POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

C++ Implementation of Time-Varying Shared Frailty Cox Models

Project Report for the PACS course
Mathematical Engineering

Author: **Giulia Romani**

Student ID: 964167
Professor: Prof. Luca Formaggia
Teaching Assistants: Dr. Matteo Caldana, Dr. Paolo Joseph Baioni
Advisors: Dr. Chiara Masci, Dr. Alessandra Ragni
Academic Year: 2022-23

Contents

Introduction

In the context of *survival analysis*, several statistical regression models have been proposed to explain the relation between the time to the occurrence of a precise event and some variables (or covariates) describing individual characteristics. Among all these models, the *Time-Varying Shared Frailty Cox Model* allows us to consider the subdivision of the population into groups and to introduce a time-varying random term (frailty), that quantifies the group characteristics and describes how they vary in time, providing information about how the individual risk of facing the event is influenced by the individual belonging to a group.
In the family of time-varying shared frailty Cox models, we study and apply the models introduced in the paper of Wintrebert et al. [15]. Since no codes fitting the models are available in [15], we implemented new ones from scratch, using the software *R* [1]. However, we soon realized that the execution time necessary to produce an output is too high, making the application of these models impossible. We thus decided to implement them in C++, exploiting also the advantages of parallel computing to speed up the data processing.
A consistent portion of the R codes has been rewritten in C++ and it is actually able to produce the same results given by R, but in a faster way.

This report is structured as follows. First of all, in Chapter 1 we briefly describe the dataset on which the models are applied. In Chapter 2, we introduce the basis of *survival analysis* and the time-varying models. In Chapter 3 and 4, we theoretically and practically describe how the codes are implemented in C++ and in Chapter 5, they are applied to the dataset and the results are presented.

1 | PoliMi Dataset

The event we previously mentioned corresponds to the recurrent phenomenon of the academic dropout and the main focus of the work thesis, this project is associated to, consists in studying how it changes across different academic faculties[1] and across time. The results we will be able to obtain from the application of these codes would be used by Politecnico di Milano to take preventive actions to avoid early students dropout.

To perform this ambitious analysis, we make use of an administrative database provided by PoliMi, that contains data of students enrolled between the beginning of 2010 and the 29th of September 2022 (*extraction date*), in one of the engineering bachelor programs actually given in a campus of PoliMi (Milano Leonardo and Bovisa).
At the moment of each student's enrollment, several data are collected to study the dropout event from different points of view. These data regard individual school background, early academic career or more strictly personal information, but only a few of them have a practical utility. Indeed, we consider the gender (*Gender*), the residence (*Origins*), the registered score at the admission test (*Admission Score*), the attended degree course (*Faculty*) and the number of CFU[2] passed by the end of the first semester (*CFUP*).
More in detail, *CFUP* and *Admission Score* are the only two numerical variables and they have been standardized to have more homogeneous data. On the other hand, the remaining are categorical, with *Origins* having three levels (*Commuter*, *Milanese*, *Offsite*) and grouping the students according to the location of their residence with respect to one of the PoliMi campus.
Finally, *Faculty* has 16 different levels corresponding to the different degree courses the students can decide to enroll in. For privacy issues related to internal policies, the faculties have been anonymized and are now represented by the four letters *EngA*, *EngB*, . . . , *EngP*.

[1]They correspond to the groups we were taking about in the Introduction.
[2]Credito Formativo Universitario

Each student is followed during his/her academic career for a maximum of 6 semesters (from now on called *Follow-Up* and equal to the theoretical length of a bachelor degree) and a particular attention is devoted to the dropout event. A new variable, called *time-to-event*, is individually defined to register the precise time instant, during the follow-up, in which a student decides to withdraw from university. Of course, if a student either graduates before the end of the third year, does not complete the studies in time or abandon university later than the follow-up, the default assigned time will correspond to one instant past the end of the follow-up[3]. The decision of distinguishing dropouts from other type of students is made according to the structure of the models we are going to apply.

One technical detail that deserves to be deepened is the unit of measure of all the temporal variables here introduced and used. They are defined in *semesters*, where a single semester is defined as half of an academic year. This choice is simply motivated by the subdivision itself of a year at university. For instance, if a student has a time-to-event $t_i = 2$, we can deduce he/she has abandoned university at the end of the second semester or, equivalently, at the end of the first year. Conversely, if a student has a value $t_i = 6.1$, that is greater than the follow-up, we do not know his/her status, but for sure he/she did not dropped by the end of the third year.

To conclude this part devoted to the description of the PoliMi dataset, we anticipate that each dataset used for the model simulations and applications corresponds to a small subset of the main database. It is thought as a matrix, composed of different rows and always 5 columns, respectively corresponding to the number of students enrolled at PoliMi in a precise academic year (either 2010, 2018 or $2017 - 2018$), and the number of covariates[4]. The reason why the *Faculty* variable is not considered as a covariate will be cleared in Section 2.2.3 and it is consequently saved in a separate vector.

[3]Any value greater than the length of the follow-up can be accepted.

[4]They are *Gender*, *Origins*, *Admission Score* and *CFUP*. Categorical variables have to be transformed into dummy variables. For n levels, $(n - 1)$ dummy variables must be introduced.

2 | Theory

2.1. Basis of survival analysis

Survival analysis is a collection of statistical procedures for data analysis for which the outcome variable of interest is the *survival time*, that measures the time from a particular starting date (*origin event*) to a precise endpoint (*event of interest*). Such type of data are called *time-to-event data.*
In general, any designed experience that could happen to an individual or to a system could be considered an event: disease incidence, relapse from remission, death after diagnosis from cancer or, more related to the context of this work, dropout from university. Individuals can enter the study at any time and the length of the study period (called *follow-up*) can be measured in days, weeks, months, years or any other well defined unit of measure.

A fundamental concept in survival analysis is *censoring.* It occurs when we have some information about individual survival time but we do not know precisely when the event happened. Three forms of censoring exist: *right*, *left*, *interval.* The former is the most common, where we may be prevented from observing the precise time to the event because either the individual did not experience the event before the end of the study (*end-of-study time*), or the study team lost contact with the individual (*loss to follow-up*) or the individual withdrew from the study for another reason. In the first case, we simply assume that the time-to-event is greater than the end-of-study time.

From now on, we only focus on right censoring. Let C be the random censoring time, T the random time-to-event and define T^* as the minimum between C and T, corresponding to the registered survival time. To know if an individual experienced the event, we have to introduce a further (binary) variable δ that indicates either censorship or failure: $\delta_i = 1$ if the individual i failed during the follow-up and $\delta_i = 0$ if i is censored, meaning that i either did not deal with the event or faced it after the end of study.
Therefore, the outcome data in survival analysis consists of a couple of random variables

(T_i^*, δ_i) for each i, $i = 0, ..., n$, where:

$$T_i^* = min\{C_i, T_i\} \quad \text{and} \quad \delta_i = 1 \text{ if } T_i \leq C_i,\ \delta_i = 0 \text{ if } T_i > C_i$$

Survival and Hazard function

Denote by t any specif value of interest of the non-negative random variable T and define the *survival function* $S(t)$ as the function that gives the probability of an individual to survive longer than the time t. In formula, $S(t) = P(T > t) = 1 - F_T(t)$, where $F_T(t)$ indicates the cumulative distribution function of T.
$S(t)$ assumes values in the range $0 - 1$ and it is non-increasing. In detail, at time $t = 0$, $S(t) = S(0) = 1$ since nobody has experienced the event yet, while at $t = \infty$ $S(t) = S(\infty) = 0$ since it is supposed that all individuals will deal with the event, sooner or later.

In case of only uncensored data, the survival function can be estimated starting from the empirical distribution function $F_n(t)$. However, in real application a censoring mechanism kicks in, and $S(t)$ can be evaluated from data through a non-parametric statistic called *Kaplan-Meier estimator* [10]. In both cases, the obtained survival is a step-function made of horizontal segments and it may not reach zero if the follow-up is finite, as it could be in real data.

While the survival function $S(t)$ indicates the percentage of subjects in a cohort that are still event-free at time t, the *hazard function* $h(t)$ represents the instantaneous risk of failure for the event under consideration, conditionally on the fact that it has not occurred yet. $h(t)$ has range in the interval $[0, \infty]$ with null value at time $t = 0$, and then its trend is mainly governed by the registered failures: the higher the hazard at a generic time t, the higher the risk of failure for an individual at t.

2.2. Cox Proportional-Hazard Model

In this section we are going to present the *Cox Proportional-Hazard* model, a popular statistical tool used in the context of survival data analysis with the aim of exploring the relationship between the survival of an individual and some explanatory variables, called covariates.

2.2.1. Time-Invariant Cox PH Model

The *Time-Invariant Cox PH Model* is the basic version of the Cox PH model in which the covariates are time-independent, i.e. they represent a quantity that does not change with time. For instance, we have here considered the score of the admission test at Politecnico or the number of credits passed at the end of the first semester.

The Cox PH model for the i^{th} subject is usually written in terms of the hazard function at time t and according to a given set of covariates $x_{i,r}$ ($\forall r = 1, ..., R$), as here reported:

$$h_i(t, \mathbf{x}_i) = h_0(t) exp\left\{\sum_{r=1}^{R} \beta_r x_{i,r}\right\} \tag{2.1}$$

In detail, the formula says that the hazard at time t is the product of two quantities. The first one is the *baseline hazard function* $h_0(t)$, that only depends on time and models the hazard in absence of covariates, i.e. when they assume null value, they are set to the default categorical level or they are simply not present.
The second term is the exponential of the linear sum of each covariate $x_{i,r}$ multiplied by the corresponding coefficient β_r.

At this point, the only parameters involved in the model are the coefficients β_r ($r = 1, ..., R$) and they are estimated by maximizing the *partial log-likelihood function*:

$$\hat{\boldsymbol{\beta}} = \underset{\beta \in R^P}{\arg\max} \ln(L(\boldsymbol{\beta})) \qquad \ln(L(\boldsymbol{\beta})) = \sum_{i=1}^{N_{fail}} \left[\boldsymbol{x}_i^T \boldsymbol{\beta} - ln \left(\sum_{k \in R(t_i)} exp(\boldsymbol{x}_k^T \boldsymbol{\beta}) \right) \right]$$

where N_{fail} indicates the number of individuals experiencing the failure event.

2.2.2. Time-Invariant Shared Frailty Cox Model

These explanatory variables describes what is usually called *observed heterogeneity* [8]. However, it is not always possible to account for all the relevant covariates in the description of the phenomenon under study, meaning that there is still some heterogeneity that cannot be measured in any possible way. It is commonly call *unobserved heterogeneity* and its effects on life-times are referred to as *frailty*. It is a random effect that acts multiplicatively on the hazard function and its spread (e.g. variance) indicates the intensity of such unobserved heterogeneity.

One of the most common model based on frailty is the *Time-Invariant Shared Gamma-Frailty Cox Model*, where the frailty is used to model the dependence of survival times in clustered data and it is shared among individuals belonging to the same cluster. In this context, the subdivision in cluster is provided by the faculty each student belongs to.
Mathematically, suppose that the whole population is divided into N clusters, singularly represented by a realization Z_j of the frailty, and n_j individuals are part of cluster j. Then, the hazard function of individual i in group j is specified by:

$$h_{ij}(t|Z_j) = Z_j h_0(t)\text{exp}(\boldsymbol{\beta}^T \mathbf{x}_{ij}) \tag{2.2}$$

The subjects in the same cluster share the same frailty term and, conditionally on it, their life-times are assumed to be independent. Thus, it follows that the hazard function for cluster j assumes the form:

$$h_j(t|Z_j) = \prod_{i=1}^{n_j} h_{ij}(t|Z_j) = \prod_{i=1}^{n_j} Z_j h_0(t)\text{exp}(\boldsymbol{\beta}^T \boldsymbol{x}_{ij})$$

Since the frailty term is a random effect, several prior distributions can be used to represent different ways of expressing unobserved heterogeneity and the most common is the *gamma*.
Suppose that each frailty term $Z_j \sim Gamma(\eta, \xi)$, $\forall j = 1, \dots, N$ is distributed according to a gamma of parameters $\eta, \xi > 0$ and, moreover, suppose they are independent. For identifiability purposes, the expectation of the frailty is fixed to 1 so that the restriction $\eta = \xi$ is applied and $Z_j \sim Gamma(\xi, \xi)$. It can be deduced that the variance of the frailty $Var(Z_j) = \frac{1}{\xi} = \theta$ quantifies the degree of between clusters variability.

