., the `cfc.survreg` function. It first creates parametric survival regression models using the `survreg` function of the **survival** package, and passes these models to `cfc` for competing-risk analysis.

We begin by setting up our environment, including a 70-30 split of our test data set, `bmt`, into training and prediction sets:

```
R> library("CFC")
R> data("bmt")
R> rel.tol <- 1e-3
R> seed.no <- 0
```

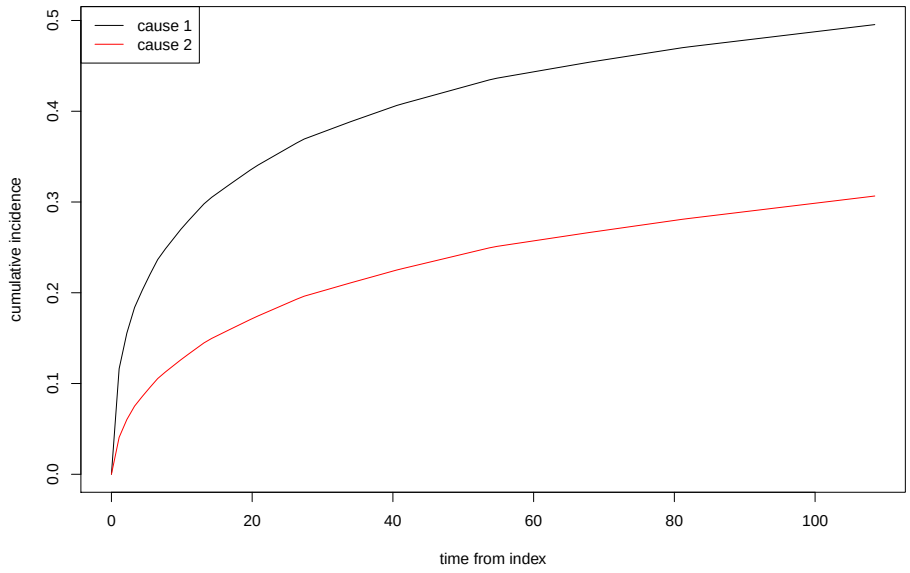

Figure 2: Cause-specific cumulative incidence functions for the Weibull survival regression models built for `bmt` data set.

```
R> set.seed(seed.no)
R> idx.train <- sample(1:nrow(bmt), size = 0.7 * nrow(bmt))
R> idx.pred <- setdiff(1:nrow(bmt), idx.train)
R> nobs.train <- length(idx.train)
R> nobs.pred <- length(idx.pred)
```

(In real-world applications, `rel.tol` must be set to a smaller number for better accuracy.) A one-line call to `sfc.survreg` is all we have to do:

```
R> out.weib <- cfc.survreg(Surv(time, cause) ~ platelet + age + tcell,
+    bmt[idx.train, ], bmt[idx.pred, ], rel.tol = rel.tol)
```

The output can be summarized for any subset of observations, using the `obs.idx` parameter:

```
R> summ <- summary(out.weib, obs.idx = which(bmt$age[idx.pred] > 0))
```

and plotted:

```
R> plot(summ, which = 1)
```

to produce and visualize the sub-population-average cumulative incidence functions for each cause (Figure 2).

It is instructive to visualize the impact of competing-risk adjustment on cumulative incidence rates:

```
R> old.par <- par(mfrow=c(1,2)); plot(summ, which = 2); par(old.par)
```

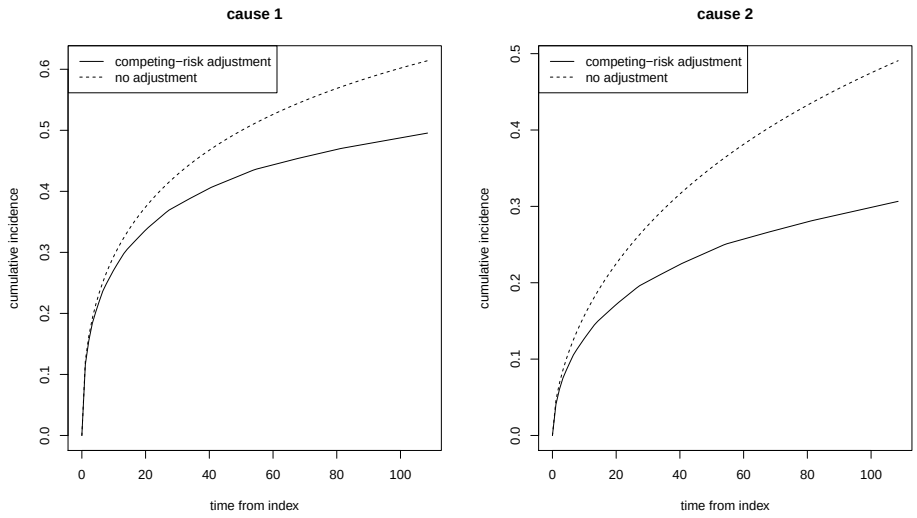

Figure 3: Comparison of adjusted and unadjusted cumulative incidence curves for Weibull survival regression models built for `bmt` data set.

We can see from Figure 3 that the competing-risk adjustment, using the CFC framework, has a very significant corrective impact on the cumulative incidence probability for both causes.

