Politecnico of Milan

School of Industrial and Information Engineering
Master of Science in Mathematical Engineering

MASTER THESIS

Including covariates and spatial information into (DRPM), a bayesian time-dependent model for clustering

Advisor
Prof. Alessandra Guglielmi

Coadvisor
Prof. Alessandro Carminati

Candidate
Federico Angelo Mor
Matr. 221429

Academic Year 2023–2024

to my cats Otto
and La Micia

Abstract

Sommario

Contents

1 Introduction **1**

2 Description of the model **3**

3 Implementation and MCMC algorithm **5**

3.1 Update rules derivation 5

4 Testing on air pollution data **9**

5 Conclusion **11**

A Computational details **13**

A.1 Spatial cohesions implementation 13

A.2 Covariates similarities implementation 16

B Algorithm codes **19**

C Further plots **21**

List of Figures

A.1 Cohesion 3 implementations comparison 15
A.2 Similarity 4 annotations comparison 18

List of Tables

List of Algorithms

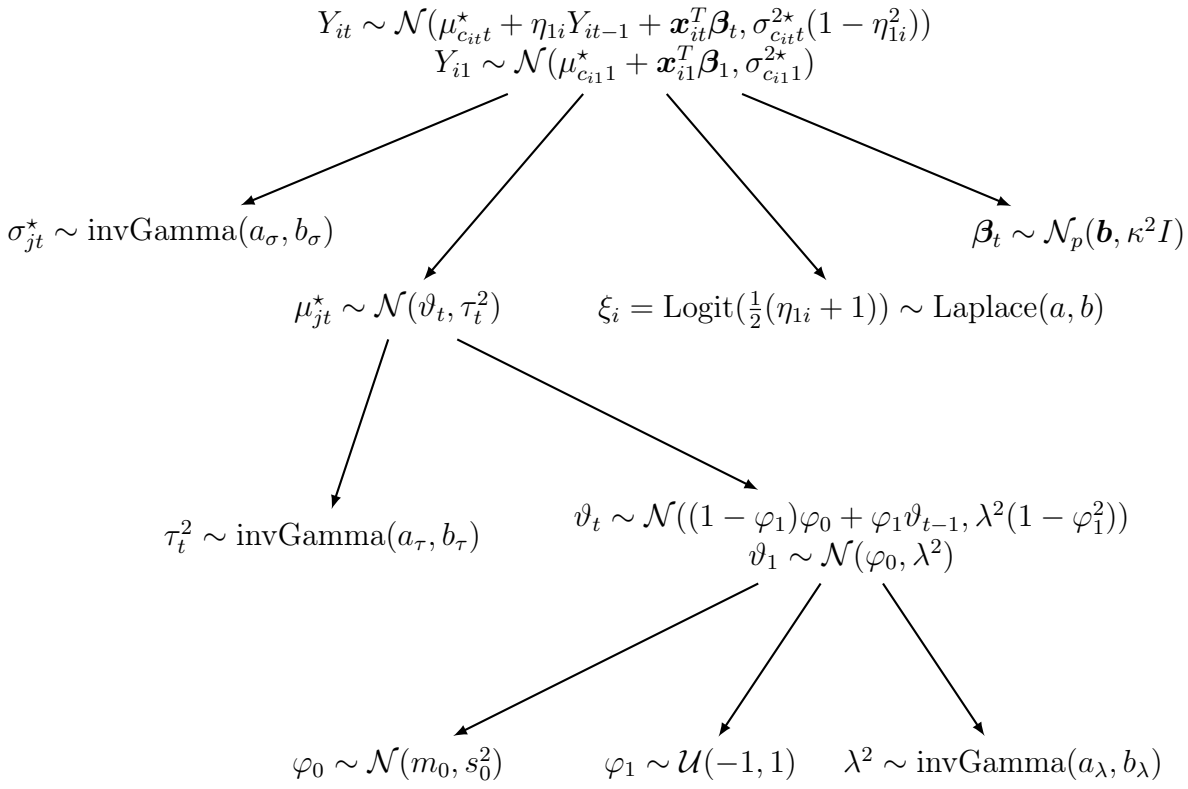
Chapter 1

Introduction

Chapter 2

Description of the model

"Come on, gentlemen, why shouldn't we get rid of all this calm reasonableness with one kick, just so as to send all these logarithms to the devil and be able to live our own lives at our own sweet will?"
— Fëdor Dostoevskij, *Notes from the Underground*



Chapter 3

Implementation and MCMC algorithm

3.1 Update rules derivation

We report here the full conditionals derivation for the parameters which had a conjugacy in the model. The other ones not included here involved instead the classical Metropolis update.