The parameters of the *Shared Gamma-Frailty Cox Model* are the regression coefficients β_r $\forall r = 1, \dots, R$ and the frailty terms Z_j $\forall j = 1, \dots, N$ and they are estimated maximizing the *partial penalized log-likelihood* represented by:

$$PPL = PL(\boldsymbol{\beta}, \mathbf{Z}; data) - g(\mathbf{Z}, \theta) \tag{2.3}$$

over $\boldsymbol{\beta}, \mathbf{Z}$, as indicated in [14].
The first term $PL(\boldsymbol{\beta}, \mathbf{Z}; data)$ corresponds to the logarithm of the usual Cox partial likelihood defined in Section 2.2.1, but accounting for the presence of the frailty term. Concerning the second term of Formula 2.3, it is a penalty function that, for a gamma distributed frailty, has shape: $g(\mathbf{Z}, \theta) = -1/\theta \sum_{j=1}^{N} [Z_j - \text{exp}(Z_j)]$.

2.2.3. Time-Varying Shared Frailty Cox Model

The problem associated to the previous model is that the frailty is assumed to be constant for the whole follow-up, indicating that the effects[1] specific to a centre do not change in time and always behave in the same way. This condition could be too restrictive in certain situation, in which the characteristics of a centre could vary over time. This motivation leads the researchers to extend the frailty model to allow a time-dependence of the frailties. Several models have been proposed in literature (e.g [12], [15], [11], [9]), under the name of *Time-Varying Shared Frailty Cox Model*, and we are going to present the ones described in [15].
One possibility to define a time-varying frailty model is to partition the whole time-domain into a certain number of intervals, potentially of different length, and to let the frailty term depends on both the centre and the time-interval. Accordingly, the baseline hazard function $h_0(t)$ is substituted by different interval-baseline components, that need to be estimated, so that this model is transformed into a parametric one.
Mathematically, assume that the whole population is divided into N number of groups (or centres) and define t_{ij} as the time-to-event for the ith individual in the jth group. Suppose to divide the time-domain into L intervals $I_k = [a_{k-1}, a_k)$, $k = 1, \dots, L$, with discrete time-points $0 = a_0 < a_1 < \cdots < a_L = \infty$. In this way, d_{ijk} is the event variable in I_k for i in j, such that $d_{ijk} = 1$ if t_{ij} is uncensored in I_k and 0 otherwise. The usual $0-1$ indicator $d_{ij} = \delta_{ij}$ of the event can be obtained as $\sum_k d_{ijk}$.

At this point, define Z_{jk} as the unobservable frailty of the jth centre for the kth time-interval. Conditionally on Z_{jk}, the hazard function h_{ijk} for the ith individual in the jth group in I_k is given by the general expression:

$$h_{ijk}(t_{ij}|Z_{jk}) = Z_{jk}\ \mathrm{e}^{(\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)} \tag{2.4}$$

where $\boldsymbol{x}_{ij}$ and $\boldsymbol{\beta}$ are, respectively, the usual individual vector of covariates and the global regressors, while ϕ_k is the newly introduced baseline log-hazard for the kth interval, $\forall k$.
Looking at the structure of the individual hazard function, we can already deduce that both the regressor coefficients and the baseline log-hazard are not parameters known a priori, but have to be estimated with a suitable procedure. Their number uniquely depends on the database setting (i.e. number of covariates) and on the partition of the time-domain, for an overall number of parameters equal to $(L + R)$.

[1]Characteristics/effects of a group/cluster/faculty may be related to the content and complexity of each faculty exam, meaning the arguments, their practical applications and the effective amount of time devoted to study, or the students' expectation of the faculty itself.

In the following paragraphs, we present the three different time-varying shared frailty Cox models, that are later implemented: the *Adapted Paik et al.'s Model*, the *Centre-Specific Frailty Model with Power Parameter* and the *Stochastic Time-Dependent Centre-Specific Frailty Model*[2]. For each one of them, we illustrate the frailty structure, the parameters involved and their statistical distribution. In order to estimate all these parameters, we need to compute the log-likelihood function and maximize it using a suitable optimization algorithm. However, as we will explain in Chapter 3, the optimization phase is not here considered.

The Adapted Paik et al.'s Model

The authors of [15] define the time-varying frailty term as $Z_{jk}(t_{ij}) = (\alpha_j + \epsilon_{jk})$ for $t_{ij} \in I_k$, that leads to the hazard function $h_{ijk}(t_{ij}|\alpha_j, \epsilon_{jk}) = (\alpha_j + \epsilon_{jk})\ \mathrm{e}^{(\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)}$, where the parameters α_j and ϵ_{jk} are chosen to be independent and distributed according to:

- $\alpha_j \sim Gamma(\mu_1/\nu, 1/\nu)\ \forall j$
- $\epsilon_{jk} \sim Gamma(\mu_2/\gamma_k, 1/\gamma_k)\ \forall j, k$

with $\mu_1, \mu_2, \nu, \gamma_k(\forall k) > 0$ and the constraint $E[Z_{jk}] = \mu_1 + \mu_2 = 1$ needed for the identifiability.

One important consideration is that the expectation of Z_{jk} is identically equal to 1 in each interval, but the variance of Z_{jk} is allowed to vary between different intervals. Moreover, thanks to the independence of α_j and ϵ_{jk} $(\forall j, k)$, this variance can be derived as the sum of the variance of its components as: $var(Z_{jk}) = \mu_1\nu + \mu_2\gamma_k$. The likelihood L_j for centre j is built starting from:

$$L_j = \int_0^\infty \prod_{i,k} \lambda_{ijk}(t_{ij}|\alpha_j, \epsilon_{jk})^{d_{ijk}}\ \mathrm{e}^{-\Lambda_{ijk}(t_{ij}|Z_{jk})} g(\alpha_j) \prod_k g(\epsilon_{jk}) d\alpha_j d\epsilon_{jk}$$

where $g(\alpha_j)$ and $g(\epsilon_{jk})$ are the gamma density functions of, respectively, α_j and ϵ_{jk} $(\forall j, k)$. Thanks to the analytical resolution of the integral through the definition of the gamma

[2]From now on, they are referred to with: the *Adapted Paik eaM*, the *CSFM with Power Parameter* and the *Stochastic Time-Dependent CSFM*.

function A.2 reported in Appendix A, we get the final form of L_j as:

$$L_j = \prod_{i,k}[\mathrm{e}^{(\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)}]^{d_{ijk}} \prod_k \sum_{l=0}^{d_{j,k}} \binom{d_{j,k}}{l} \frac{\Gamma(\mu_1/\nu+l)}{\Gamma(\mu_1/\nu)} \frac{(1/\nu)^{\mu_1/\nu}}{(1/\nu+A_{j..})^{(\mu_1/\nu+l)}} \cdot \\ \cdot \frac{\Gamma(\mu_2/\gamma_k+d_{j.k}-l)}{\Gamma(\mu_2/\gamma_k)} \frac{(1/\gamma_k)^{\mu_2/\gamma_k}}{(1/\gamma_k+A_{j.k})^{(\mu_2/\gamma_k+d_{j.k}-l)}}$$

The full likelihood is $L = \prod_{j=1}^N L_j$ and thus we derive the full log-likelihood simply computing $l = \text{log}L = \sum_{j=1}^N \text{log}L_j$. Finally:

$$l = \sum_{j=1}^N \left[\sum_{i,k} d_{ijk}(\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k) - \frac{\mu_1}{\nu}\text{log}(1+\nu A_{j..}) + \sum_k \left[\frac{-\mu_2}{\gamma_k}\text{log}(1+\gamma_k A_{j.k}) \right] \right] + \\ + \sum_{j=1}^N \left[\sum_k \left[\text{log} \left(\sum_{l=0}^{d_{j.k}} \binom{d_{j,k}}{l} \frac{\Gamma(\mu_2/\gamma_k+d_{j.k}-l)}{\Gamma(\mu_2/\gamma_k)} \frac{\Gamma(\mu_1/\nu+l)}{\Gamma(\mu_1/\nu)} \frac{(A_{j.k}+1/\gamma_k)^{(l-d_{j.k})}}{(A_{j..}+1/\nu)^l} \right) \right] \right] \tag{2.5}$$

where: $A_{ijk} = e_{ijk}\, \mathrm{e}^{(\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)}$, $A_{j.k} = \sum_i A_{ijk}$, $A_{j..} = \sum_{i,k} A_{ijk}$, $d_{j.k} = \sum_i d_{ijk}$ and e_{ijk} is introduced by Formula A.1.
The full list of the model parameters is composed of $\mu_1, \nu, \boldsymbol{\beta}, \gamma_k$ and ϕ_k $(\forall k)$ and they can be estimated by maximizing this new function l.

A Centre-Specific Frailty Model with Power Parameter

The time-varying frailty term $Z_{jk}(t_{ij})$ is defined in [15] as $Z_{jk}(t_{ij}) = \alpha_j^{\gamma_k} = \mathrm{e}^{Y_j\gamma_k}$ for $t_{ij} \in I_k$, where γ_k is a fixed but unknown parameter, defined for each time-interval I_k $(k = 1, \ldots, L)$, and $Y_j = \log(\alpha_j) \sim N(0, \sigma^2)$.
This distribution choice implies $E[Z_{jk}] = E[\alpha_j]^{\gamma_k} = e^{\frac{\sigma^2\gamma_k^2}{2}}$, that approaches 1 when σ is small, and $var(Z_{jk}) = e^{2\gamma_k^2\sigma^2} - e^{\gamma_k^2\sigma^2}$. Thanks to $\sigma \to 0$ and using the notable limit $lim_{x\to 0}(\mathrm{e}^x - 1) = x$ with a generic x, we can rewrite the variance also in this way: $var(Z_{jk}) = (\gamma_k\sigma)^2$.

The likelihood L_j for centre j has the following form:

$$L_j = \int_{-\infty}^{\infty} \prod_{i,k} \left[\mathrm{e}^{(Y_j\gamma_k+\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)}\right]^{d_{ijk}} \; \mathrm{e}^{-e_{ijk}\mathrm{e}^{(Y_j\gamma_k+\boldsymbol{x}_{ij}\boldsymbol{\beta}+\phi_k)}} g(Y_j)dY_j$$

where $g(Y_j)$ is the gaussian density function of the random variable Y_j. Exploiting this choice of distribution, we can solve numerically the integral using the Gauss-Hermite quadrature formula A.3, choosing 9 points θ_q and weights w_q [7], as indicated in [15]. Finally, the full log-likelihood is:

$$l = \sum_{j=1}^{N} \left[\sum_{i,k} d_{ijk}(\boldsymbol{X}_{ij}\boldsymbol{\beta} + \phi_k) \right] - \frac{N}{2}\log(\pi) + \\ + \sum_{j=1}^{N} \log \left[\sum_{q=1}^{9} w_q \mathrm{e}^{(\sqrt{2}\sigma\theta_q \sum_{i,k} d_{ijk}\gamma_k - \sum_{i,k} e_{ijk} \; \mathrm{e}^{(\sqrt{2}\sigma\gamma_k\theta_q + X_{ij}\beta + \phi_k)})} \right] \tag{2.6}$$

and, through its maximization, we can find the unknown model parameters σ, β, γ_k and ϕ_k $(\forall k)$.