Users can switch from Weibull to other distributions through the parameter `dist`. For example, the following command switches the survival models from Weibull to exponential:

```
R> out.expo <- cfc.survreg(Surv(time, cause) ~ platelet + age + tcell,
+    bmt[idx.train, ], bmt[idx.pred, ],
+    dist = "exponential", rel.tol = rel.tol)
```

We can even use different distributions for each cause, by passing a vector as the `dist` argument:

```
R> out.mix <- cfc.survreg(Surv(time, cause) ~ platelet + age + tcell,
+    bmt[idx.train, ], bmt[idx.pred, ],
+    dist = c("weibull", "exponential"), rel.tol = rel.tol)
```

4.2. Example 2: Bayesian CFC in R

Utilizing the function `cfc` requires that cause-specific, unadjusted survival functions be available or implemented, in R or C++. This is in contrast to `cfc.survreg` which has encapsulated the survival functions corresponding to the class of survival models in **survival** package.

First, we use the utility function `cfc.prepdata` to prepare the `bmt` data set and set up formulas for cause-specific survival analysis:

```
R> out.prep <- cfc.prepdata(Surv(time, cause) ~ platelet + age + tcell, bmt)
R> f1 <- out.prep$formula.list[[1]]
R> f2 <- out.prep$formula.list[[2]]
```

```
R> dat <- out.prep$dat
R> tmax <- out.prep$tmax
```

Next, we create cause-specific Bayesian Weibull survival regression models, using **BSGW** package (Mahani and Sharabiani 2015a):

```
R> library("BSGW")
R> seed.no <- 0
R> set.seed(seed.no)
R> nsmp <- 10
R> reg1 <- bsgw(f1, dat[idx.train, ],
+    control = bsgw.control(iter = nsmp),
+    ordweib = T, print.level = 0)
R> reg2 <- bsgw(f2, dat[idx.train, ],
+    control = bsgw.control(iter = nsmp),
+    ordweib = T, print.level = 0)
```

(In real-world problems, `nsmp` must be set to a larger number.) To perform CFC, we must take two interconnected steps: 1) implement the cause-specific survival functions for this model, 2) prepare arguments feeding into these survival functions. In this example, since we use the same model for both causes, we only need to implement one survival function:

```
R> survfunc <- function(t, args, n) {
+    nobs <- args$nobs; natt <- args$natt; nsmp <- args$nsmp
+    alpha <- args$alpha; beta <- args$beta; X <- args$X
+    idx.smp <- floor((n - 1) / nobs) + 1
+    idx.obs <- n - (idx.smp - 1) * nobs
+    return (exp(- t ^ alpha[idx.smp] *
+                  exp(sum(X[idx.obs, ] *
+                            beta[idx.smp, ]))));
+  }
R> f.list <- list(survfunc, survfunc)
R> X.pred <- as.matrix(cbind(1, bmt[idx.pred, c("platelet", "age", "tcell")]))
R> arg.1 <- list(nobs = nobs.pred, natt = 4, nsmp = nsmp,
+    X = X.pred, alpha = exp(reg1$smp$betas),
+    beta = reg1$smp$beta)
R> arg.2 <- list(nobs = nobs.pred, natt = 4, nsmp = nsmp,
+    X = X.pred, alpha = exp(reg2$smp$betas),
+    beta = reg2$smp$beta)
R> arg.list <- list(arg.1, arg.2)
```

The argument `n` is the single iterator that covers the joint space of observations (`nsmp`) and samples (`nobs`). The function must therefore extract the sample and observation indexes (`idx.smp` and `idx.ob`, respectively) from `n`. The same convention must be used while interpretting the returned arrays from `cfc`.

```
R> rel.tol <- 1e-4
R> tout <- seq(from = 0.0, to = tmax, length.out = 10)
```

```
R> t.R <- proc.time()[3]
R> out.cfc.R <- cfc(f.list, arg.list, nobs.pred * nsmp, tout,
+    rel.tol = rel.tol)
R> t.R <- proc.time()[3] - t.R
R> cat("t.R:", t.R, "sec\n")

t.R: 25.095 sec
```

Summarizing and plotting: One advantage of Bayesian techniques is their consistent representation and treatment of uncertainty. It is, therefore, useful for **CFC** to enable users to quantify and visualize the uncertainty associated with outputs produced by `cfc`. This can be done by calling `summary.cfc`.

In Bayesian **CFC**, the cumulative incidence and survival arrays returned have three dimensions: 1) time, 2) cause, 3) `n` iterator (described above). In summarizing the arrays, we should not reduce/collapse the first two dimensions. The last dimension is often a flattened version of two dimensions: observations and MCMC samples. How this 2D space is mapped to the 1D iterator is left to the users in their specification of the survival function. We have similarly aimed for flexibility in designing the `summary.cfc` function, where the `f.reduce` argument is required from the user in order to determine how each sub-array of the 3D arrays returned by `cfc` must be processed/reduced, before being passed to the `quantile` function to determine the median and credible bands for each combination of time and cause. For the example provided in this section, a suitable reduction function can be defined as:

```
R> my.f.reduce <- function(x, nobs, nsmp) {
+    return (colMeans(array(x, dim = c(nobs, nsmp))))
+  }
```