- update φ_0

$$
\begin{aligned}
f(\varphi_0|-) &\propto f(\varphi_0) f((\vartheta_1,\dots,\vartheta_T)|\varphi_0,-) \\
&= \mathcal{L}_{\mathcal{N}(m_0,s_0^2)}(\varphi_0)\left[\mathcal{L}_{\mathcal{N}(\varphi_0,\lambda^2)}(\vartheta_1)\prod_{t=2}^{T}\mathcal{L}_{\mathcal{N}\left((1-\varphi_1)\varphi_0+\varphi_1\vartheta_{t-1},\lambda^2(1-\varphi_1^2)\right)}(\vartheta_t)\right] \\
&\propto \exp\left\{-\frac{1}{2s_0^2}(\varphi_0-m_o)^2\right\}\exp\left\{-\frac{1}{2\lambda^2}(\varphi_0-\vartheta_1)^2\right\} \\
&\cdot \exp\left\{-\frac{1}{2\frac{\lambda^2(1-\varphi_1^2)}{(T-1)(1-\varphi_1)^2}}\left(\varphi_0-\frac{(1-\varphi_1)(\mathrm{SUM}_t)}{(T-1)(1-\varphi_1)^2}\right)^2\right\} \\
&\text{where } \mathrm{SUM}_t=\sum_{t=2}^{T}(\vartheta_t-\varphi_1\vartheta_{t-1}) \\
\implies f(\varphi_0|-) &\propto \text{kernel of a } \mathcal{N}\left(\mu_{\varphi_0(\mathrm{post})},\sigma^2_{\varphi_0(\mathrm{post})}\right) \text{ with} \\
\sigma^2_{\varphi_0(\mathrm{post})} &= \frac{1}{\frac{1}{s_0^2}+\frac{1}{\lambda^2}+\frac{(T-1)(1-\varphi_1)^2}{\lambda^2(1-\varphi_1^2)}} \\
\mu_{\varphi_0(\mathrm{post})} &= \sigma^2_{\varphi_0(\mathrm{post})}\left[\frac{m_0}{s_0^2}+\frac{\vartheta_1}{\lambda^2}+\frac{1-\varphi_1}{\lambda^2(1-\varphi_1^2)}\mathrm{SUM}_t\right]
\end{aligned}
$$

- update ϑ_t

for $t = T$:

$$
\begin{aligned}
f(\vartheta_t|-) &\propto f(\vartheta_t) f(\boldsymbol{\mu}_t^\star, -) \\
&= f(\vartheta_t) \prod_{j=1}^{k_t} f(\mu_{jt}^\star|\vartheta_t, -) \\
&= \mathcal{L}_{\mathcal{N}((1-\varphi_1)\varphi_0+\varphi_1\vartheta_{t-1},\lambda^2(1-\varphi_1^2))}(\vartheta_t) \prod_{j=1}^{k_t} \mathcal{L}_{\mathcal{N}(\vartheta_t,\tau_t^2)}(\mu_{jt}^\star) \\
&\propto \exp\left\{-\frac{1}{2(\lambda^2(1-\varphi_1^2))}\Big(\vartheta_t - ((1-\varphi_1)\varphi_0 + \varphi_1\vartheta_{t-1})\Big)^2\right\} \\
&\quad \cdot \exp\left\{-\frac{k_t}{2\tau_t^2}\left(\vartheta_t - \frac{\sum_{j=1}^{k_t}\mu_{jt}^\star}{k_t}\right)\right\} \\
\implies f(\vartheta_t|-) &\propto \text{kernel of a } \mathcal{N}\left(\mu_{\vartheta_t(\text{post})}, \sigma^2_{\vartheta_t(\text{post})}\right) \text{ with} \\
\sigma^2_{\vartheta_t(\text{post})} &= \frac{1}{\frac{1}{\lambda^2(1-\varphi_1^2)} + \frac{k_t}{\tau_t^2}} \\
\mu_{\vartheta_t(\text{post})} &= \sigma^2_{\vartheta_t(\text{post})}\left[\frac{\sum_{j=1}^{k_t}\mu_{jt}^\star}{\tau_t^2} + \frac{(1-\varphi_1)\varphi_0 + \varphi_1\vartheta_{t-1}}{\lambda^2(1-\varphi_1^2)}\right]
\end{aligned}
\tag{3.1}
$$