A Stochastic Time-Dependent Centre-Specific Frailty Model

The model presented in this paragraph shows how it is possible to define a time-varying frailty in which the frailty term does not depend on a precise time interval I_k, but on the whole temporal domain. Indeed, it can be easily expressed through its logarithm as $\log Z_j(t_{ij}) = (c_j + b_j t_{ij})$ for $t_{ij} \in [0, \infty)$, where the vector $\boldsymbol{cb} = (c_j, b_j) \sim N_2(\boldsymbol{0}, \Sigma)$ and the covariance matrix Σ is given by:

$$\Sigma = \begin{bmatrix} \sigma_c^2 & \sigma_{cb} \\ \sigma_{cb} & \sigma_b^2 \end{bmatrix}$$

In [15], the frailty variance $var(Z_{jk})$ is declared to be almost equal to the variance of the logarithmic frailty, which can be easily computed as $var(log(Z_{jk})) = (\sigma_c^2 + \sigma_b^2 t_{ij}^2 + 2\sigma_{cb} t_{ij})$, where the time-instant t_{ij} may coincide with an internal nodes of the discretized time-domain or any other value contained in there.

However, even if the frailty depends only on the individual time instant t_{ij}, the baseline log-hazard is still defined on each interval I_k, so that the hazard function has form $h_{ijk}(t_{ij}|c_j, b_j) = \mathrm{e}^{(c_j + b_j t_{ij} + \boldsymbol{X}_{ij}\boldsymbol{\beta} + \phi_k)}$ for $t_{ij} \in I_k$, and, consequently, the conditional cumulative hazard function can be derived as follows:

$$\Lambda_{ijk}(t_{ij}|c_j, b_j) = \int_0^{t_{ij}} \lambda_{ij}(s|c_j, b_j) ds = \frac{\mathrm{e}^{(c_j + \boldsymbol{x}_{ij}\boldsymbol{\beta})}}{b_j} f_{ijk}(b_j)$$

where $f_{ijk}(x)$ is a function introduced to evaluate the integral with respect to time:

$$f_{ijk}(x) = \begin{cases} 0 & \text{if } t_{ij} < a_{k-1} \\ \mathrm{e}^{\phi_k}(\mathrm{e}^{xt_{ijk}} - \mathrm{e}^{xa_{k-1}}) & \text{if } t_{ij} \in I_k \\ \mathrm{e}^{\phi_k}(\mathrm{e}^{xa_k} - \mathrm{e}^{xa_{k-1}}) & \text{if } t_{ij} \geq a_k \end{cases}$$

At this point, the likelihood L_j for the centre j can be obtained from the initial form:

$$L_j = \iint_{-\infty}^{\infty} \prod_{i,k} \left[\mathrm{e}^{(c_j + b_j t_{ij} + \boldsymbol{x}_{ij}\boldsymbol{\beta} + \phi_k)}\right]^{d_{ijk}} \ \mathrm{e}^{-\Lambda(t_{ij}|c_j,b_j)} g(c_j, b_j) dc_j db_j$$

where $g(c_j, b_j)$ is the joint gaussian density function. As done in the previous case, we exploit this choice of distribution and we solve the integral numerically with the Gauss-Hermite quadrature formula A.3, using some nodes θ_q and weights w_q, [7]. It's necessary to choose an even number of points to avoid the presence of the null node, for which the function $G(V)$, that will be introduced in a moment, is not defined.
Finally, the full log-likelihood is:

$$l = -N\log(\pi) + \sum_{j=1}^{N} \left[\sum_{i,k} d_{ijk}(\boldsymbol{X}_{ij}\boldsymbol{\beta} + \phi_k) + \log\left(\sum_{q=1}^{10} w_q \ \mathrm{e}^{\sqrt{2}\sigma_r\theta_q d_{j..}} G(\theta_q) \right) \right] \tag{2.7}$$

The function $G(V)$ is defined as:

$$G(V) = \sum_{u=1}^{10} w_u \ \exp\left(\sqrt{2}\sigma_b\theta_u(\gamma d_{j..} + \sum_i d_{ij.}t_{ij}) - \frac{\mathrm{e}^{\left(\sqrt{2}\sigma_r V + \sqrt{2}\sigma_b\gamma\theta_u\right)}}{\sqrt{2}\sigma_b\theta_u} \sum_{i,k} \mathrm{e}^{\boldsymbol{x}_{ij}\boldsymbol{\beta}} f_{ijk}(\sqrt{2}\sigma_b\theta_u) \right)$$

and $d_{ij.} = \sum_k d_{ijk}, d_{j..} = \sum_i \sum_k d_{ijk}, \gamma = \frac{\sigma_{cb}}{\sigma_b^2}$ and $\sigma_r^2 = \sigma_c^2 - \gamma^2\sigma_b^2$.
Maximizing this function l, we can find the optimal value for the unknown model parameters $\sigma_c, \sigma_{cb}, \sigma_b, \beta$ and ϕ_k ($\forall k$).

3 | Technical Details behind the Implementation

In this chapter we discuss some technical details about the implementation of the *Time-Varying Shared Frailty Cox Models* and motivate some adopted decisions.

3.1. Model Setup

Time-Varying Variables

Being the individual time-to-event variable t_{ij} known for each individual, we use it to define the time-varying variables necessary to the analysis: the time-dropout d_{ijk} and the e-time variable e_{ijk}.
The time-dropout variable d_{ijk}, $\forall i, k$, is the temporal extension of the usual dropout variable d_{ij}. Indeed, it assumes value $d_{ijk} = 1$ if individual i in group j fails in interval I_k (i.e. $t_{ij} \in I_k$) and 0 otherwise.
The e-time variable e_{ijk}, $\forall i, k$, is obtained through Formula A.1 of Appendix A, where the generic time-instant is now substituted by the time-to-event t_{ij}.
For convenience, both d_{ijk} and e_{ijk} are built as matrices, with a number of rows and columns equal, respectively, to the number of individuals and intervals.

Vector of Parameters

All models have some parameters in common, such as the baseline log-hazard ϕ_k $\forall k = 1, \dots, L$ and the regression coefficients β_r $\forall r = 1, \dots, R$, while the others are model-specific and differ both in meaning and numerosity, as here indicated:

- *Adapted Paik eaM*: $\boldsymbol{p} = [\phi_1, \dots, \phi_L, \beta_1, \dots, \beta_R, \mu_1, \nu, \gamma_1, \dots, \gamma_L]$, with $\mu_1, \nu, \gamma_1, \dots, \gamma_L > 0$
- *CSFM with Power Parameter*: $\boldsymbol{p} = [\phi_1, \dots, \phi_L, \beta_1, \dots, \beta_R, \gamma_1, \dots, \gamma_L, \sigma]$, with $\gamma_1, \dots, \gamma_L, \sigma > 0$.
- *Stochastic Time-Dependent CSFM*: $\boldsymbol{p} = [\phi_1, \dots, \phi_L, \beta_1, \dots, \beta_R, \sigma_c, \sigma_{cb}, \sigma_b]$,

with $\sigma_c, \sigma_b > 0$.

The numerosity of some parameters depends on the number of regressors or intervals of the time-domain and some of them are well-defined if and only if assume value in a suitable $range = [range_{min}, range_{max}]$, that must be supplied. For these reasons, we decide to gather parameters with the same meaning into the same category, to be able to provide them a unique range during the constraining step, extract them together from $\boldsymbol{p}$, just specifying their position and numerosity.
The categories are indicated by the parameter symbol, that is written in bold only if it groups multiple elements, and the number of categories $n_{category}$ is model dependent. For each model, we report the new structure of $\boldsymbol{p}$, the formula for the computation of the number of parameters and the number of categories:

- *Adapted Paik eaM*: $\boldsymbol{p} = [\boldsymbol{\phi}, \boldsymbol{\beta}, \mu_1, \nu, \boldsymbol{\gamma}]$, $n_p = 2L + R + 2$, $n_{category} = 5$
- *CSFM with Power Parameter*: $\boldsymbol{p} = [\boldsymbol{\phi}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \sigma]$, $n_p = 2L + R$, $n_{category} = 4$
- *Stochastic Time-Dependent CSFM*: $\boldsymbol{p} = [\boldsymbol{\phi}, \boldsymbol{\beta}, \sigma_c, \sigma_b, \sigma_{cb}]$, $n_p = L+R+3$, $n_{category} = 5$

Each overall log-likelihood l defined in 2.5, 2.6 and 2.7 depends on its own parameter vector $\boldsymbol{p}$ and, in order to estimate it, we have to maximize l using a suitable constrained optimization procedure, that looks for the optimal solution in a region of the R^{n_p} space. However, since n_p could be really big, we decide to supply a lower and an upper bound also for the unconstrained parameters, as the regressors β_r or the baseline log-hazard ϕ_k.

Concerning the third model, σ_c, σ_{cb} and σ_b constitute the variance-covariance matrix of the bivariate normal distribution of (c, b) and, by definition, this matrix has to be symmetric and positive semi-definite. However, during the execution of the whole algorithm, checking this well-poseness condition could be very difficult and almost impossible. Therefore, we decide to use the *spectral theorem* and rewrite this matrix as $\Sigma = Q\Lambda Q^T$, where Q is an orthonormal matrix written in terms of the cosine and sine of a generic angle α and Λ is a diagonal matrix formed by the eigenvalues λ_1, λ_2 of Σ. More precisely:

$$\Sigma = \begin{bmatrix} \sigma_c^2 & \sigma_{cb} \\ \sigma_{cb} & \sigma_b^2 \end{bmatrix} = \begin{bmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{bmatrix} \tag{3.1}$$

$$\Rightarrow \begin{cases} \sigma_c^2 & = \lambda_1 cos^2(\alpha) + \lambda_2 sin^2(\alpha) \\ \sigma_{cb} & = \lambda_1 sin(\alpha)cos(\alpha) - \lambda_2 sin(\alpha)cos(\alpha) \\ \sigma_b^2 & = \lambda_1 sin^2(\alpha) + \lambda_2 cos^2(\alpha) \end{cases} \tag{3.2}$$

Expanding the product of the matrices on the right hand side of Formula 3.1, we get the useful relations expressed by 3.2, between the old variables $\sigma_c, \sigma_{cb}, \sigma_b$ and the newly introduced ones $\lambda_1, \lambda_2, \alpha$. At this point, we can ensure the positive semi-definiteness of Σ, because it is automatically satisfied by these substitutions, with the additive conditions that $\lambda_1, \lambda_2 > 0$ and $\alpha \in [0, 2\pi]$. To be precise, we provide the minimum and maximum range for λ_1, λ_2 and α, but the log-likelihood function 2.7 is still defined and evaluated in terms of σ_c, σ_b and σ_{cb}, that at each iteration are computed using Equation 3.2.

Log-Likelihood Function

For all the time-varying shared frailty Cox models previously discussed, the overall log-likelihood function l can be obtained as the sum of each log-likelihood l_j related to the jth group, as indicated by $l = \sum_{j=1}^{N} l_j$.
The models differ in the way l_j is computed and their structures are reported in Formula 2.5, 2.6 and 2.7. Beyond the frailty distribution, their main differences are related to the way the spatial integrals are solved (definition of gamma function vs Gauss-Hermite quadrature formula, see Appendix A) and the presence of further external function to be called, as it happens for the *Stochastic Time-Dependent CSFM*. These aspects obviously influence the amount of time required by a single evaluation of l, making the *Adapted Paik eaM* the fastest and the *Stochastic Time-Dependent CSFM* the slowest among the models.

Each l_j is computed starting from the individuals belonging to group j and they can be easily distinguished thanks to the *Faculty* variable. Since we often need to access the members of a group during the model execution, it is of paramount importance to find a way of storing these group members. We do not go into details at this stage of the report, but we anticipate that we need to create a structure ad hoc (i.e. an ordered map) that associates to each faculty the list of the dataset rows (positional indexes) associated to the individual in a precise group. In this way, every time we need to get them, we just need to pass to a suitable method the group name we are looking for.

Optimization Method and its Limitations

As just said, to estimate the whole vector of parameters, we have to maximize the log-likelihood function using a constrained optimization method in multidimension. The method we applied in our work thesis is a reinterpretation of the *Powell's method in multidimension* [13], whose original version is applied in [15]. Briefly, it minimizes (maximizes) a multidimensional function without computing any derivatives and it takes a set of n_p linearly independent directions and minimizes (maximizes) the function along one direction at the time, using a one-dimensional optimization method. This can coincide with the *Brent's method* [13], that uses a combination of *golden section search* and *successive parabolic interpolation* to locate the minimum (maximum) in a precise interval, whose extrema has to be provided.