This function calculates the population average (for each time, cause and MCMC sample). As such, when supplied to `summary.cfc`, it produces the credible bands for population-average values for cause-specific cumulative incidence and survival probabilities. The output of the `summary` function can be passed to the `plot` function to produce Figure 4.

```
R> my.cfc.summ <- summary(
+    out.cfc.R, f.reduce = my.f.reduce
+    , nobs = nobs.pred, nsmp = nsmp)
R> oldpar <- par(mfrow = c(2, 2))
R> plot(my.cfc.summ)
R> par(oldpar)
```

Parallelization: We saw that running CFC in R takes nearly 25 seconds on our test server. (See Section B for session information.) This is for a small data set (`nobs.pred=123`), a handful of samples (`nsmp=10`) and a lenient error tolerance (`rel.tol=1e-4`). By extrapolation, to perform CFC for a data set of size 1000, with 1000 samples, a somewhat more realistic scenario, we would need nearly 4.5 hours. (Note that execution time is nearly independent of the length of `tout`, since the latter only affects the last – interpolation – step, which is computationally cheap.) An easy way to improve performance is by using multi-threaded parallelization on a multicore machine. This can be done via the `ncore` parameter:

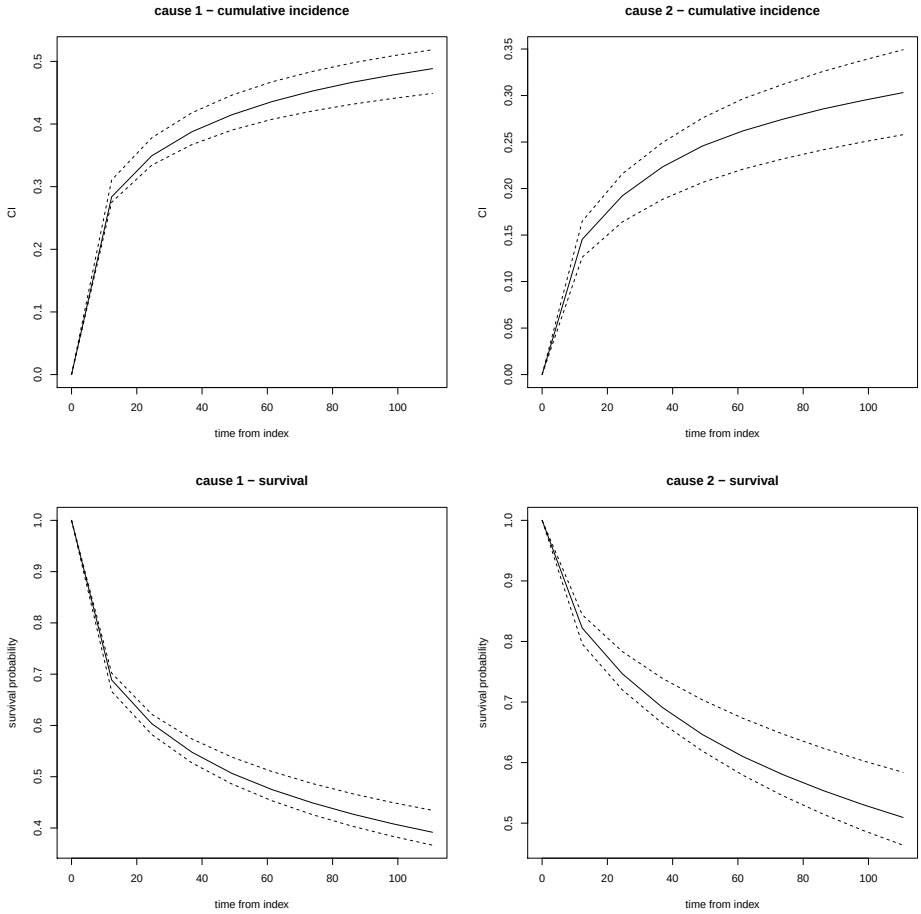

Figure 4: Median and 95% credible bands for population-averaged cause-specific cumulative-incidence and survival probabilities, corresponding to Example 2.

```
R> ncores <- 2
R> tout <- seq(from = 0.0, to = tmax, length.out = 10)
R> t.R.par <- proc.time()[3]
R> out.cfc.R.par <- cfc(f.list, arg.list, nobs.pred * nsmp, tout,
+    rel.tol = rel.tol, ncores = ncores)
R> t.R.par <- proc.time()[3] - t.R.par
R> cat("t.R.par:", t.R.par, "sec\n")
```

The speedup is close to linear, which is expected given the low amount of coordination needed among threads:

```
R> cat("parallelization speedup - R:", t.R / t.R.par, "\n")

parallelization speedup - R: 1.916088
```

A more powerful to improve performance is by using the C++ interface of **CFC**, which is illustrated next.