for $1 < t < T$:

$$
\begin{aligned}
f(\vartheta_t|-) &\propto \underbrace{f(\vartheta_t) f(\boldsymbol{\mu}_t^\star, -)}_{\text{as in the case } t=T} f(\vartheta_{t+1}|\vartheta_t, -) \\
&= \mathcal{L}_{\mathcal{N}\left(\mu_{\vartheta_t(\text{post})},\sigma^2_{\vartheta_t(\text{post})}\right)}(\vartheta_t) \mathcal{L}_{\mathcal{N}\left((1-\varphi_1)\varphi_0+\varphi_1\vartheta_t,\lambda^2(1-\varphi_1^2)\right)}(\vartheta_{t+1}) \\
&\propto \exp\left\{-\frac{1}{2\sigma^2_{\vartheta_t(\text{post})}}\left(\vartheta_t - \mu_{\vartheta_t(\text{post})}\right)^2\right\} \cdot \\
&\quad \exp\left\{-\frac{1}{2\frac{\lambda^2(1-\varphi_1^2)}{\varphi_1^2}}\left(\vartheta_t - \frac{\vartheta_{t+1} - (1-\varphi_1)\varphi_0}{\varphi_1}\right)^2\right\} \\
\implies f(\vartheta_t|-) &\propto \text{kernel of a } \mathcal{N}\left(\mu_{\vartheta_t(\text{post})}, \sigma^2_{\vartheta_t(\text{post})}\right) \text{ with} \\
\sigma^2_{\vartheta_t(\text{post})} &= \frac{1}{\frac{1+\varphi_1^2}{\lambda^2(1-\varphi_1^2)} + \frac{k_t}{\tau_t^2}} \\
\mu_{\vartheta_t(\text{post})} &= \sigma^2_{\vartheta_t(\text{post})}\left[\frac{\sum_{j=1}^{k_t}\mu_{jt}^\star}{\tau_t^2} + \frac{\varphi_1(\vartheta_{t-1}+\vartheta_{t+1}) + \varphi_0(1-\varphi_1)^2}{\lambda^2(1-\varphi_1^2)}\right]
\end{aligned}
$$

for $t = 1$:

$$
\begin{aligned}
f(\vartheta_t|-) &\propto f(\vartheta_t) f(\vartheta_{t+1}|\vartheta_t, -) f(\boldsymbol{\mu}_t^\star|\vartheta_t, -) \\
&= \mathcal{L}_{\mathcal{N}(\varphi_0, \lambda^2)}(\vartheta_t) \mathcal{L}_{\mathcal{N}\left((1-\varphi_1)\varphi_0 + \varphi_1 \vartheta_{t-1}, \lambda^2(1-\varphi_1^2)\right)}(\vartheta_{t+1}) \prod_{j=1}^{k_t} \mathcal{L}_{\mathcal{N}(\vartheta_t, \tau_t^2)}(\mu_{jt}^\star) \\
&\propto \exp\left\{ -\frac{1}{2\lambda^2}(\vartheta_t - \varphi_0)^2 \right\} \\
&\quad \cdot \exp\left\{ -\frac{1}{2\frac{\lambda^2(1-\varphi_1^2)}{\varphi_1^2}} \left(\vartheta_t - \frac{\vartheta_{t+1} - (1-\varphi_1)\varphi_0}{\varphi_1} \right)^2 \right\} \\
&\quad \cdot \exp\left\{ -\frac{k_t}{2\tau_t^2} \left(\vartheta_t - \frac{\sum_{j=1}^{k_t} \mu_{jt}^\star}{k_t} \right) \right\} \\
\implies f(\vartheta_t|-) &\propto \text{kernel of a } \mathcal{N}\left(\mu_{\vartheta_t(\text{post})}, \sigma^2_{\vartheta_t(\text{post})}\right) \text{ with} \\
\sigma^2_{\vartheta_t(\text{post})} &= \frac{1}{\frac{1}{\lambda^2} + \frac{\varphi_1^2}{\lambda^2(1-\varphi_1^2)} + \frac{k_t}{\tau_t^2}} \\
\mu_{\vartheta_t(\text{post})} &= \sigma^2_{\vartheta_t(\text{post})} \left[\frac{\varphi_0}{\lambda^2} + \frac{\varphi_1(\vartheta_{t+1} - (1-\varphi_1)\varphi_0)}{\lambda^2(1-\varphi_1^2)} + \frac{\sum_{j=1}^{k_t} \mu_{jt}^\star}{\tau_t^2} \right]
\end{aligned}
$$

Chapter 4

Testing on air pollution data

Chapter 5

Conclusion

Appendix A

Computational details

One major issue encountered during the development and testing of the fitting function in Julia was the enormous amount of memory and allocations that some functions or structures would require.

A.1 Spatial cohesions implementation

The memory allocation problem was especially seen in the spatial cohesion computation. In fact, such computation appears twice in both sections of updating γ and updating ρ, which are inside the outer loops on draws, time, and units, and involve themselves some other loops on clusters. All this implied those cohesion computations to be executed possibly millions of times during a fit. A simple and quick inspection suggests a value between $\Omega(dTn)$ and $O(dTn^2)$, being d the number of iterations, T the time horizon and n the number of units. We can't provide a Θ estimate due to the variability and randomicity of the inside loops, which depend on the distribution of the clusters. Considering all of this, it was crucial to optimize the performance of the cohesion functions.

Function calls in Julia are very cheap, so the only optimizing task consisted in carefully designing the cohesion function computation. Cohesions 1, 2, 5 and 6 didn't exhibit any complication or need for optimization. Their natural conversion in code from their mathematical models proved to be already optimally performing. The main problem was with cohesions 3 and 4, the auxiliary and double dippery, which by nature would involve some linear algebra computation. However, the vector implementation of those cohesions turned out to be really slow, due to the overhead generated by allocating and then freeing, at each call, the memory devoted to vectors and matrices. This in the end meant that most of the execution time was actually spent by the garbage collector, rather than in the real and useful operations. The solution was then to resort to a scalar implementation, i.e. working singularly on the components of the vectors and matrices involved. A less elegant solution but way more efficient, having to deal with scalars values only. Figure A.1 shows the comparison about their performances.

Listing 1: Definitions of the two versions of the cohesion 3 function.

```
# vector version
function cohesion3_vect(s1::AbstractVector{Float64},
↪  s2::AbstractVector{Float64}, mu_0::AbstractVector{Float64}, k0::Real,
↪  v0::Real, Psi::Matrix{Float64}; Cohesion::Int, lg::Bool, M::Real=1.0)
    sdim = length(s1)
    sp = [s1 s2]
    # vector of sample means
    sbar = [mean(s1), mean(s2)]
    # deviation matrix
    S = zeros(2,2)
    for i in 1:sdim
        vtemp1 = sp[i,:] - sbar
        S += (vtemp1)*(vtemp1)'
    end

    # updated parameters for cohesion 3
    kn = k0+sdim
    vn = v0+sdim
    vtemp2 = sbar-mu_0
    Psi_n = Psi + S + (k0*sdim)/(k0+sdim)*vtemp2*vtemp2'

    out = -sdim * logpi + G2a(0.5 * vn, true) - G2a(0.5 * v0, true) + 0.5 * v0 *
    ↪  logdet(Psi) - 0.5 * vn * logdet(Psi_n) + log(k0) - log(kn)
    return lg ? out : exp(out)
end

# scalar version
function cohesion3_scal(s1::AbstractVector{Float64},
↪  s2::AbstractVector{Float64}, mu_0::AbstractVector{Float64}, k0::Real,
↪  v0::Real, Psi::Matrix{Float64}; lg::Bool, M::Real=1.0)
    sdim = length(s1)
    # "vector" of sample means
    sbar1 = mean(s1)
    sbar2 = mean(s2)
    # deviation "matrix"
    S1, S2, S3, S4 = 0.0, 0.0, 0.0, 0.0
    @inbounds for i in 1:sdim
        s_sbar1 = s1[i] - sbar1
        s_sbar2 = s2[i] - sbar2
        S1 += s_sbar1 * s_sbar1
        S4 += s_sbar2 * s_sbar2
        S2 += s_sbar1 * s_sbar2
    end
    S3 = copy(S2)

    # updated parameters for cohesion 3
    kn = k0 + sdim
    vn = v0 + sdim

    auxvec1_1 = sbar1 - mu_0[1]
    auxvec1_2 = sbar2 - mu_0[2]

    auxmat1_1 = auxvec1_1^2
    auxmat1_2 = auxvec1_1 * auxvec1_2
    auxmat1_3 = copy(auxmat1_2)
    auxmat1_4 = auxvec1_2^2
```

```
    auxconst1 = k0 * sdim
    auxconst2 = k0 + sdim
    Psi_n_1 = Psi[1] + S1 + auxconst1 / (auxconst2) * auxmat1_1
    Psi_n_2 = Psi[2] + S2 + auxconst1 / (auxconst2) * auxmat1_2
    Psi_n_3 = Psi[3] + S3 + auxconst1 / (auxconst2) * auxmat1_3
    Psi_n_4 = Psi[4] + S4 + auxconst1 / (auxconst2) * auxmat1_4

    detPsi_n = Psi_n_1 * Psi_n_4 - Psi_n_2 * Psi_n_3
    detPsi = Psi[1] * Psi[4] - Psi[2] * Psi[3]

    out = -sdim * logpi + G2a(0.5 * vn, true) - G2a(0.5 * v0, true) + 0.5 * v0 *
    ↪  log(detPsi) - 0.5 * vn * log(detPsi_n) + log(k0) - log(kn)
    return lg ? out : exp(out)
end
```

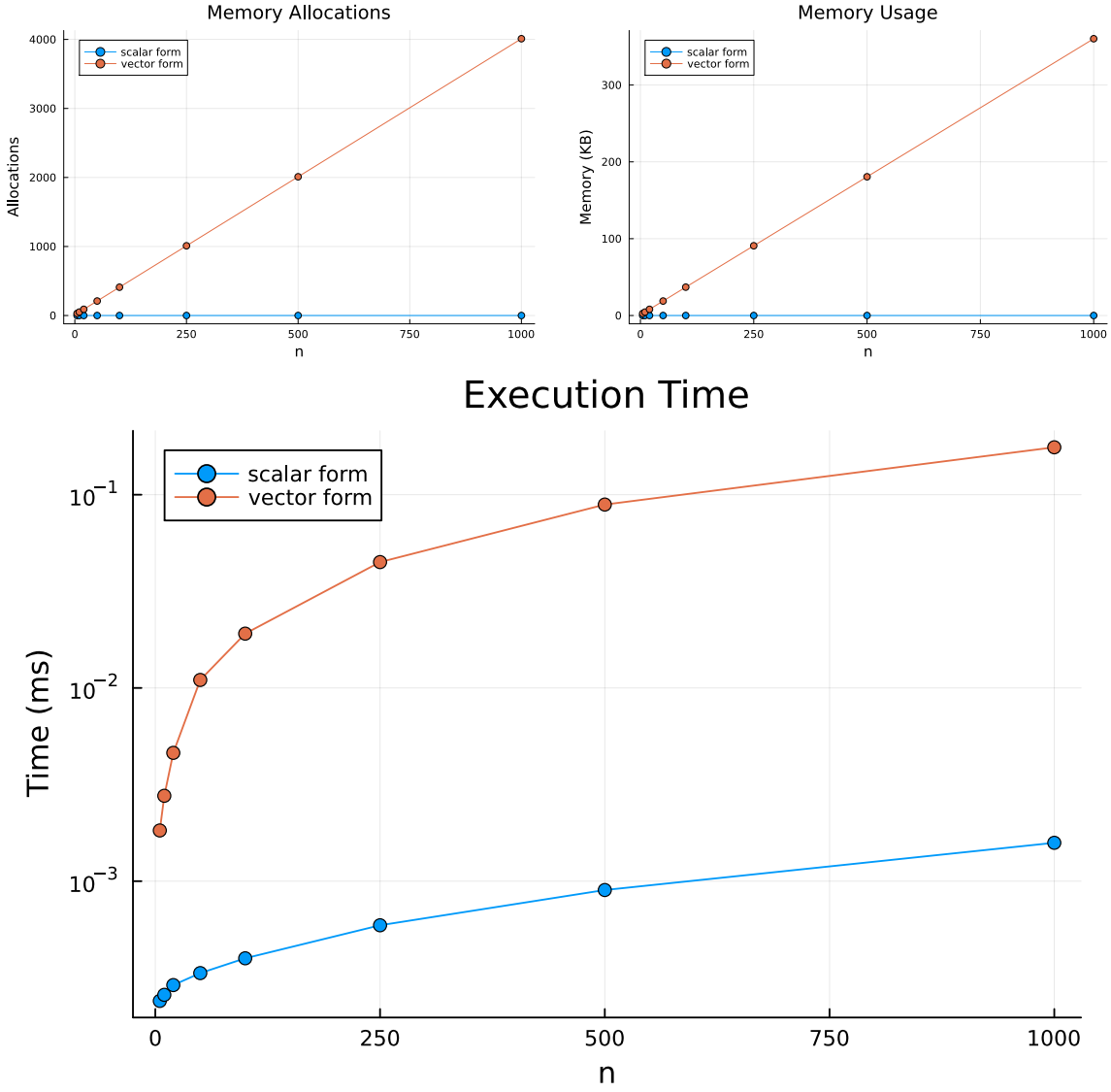


Figure A.1: Performance comparison between the two versions of the cohesion 3 function. Tests ran through the `BenchmarkTools` package of Julia and by randomly generating the spatial coordinates of the various test sizes n. Similar results stand about cohesion 4.

Noticeably, the C implementation of the model didn't have to worry about this, since C can't agilely work with vectors and matrices. So one point for the old lad.

The cohesion 3 in the scalar form code actually could be further optimized, just slightly, by noticing that some variables are not really required, due to the symmetry of the matrices involved, so some variables and relative `copy`es could be removed. However we decided to keep the full version so to keep it more readable and understandable in case of future updates or revisions of the code.

Another solution could have been keeping the vector implementation of the function and preallocating all those non-scalar variables, to reuse them at every function call. However this would have made the code more crowded of variables going around until their place of interest; a way that would have made the code less readable and uniform in his structure (since the other cohesions do not need any special preallocation and recycling of variables).

A.2 Covariates similarities implementation

Another problem was how to speed up the computation of the similarity functions, since those would also be called the possibly millions of times as the cohesion ones. Actually even more, considering that we could include more than one covariate into the clustering process, so there would be an additional loop based on p, the number of covariates elicited for the clustering.

As in the previous case, some of the functions didn't show any special need or room for relevant optimizations. The fourth one instead, the auxiliary similarity function, was essential to be optimized, since it is one the most frequent choice among all the similarities and because it involves a heavy sum of the squares of the covariate values, as we can see in Listing 2.

Listing 2: Final version of the similarity 4 function.

```
function similarity4(X_jt::AbstractVector{<:Real}, mu_c::Real, lambda_c::Real,
↪  a_c::Real, b_c::Real; lg::Bool)
    n = length(X_jt)
    nm = n/2
    xbar = mean(X_jt)
    aux2 = 0.0
    @inbounds @fastmath @simd for i in eachindex(X_jt)
        aux2 += X_jt[i]^2
    end
    aux1 = b_c + 0.5 * (aux2 - (n*xbar + lambda_c*mu_c)^2/(n+lambda_c) +
    ↪  lambda_c*mu_c^2 )
    out = -nm*log2pi + 0.5*log(lambda_c/(lambda_c+n)) + lgamma(a_c+nm) -
    ↪  lgamma(a_c) + a_c*log(b_c) + (-a_c-nm)*log(aux1)
    return lg ? out : exp(out)
end
```

The idea to optimize it has been to annotate the loop with some macros provided by Julia. They are the following:

- `@inbounds` eliminates the array bounds checking within expressions. This allows to skip the checks, at the cost to guarantee, by code design, that no out-of-bounds or wrong accesses will occur in the code and therefore no undefined behaviour will take place.

- `@fastmath` executes a transformed version of the expression, which calls functions that may violate strict IEEE semantics[1]. For example, its use could make $a + b \neq b + a$ (but just in very pathological cases). Anyway, this is not a problem in our case, where we are computing $\sum X_i^2$, which does not have a "right order" in which has to be done.

- `@simd` (single instruction multiple data) annotate a for loop to allow the compiler to take extra liberties to allow loop re-ordering. This is a sort of parallelism technique, but rather than distributing the computational load on more processors we just *vectorize* the loop, i.e. we enable the CPU to perform that single instruction (summing the square of the ith component to a reduction variable) on multiple data chunks at once, using vector registers, rather than working on each element of the vector individually.

As we can see from Figure A.2, the actual performance difference, as expected, basically derives just from the use of `@simd`, with the other two annotations making not much of a difference.

[1]Institute of Electrical and Electronics Engineers. The IEEE-754 standard defines floating-point formats, i.e. the ways to represent real numbers in hardware, and the expected behavior of arithmetic operations on them, including precision, rounding, and handling of special values (e.g. NaN (Not a Number) and infinity).

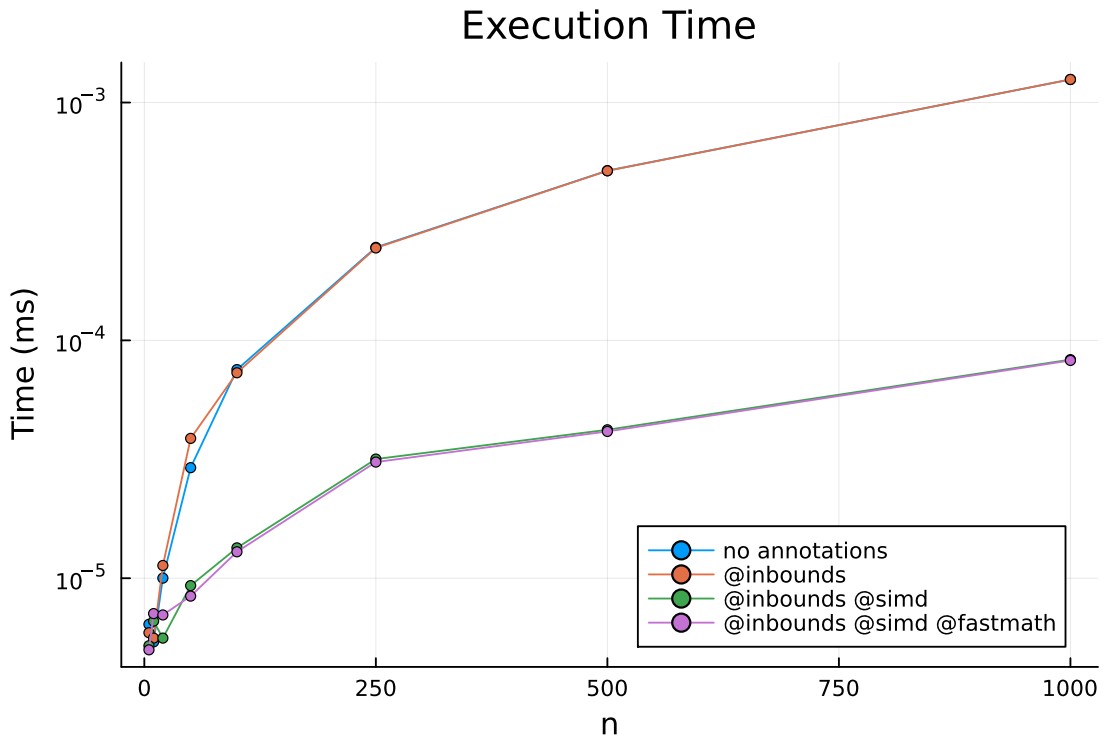


Figure A.2: Comparison of the performances of the different possible loop annotations in the similarity 4 implementation. Their numerical output results are indeed the same for all cases.

Appendix B

Algorithm codes

Appendix C

Further plots

Acknowledgements

Ringraziamenti

"Ecco il mio ponte d'approdo per molti lunghi anni, il mio angoletto, nel quale faccio il mio ingresso con una sensazione così diffidente, così morbosa... Ma chi lo sa? Forse, quando tra molti anni mi toccherà abbandonarlo, magari potrei anche rimpiangerlo!"

— Fëdor Dostoevskij, *Memorie da una casa di morti*