The problem associated to our implemented version is related to the choice of the directions along which the one-dimensional function must be optimized. Indeed, different versions of the Powell's method can be obtained just changing the way these directions are chosen at each iterations. In our approach, we use several sets of predefined directions (i.e. orthogonal basis, which components are differently ordered) and the method has to go through all of them until the convergence[1] is reached. However, this approach is extremely slow and inefficient, because we perform little steps in all directions and we may move along a direction, even if the function assumes a constant behaviour along it. If we also consider that a -possible- slow log-likelihood function must be evaluated lots of times inside the optimization phase, we may take several hours to reach convergence. Unfortunately, this is not an exaggeration, but the time required by both the *CSFM with Power Parameter* and *Stochastic Time-Dependent CSFM* to reach the maximum number of iterations and not even convergence.

At this point, we propose three possible solutions: keep the Powell's method in multidimension and change the way the directions are chosen at each iteration; change the optimization method or speed up the evaluation of the log-likelihood function through, for instance, its parallel implementation. We think that any optimization method different from the one we actually apply should be tested to prove the improvements it could bring to the overall time-varying models and, due to our time-constraint, we decide not to go into this direction. Therefore, we decide to provide a parallel version of the log-likelihood function, exploiting the simple relationship between l and l_j. In this way, any evaluation of this function will require less time and could be used inside a suitable and more performing optimization method.
Therefore, due to the decision of not optimizing the log-likelihood function, the provided input files already contain the optimized vectors of parameters and we just demonstrate that the parallel version is able to bring a remarkable reduction to the execution time required by a single evaluation of the log-likelihood function.

[1]Being $ll_{optimal}$ and $ll_{current}$ respectively the best obtained log-likelihood value and the log-likelihood of the current iteration, we say the optimization method has reached convergence when $|ll_{optimal} - ll_{current}| < tol_{ll}$, being tol_{ll} a small predefined tolerance.

Computation of Parameters Standard Error

To quantify the accuracy of our estimated parameters $\hat{\mathbf{p}}_{optimal}$, we compute the *standard error* of each one of them as:

$$\boldsymbol{se}_{pp}(\hat{\boldsymbol{p}}_{optimal}) = 1/\sqrt{I_{pp}(\hat{\boldsymbol{p}}_{optimal})} \qquad I(\hat{\boldsymbol{p}}_{optimal}) = -H(\hat{\boldsymbol{p}}_{optimal})$$

where $I_{pp}(\hat{\boldsymbol{p}}_{optimal})$ is the pth element of the diagonal of the *information matrix*, computed as the opposite of the *hessian matrix* evaluated at the $\hat{\boldsymbol{p}}_{optimal}$.
To be coherent with the optimization procedure we applied in R and due to the impossibility of directly computing the second derivative of the log-likelihood function, we decide to compute the diagonal of the hessian matrix and to numerically approximate the second derivative. We adopt the following centered finite difference scheme, with an accuracy of the second order, and with a discretization step h_d:

$$f''(\hat{p}) \approx \frac{f(\hat{p}_{-1}) - 2f(\hat{p}) + f(\hat{p}_{+1})}{h_d^2} + O(h_d^2) \qquad \hat{p}_{-1} = \hat{p} - h_d \quad \hat{p}_{+1} = \hat{p} + h_d$$

having imposed $\hat{p} = \hat{p}_{optimal}$.
The entire procedure is summarized in Algorithm 3.1:

Algorithm 3.1 Computation of optimal parameters standard error

```
Require: $\hat{\boldsymbol{p}}_{optimal}$, $h_d$
1: Compute number of parameters $n_p$ and initialize $\boldsymbol{se}_{optimal} = \mathbf{0}$
2: for $pp = 1, \dots, n_p$ do
3:     Extract $\hat{p} = \hat{\boldsymbol{p}}[pp]$ and define $\hat{p}_+ = (\hat{p} + h_d)$, $\hat{p}_- = (\hat{p} - h_d)$.
4:     Re-assign $\hat{\boldsymbol{p}}_+[pp] = \hat{p}_+$, $\hat{\boldsymbol{p}}_-[pp] = \hat{p}_-$
5:     Compute $ll_+ = ll(\hat{\boldsymbol{p}}_+)$, $ll_- = ll(\hat{\boldsymbol{p}}_-)$ and $ll = ll(\hat{\boldsymbol{p}})$.
6:     Compute $H(\hat{p}) = (ll_- - 2ll + ll_+)/h_d^2$
7:     Compute $\boldsymbol{se}_{optimal}(\hat{p}) = 1/\sqrt{-H(\hat{p})}$
8: end for
9: return $\boldsymbol{se}_{optimal}$
```

3.2. Getline and GetPot

As we will see, two input files must be provided to the main source file and they are read using *Getline* and *GetPot*. The first one is used to read consecutive lines, not subdivided in blocks, and it stops when it reaches the end of the file. On the other hand, GetPot is used to read variables distributed in blocks and it may be called several times in the program, according to the block variables we need.

A problem we encountered is related to the types of variables GetPot is able to read and store. Indeed, looking at the *operator(...)* method present in the GetPot class, we see that both its input and returned types[2] are either a *double* or an *int*, while some of our variables must be compulsory non-negative and cannot be saved without controlling the sign. Therefore, we have adopted the following strategy for an *unsigned int* variable: we read it from input, save it into an *int*, check its sign and if it is wrong (negative), we stop the execution of the program, otherwise we cast it into the right type through a *static_cast<unsigned int>(int)*[3]. For a double type, we just control the sign and if it is wrong, we stop the program execution, otherwise we do not cast it since it is already saved into the right type.

3.3. Eigen Library and Parallel Computing

Eigen

The dataset containing the students covariates and some of the variables we defined so far (e.g. time-dropout d_{ijk}, e-time e_{ijk}) are theoretically built as matrices, while other variables as vectors. When moving to C++, we decide to keep these data structures and to take advantage of *Eigen* library [3]. Therefore, we implement all the matrices as dynamic eigen matrices and the vector variables that will multiply a matrix, as dynamic eigen vector. Conversely, all the other vectors are implemented as normal *std::vector*. For instance, the parameter vector is a dynamic eigen vector, while the time-domain vector is a vector of the standard library.

OpenMP

As we previously anticipated, the necessity of speeding-up a single evaluation of the log-likelihood function leads to the introduction and application of parallel computing. More precisely, we decide to adopt a *MIMD*[4] architecture and a *Shared Memory System* with protocol *OpenMP* [5], so that the computer RAM memory is shared among the cores.
Thanks to the structure of the overall log-likelihood ($l = \sum_{j=1}^{N} l_j$), the idea is to assign the computation of the log-likelihood function l_j related to a group j to a thread (core), exploiting the fact that the all the individuals belonging to that group can be easily get through the map we have built and this map can be accessed by all threads, because all of them have access to the same memory.

[2]The ones we are interested in.
[3]We could have used an implicit conversion, but we preferred to make our intention clear.
[4]Multiple Instruction, Multiple Data

4 | Implementation

In this chapter we describe how the time-varying shared frailty Cox models are implemented in C++. We start from the construction of the basic data structures, then we move to the implementation of the models and, at the end, we discuss how to obtain a parallel version of the log-likelihood function. We also present the two input .txt files, in which several user-provided variables are stored.
The project has been implemented under the *namespace TVSFCM* (*Time-Varying Shared Frailty Cox Model*) and it is available at
`https://github.com/GiuliaRomani/TimeVaryingSharedFrailtyCoxModels`.

4.1. Structure of the folder

The folder *TimeVaryingSharedFrailtyCoxModels* is composed of a *Makefile*, a *READMI.md* file and the following folders:

- *BashScript*: It contains several bash scripts for the direct execution on terminal of the models. Four bash scripts are available and they only differ in the couple of used input .txt files.
- *Data*: It contains two subfolders (*DataIndividuals* and *DataTool*) of input files. Each subfolder contains data files related to the academic years 2010, 2018 and $2017 - 2018$ and a data file for model testing.
- *Doc*: It contains the configuration file for the *doxygen* [2] documentation and the report.
- *Src*: It contains all the header and source files, and the *Makefile* for compiling them.

4.2. Input .txt files

DataIndividualsFile.txt

Each file of the form *DataIndividualsFile.txt* contains the PoliMi dataset described in Chapter 1 and it is read through *getline*. It is composed of as many rows (+1) as the

number of individuals composing the dataset and as many columns (+2) as the number of covariates. The first row is composed of two non-negative integers respectively indicating how many rows and columns constitute the dataset. They are provided to properly set the dimension of the dynamic eigen matrix in which the dataset is saved.
Concerning the other rows, their first element corresponds to the faculty each student belongs to: it is composed of four letters in quotes and it is saved into a suitable dynamic eigen vector. The last element of the row is the non-negative time-to-event, saved into another dynamic eigen vector.

DataToolFile.txt

Each file of the form *DataToolFile.txt* is composed of several blocks read through *GetPot* and each one refers to a precise component of the time-varying models:

- TimeDomain: It contains the time-domain vector and its dimension, used to properly make a resize of the *std::vector*.
- Parameters: It is composed by other two blocks:
    - ◇ Ranges: For each time-varying model, the minimum and maximum range of each category is provided.
    - ◇ Values: For each time-varying model, it contains the optimal vector of parameters.
- DiscretizationStep: It is the discretization step used for the numerical approximation of the second derivative of the log-likelihood function.
- Model: It is the numeric id associated to the time-varying model. The possible values are 1,2,3 (more details will follow).
- ParallelVersion: It is composed of three non-negative integer values, that are: the number of threads, the chunk size and the id of the scheduling strategy to be used into the parallel for loop.

The user needs to fill these two files and every time we extract their variables, we check that they respect the required existence condition, to avoid unexpected and wrong behaviours. For instance, we control that the non-negative variables are exactly non-negative.

4.3. Basic structures

The general idea is to define each component necessary for the global execution of a model in a dedicated structure, that could coincide either with a class or a struct, according to

its meaning and utility.
The most important structures are the ones related to the temporal domain (*TimeDomain*), the dataset (*DatasetInfo*) and, probably the most complicated, the parameters (*Parameters*).
As discussed in Section 2.2.3, some individual (e.g. time-dropout d_{ijk}) and global (e.g. parameter vector $\boldsymbol{p}$) variables depend on the time-domain and, moving forward, all models depend on both the dataset and the parameters for the evaluation of the log-likelihood. For these reasons and for some later specified classes, we have adopted both a composition by (public) inheritance and polymorphism, as illustrated in Figure 4.1.

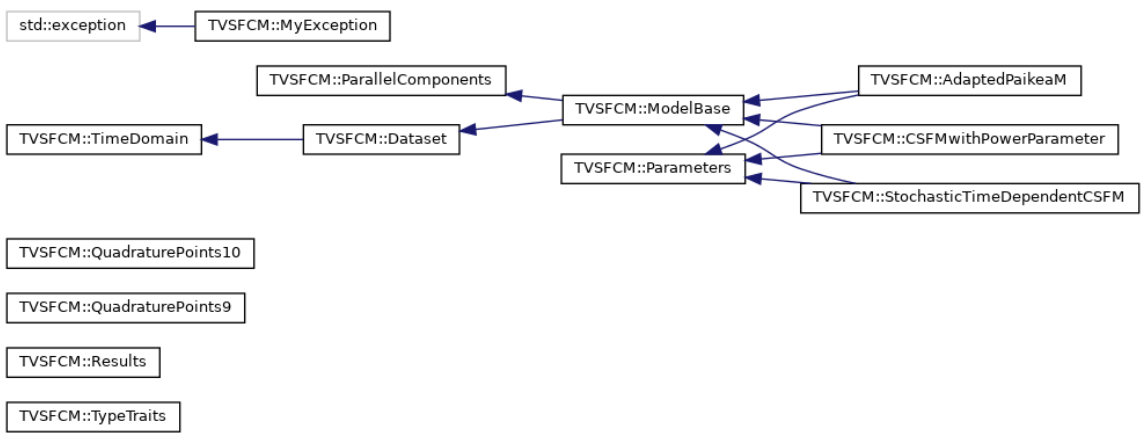

Figure 4.1: Inheritance scheme of the classes.