4.3. Example 3: High-performance Bayesian CFC (in C++)

The first step is to implement the data structure needed by the survival model. For Bayesian Weibull regression, we have:

```
struct weib {
  int nobs; // number of observations
  int natt; // number of attributes
  int nsmp; // number of MCMC samples
  mat X; // nobs-by-natt
  vec alpha; // nsmp
  mat beta; // natt-by-nsmp
};
```

The initializer function is responsible for converting the incoming `List` from R to the `weib` structure:

```
void* weib_init(List arg) {
  weib* myweib = new weib;
  myweib->nobs = arg[0];
  myweib->natt = arg[1];
  myweib->nsmp = arg[2];
  myweib->X = mat(REAL(arg[3]), myweib->nobs, myweib->natt, true, true);
  myweib->alpha = vec(REAL(arg[4]), myweib->nsmp, true, true);
  myweib->beta = mat(REAL(arg[5]), myweib->natt, myweib->nsmp, true, true);
  return (void*)myweib;
}
```

By implementing the data structure in terms of `Armadillo` classes `vec` and `mat`, we isolate the dependence on R data structures to the initializer, which allows for easier porting of

the survival model to other environments. We must also create an external pointer to the initializer, which will be supplied to `cfc`. This can be accomplished by calling the following function from R, as we will demonstrate later:

```
// [[Rcpp::export]]
XPtr<initfunc> weib_getPtr_init() {
  XPtr<initfunc> p(new initfunc(&weib_init), true);
  return p;
}
```

Recall that the `initfunc` function pointer has been `typedef`'ed before. Since the initializer creates a new `weib` data structure on the heap, it is best practice to release this memory once we are finished. We do so by implementing a `freefunc`, as well as a companion function for creating an external pointer to it:

```
void weib_free(void *arg) {
  delete (weib*)arg;
}
// [[Rcpp::export]]
XPtr<freefunc> weib_getPtr_free() {
  XPtr<freefunc> p(new freefunc(&weib_free), true);
  return p;
}
```

Finally, the survival function itself must be implemented, which we do here by using **RcppArmadillo** linear algebra methods. Note a similar approach to the R implementation for extracting the observation and sample indexes (zero-based here) from the flat iterator `n`:

```
vec weib_sfunc(vec t, void *arg, int n) {
  weib *argc = (weib*)arg;
  int nsmp = argc->nsmp, nobs = argc->nobs, natt = argc->natt;
  int idx_smp = n / nobs;
  int idx_obs = n - idx_smp * nobs;
  mat X = argc->X;
  mat beta = argc->beta;
  vec alpha = argc->alpha;
  mat exbeta = exp(X.row(idx_obs) * beta.col(idx_smp));
  return exp(- pow(t, alpha(idx_smp)) * exbeta(0,0));
}
// [[Rcpp::export]]
XPtr<func> weib_getPtr_func() {
  XPtr<func> p(new func(&weib_sfunc), true);
  return p;
}
```

We can compile the entire C++ code for this model by running `Rcpp::sourceCpp`, inside an R session, against the source file (`weib.cpp`). We are now ready to apply `cfc` to this C++ implementation. The call looks similar to the R version, except for the first parameter `f.list`,

which must now be a list of pointers to the survival function, the initializer function, and the free function:

```
R> tout <- seq(from = 0.0, to = tmax, length.out = 10)
R> library("Rcpp")
R> Rcpp::sourceCpp("weib.cpp")
R> f.list.Cpp.1 <- list(weib_getPtr_func(), weib_getPtr_init(),
+    weib_getPtr_free())
R> f.list.Cpp <- list(f.list.Cpp.1, f.list.Cpp.1)
R> t.Cpp <- proc.time()[3]
R> out.cfc.Cpp <- cfc(f.list.Cpp, arg.list, nobs.pred * nsmp, tout,
+    rel.tol = rel.tol)
R> t.Cpp <- proc.time()[3] - t.Cpp
R> cat("t.Cpp:", t.Cpp, "sec\n")

t.Cpp: 0.183 sec
```

We can verify that the C++ results are identical to the R results:

```
R> all.equal(out.cfc.R, out.cfc.Cpp)

[1] TRUE
```

Note the impressive speedup achieved by the C++ implementation:

```
R> cat("C++-vs-R speedup:", t.R / t.Cpp, "\n")

C++-vs-R speedup: 137.1311
```

This performance level makes it feasible to use more realistic parameters, e.g., `nsmp = 1000`:

```
R> nsmp <- 1000
R> reg1 <- bsgw(f1, dat[idx.train, ],
+    control = bsgw.control(iter = nsmp),
+    ordweib = T, print.level = 0)
R> reg2 <- bsgw(f2, dat[idx.train, ],
+    control = bsgw.control(iter = nsmp),
+    ordweib = T, print.level = 0)
R> arg.1 <- list(nobs = nobs.pred, natt = 4, nsmp = nsmp,
+    X = X.pred, alpha = exp(reg1$smp$betas),
+    beta = reg1$smp$beta)
R> arg.2 <- list(nobs = nobs.pred, natt = 4, nsmp = nsmp,
+    X = X.pred, alpha = exp(reg2$smp$betas),
+    beta = reg2$smp$beta)
R> arg.list <- list(arg.1, arg.2)
R> t.Cpp.1000 <- proc.time()[3]
R> out.cfc.Cpp.1000 <- cfc(f.list.Cpp, arg.list, nobs.pred * nsmp, tout,
+    rel.tol = rel.tol)
R> t.Cpp.1000 <- proc.time()[3] - t.Cpp.1000
R> cat("t.Cpp - 1000 samples", t.Cpp.1000, "sec\n")
```

```
t.Cpp - 1000 samples 37.016 sec
```

Further speedup can be achieved by multi-threading, using the `ncores` parameter:

```
R> ncores <- 2
R> t.Cpp.1000.par <- proc.time()[3]
R> out.cfc.Cpp.1000.par <- cfc(f.list.Cpp, arg.list, nobs.pred * nsmp, tout,
+    rel.tol = rel.tol, ncores = ncores)
R> t.Cpp.par.1000.par <- proc.time()[3] - t.Cpp.1000.par
R> cat("t.Cpp.par - 1000 samples:", t.Cpp.1000.par, "sec\n")

t.Cpp.par - 1000 samples: 19.605 sec
```

The speedup for `nsmp=1000` remains quite acceptable:

```
R> cat("parallelization speedup - C++:", t.Cpp.1000 / t.Cpp.1000.par, "\n")

parallelization speedup - C++: 1.88809
```

4.4. Example 4: Combining parametric and non-parametric survival models in CFC

The logical separation of survival models and competing-risk analysis in CFC offers the flexibility to use entirely different type of models for different causes. We saw an example of this in Section 4.1, where we used Weibull and exponential distributions in the `cfc.survreg` function. It is even possible to combine parametric and non-parametric survival models in CFC, as we illustrate next.

The random forest survival model in **randomForestSRC** package produces discretized survival curves. When survival curves for all causes are discrete, combining them does not require integration, and the `survfit` function in **survival** package offers this functionality. However, if at least one cause has a continuous survival function, we can use **CFC** to produce continuous output. The key step is to write a wrapper function around the discrete survival functions that uses interpolation to create a continuous interface.

As before, we begin by using the utility function `cfc.prepdata` to prepare the data for cause-specific survival analysis:

```
R> prep <- cfc.prepdata(Surv(time, cause) ~ platelet + age + tcell, bmt)
R> f1 <- prep$formula.list[[1]]
R> f2 <- prep$formula.list[[2]]
R> dat <- prep$dat
R> tmax <- prep$tmax
```

We choose a parametric Weibull regression for the first cause, taking care to keep `x` for prediction:

```
R> library("survival")
R> reg1 <- survreg(f1, dat, x = TRUE)
```

For the second cause, we build a random forest survival model. This is followed by implementing a function to provide a continuous-output interface to the prediction function provided by the package:

```
R> library("randomForestSRC")
R> reg2 <- rfsrc(f2, dat)
R> rfsrc.survfunc <- function(t, args, n) {
+    which.zero <- which(t < .Machine$double.eps)
+    ret <- approx(args$time.interest, args$survival[n, ], t, rule = 2)$y
+    ret[which.zero] <- 1.0
+    return (ret)
+  }
```

Finally, we construct the function and argument lists for `cfc` and call the function:

```
R> f.list <- list(cfc.survreg.survprob, rfsrc.survfunc)
R> arg.list <- list(reg1, reg2)
R> tout <- seq(0.0, tmax, length.out = 10)
R> cfc.out <- cfc(f.list, arg.list, nrow(bmt), tout, rel.tol = 1e-4)
```

This `cfc` object can be summarized as before, but this time we will simply average across all observations, i.e., there will be no samples remaining since the framework was not Bayesian. See Figure 5. Note that, in this case, our reduction function isn't producing samples but point estimates, and therefore quantile calculation is meaningless.

```
R> plot(summary(cfc.out, f.reduce = mean))
```

5. Discussion

Summary: Bayesian techniques offer many, well-recognized methodological advantages – particularly in survival analysis – including a general estimation framework, consistent treatment and propagation of uncertainty, validity for small (and large) samples, ability to incorporate prior information, ease of model comparison and validation, and natural handling of missing data (Ibrahim, Chen, and Sinha 2005). Translating this broad appeal into widespread *adoption* of Bayesian techniques is critically dependent on the availability of software for their easy and efficient estimation and prediction. Most effort in developing high-performance Bayesian software, however, has been focused on the estimation side, with research covering areas such as efficient (Girolami and Calderhead 2011; Mahani, Hasan, Jiang, and Sharabiani 2015), self-tuning (Homan and Gelman 2014), and parallel (Mahani and Sharabiani 2015b; Gonzalez, Low, Gretton, and Guestrin 2011) MCMC sampling, among others. In contrast, relatively little attention has been paid to providing techniques and tools for full Bayesian *prediction*, leaving many practitioners with no choice but to use premature, point summaries of model parameters to produce approximate, mean values for predicted entities. (See, however, **?**.)

We presented the R package **CFC** for Bayesian, and non-Bayesian, cause-specific competing-risk analysis of parametric and non-parametric surviva models with an arbitrary number of