Type Traits

Refer to: *Project_PACS/Src/TypeTraits.hpp*.
Several types of variables are used in the models implementation and they are collected in a unique *struct*. To avoid reporting the word "TypeTraits" every time we want to use a type here present, we use the type alias: $T = TypeTraits$.

We briefly describe two types, while the others are presented at the occurrence.
$VariableType = double$ is the basic type used for the PoliMi dataset and also for other created variables. Even if some covariates of the dataset may be binary, they are directly stored as a double and not converted from a bool. Indeed, since we are in a regression problem context, each one is multiplied by a scalar double coefficient β (see Section 2.2.3) and thus is more convenient to have it as a double. The same holds also for other binary variables, such as the time-dropout d_{ijk}. Moreover, when reading the input data, we cannot know in advance which and how many covariates are binary and to allow all the possibilities, we adopt a double.

Another important type is related to the temporal variables. In survival analysis the follow-up period and the time-to-event can be measured in several ways, for instance days, years or also semesters, as in our work. To permit type flexibility, we decide to introduce the type alias *TimeType* = *double*.

Exceptions

Refer to: *Project_PACS/Src/MyException.hpp*

Every time we extract a variable from a file, we need to check that it is well defined and it satisfies the required conditions. For instance, each optimal parameter must be effectively contained in its range or the number of threads has to be non-negative. In case at least a condition is not satisfied, the program stops and reports the cause of the problem. We have created a *MyException* class, publicly derived from the *std::exception*, that lets us throw an exception with a problem-related message of type *ExceptionType* = *std::string*. We report its structure in Code 1:

Listing 1 MyException Class

```cpp
class MyException: public std::exception{
public:
    explicit MyException(const T::ExceptionType& message_):
        message(message_){};
    explicit MyException(const char* message_):
        message(message_){};
    virtual const char* what() const noexcept {
        return message.c_str();
    };
    virtual ~ MyException() = default;
protected:
    T::ExceptionType message;           //! Content of the exception
};
```

Thanks to the fact that the message is not predefined and can be passed in input to the class constructor, the exception can be thrown whenever we encounter an error or a problem and we need to stop the whole execution. The possible causes are: missing, incomplete, wrong or negative values, parameter out of range or wrong range assignment, or not existing file.

For what concerns the missing values, if a required element is not provided, we assign it a default value equal to NaN, so that we can detect it through a *std::isnan(. . .)* and later communicate the problem. We cannot provide a different default initialization because the correct value may depend on several factors related to the dataset, the temporal domain and the chosen time-varying model. The only cases in which a not NaN value is default assigned are the ones not related to the dataset, the time-domain and the parameters, such as, the value of the discretization step h_d for the numerical approximation of the second derivative or the number of threads for the parallel computing.

All the thrown exceptions are caught during the global execution of the process in the *main.cpp* source file, in a unique *try and catch* block, illustrated in Code 2. They are thrown by several ad hoc methods defined as *check_ condition(...)*.

Listing 2 Try and Catch block

```
try{ [... something ...] }
catch(const MyException& e) {
    std::cout << e.what() << std::endl;
};
```

Quadrature Nodes and Weights

Refer to: *Project_PACS/Src/QuadraturePoints.hpp.*

As we may recall from the definition of the log-likelihood functions 2.6 and 2.7, the spatial integrals are solved through the Gauss-Hermite quadrature formula, using different number of points. We have defined two structs, called *QuadraturePoints::Points9* and *QuadraturePoints::Points10*, containing the initialized array of nodes and weights [7]. Since we know in advance their dimension, respectively equal to 9 and 10, their type is either *std::array<VariableType, 9>* or *std::array<VariableType, 10>*.

Time Domain

Refer to: *Project_PACS/Src/TimeDomain.hpp* and *Project_PACS/Src/TimeDomain.cpp*

We have created a simple class called *TimeDomain* that contains the partitioned time-domain and its number of intervals (L), which corresponds to the length of the vector minus 1.

Since the information contained in this class are used throughout the program by other classes and methods, we have decided to define this class as a base one, with public and protected members, from which other classes can derive and inherit its components.

Dataset

Refer to: *Project_PACS/Src/Dataset.hpp* and *Project_PACS/Src/Dataset.cpp*

The class *Dataset* contains the dataset and builds all the variables that can be extracted from it and used for the evaluation of the model log-likelihood. Some of them need to be computed starting from the knowledge of the time-domain, its intervals and their number. For this reason, we have decided to derive the current class from the *TimeDomain* class, through the composition by (public) inheritance. Since the data contained in this class are necessary for the computation of the model log-likelihood, it is built as a base class.

As we anticipated in Section 3.1, one fundamental step is the grouping of all individuals into a certain number of clusters, equal to the engineering PoliMi faculties (16). This is performed through the ordered map, *map_ groups*, that associates to each faculty a shared pointer to a vector, which elements correspond to the dataset rows (positional indexes) of the individuals in that group. To be clearer, consider the following example where the vector of individual group membership is: $["EngA", "EngC", "EngB", "EngB", "EngA"]$. The map is thus formed by the couples: $("EngA", \rightarrow (0,4)), ("EngB", \rightarrow (2,3))$, $("EngC", \rightarrow (1))$. In this way, when we compute the overall log-likelihood l, we simply loop over the faculties contained in the map, extract the associated shared pointer and compute the group log-likelihood l_j with all and only the individuals of faculty j.

The types of the *map_ groups* and shared pointer are:
MapType = std::map<GroupNameType, SharedPtrType>
SharedPtrType = std::shared_ ptr<VectorIndexType>, where
GroupNameType = std::string, *VectorIndexType = std::vector<IndexType>* and
IndexType = unsigned int.

Parameters

<u>Refer to</u>: *Project_ PACS/Src/Parameters.hpp* and *Project_ PACS/Src/Parameters.cpp*
Each model is characterized by some unknown parameters, whose value should be properly determined through a constrained optimization procedure in multidimension. However, this optimization is not performed and the optimal parameters are already provided in input, together with the minimum and maximum range of the categories presented in Section 3.1. These ranges are provided to check that the parameters are effectively contained in there because, otherwise, some well-poseness condition may not be satisfied.

The numerosity of the parameter vector depends on some variables that are not known a priori and are determined only once the *Dataset* and *TimeDomain* classes are initialized and the specific model is chosen. The constructor of the class *Parameters* needs the variables L, R, n_p, $n_{category}$, and a vector containing the order according to which the categories are distributed in the parameter vector. To be clearer, if we consider the *Adapted Paik eaM*, this vector variable is called *all_ n_ parameter* and has shape $[L, R, 1, 1, L]$, where each element correspond to the dimension of each element of $\boldsymbol{p} = [\boldsymbol{\phi}, \boldsymbol{\beta}, \mu_1, \nu, \boldsymbol{\gamma}]$.

The order according to which the categories are registered is fundamental both when checking that each input parameter is properly contained in the range of its category (well-poseness condition) and when extracting the parameters of a category from the entire parameter vector. An example of the first application is reported in Code 3.

Listing 3 Method for checking parameters well-poseness condition

```cpp
void Parameters::check_condition(const T::VectorXdr& v_parameters_) const{
    T::NumberType n = 0;            // Numerosity of a category
    T::IndexType actual_j = 0;      // Index for the parameter vector
    T::VariableType a,b = 0.;       // Min and max range
    // Loop over the categories in all_n_parameters
    for(T::IndexType i = 0; i < n_ranges; ++i){
        n = all_n_parameters[i];
        a = range_min_parameters[i];
        b = range_max_parameters[i];
        // Loop over the parameters in the category
        for(T::IndexType j = 0; j < n; ++j){
            if(std::isnan(v_parameters(actual_j))){
                throw MyException("At least one parameter is not provided ");
                    }
              else if((v_parameters(actual_j) < a) || (v_parameters(actual_j) > b)){
                throw MyException(...Value of parameter not in the range...);
                }
            actual_j += 1;
            }
    }
};
```

For what concerns the extraction of the parameters from the parameter vector, instead of using *all_n_parameter*, we exploit some operations of the Eigen library, such as *head(n)*, *block(n)* and *tail(n)* that allow us to extract the first, the last or a block of n elements from the dynamic vector. Their functioning is observable in the method *extract_parameters(...)*, which definition is reported in Code 4 and it is valid for *Adapted Paik eaM*. Even if this method does not belong to the *Parameters* class, we have decided to present it here for clarifying this discussion.
We can observe that the returned type *TuplePaikType* does not fully coincide with the list of categories and this is due to the fact that some model parameters, used in the

computation of the log-likelihood, are get starting from the parameters in $\boldsymbol{p}$, imposing some constraints. For instance, if we still consider the *Adapted Paik eaM*, μ_1 is unknown and contained in $\boldsymbol{p}$, while μ_2 can be get imposing $\mu_2 = 1 - \mu_1$.

Listing 4 Method for extracting parameters from the vector of parameters

```
T::TuplePaikType
AdaptedPaikeaM::extract_parameters(T::VectorXdr& v_parameters_) noexcept{
    T::VectorXdr phi = v_parameters_.head(Dataset::n_intervals);
    T::VectorXdr betar = v_parameters_.block(Dataset::n_intervals, 0,
                            Dataset::n_regressors, 1);
    T::VariableType mu1 = v_parameters_(Dataset::n_intervals + Dataset::n_regressors);
    T::VariableType mu2 = 1 - mu1;
    T::VariableType nu = v_parameters_(Dataset::n_intervals+Dataset::n_regressors+1);
    T::VectorXdr gammak = v_parameters_.tail(Dataset::n_intervals);
    return std::make_tuple(phi, betar, mu1, mu2, nu, gammak);
};
```

According to the model and what we need to extract, several extraction methods with the form just discussed are defined and each one has its own returned type.

ParallelComponents

Refer to: *Project_PACS/Src/ParallelComponents.hpp* and
Project_PACS/Src/ParallelComponents.cpp
The class *ParallelComponents* contains four protected data members used to set the parallel for loop of the parallel implementation of the overall log-likelihood function. They are protected because each time-varying model is publicly derived from this class.
The data members are provided in input by the user, because we let him/her the possibility of choosing the preferred setup according to the data at disposal, as it will be explained in Section 5.1. These members are:

- *n_threads*: It is the number of threads(cores) to use. If the user selects 1, then the parallel program is not executed and the log-likelihood function is computed in the serial way. If the value is not provided, the default assigned is 1.
- *chunk_size*: It is the maximum number of iterations each thread may execute. The value depends also on the choice of the scheduling strategy. If the value is not provided, the default assigned is 0. Indeed, when a value less than 1 is indicated, the *omp_set_schedule(...)* routine assigns it another default value.
- *schedule_type*: It is the numeric id of the scheduling strategy to apply and its name

is saved in *schedule_type_name*. There are only four possibilities $(1, 2, 3, 4)$ and in case it is not provided, the default value is 1.

Since the scheduling strategy is decided runtime, we set the clause *schedule(runtime)* in the parallel for loop. Then, to pass from the numeric id to the omp strategy type, we take advantage of the enumeration type *omp_sched_t*, defined in OpenMP as indicated in Code 5, and we impose *omp_sched_t(ParallelComponents::schedule_type)* inside the *omp_set_schedule* routine to select the desired scheduling strategy (see Section 4.5).

Listing 5 omp_sched_t type

```
typedef enum omp_sched_t {
  omp_sched_static = 0x1,
  omp_sched_dynamic = 0x2,
  omp_sched_guided = 0x3,
  omp_sched_auto = 0x4,
} omp_sched_t;
```

Results

Refer to: *Project_PACS/Src/Results.hpp* and *Project_PACS/Src/Results.cpp*
The class *Results* contains the results of the application of one of the time-varying shared frailty Cox models and they coincide with: the value of the optimal log-likelihood function and the Akaike Information Criterion (AIC), the optimized parameters, their number and their standard error and, eventually, the standard deviation of the frailty. Moreover, the class contains also information about the parallel implementation of the log-likelihood function, such as the number of threads, the chunk size and the scheduling strategy.
With the exception of the AIC, that is computed with a method inside the class constructor, the other parameters must be provided to the constructor.
At the end, everything is printed in a customized way distinguishing in which way the log-likelihood function has been evaluated (parallel vs serial way).