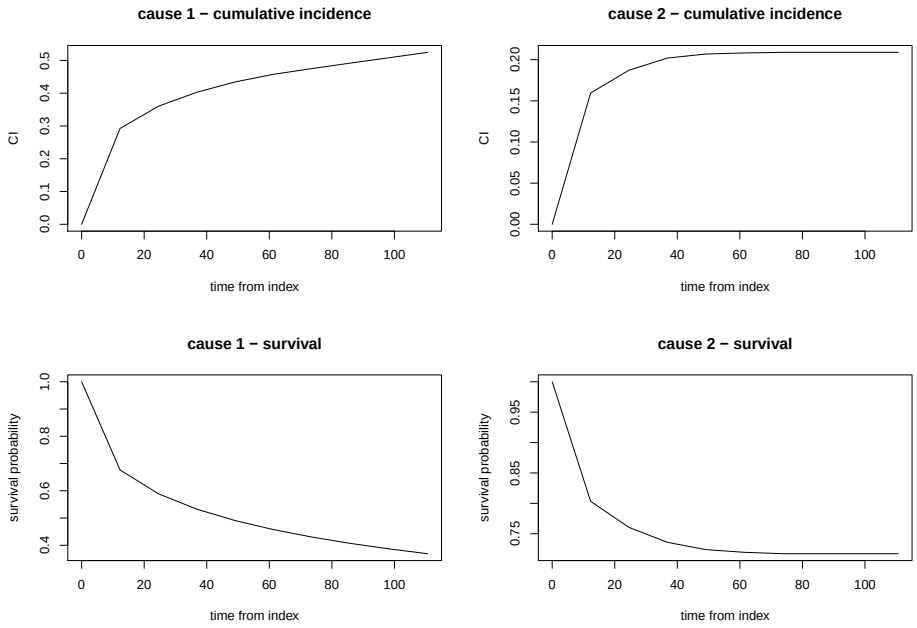

Figure 5: Population-average, cause-specific cumulative incidence and survival functions corresponding to example 4.

causes. Three usage modes available in **CFC** offer a combination of ease-of-use and extensibility: While a single-line call to `cfc.survreg` performs parametric survival regression followed by competing-risk analysis, the core function `cfc` allows users to include other survival models, including non-parametric ones, in cause-specific competing-risk analysis. The R interface can be used for small data sets and/or non-Bayesian models, where the computational workload is modest. It can also serve as a reference for implementing the C++ version of the survival functions in order to significantly improve peformance for computationally-demanding problems. The quadrature algorithm used in **CFC** can be considered an implicit variable transformation method that circumvents potential end-point singularities, and also enhances usability by removing the need to supply the cause-specific hazard functions. In addition to the C++ API, other performance optimization techniques in **CFC** such as cross-cause worksharing and OpenMP parallelization have combined to put a full Bayesian approach to survival and competing-risk analysis within the reach of practitioners.

Potential future work: According to Equations 3 and 4, we must have:

$$\sum_k \Delta F_k = -\Delta E = -\Delta \prod_k S_k, \tag{15}$$

where Δ refers to the change in a quantity during an integration time step. However, the generalized Simpson step (Equations 13a and 13b), when applied to all causes, does not mathematically satisfy this condition. In other words, the sum of event-free probability and all cumulative incidence functions does not mathematically add up to 1 after discrete time evolutions. Similarly, the generalized trapezoidal rule of Equation 14, when applied to each cause in isolation, does not satisfy this property in general (but it does for $K = 2$). It is possible to extend the trapezoidal step to satisfy this property, but without an equivalent

extension of the Simpson rule, we would need to develop an alternative approach to error analysis. This is because, absent the Simpson rule as the main method, the trapezoidal step would change role from reference to main method to become the return value of the integral. Developing a coherent framework that satisfies Equation 15 and includes proper error analysis is an interesting potential area of research.

In terms of software development, current implementation of `cfc.survreg` is R based. This is partially justified since the underlying models, from **survival** package, are non-Bayesian. Therefore, as long as data sizes are small, computational workloads in `cfc.survreg` remain manageable without porting to C++. However, for large data sets this will be inadequate, and therefore a high-performance implementation is warranted to cover the emerging, big-data use-cases.

OpenMP parallelization of `cfc` provides an immediate and significant performance gain, but there are other, more advanced opportunities for performance optimization. For example, Single-Instruction, Multiple-Data (SIMD) parallelization has recently been successfully applied to Bayesian problems (Mahani and Sharabiani 2015b). Given the increasing width of vector registers in modern CPUs (Jeffers and Reinders 2013), taking advantage of SIMD parallelization offers an opportunity for meaningful performance improvements. A second area of investigation, especially for large data sets with memory-bound performance ceilings, is reducing data movement throughout the memory hierarchy. Techniques such as improving data layout to permit unit-stride access, and NUMA-aware memory allocation to minimize cross-socket data transfer over slower bus interconnects (Mahani and Sharabiani 2015b) can help minimize data movement and improve cache and memory bandwidth utilization.

References

Analytics R, Weston S (2015a). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.10, URL http://CRAN.R-project.org/package=doParallel.

Analytics R, Weston S (2015b). *foreach: Provides Foreach Looping Construct for R*. R package version 1.4.3, URL http://CRAN.R-project.org/package=foreach.

Andersen PK, Klein JP, Rosthøj S (2003). "Generalised linear models for correlated pseudo-observations, with applications to multi-state models." *Biometrika*, **90**(1), 15–27.

Bailey DH, Jeyabalan K, Li XS (2005). "A comparison of three high-precision quadrature schemes." *Experimental Mathematics*, **14**(3), 317–329.

Breiman L (2001). "Random forests." *Machine learning*, **45**(1), 5–32.

Chen CH, Chang IS, Hsiung CA (2015). *NPMLEcmprsk: Type-Specific Failure Rate and Hazard Rate on Competing Risks Data*. R package version 2.1, URL http://CRAN.R-project.org/package=NPMLEcmprsk.

Datta GS (2005). "Location–Scale Family." *Encyclopedia of Biostatistics*.

Eddelbuettel D (2013). "Calling R Functions from C++." URL http://gallery.rcpp.org/articles/r-function-from-c++/.

Eddelbuettel D, François R, Allaire J, Chambers J, Bates D, Ushey K (2011). "Rcpp: Seamless R and C++ integration." *Journal of Statistical Software*, **40**(8), 1–18.

Eddelbuettel D, Sanderson C (2014). "RcppArmadillo: Accelerating R with high-performance C++ linear algebra." *Computational Statistics and Data Analysis*, **71**, 1054–1063. URL http://dx.doi.org/10.1016/j.csda.2013.02.005.

Fine JP, Gray RJ (1999). "A proportional hazards model for the subdistribution of a competing risk." *Journal of the American statistical association*, **94**(446), 496–509.

Friedman JH (2001). "Greedy function approximation: a gradient boosting machine." *Annals of statistics*, pp. 1189–1232.

Gelman A, Hill J (2006). *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press.

Gilbert P, Varadhan R (2012). *numDeriv: Accurate Numerical Derivatives*. R package version 2012.9-1, URL http://CRAN.R-project.org/package=numDeriv.

Girolami M, Calderhead B (2011). "Riemann manifold langevin and hamiltonian monte carlo methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **73**(2), 123–214.

Gonzalez J, Low Y, Gretton A, Guestrin C (2011). "Parallel gibbs sampling: From colored fields to thin junction trees." In *International Conference on Artificial Intelligence and Statistics*, pp. 324–332.

Gray B (2014). *cmprsk: Subdistribution Analysis of Competing Risks*. R package version 2.2-7, URL http://CRAN.R-project.org/package=cmprsk.

Haller B, Schmidt G, Ulm K (2013). "Applying competing risks regression models: an overview." *Lifetime data analysis*, **19**(1), 33–58.

Homan MD, Gelman A (2014). "The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo." *The Journal of Machine Learning Research*, **15**(1), 1593–1623.

Ibrahim JG, Chen MH, Sinha D (2005). *Bayesian survival analysis*. Wiley Online Library.

Ishwaran H, Gerds TA, Kogalur UB, Moore RD, Gange SJ, Lau BM (2014). "Random survival forests for competing risks." *Biostatistics*, **15**(4), 757–773.

Ishwaran H, Kogalur U (2015). *Random Forests for Survival, Regression and Classification (RF-SRC)*. R package version 1.6.1, URL http://cran.r-project.org/web/packages/randomForestSRC/.

Jeffers J, Reinders J (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

Larson MG, Dinse GE (1985). "A mixture model for the regression analysis of competing risks data." *Applied statistics*, pp. 201–211.

Laurie D (1997). "Calculation of Gauss-Kronrod quadrature rules." *Mathematics of Computation of the American Mathematical Society*, **66**(219), 1133–1145.

Mahani AS, Hasan A, Jiang M, Sharabiani MT (2015). *sns: Stochastic Newton Sampler (SNS)*. R package version 1.1.0.

Mahani AS, Sharabiani MT (2015a). *BSGW: Bayesian Survival Model with Lasso Shrinkage Using Generalized Weibull Regression*. R package version 0.9.1, URL http://CRAN.R-project.org/package=BSGW.

Mahani AS, Sharabiani MT (2015b). "SIMD parallel MCMC sampling with applications for big-data Bayesian analytics." *Computational Statistics & Data Analysis*, **88**, 75–99.

Maja Pohar Perme, Gerster M (2012). *pseudo: Pseudo - observations*. R package version 1.1, URL http://CRAN.R-project.org/package=pseudo.

Nicolaie M, van Houwelingen HC, Putter H (2010). "Vertical modeling: A pattern mixture approach for competing risks modeling." *Statistics in medicine*, **29**(11), 1190–1205.

Okamura H (2015). *deformula: Integration of One-Dimensional Functions with Double Exponential Formulas*. R package version 0.1.1, URL http://CRAN.R-project.org/package=deformula.

Piessens R, de Doncker-Kapenga E, Überhuber CW, Kahaner DK (2012). *QUADPACK: a subroutine package for automatic integration*, volume 1. Springer Science & Business Media.

Prentice RL, Kalbfleisch JD, Peterson Jr AV, Flournoy N, Farewell V, Breslow N (1978). "The analysis of failure times in the presence of competing risks." *Biometrics*, pp. 541–554.

Press WH (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

Ridgeway G (2015). *gbm: Generalized Boosted Regression Models*. R package version 2.1.1, URL http://CRAN.R-project.org/package=gbm.

Stroud AH, Secrest D (1966). *Gaussian quadrature formulas*, volume 39. Prentice-Hall Englewood Cliffs, NJ.

Takahasi H, Mori M (1974). "Double exponential formulas for numerical integration." *Publications of the Research Institute for Mathematical Sciences*, **9**(3), 721–741.

Therneau T (2013). "A Package for Survival Analysis in S. R package version 2.37-4. 2013."

Therneau T (2015). *A package for survival analysis in S*. R package version 2.38, URL http://CRAN.R-project.org/package=survival.