MathFunctions

Refer to: *Project_PACS/Src/MathFunctions.hpp*.
This header file contains just two functions necessary for the computation of the binomial coefficient of two integers, required in the log-likelihood 2.5. We have decided to exploit the fact that the gamma function of an integer number is equivalent to $\Gamma(n) = (n-1)!$

[6], and to re-write the binomial coefficient starting from its logarithm [4], as shown:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{if } n,k \in N, 0 \leq k \leq n$$
$$\ln\binom{n}{k} = \ln(\Gamma(n+1)) - \ln(\Gamma(k+1)) - \ln(\Gamma(n-k+1)) \Rightarrow \binom{n}{k} = \exp\left(\ln\binom{n}{k}\right)$$

Eventually, the binomial coefficient is computed using the function *binom (...)*, as the exponential of the logarithmic binomial obtained by *logbinom(...)*, as indicated in Code 6:

Listing 6 Methods for computing the the binomial coefficient

```
//! Compute the logarithmic binomial coefficient
inline T::VariableType logbinom(T::NumberType n, T::NumberType k) noexcept{
    return std::lgamma(n+1) - std::lgamma(n-k+1) - std::lgamma(k+1);
};
//! Compute the binomial coefficient
inline T::VariableType binom(T::NumberType n, T::NumberType k) noexcept{
    return std::exp(logbinom(n,k));
};
```

4.4. Models

In this section we are going to present the three *Time-Varying Shared Frailty Cox Models*. Our idea is that the user has the possibility of choosing which model he/she wants to apply, simply providing a numeric id associated to the model. To permit this, we have decided to use polymorphism (see Figure 4.1) and to define a base class that contains all the data members, not related to any particular model, and all the function members that can be inherited by the derived classes and customized. For this purpose, they are marked as virtual.
Moreover, since the base class does not implement any log-likelihood function, we have built it as abstract: the virtual methods are pure and no implementation is provided.

4.4.1. Base Class

Refer to: *Project_PACS/Src/ModelBase.hpp* and *Project_PACS/Src/ModelBase.cpp*
ModelBase is the base class for the time-varying models and it contains all their common data members. First of all, it is publicly derived from *Dataset* and *ParallelComponents*, thanks to the composition by inheritance, so that it is possible to access all the variables

of the dataset and the ones necessary to the parallel computing. Then, it is composed of the class *Result*, for the storage of the model results.

The most important pure virtual method is *evaluate_loglikelihood()*. It is public and not protected, because it is called in the *main.cpp* source file to evaluate the log-likelihood function. Inside it, other methods are called for computing the standard error associated to each parameter and the standard deviation of the frailty.
One method not marked as virtual is *print_results()*: it simply calls the same method contained in *Results* class, once it has been initialized.

4.4.2. Derived Classes

The are three derived classes, each one valid for one of the time-varying models, and all publicly derived from *ModelBase* and *Parameters*. The latter class is initialized with the elements characterizing each model, as the number of parameters, the number of categories and the vector containing the ordered numerosity of the categories. While both the vector and the number of categories are already built and provided to the *Parameters* constructor, the number of parameters is obtained through the method *compute_n_parameters()* and the same computation there contained is also passed to the *Parameters* constructor.
Each model contains several lambda functions, implementing the group and overall log-likelihood, both in the serial and parallel version, the numerical approximation of the second derivative and, only for the *Stochastic Time-Dependent CFSM*, both $G(\dots)$ and $f_{ijk}(\dots)$ functions. They are all initialized inside the constructor of the derived class, through the methods *build_loglikelihood()*, *build_dd_loglikelihood()* and *build_loglikelihood_parallel()*.
All models implement the parallel version of the overall log-likelihood function and the differences are not in their structure, but only in which model-specific group log-likelihood is called. For this reason, we just describe the functioning of the parallel version for the *Adapted Paik eaM* in Section 4.5.

The Adapted Paik eaM

Refer to: *Project_PACS/Src/ModelDerived.hpp* and *Project_PACS/Src/ModelDerivedPaik.cpp*.
AdaptedPaikeaM is the class associated to the *Adapted Paik eaM* and the most important members of this class are the two lambda functions, *ll_group_paik(...)* and *ll_paik(...)*, that builds respectively the group log-likelihood and overall log-likelihood function 2.5, which definitions are reported in Code 7 and 8.
In Code 7 the use of the *map_groups* variable is illustrated: through an iterator we loop

over the groups in the map and extract the shared pointer to the individuals contained in the group itself.

Listing 7 Method for computing the overall log-likelihood function

```
ll_paik = [this] (T::VectorXdr& v_parameters_){
    T::VariableType log_likelihood_group, log_likelihood = 0;
    T::MapType::iterator it_map = Dataset::map_groups.begin();
    T::MapType::iterator it_map_end = Dataset::map_groups.end();
    for(; it_map != it_map_end; ++it_map){
        // Extract the shared pointer
        const auto& indexes_group = it_map->second;
        log_likelihood_group = ll_group_paik(v_parameters_, indexes_group);
        log_likelihood += log_likelihood_group;
    }
    return log_likelihood;
};
```

For the evaluation of the group log-likelihood, we need to extract some data variables specifically related to the cluster we are evaluating (e.g. $A_{ijk}, d_{ijk}, \dots$) and we apply a procedure very similar to the one used for the extraction of the parameters from their vector (i.e. *extract_parameters(...)*). The methods are *extract_matrixA_variables(...)* and *extract_dropout_variables(...)*: they receive, as input, the shared pointer to the vector of group individuals and they return a tuple of three elements, of type dependent on the form of the extracted variables.
In Code 8, we report a part of the construction of *ll_group_paik(...)* to show how the variables are extracted and how the method works, even if it precisely follows the structure of Equation 2.5.

CSFM with Power Parameter and Stochastic Time-Dependent CFSM

Refer to: *Project_PACS/Src/ModelDerived.hpp*, *Project_PACS/Src/ModelDerivedPP.cpp* and *Project_PACS/Src/ModelDerivedLF.cpp*.

CSFMwithPowerParameter and *StochasticTimeDependentCSFM* are the other two time-varying models and they have a structure similar to the one just explained, except for the fact that each one is composed of one quadrature formula struct, assigned as indicated:

- *QuadraturePoints::Points9* for the *CSFMwithPowerParameter*
- *QuadraturePoints::Points10* for the *StochasticTimeDependentCSFM*

However, inevitable changes have been performed on the construction of the two log-likelihood functions and on the way the parameters and the group temporal variables are extracted from, respectively, the parameter vector and the dataset. Moreover, also the procedure for the computation of the standard deviation of the frailty changes, as indicated in Section 2.2.3.

Listing 8 Implementaion of group log-likelihood for the *Adapted Paik et al.'s Model*

```cpp
ll_group_paik = [this] (T::VectorXdr& v_parameters_, T::SharedPtrType indexes_group_){
    // Extract variables and parameters from the vector
    auto [phi, betar, mu1, mu2, nu, gammak] = extract_parameters(v_parameters_);
    auto [A_ijk, A_ik, A_i] = extract_matrixA_variables(indexes_group_, phi, betar);
    auto [d_ijk, d_ik, d_i] = extract_dropout_variables(indexes_group_);
    // Compute the first term of the formula and then subtract the second term
    T::VariableType  dataset_betar, loglik1 = 0.;
    for(const auto &i: *indexes_group_){
        dataset_betar = Dataset::dataset.row(i) * betar;
        for(T::NumberType k = 0; k < Dataset::n_intervals; ++k){
            loglik1 += (dataset_betar + phi(k)) * Dataset::dropout_intervals(i,k);
        }
    }
    loglik1 -= (mu1/nu) * log(1 + nu * A_i);
    // Compute the rest of the formula
    [... loglik2, loglik3 ...]
    // Sum the terms
    result = loglik1 + loglik2 + loglik3;
    return (result);
};
```

4.4.3. Time-Varying Object Factory

Refer to: *Project_PACS/Src/MethodFactory.hpp*
The models we described so far have different characteristics, limitations and advantages and the user may be interested in only one of them according to the dataset he/she has. For instance, both the *Adapted Paik eaM* and the *CSFM with Power Parameter* are quite fast and produce very accurate results. On the other hand, the *Stochastic Time-Dependent CSFM* could take some minutes just for computing the standard error of the parameters, but its results may be slightly different from the others. Therefore, the user can be interested in applying only one of the three models: to make this possible we have adopted the mechanism of *polymorphism* and saved the model objects into an *object factory*.

First of all, we have decided to assign each model a couple of variables with types *IdType* = *unsigned int* and *IdNameType* = *std::string*, indicating the numerical id and the extended name of the model. The only possible registered couples are: (1, *"Adapted Paik eaM"*), (2, *"CSFM with Power Parameter"*) and
(3, *"Stochastic TimeDependent CSFM"*).
Then, we built an object of type *FactoryType* = *std::map<IdType, IdNameType>*, that associates to each numerical id the corresponding extended name. This object is created as a *static* variable and it is filled with the couples, using a specific method. There are no possibilities of adding other models, because as far as we know no other time-varying frailty models with these structure and components exist.

In the end, we implement a method called *MakeLikelihoodModel(...)* that receives the numeric id of the model we want to apply and the two input files. It returns a unique pointer to the base class, initialized with the model we have required, so that we can call the virtual method *evaluate_ loglikelihood()* for the computations. In case an unregistered numeric id is passed, an exception is thrown.
Internally, a switch function is used to switch among the three models and it calls the constructor of a unique pointer, with the model and input provided. We report the structure of the factory in Code 9:

Listing 9 Time-Varying Shared Frailty Cox Model Factory

```
std::unique_ptr<ModelBase>
MakeLikelihoodModel(const T::IdType id, const T::FileNameType& filename1_,
                    const T::FileNameType& filename2_) {
    switch(id){
        case 1:  return
                 std::make_unique<AdaptedPaikeaM>(filename1_, filename2_);
        case 2:  return
                 std::make_unique<CSFMwithPowerParameter>(filename1_, filename2_);
        case 3:  return
                 std::make_unique<StochasticTimeDependentCSFM>(filename1_, filename2_);
        default:  throw MyException("Not existent or not provided id method!");
    };
};
```

4.5. OpenMP

As we already mentioned, the idea behind the introduction of parallel computing is to speed-up the execution of a single log-likelihood function, taking advantage of the way it is computed starting from the group log-likelihood, and that is, summing the contribute provided by each group.
Looking at the implementation of *ll_paik(...)* reported in Code 7, we aim at performing in parallel the for loop so that each thread is able to evaluate the log-likelihood related to at least a group. To do this, we perform a for loop in parallel using the OpenMP directive *omp parallel for* and we specify some clauses. We report the parallel implementation of the overall log-likelihood function in Code 10 and we comment it.

First of all, we use the *omp_set_schedule* routine to impose both the scheduling strategy and the chunk size the user has decided to use. Then, the *omp parallel for* directive requires several clauses:

- *num_threads*: It is the number of threads decided by the user and read in input. It belongs to the ParallelComponents class.
- *firstprivate*: The variable *it_map* is replicated in each thread, but defined outside.
- *schedule*: Since the scheduling strategy is read in input and it is not available when compiling the codes, we have to set it to *runtime*.
- *reduction*: Each thread is responsible for the evaluation of the log-likelihood function related to a group and the result it gets must be summed to the overall log-likelihood. This is automatically done through the clause *reduction(+:log_likelihood)*, where the *log_likelihood* variable is initialized to 0 outside the parallel region and replicated in each thread.

The way we go through the map is not directly done using a map iterator because the parallel for loop only supports random access iterators and the iterator of a map is not of this type.
Since the number of pairs constituting the map corresponds to the number of faculties and this value has been saved in the *Dataset* class, we decide to execute a parallel for loop with an *unsigned int* counter variable, that loops from $j = 0$ up to $j = n_{groups} - 1 = 15$, and to define outside the parallel environment the iterators *it_map* (not initialized) and *it_map_begin = Dataset::map_groups.begin()*. Inside the parallel region, every time a thread has to compute the log-likelihood of the jth group, it increments *it_map_begin* of j positions, so that it is actually pointing at the jth element of the map. To increment

the iterator by value j we use the *std::next(it_ map_ begin, j)* operation, that returns the moved iterator *it_ map*. This operation has been preferred to *std::advance(it_ map, j)* because the latter has void returned type and, at every iteration, we should have re-initialized *it_ map* to *it_ map_ begin*.
Note that, at each iteration of the for loop, we do not need to check that
it_ map != Dataset::map_ groups.end() because we advance *it_map_begin* of at most $(n_{groups} - 1)$ positions, precisely arriving at the last element of the container.

Listing 10 Method for computing the overall log-likelihood function in parallel

```cpp
ll_paik_parallel = [this] (T::VectorXdr& v_parameters_){
    T::VariableType log_likelihood = 0;            // Overall log-likelihood value

    // Loop over the map through an iterator
    T::MapType::iterator it_map_begin = Dataset::map_groups.begin();
    T::MapType::iterator it_map = it_map_begin;

// Parallel region
omp_set_schedule(omp_sched_t(ParallelComponents::schedule_type),
                 ParallelComponents::chunk_size);
#pragma omp parallel for num_threads(ParallelComponents::n_threads)
#                        firstprivate(it_map)
#                        schedule(runtime)
#                        reduction(+:log_likelihood)
    for(T::IndexType j = 0; j < n_groups; ++j){
        it_map = std::next(it_map_begin, j);
        const auto& indexes_group = it_map->second;
        log_likelihood += ll_group_paik(v_parameters_, indexes_group);
    }
    return log_likelihood;
};
```

For what concerns the choice of the scheduling strategy, there are four possibilities: *static* is the first one, where the iterations are distributed approximately equally among the threads and each thread is responsible for the evaluation of $(n_{groups}/n_{threads})$ or $(n_{groups}/n_{threads} + 1)$ iterations, so that the first group of iterations is assigned a thread, the second group to another thread and so on. Since we allow the user to specify the chunk size, the number of iterations is distributed as uniform as possible, with a maximum equal to chunk size.
Due to the fact that the groups may have different numerosity, this strategy is not the

best one because the first group of iterations could be composed of less individual than the second group. Therefore, each thread may take a different amount of time to evaluate the assigned groups log-likelihood and the amount of work is not efficiently distributed among the threads. A possible alternative is given by *dynamic*, where the chunk size is equal to 1 if not specified or each thread executes *chunk_ size* iterations before being available for another iterations. This strategy performs better when the cost of each iteration is not uniform, as in our case. A similar strategy is represented by *guided*, where the chunk size is dynamically determined by the system and decreases with the iterations. Conversely, if we specify the chunk size, we are specifying the maximum number of iterations for each thread. The last possibility is *auto* where we let the system choose the scheduling strategy.

As we previously anticipated, according to the number of threads provided in input, either the serial or the parallel version are built inside the class constructor. Precisely, if $n_{threads} > 1$, both the serial and the parallel version are built, but only the parallel one is used. Indeed, *ll_ paik_ parallel* requires the function *ll_ group_ paik(. . .)* that is built in the serial way. Conversely, when we compute the standard error of the parameters, we do not distinguish between the two cases and execute only the serial version.

4.6. Makefile

The codes we described so far are contained in the folder *TimeVaryingSharedFrailtyCox-Models/Src*, where a *Makefile* is present to compile them. Since we use the *Eigen* library, we need to specify where to look for its header files and, to get the full speed from it, we compile the codes with optimization enable and no debugging (-O3 -DNDEBUG).
On the other hand, to use *OpenMP*, we need to include the header file *<omp.h>* in the files where it is required[1] and then we need to activate OpenMP through the compiler option *-fopnemp*.
In the main folder, another *Makefile* is present and it compiles the codes in the subfolders indicated and also produces the doxygen documentation, which doxygen configuration file is contained in the *Doc* folder.

4.7. Bash Script

The source file *main.cpp* has been implemented to be able to receive the name of the two .txt files from the command line and, if the number of passed arguments is not correct,

[1]The same is done with *<Eigen/Dense>* for the Eigen library and with all other header files and libraries.

an exception is thrown. Moreover, it is compulsory that the first file belongs to the folder *Data/DataTool* and the second one to *Data/DataIndividuals*. There are four possible couples of files, for four different applications.
To avoid typing their name every time and selecting the right couple, we have already created four bash scripts, that can be executed directly from the terminal of the *BashScript* folder: *bash_ test.sh* executes the main source file with test data, while *bash_ app2010.sh* executes data of 2010, and so on.

As also reported in the *READMI.md* file, the user can modify only the indicated values of the files contained in the folder *Data/DataTool*, to play with the models and their setup. He/She can also change the non-indicated values but he/she may obtain different results or errors highlighted by the exceptions. The files in *Data/DataIndividuals* simply contain the dataset and they should not be modified.
The bash scripts are commented and they indicate what needs to be done to play with the different models and setup, without re-compiling every time the codes.

4.8. System Characteristics

We implement and execute the codes of a Virtual Machine (*Oracle VM VirtualBox 6.1*) installed on a computer *Dell Inspiron 15 5000 series, Intel Core i7* with 16*GB RAM* and 4 cores. The virtual machine has operative system *Ubuntu 22.04.01*, with 4*GB RAM* and 2 cores.

5 | Testing and Application to PoliMi Dataset

In this section we test the functioning of all the codes, typing on the *BashScript* folder terminal *./bash_test.sh*, that uses test dataset and test variables contained in *DataToolFiletest.txt* and *DataIndividualsTest.txt*; we show what happens when we provide wrong input data or we misspell the name of the files. In Table 5.1 and 5.2 we report the description of what we execute, what we expect to obtain and the final output.

Input	**Expected Output**	**Actual Output**
Adapted Paik eaM with random parameters in their range. No errors.	Working	LogLikelihood=-47.810 AIC=141.62, n_p=23
CSFM with Power Parameter with random parameters in their range. No errors.	Working	LogLikelihood=-12.913 AIC=67.827, n_p=21
Stochastic Time-Dependent CSFM with random parameters in their range. No errors.	Working	LogLikelihood=-10.876 AIC=53.752, n_p=16
Any model has third random parameter not in its range.	MyException	Value of parameter in position 3 not in the range.
For any model, a parameter is removed	MyException	At least a parameter is not provided.
For any model, category 0 has range_min > range_max	MyException	For category 0, min range is greater than max range.
For any model, one range_min is removed.	MyException	Either the minimum or the maximum range for a category is not provided.
For any model, smaller declared length of the vector of intervals	Working	Classic output, but different vector of interval
For any model, greater declared length of the vector of intervals	MyException	Wrong information about the length of the time intervals vector

Table 5.1: Testing the Codes

Input	Expected Output	Actual Output
For any model, omitted length of the vector of intervals	MyException	Null or not provided number of subdivision of the time domain
For any model, omitted vector of intervals	MyException	Vector of time intervals is not provided
For any model, negative number of threads	MyException	Provided negative value for number of threads
For any model, misspell the name of the first file	MyException	File DataToolFileTest.txt does not exist
For any model, misspell the name of the second file	MyException	File DatIndividualsFileTest.txt does not exist

Table 5.2: Testing the Codes

As we can observe, if we omit some values or vectors, we assign negative value to non-negative variables, we provide parameters out of their range, or the left boundary of any range is greater than the right boundary, the process stops because it has not the correct information to proceed. The same happens when we misspell the name of the files.
Different results can be obtained when we wrongly declare the dimension of the time-domain vector $vector\_intervals$ (i.e. $length\_vector\_intervals = l_{vector}$)[1]. Indeed, if this l_{vector} is smaller than the actual size of $vector\_intervals$, then the method consider only its first l_{vector} components and ignores the other. On the other hand, if l_{vector} is greater than the actual size, the method stops because it cannot assign default values to the missing time instants.

5.1. Application to the Polimi Dataset

Once we checked that the models properly work in any conditions, we apply them to the PoliMi dataset. We choose the files *DataToolFile2010.txt* and *DataIndividualsFile2010.txt* and we execute *bash_ app2010.sh.* We compare these results with those obtained in *R*, to prove they effectively work, and we report them in Table 5.3. Precisely, we indicate the value of the log-likelihood and the AIC, the first three elements of both the vector of parameter standard error and the vector of the frailty standard deviation, and the elapsed time for the complete evaluation.

[1]Be careful: it is not the number of intervals, but the length of the vector!

Model	Output in R	Output in C++
Adapted Paik eaM	Loglikelihood = -1498.5040, AIC = 3039.0080 $se[1:3] = [1.6523e-1, 9.1589e-2, 2.355e-1]$ $sd[1:3] = [3.6896e-1, 1.8987e-1, 6.3039e-1]$ Elapsed time: 6.84s	LogLikelihood = -1498.5044, AIC = 3039.0087 $se[1:3] = [1.6523e-1, 9.1602e-2, 2.3255e-1]$ $sd[1:3] = [3.6892e-1, 1.8987e-1, 6.3040e-1]$ Elapsed time: 0.0608s
CSFM with Power Parameter	Loglikelihood = -1511.6165, AIC = 3061.2330 $se[1:3] = [1.2809e-1, 7.4126e-2, 1.5618e-1]$ $sd[1:3] = [4.6784e-1, 2.1098e-7, 2.1098e-7]$ Elapsed time: 14.23s	LogLikelihood = -1511.6165, AIC = 3061.2330 $se[1:3] = [1.2810e-1, 7.4123e-2, 1.5621e-1]$ $sd[1:3] = [4.6787e-1, 2.1096e-7, 2.1101e-7]$ Elapsed time: 0.315s
Stochastic Time-Dependent CSFM	Loglikelihood = -1500.479, AIC = 3030.9580 $se[1:3] = [1.4064e-1, 8.9612e-2, 1.5858e-1]$ $sd[1:3] = [3.0438e-1, 2.1617e-1, 1.3134e-1]$ Elapsed time: $\approx 5min$	LogLikelihood = -1500, AIC = 3031 $se[1:3] = [1.406e-1, 8.960e-2, 1.585e-1]$ $sd[1:3] = [3.043e-1, 2.511e-1, 1.899e-1]$ Elapsed time: 35.3s

Table 5.3: Applying the Codes to the Polimi dataset (3000 students enrolled in 2010)

First of all, the correctness of our C++ implementation is confirmed by the correspondence of the results: the log-likelihood, the AIC and the frailty standard deviation coincide, and the same holds for the parameters standard error that has been numerically approximated. The major improvement is provided by each model execution time[2]. As we can observe in both cases, it increases with the complexity of the time-varying models and, especially, when the spatial integrals are solved with the Gauss-Hermite quadrature formula, as it happens for the *CSFM with Power Parameter* and *Stochastic Time-Dependent CSFM*. The time required by the R model execution is almost 113 times the one taken by the C++ implementation for the first model, 45 times for the second one and 9 times for the most complex of the three models. It is thus evident the necessity of performing some changes in the R implementations and, for instance, adopting a *RCpp binding*, that permits to exploit the power and efficiency of the C++ codes.
However, also the time required by the C++ version can be improved with the introduction of the parallel evaluation of the log-likelihood function. In Table 5.4, 5.5 and 5.6, we illustrate the results obtained for different scheduling strategies of the parallel for loop, on several dataset, trying to find a relationship between the dimensionality of the dataset, the best strategy and version (parallel vs serial). The reported results refer to the single evaluation of the log-likelihood function and not to the computation of the pa-

[2]All measurements are made using the *chrono* library and they are influenced by the processes running on the computer. We repeat a model execution several times and the reported value is the smallest measured.

rameter standard error and the frailty standard deviation, which method calls have been commented in the *evaluate_loglikelihood()* method. Indeed, our aim is to improve the performance of the function for the optimization step, where those two quantities are not computed. They are evaluated only once the optimal parameters have been determined.

As expected, the first thing we notice is that the execution time increases with the increase of the number of students present in the dataset and with the complexity of the time-varying models. Concerning the *Adapted Paik eaM*, the elapsed times are very small and they significantly change every time we repeat the execution of the model. Despite this fact, we can observe that the parallel executions have measured times aligned with the serial one or even worse, suggesting that it is not convenient applying the parallel version because we already have the best execution time. These results are due to the simplicity of the model, and, on the other side, to the additional costs related to the parallel environment (mainly cost of communication), that influence and increase the real execution time.

Schedule	**(n_threads, chunk_size)**	**Adapted Paik eaM**	**CSFM with Power Parameter**	**Stochastic Time-Dependent CSFM**
static (id=1)	(4, 4)	0.00161s	0.0131s	0.405s
dynamic(id=2)	(4, default)	0.00152s	0.00648s	0.373s
guided(id=3)	(4, default)	0.0020s	0.0126s	0.458s
auto(id=4)	(4, default)	0.00146s	0.00713s	0.663s
serial	–	0.000998s	0.00668s	0.722s

Table 5.4: Parallel computing on PoliMi dataset (3000 students enrolled in 2010)

Slightly different considerations are reserved to the *CSFM with Power Parameter*. As we can observe in Table 5.4 and 5.5, when the dataset contains only 3000 or 5054 students, the serial version of the log-likelihood function has still the best performance; but when we move to the biggest dataset composed of 9942 individuals (Table 5.6), the parallel version with *dynamic* scheduling strategy is able to reduce a little the elapsed time. The *static* strategy performs worse and this result is not unexpected at all since the cardinality of each faculty is different: therefore, assigning consecutive blocks of faculties to the threads causes at least a thread to have more work to do than another one, implying that the other threads have to wait until it finishes.

Schedule	**(n_threads, chunk_size)**	**Adapted Paik eaM**	**CSFM with Power Parameter**	**Stochastic Time-Dependent CSFM**
static (id=1)	(4, 4)	0.00443s	0.0121s	0.802s
dynamic(id=2)	(4, default)	0.0035s	0.0125s	0.691s
guided(id=3)	(4, default)	0.0041s	0.0135s	0.699s
auto(id=4)	(4, default)	0.00638s	0.0128s	0.744s
serial	–	0.00218s	0.0126s	1.37s

Table 5.5: Parallel computing on PoliMi dataset (5054 students enrolled in 2018)

The best improvement is for sure obtained with the *Stochastic Time-Dependent CSFM* and this is an optimal result considering that this model is the slowest among the time-varying shared frailty Cox models.
We can notice that, as long as the dimensionality of the dataset increases, any scheduling strategy performs better than the serial version; but, keeping in mind the pros and cons of all the strategies, *dynamic* is the best one and is able to progressively reduce the execution time, reaching 49% of the serial time value. These results confirm the necessity and efficiency of the parallel implementation for reducing the execution time of this complex time-varying model.

Schedule	**(n_threads, chunk_size)**	**Adapted Paik eaM**	**CSFM with Power Parameter**	**Stochastic Time-Dependent CSFM**
static (id=1)	(4, 4)	0.00374s	0.0145s	1.34s
dynamic(id=2)	(4, default)	0.00277s	0.0116s	1.24s
guided(id=3)	(4, default)	0.00342s	0.0117s	1.28s
auto(id=4)	(4, default)	0.00355s	0.0120s	1.34s
serial	–	0.00305s	0.0201s	2.34s

Table 5.6: Parallel computing on PoliMi dataset (9942 students enrolled in 2017 and 2018)

6 | Conclusion and Future Developments

In the context of survival analysis and in the context of this work, the *Time-Varying Shared Frailty Cox Models* are of paramount importance for the comprehension of the dropout phenomenon and for studying if and how the subdivision of the students into faculties influences over time the risk of facing the event.
The only available numerical codes have been implemented by me and my advisors in R, but they suffer the slowness of the evaluation of any model log-likelihood function and the non-performing applied optimization procedure. Therefore, motivated by the necessity of accelerating these codes, we have implemented them also in C++, completely from scratch.
We have described these codes, highlighted the most important features and demonstrated that they exactly performed as in R (i.e. same results), but in a faster way. At this point, exploiting the simple relationship between the group and overall log-likelihood, we tried to improve the performance of the codes adopting a parallel scheme for the computation of the overall log-likelihood and, at the end, we compared the results with the serial version of this function. Given the different and increasing complexity of the time-varying shared frailty Cox models, the adopted parallel scheme performs greatly only when applied to the *Stochastic Time-Dependent CSFM* and it is able to reduce the execution time of the serial version up to 49%, according to the dimension of the dataset. The best scheduling strategy is for sure *dynamic*, while *static* is not suggested since the students are not uniformly distributed among the groups. Concerning the other models, as long as the cardinality of the dataset is moderate, there are no improvements provided by the parallel version and thus using it is not recommended. Indeed, applying the *Adapted Paik eaM*, the parallel execution time is greater than the time of the serial version and this is mainly due to the additional costs of communication among the threads, affecting its efficiency.

We are perfectly aware that these numerical codes only work when the optimal parameters are provided in input and that the entire project misses the optimization phase.

We have encountered several difficulties with the individuation of the right constrained optimization method in multidimension and we have preferred not to implement the same inefficient scheme used in R.
However, the codes have been implemented to be editable for a future optimization method, that should take in input the log-likelihood function (indeed, it is a *lambda* function with parameter vector in input!) and other optimization-related variables (e.g. tolerances, number of iterations), that could be read from a file and stored into a base class, from which the *ModelBase* class can inherit its components.
At the end, it could be also possible to take advantage of these codes and make them available in R, for instance through a *RCpp* binding, so that they could be used together with the existent survival analysis models.
Of course these are only suggestions, but we really hope that someone could be interested in completing these codes and producing a functional instruments for the study, the comprehension and the analysis of the academic dropout phenomenon.

Bibliography

[1] The r project for statistical computing. URL `https://www.r-project.org/`.

[2] Doxygen. URL `https://www.doxygen.nl/`.

[3] Eigen library. URL `https://eigen.tuxfamily.org/dox/index.html`.

[4] std::lgamma. URL `https://en.cppreference.com/w/cpp/numeric/math/lgamma`.

[5] The openmp api specification for parallel programming. URL `https://www.openmp.org/`.

[6] std::tgamma. URL `https://en.cppreference.com/w/cpp/numeric/math/tgamma`.

[7] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*. Dovers Publications, New York, 1965.

[8] T. A. Balan and H. Putter. A tutorial on frailty models. *Statistical Methods in Medical Research*, 29(11):3424–3454, 2020. URL `https://doi:10.1177/0962280220921889`.

[9] P. C. Farrington, S. Unkel, and K. Anaya-Izquierdo. The relative frailty variance and shared frailty models. *Journal of Royal Statistical Society. Series B*, 74(4):673–696, 2012. URL `https://doi.org/10.1111/j.1467-9868.2011.01021.x`.

[10] D. Kleinbaum and M. Kelin. *Survival Analysis, A Self-Learning Text, Third Edition*. Statistics for Biology and Health. Springer Science+Business Media Inc, 2012.

[11] M. Munda, C. Legrand, L. Duchateau, and P. Janssen. Testing for decreasing heterogeneity in a new time-varying frailty model. *TEST*, 25:591–606, 2016. URL `https://doi.org/10.1007/s11749-015-0468-9`.

[12] M. C. Paik, W.-Y. Tsai, and R. Ottman. Multivariate survival analysis using piecewise gamma frailty. *Biometrics*, 50(4):975–988, 1994. URL `https://www.jstor.org/stable/2533437`.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press, 1988-1992.

[14] T. M. Therneau, P. M. Grambsch, and V. S. Pankratz. Penalized survival models and frailty. *Journal of Computational and Graphical Statistics*, 12(1):156–175, 2003. URL `https://www.jstor.org/stable/1391074`.

[15] C. M. A. Wintrebert, H. Putter, A. H. Zwinderman, and J. C. van Houwelingen. Centre-effect on survival after bone marrow transplantation: Application of time-dependent frailty models. *Biometrical Journal*, 46(5):512–525, 2004. URL `https://doi.org/10.1002/bimj.200310051`.

A | Appendix A

Here we report the general procedure for building the likelihood of one of the time-varying models and we also recall the main components needed to define it.

Let N be the number of groups in which the whole population is divided according to the value assumed by a special feature, corresponding to *Faculty* in our work.
Let t_{ij} be the time-to-event for the ith individual in the jth group and let the time-domain be divided into L intervals $I_k = [a_{k-1}, a_k)$, $k = 1, \ldots, L$, with discrete time-points $0 = a_0 < a_1 < \cdots < a_L = \infty$.
Moreover, let d_{ijk} be the event variable for the ith individual in the jth group in I_k, such that $d_{ijk} = 1$ if t_{ij} is uncensored in I_k and 0 otherwise. The usual $0 - 1$ indicator d_{ij} of event is here $\sum_k d_{ijk}$ [15].

Let Z_{jk} be the unobservable frailty of the jth centre for the kth time-interval. Conditionally on Z_{jk}, the hazard function λ_{ijk} for the ith individual, in the jth group in I_k, is given by the general expression [15]:

$$\lambda_{ijk}(t_{ij}|Z_{jk}) = Z_{jk}\ \mathrm{e}^{(X_{ij}\beta + \phi_k)}$$

Thanks to the further assumption that subjects in the same centre are independent, the likelihood $L_j(t|Z_{j.})$ conditionally on the centre frailty term $Z_{j.}$ can be obtained as:

$$L_j(t|Z_{j.}) = \prod_{i,k} \lambda_{ijk}(t_{ij}|Z_{jk})^{d_{ijk}}\ \mathrm{e}^{-\Lambda_{ijk}(t_{ij}|Z_{jk})}$$

where $\Lambda_{ijk}(t_{ij}|Z_{jk}) = \int_0^{t_{ij}} \lambda_{ijk}(s|Z_{jk})ds$ is the conditional cumulative hazard function at t_{ij}. Unless it is differently declared, the latter integral is resolved introducing the variable

$$e_{ijk} = \begin{cases} 0 & \text{if } t_{ij} < a_{k-1} \\ t_{ij} - a_{k-1} & \text{if } t_{ij} \in I_k \\ a_k - a_{k-1} & \text{if } t_{ij} \geq a_k \end{cases} \tag{A.1}$$

so that $\Lambda_{ijk}(t_{ij}|Z_{jk}) = e_{ijk}\lambda_{ijk}(s|Z_{jk})$.

Eventually, if $g(Z_{j.})$ is a general density function for the frailty $Z_{j.}$, then the likelihood L_j can be derived as:

$$L_j = \int_{Dom(Z_j)} L_j(t|Z_{j.})g(Z_{j.})dZ_{j.}$$

Substituting both the jth conditional likelihood $L_j(t|Z_j)$ and the density function $g(Z_j)$ into L_j and solving the integral with respect to space, we arrive at the extended definition of the group likelihood function L_j. The full (overall) likelihood $L = \prod_{j=1}^{N=n_{groups}} L_j$ and then we move to the log-likelihood $l = \sum_{j=1}^{N=n_{groups}} l_j$, being $l_j = log(L_j)$ the group log-likelihood.

Concerning the resolution of the spatial integral, two techniques can be used. For the *Adapted Paik et al.'s Model*, the integral is solved using the definition of the gamma function, reported in Formula A.2.
If $x \sim Gamma(\zeta, \psi)$ with $\zeta, \psi > 0$, then its density function $g(x)$ is:

$$g(x) = \frac{\psi^{\zeta}}{\Gamma(\zeta)} x^{\zeta-1}\ \mathrm{e}^{-\psi x} \quad \text{with} \quad \Gamma(\zeta) = \int_0^{\infty} x^{\zeta-1}\ \mathrm{e}^{-x} dx \tag{A.2}$$

On the other hand, the other two models solve the integral through the numerical approximation provided by the Gauss-Hermite quadrature formula indicated in A.3, where $f(x)$ is a generic function and Q is the number of points. This particular method is chosen because of the gaussian density involved in the models and the presence of the term e^{-x^2} in the integral in A.3.

$$\int_{-\infty}^{\infty} f(x)\ \mathrm{e}^{-x^2} dx = \sum_{q=1}^{Q} w_q f(\theta_q) \tag{A.3}$$