Wynn P (1966). "On the convergence and stability of the epsilon algorithm." *SIAM Journal on Numerical Analysis*, **3**(1), 91–122.

A. Proof of generalized Simpson rule

Our objective is to derive an approximation for $\int_a^b f(t)\,dg(t)$, using a second-order Taylor-series expansion of $f(t)$ in terms of $g(t)$ over the interval $[a, b]$:

$$f(t) = f_a + \alpha\,(g(t) - g_a) + \frac{1}{2}\beta\,(g(t) - g_a)^2 \tag{16}$$

where $f_a \equiv f(t = a)$ and similarly for f_b, g_a and g_b. To find α and β, we require that this quadratic function passes through (f_a, g_a), (f_b, g_b), and (f_m, g_m), where $f_m \equiv f(m = (a + b)/2)$, and similarly for g_m. Th first of three conditions, at $t = a$, is already satisfied in Equation 16. The next two conditions lead to

$$f_b = f_a + \alpha\,(g_b - g_a) + \frac{1}{2}\beta\,(g_b - g_a)^2, \tag{17a}$$

$$f_m = f_a + \alpha\,(g_m - g_a) + \frac{1}{2}\beta\,(g_m - g_a)^2. \tag{17b}$$

Solving for α and β leads to:

$$\alpha = \frac{(f_m - f_a)(g_b - g_a)^2 - (f_b - f_a)(g_m - g_a)^2}{(g_m - g_a)(g_b - g_m)(g_b - g_a)}, \tag{18a}$$

$$\beta = \frac{2\{(f_b - f_a)(g_m - g_a) - (f_m - f_a)(g_b - g_a)\}}{(g_m - g_a)(g_b - g_m)(g_b - g_a)}. \tag{18b}$$

Integrating Equation 16 over $[a, b]$ leads to the following approximation:

$$\int_a^b f(t)\,dg(t) \cong I_{gs}(f, g; a, b) = f_a(g_b - g_a) + \frac{1}{2}\alpha\,(g_b - g_a)^2 + \frac{1}{6}\beta\,(g_b - g_a)^2. \tag{19}$$

Substituting α and β from Equations 18a and 18b into Equation 19, while defining $g_1 \equiv g_m - g_a$ and $g_2 \equiv g_b - g_m$, and some algebraic manipulation, leads to:

$$I_{gs} = \frac{g_1 + g_2}{6\,g_1\,g_2}\left\{f_a\,(2\,g_1\,g_2 - g_2^2) + f_m\,(g_1 + g_2)^2 + f_b\,(2\,g_1\,g_2 - g_1^2)\right\}. \tag{20}$$

A second change of variable, using $h \equiv g_1 + g_2 = g_b - g_a$ and $\delta \equiv g_1 - g_2 = 2g_m - (g_a + g_b)$ allows us to re-express the above in the following form, after some further algebraic manipulations:

$$I_{gs} = \frac{1}{6}\frac{h}{h^2 - \delta^2}\left\{f_a(h^2 + 2h\delta - 3\delta^2) + 4f_m h^2 + f_b(h^2 - 2h\delta - 3\delta^2)\right\}. \tag{21}$$

A final symbol definition, $r \equiv h/\delta$, readily leads to Equation 13a.

B. Setup

Below is the R session information used in producing R output in Section 4.

```
R> sessionInfo()

R version 3.3.3 (2017-03-06)
Platform: x86_64-redhat-linux-gnu (64-bit)
Running under: Amazon Linux AMI 2017.03

Matrix products:

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
```

```
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base

other attached packages:
[1] randomForestSRC_2.4.2 Rcpp_0.12.11          CFC_1.1.0
[4] Hmisc_4.0-3           ggplot2_2.2.1         Formula_1.2-1
[7] survival_2.41-3       lattice_0.20-34       BSGW_0.9.2

loaded via a namespace (and not attached):
 [1] compiler_3.3.3            MfUSampler_1.0.4
 [3] RColorBrewer_1.1-2        plyr_1.8.4
 [5] methods_3.3.3             base64enc_0.1-3
 [7] iterators_1.0.8           tools_3.3.3
 [9] rpart_4.1-10              digest_0.6.12
[11] tibble_1.3.3              gtable_0.2.0
[13] htmlTable_1.9             checkmate_1.8.2
[15] rlang_0.1.1               Matrix_1.2-8
[17] foreach_1.4.3             parallel_3.3.3
[19] RcppArmadillo_0.7.900.2.0 gridExtra_2.2.1
[21] coda_0.19-1               stringr_1.2.0
[23] cluster_2.0.5             knitr_1.16
[25] htmlwidgets_0.8           grid_3.3.3
[27] nnet_7.3-12               data.table_1.10.4
[29] HI_0.4                    foreign_0.8-67
[31] latticeExtra_0.6-28       magrittr_1.5
[33] scales_0.4.1              backports_1.1.0
[35] codetools_0.2-15          htmltools_0.3.6
[37] ars_0.5                   splines_3.3.3
[39] abind_1.4-5               colorspace_1.3-2
[41] stringi_1.1.5             acepack_1.4.1
[43] lazyeval_0.2.0            doParallel_1.0.10
[45] munsell_0.4.3
```

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.
1725 I St NW
Washington, DC 20006
E-mail: alireza.s.mahani@gmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd