BKTR: An efficient spatiotemporally varying coefficient regression package in R and Python

Julien Lanthier
HEC Montréal

Mengying Lei
McGill University

Lijun Sun
McGill University

Aurélie Labbe
HEC Montréal

Abstract

BKTR is a new software library for spatiotemporal regression analysis with varying coefficients that allows for efficient and easy-to-use inference over datasets that vary in time and space. The library is implemented in R and Python, providing a flexible and easy-t-use framework for spatiotemporal regression models. One of the main challenges in spatiotemporal modelling when using local regression is the computational cost. **BKTR** addresses this computational challenge by implementing a tensor regression approach, which greatly reduces the model's computational cost. The calculation speed is further improved using the specialized tensor library **torch** (in both R and Python), which enables optimal matrix and tensor computation on GPUs and TPUs. The framework also assumes Gaussian process (GP) priors to capture the spatial and temporal dependencies of the data in a Bayesian context. Hence, the full name of the framework, Bayesian Kernelized Tensor Regression, refers to the combined usage of tensor regression and GP models. The Python **pyBKTR** package is available on PyPI and the R **BKTR** package has been submitted to CRAN.

Keywords: Gaussian process, Tensor regression, Local spatiotemporal regression, R, Python.

---

1. Introduction

With the rise of the Internet of Things (IoT) and the considerable increase in mobile device and sensor use, the amount of available data varying through time and space has been growing rapidly. Thus, statistical analyses such as spatiotemporal regressions that take into account the spatial and temporal aspects of data have become increasingly valuable. Spatiotemporal regressions are especially useful for analyzing and predicting complex phenomena like weather patterns, agricultural output, transportation, epidemiology outbreaks and much more. It is possible to regroup spatiotemporal regressions into two main regression categories: global and local. The main difference between the two is their approach to modelling relationships between variables over space and time. Global regression assumes that the relationships between variables are constant across space and time. In contrast, local regression allows for the relationship between variables to vary across different locations and time points. Local regression is often viewed as a more flexible and better suited method for capturing changes

in variables' relationships in time and space.

Even if local regression is usually more flexible and leads to a better fit, this method is much more computationally expensive than global regression. This is due to the fact that local regression needs to estimate coefficients at each location and time point studied. In fact, for a response matrix $Y \in \mathbb{R}^{M \times N}$ observed from a set of locations $S = \{s_1, \ldots, s_M\}$ and a set of time points $T = \{t_1, \ldots, t_N\}$, we can define the model over a Cartesian product $S \times T = \{(s_m, t_n) : m = 1, \ldots, M,\ n = 1, \ldots, N\}$ and formulate local spatiotemporal regression as:

$$y(s_m, t_n) = \boldsymbol{x}(s_m, t_n)^T \boldsymbol{\beta}(s_m, t_n) + \epsilon(s_m, t_n), \tag{1}$$

where $y(s_m, t_n)$ is the response variable at location s_m and time t_n, and $x(x_m, t_n)$ and $\beta(s_m, t_n)$ are the covariates and coefficients at location s_m and time t_n, respectively. Local spatiotemporal regression has already been explored and implemented in the literature. For example, Gelfand *et al.* (2003) have already suggested using a separable kernel to build a spatiotemporally varying coefficient model (STVC). However, even with the advent of new computing technologies, local regression models for spatiotemporal data like STVC is still unable to run on even medium-sized datasets, due to the sheer number of local regression's parameters. To the best of our knowledge, no readily available software package currently allows for the efficient use of local spatiotemporal regression. The Bayesian Kernelized Tensor Regression (BKTR) method proposed by Lei *et al.* (2023) overcomes this limitation by using a tensor decomposition to estimate coefficients. In so doing, the time complexity of each sampling iteration changes from $\mathcal{O}(M^3N^3P^3)$ for the STVC method to $\mathcal{O}(R^3(M^3 + N^3 + P^3))$, where R is usually an arbitrary small value denoting the rank of the tensor decomposition. The BKTR method also uses a Gaussian process (GP) prior with a spatial and a temporal kernel to model the spatiotemporal dependence of the coefficients.

From a software perspective, a wide range of R (R Core Team 2023) packages is currently available on the Comprehensive R Archive Network (CRAN) for exploring the spatial characteristics of datasets. Some general spatial packages including **sp** (Pebesma and Bivand 2005), **spatial** (Venables and Ripley 2002), **spacetime** (Pebesma 2012) and **spatstat** (Baddeley and Turner 2005) enable the analysis and visualization of spatial patterns. Additional packages such as **splm** (Millo and Piras 2012), and **fields** (Nychka *et al.* 2021) facilitate spatial regressions and kriging. The**gstat** (Pebesma 2004) package initially focused on spatial models, but now enables spatiotemporal modelling using covariance models (Gräler *et al.* 2016).

Significant advancements in spatio-temporal data modelling are linked to the integration of regression models based on the Bayesian paradigm. Implementing Bayesian Markov chain Monte Carlo (MCMC) inference can be accomplished through packages like **R2WinBUGS** (Sturtz *et al.* 2005), **rjags** (Plummer 2023) and **rstan** (Stan Development Team 2023), which are R wrappers for the **WinBugs** (Lunn *et al.* 2000), **JAGS** (Plummer 2003) and **Stan** (Carpenter *et al.* 2017) general MCMC frameworks, written in low-level languages. An alternative to MCMC for Bayesian inference, which can be more computationally efficient, is the Integrated Nested Laplace Approximation (INLA) framework (Rue *et al.* 2009). This approach has been successfully applied to different spatial datasets (Lindgren and Rue 2015) using the **R-INLA** package. While these packages provide versatile Bayesian frameworks, the complexity of implementing spatiotemporal models, especially with localized spatio-temporal variations, often requires specialized expertise.

New integrated Bayesian regression software solutions such as **spate** (Sigrist *et al.* 2015), simplifying spatiotemporal regression while accounting for temporal aspects of the data. However,

spate lacks the ability to incorporate local spatial regression, which restricts the inclusion of locally varying temporal patterns. To address this limitation, packages such as **spTimer** (Bakar and Sahu 2015), **spBayes** (Finley *et al.* 2015) and **spTDyn** (Bakar *et al.* 2016) have emerged, making it easier to implement spatio-temporal regression models with spatially varying coefficients. As demonstrated in Table 1, an unmet need for local temporal regression has persisted, but this failing has been rectified by the introduction of the **BKTR** package. By efficiently incorporating spatiotemporally varying coefficients, BKTR enhances spatiotemporal analyses and broadens the toolkit available to researchers and analysts

R Package	Regression Types				
	Spatial	Temporal	Bayesian	Local in Space	Local in Time
splm	✓				
fields	✓				
gstat	✓	✓			
spate	✓	✓	✓		
spBayes	✓	✓	✓	✓	
spTimer	✓	✓	✓	✓	
spTDyn	✓	✓	✓	✓	
BKTR	✓	✓	✓	✓	✓

Table 1: Summary of the spatial regression packages available in R

The packages presented in this paper, **BKTR** and **pyBKTR**, implement the BKTR method and provide a user-friendly interface for estimating coefficients for local spatiotemporal regression. The **BKTR** package is implemented in R and is available on CRAN. The **pyBKTR** library is implemented in Python and is available on PyPI (`https://pypi.org/project/pyBKTR`). Both packages provide the same functionalities and are designed to be used in a similar fashion. To mimic the object-oriented patterns of Python, we used the **R6** package (Chang 2021) in **BKTR** for classes and methods. Also, to be able to use a similar approach for tensor operations, we used **Torch** in both R and Python packages which translate to **torch** (Falbel and Luraschi 2023) and **pytorch** (Paszke *et al.* 2019), respectively. The results can be visualized using the **ggplot2** package (Wickham 2016) in R and **plotly** (Plotly Technologies Inc. 2015) in Python. Furthermore, to be able to use Wilkinson formulae (Wilkinson and Rogers 1973), as in the R formula object, we use the **Formulaic** (Wardrop 2022) python package. For DataFrame usage, we use the **pandas** package (The Pandas Development Team 2022) in Python and **data.table** (Dowle and Srinivasan 2023) in R. All examples given in this paper are available in a GitHub repository (`https://github.com/julien-hec/bktr-examples`). This paper focuses mainly on the R implementation of **BKTR**, but the syntaxes of the R and Python packages we implemented are very similar. To convert covered examples from R to Python, it should be sufficient to convert all base 1 indexes to base 0 and to change "`$`" to "`. `", "`<-`" to "`=`" and "`$new()`" to "`()`". Also, the source code is available on GitHub at `https://github.com/julien-hec/BKTR/` and `https://github.com/julien-hec/pyBKTR/` for **BKTR** and **pyBKTR**, respectively. Note that when we refer to the **BKTR** packages in this paper (plural form), we are referring to both the R and Python implementations at the same time.

The rest of this paper is organized as follows. Section 2 presents the BKTR algorithm. Section 3 presents the `BKTRRegressor` class and its attributes. Section 4 presents the different kernels

available in the package. Section 5 presents a simulation study to validate our implementation of the BKTR regression. Section 6 presents an experimental study on bike sharing data. Finally, Section 7 concludes the paper.

2. BKTR algorithm

The objective of Bayesian Kernelized Tensor Regression (BKTR) is to model a response variable $\boldsymbol{Y}$ as a function of spatiotemporal covariates $\boldsymbol{\mathcal{X}}$, with the model's coefficients allowed to vary in time and space. In addition to the spatial and temporal framework, BKTR also considers the case where a proportion of the response matrix $\boldsymbol{Y}$ can be unobserved or corrupted, given observed values of the covariates $\boldsymbol{\mathcal{X}}$. The covariates $\boldsymbol{\mathcal{X}}$ represent a tensor of size $M \times N \times P$ where M is the number of locations, N is the number of time points and P is the number of covariates at each location and time point we want to model. The response variable $\boldsymbol{Y}$, for its part, is a matrix of size $M \times N$. One application of this could be to model the yearly hospital beds number per 10,000 people of M districts through N months, using P socioeconomic covariates that change through time and space.

This section is based considerably on the work of Lei *et al.* (2023) and aims to summarize the BKTR model they have described. In Section 2.3, we discuss a previously unexplored aspect of BKTR regarding interpolation.

2.1. Model definition

It is possible to reshape the above-mentioned covariates tensor $\boldsymbol{\mathcal{X}}$ and the response variable $\boldsymbol{Y}$ to obtain a vectorized version of Equation 1:

$$\boldsymbol{y} = (\boldsymbol{I}_{MN} \odot \boldsymbol{X}_{(3)})^\top \text{vec}(\boldsymbol{B}_{(3)}) + \boldsymbol{\epsilon}, \tag{2}$$

where $\boldsymbol{y}$ is the vectorized version in $\mathbb{R}^{MN}$ of $\boldsymbol{Y}$, and $\boldsymbol{X}_{(3)}$ and $\boldsymbol{B}_{(3)}$ are the mode-3 unfolding of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{B}}$, respectively. The $\odot$ operator is the Khatri-Rao product (Khatri and Rao 1968) and the product $\boldsymbol{I}_{MN} \odot \boldsymbol{X}_{(3)}$ is a sparse expansion of the covariates. The error term $\boldsymbol{\epsilon}$ is assumed to follow a multivariate normal distribution such that $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \tau^{-1}\boldsymbol{I}_{MN})$, where $\boldsymbol{I}_{MN}$ is an identity matrix of size $MN \times MN$. Assuming that $\boldsymbol{\mathcal{B}}$ admits a CANDECOMP/PARAFAC (CP) (Kolda and Bader 2009) decomposition with rank $R \ll \min\{M, N\}$, i.e., $\boldsymbol{\mathcal{B}} = \sum_{r=1}^{R} \boldsymbol{u}_r \circ \boldsymbol{v}_r \circ \boldsymbol{w}_r$, the model can be rewritten as

$$\boldsymbol{y} = \tilde{\boldsymbol{X}} \text{vec}(\boldsymbol{W}(\boldsymbol{V} \odot \boldsymbol{U})^\top) + \boldsymbol{\epsilon}, \tag{3}$$

where $\tilde{\boldsymbol{X}} = (\boldsymbol{I}_{MN} \odot \boldsymbol{X}_{(3)})^\top$ and $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ are matrices of size $M \times R$, $N \times R$ and $P \times R$, respectively, containing the concatenated vectors $\boldsymbol{u}_r$, $\boldsymbol{v}_r$ and $\boldsymbol{w}_r$ of the CP decomposition of $\boldsymbol{\mathcal{B}}$.

To account for missing values in response variable $\boldsymbol{Y}$, we can rewrite the model of Equation 3 as:

$$\boldsymbol{y}_\Omega \sim \mathcal{N}\left(\boldsymbol{O}\left(\tilde{\boldsymbol{X}} \text{vec}(\boldsymbol{W}(\boldsymbol{V} \odot \boldsymbol{U})^\top)\right), \tau^{-1}\boldsymbol{I}_{|\Omega|}\right). \tag{4}$$

where $\boldsymbol{y}_\Omega$ is the vectorized version of $\boldsymbol{Y}$ restricted to the observed entries Ω, $\boldsymbol{O}$ is the operator selecting the observed entries of a vector and $|\Omega|$ is the number of observed entries,

so that given $\boldsymbol{O}$, $\boldsymbol{y}_\Omega = \boldsymbol{O}\boldsymbol{y}$. To account for the spatial and temporal correlation during CP decomposition, we use GP priors on the spatial and temporal component vectors:

$$\begin{aligned} \boldsymbol{u}_r &\sim \mathcal{GP}(0, \boldsymbol{K}_s),\ r = 1, \ldots, R, \\ \boldsymbol{v}_r &\sim \mathcal{GP}(0, \boldsymbol{K}_t),\ r = 1, \ldots, R, \end{aligned} \tag{5}$$

where $\boldsymbol{K}_s$ and $\boldsymbol{K}_t$ are covariance matrices of size $M \times M$ and $N \times N$, respectively, derived from two kernel functions $k_s(s_m, s_{m'}; \Phi)$ and $k_t(t_n, t_{n'}; \Gamma)$, where Φ is a vector of J spatial kernel hyperparameters $\Phi = \{\phi_1, \ldots, \phi_J\}$ and Γ is a vector of L temporal kernel hyperparameters $\Gamma = \{\gamma_1, \ldots, \gamma_L\}$. The priors of kernel hyperparameters are

$$\begin{aligned} \log(\phi_i) &\sim \mathcal{N}(\mu_{\phi_i}, \tau_{\phi_i}^{-1}),\ i = 1, \ldots, J, \\ \log(\gamma_i) &\sim \mathcal{N}(\mu_{\gamma_i}, \tau_{\gamma_i}^{-1}),\ i = 1, \ldots, L. \end{aligned} \tag{6}$$

It can be noted that, for BKTR, $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ are in fact correlation matrices, as we set the kernel variance to 1 and capture the variance through $\boldsymbol{W}$.

For the components of factor matrix $\boldsymbol{W}$, we use the following priors:

$$\begin{aligned} \boldsymbol{w}_r &\sim \mathcal{GP}(0, \boldsymbol{\Lambda}_w^{-1}),\ r = 1, \ldots, R, \\ \boldsymbol{\Lambda}_w &\sim \mathcal{W}(\boldsymbol{\Psi_0}, \nu_0), \end{aligned} \tag{7}$$

where $\boldsymbol{\Psi_0}$ is a $P \times P$ scale matrix and ν_0 is the number of degrees of freedom.

The prior on the noise precision τ in Equation 4 is $\tau \sim \text{Gamma}(a_0, b_0)$.

An illustration of the BKTR framework taken from (Lei *et al.* 2023) is shown in Figure 1 to help visualize the dependencies between variables and the different steps of the algorithm.

2.2. Sampling

This section provides only a brief overview of the sampling algorithm and its main steps, which are described in Algorithm 1. More details about the conditional posterior distributions from which BKTR parameters are sampled can be found in Lei *et al.* (2023).

The BKTR MCMC algorithm uses Gibbs sampling (Geman and Geman 1984) for the parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}, \tau$ and the precision matrix $\boldsymbol{\Lambda}_w$. For the hyperparameters Φ and Γ, a slice sampling algorithm (Neal 2003) is used, as the conditional posterior distribution of these parameters is not easy to sample from.

The sampling process uses K_1 burn-in iterations and K_2 iterations to sample the posterior distribution of the parameters. The number of iterations K_1 and K_2 are arbitrary, chosen by the user and ideally large enough to ensure the Markov chains reach a stationary state before sampling begins.

The posterior samples $\{\boldsymbol{\mathcal{B}}^{(k)}\}_{k=1}^{K_2}$ are used to approximate the posterior distribution of the coefficients $\boldsymbol{\mathcal{B}}$. The posterior samples can then be used directly to impute unobserved response variables with given covariate values and to analyze the spatial and temporal patterns of the coefficients.

2.3. Interpolation

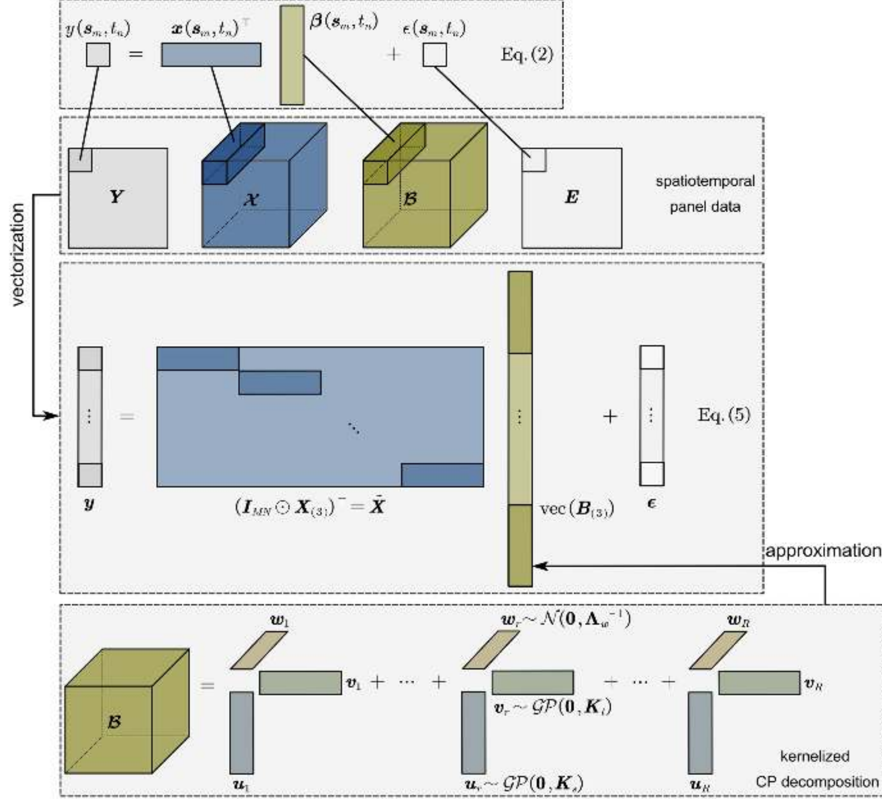


Figure 1: Illustration of the BKTR framework (Source: Figure 1 from Lei *et al.* (2023))

An important addition to the BKTR algorithm is the capacity to do interpolation. By interpolation, we mean the ability to estimate new regression coefficients $\boldsymbol{\mathcal{B}}^{\text{new}}$ and response values $\boldsymbol{Y}^{\text{new}}$ at unobserved time points and locations. In the literature, this process is often named Bayesian kriging. As a process, interpolation differs from imputation, which was already covered in BKTR. Imputation is used when parts of the response variables are missing at some of the M^{obs} locations or N^{obs} time points employed during regression. In contrast, interpolation is accomplished in a completely different step after MCMC sampling, and it is performed at M^{new} new locations and N^{new} time points. To perform interpolation, we need to estimate, for M^{new} unobserved locations and N^{new} unobserved time steps, the posterior distributions of the related coefficients $\boldsymbol{\mathcal{B}}^{\text{new}}$. The formal representation of the $\boldsymbol{\mathcal{B}}^{\text{new}}$ coefficients can be somewhat difficult to visualize, as it cannot be stored in a simple tensor format. Thus, we use a representation similar to the one presented by Takeuchi *et al.* (2017) and we illustrate it in Figure 2. Using this representation, the prediction results of $\boldsymbol{\mathcal{B}}^{\text{new}}$ can be represented by three tensors: $\boldsymbol{\mathcal{B}}^1$, the coefficients for new locations at observed time points, $\boldsymbol{\mathcal{B}}^2$, the coefficients for new time points at observed locations and $\boldsymbol{\mathcal{B}}^3$, the beta coefficients for new locations at new time points. We also include in Figure 2, the equivalent illustration for the newly provided covariates $\boldsymbol{\mathcal{X}}^{\text{new}}$ composed of $\boldsymbol{\mathcal{X}}^1$, $\boldsymbol{\mathcal{X}}^2$, $\boldsymbol{\mathcal{X}}^3$, on top of the related new response variable matrices $\boldsymbol{Y}^{\text{new}}$ composed of $\boldsymbol{Y}^1$, $\boldsymbol{Y}^2$ and $\boldsymbol{Y}^3$.

We estimate the distributions of $\boldsymbol{\mathcal{B}}^{\text{new}}$ via MCMC sampling with an approach similar to the one presented by Gamerman *et al.* (2008). The posterior coefficients $\boldsymbol{\mathcal{B}}^{\text{new}}$ are estimated

Algorithm 1 Simplified BKTR MCMC sampling process

Input: $\boldsymbol{y}_\Omega, \boldsymbol{\mathcal{X}}, R, K_1, K_2, k_s, k_t, \Phi, \Gamma, \tau_\phi, \tau_\gamma, a_0, b_0, \tau$

1: Intialize $\{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}\}$ as normally distributed random values
2: Set $\mu_{\phi_i} = \log(\phi_i)\ \forall \phi_i \in \Phi,\ \mu_{\gamma_i} = \log(\gamma_i)\ \forall \gamma_i \in \Gamma$
3: Sample $\Lambda_\omega \sim \mathcal{W}(\boldsymbol{I}_P, P)$
4: **for** $k = 1 : K_1 + K_2$ **do**
5: Sample kernel hyperparameters Φ, Γ
▷ from slice sampling given $\boldsymbol{V}, \boldsymbol{W}, y_\Omega, \boldsymbol{\mathcal{X}}, \Phi, \tau_{\phi_1}, \ldots, \tau_{\phi_J}, \Gamma, \tau_{\gamma_1} \ldots, \tau_{\gamma_L}$
6: Sample hyperparameters $\boldsymbol{\Lambda}_w$ ▷ from Wishart distribution given $\boldsymbol{W}$
7: Sample factor vec($\boldsymbol{U}$) ▷ from Normal distribution given $\boldsymbol{W}, \boldsymbol{V}, y_\Omega, \boldsymbol{\mathcal{X}}, \tau, k_s$
8: Sample factor vec($\boldsymbol{V}$) ▷ from Normal distribution given $\boldsymbol{W}, \boldsymbol{U}, y_\Omega, \boldsymbol{\mathcal{X}}, \tau, k_t$
9: Sample factor vec($\boldsymbol{W}$) ▷ from Normal distribution given $\boldsymbol{U}, \boldsymbol{V}, y_\Omega, \boldsymbol{\mathcal{X}}, \tau, \Lambda_\omega$
10: Sample precision τ ▷ from Gamma distribution given $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, y_\Omega, \boldsymbol{\mathcal{X}}, a_0, b_0$
11: **if** $k > K_1$ **then**
12: Collect the samples $\boldsymbol{U}^{(k-K_1)} = \boldsymbol{U}, \boldsymbol{V}^{(k-K_1)} = \boldsymbol{V}, \boldsymbol{W}^{(k-K_1)} = \boldsymbol{W}$;
13: Compute and collect $\boldsymbol{\mathcal{B}}^{(k-K_1)}$ given $\boldsymbol{\mathcal{B}} = \sum_{r=1}^{R} \boldsymbol{u}_r \circ \boldsymbol{v}_r \circ \boldsymbol{w}_r$
14: **end if**
15: **end for**
16: **return** $\{\boldsymbol{\mathcal{B}}^{(k)}\}_{k=1}^{K_2}$ to approximate posterior coefficients and estimate unobserved data.

Figure 2: Illustration of the data components $\boldsymbol{Y}$, $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{B}}$ for interpolation at M^{new} unobserved locations and N^{new} unobserved time steps.

by doing interpolation on the spatial decomposition $\boldsymbol{U}$ and the temporal decomposition $\boldsymbol{V}$ separately. We can formulate the joint multivariate normal distribution for the rth component of the spatial decomposition which includes the observed ($\boldsymbol{u}_r^{\text{obs}}$) and the new ($\boldsymbol{u}_r^{\text{new}}$) decomposition vectors as follows:

$$\begin{pmatrix} \boldsymbol{u}_r^{\text{obs}} \\ \boldsymbol{u}_r^{\text{new}} \end{pmatrix} \mid \Phi \Bigg) \sim \mathcal{N}\left[\begin{pmatrix} \boldsymbol{0}_{M^{\text{obs}}} \\ \boldsymbol{0}_{M^{\text{new}}} \end{pmatrix}, \begin{pmatrix} \boldsymbol{R}_\Phi^{\text{obs}} & \boldsymbol{R}_\Phi^{\text{obs,new}} \\ \boldsymbol{R}_\Phi^{\text{new,obs}} & \boldsymbol{R}_\Phi^{\text{new}} \end{pmatrix}\right], \tag{8}$$

where $\boldsymbol{R}_\Phi^{\text{obs}}$ is the covariance matrix corresponding to the observed locations and $\boldsymbol{R}_\Phi^{\text{new}}$ is the covariance matrix corresponding to the unobserved locations for a known set of spatial kernel hyperparameters Φ. The covariance matrix between the observed and unobserved locations is $\boldsymbol{R}_\Phi^{\text{obs,new}}$ and $\boldsymbol{R}_\Phi^{\text{new,obs}}$ is its transpose. Finally, $\boldsymbol{0}_M$ represents a vector of zeros of size M.

The same approach can be used to formulate the joint multivariate normal distribution for the temporal decomposition $\boldsymbol{V}$ using the temporal kernel hyperparameters Γ.

From Equation 8 (and its equivalent for temporal decomposition), we can find the conditional distribution of the decompositions at new locations and time points

$$\begin{aligned}\boldsymbol{u}_r^{\text{new}}|\boldsymbol{u}_r^{\text{obs}} &\sim \mathcal{N}(\boldsymbol{R}_\Phi^{\text{new,obs}}\boldsymbol{R}_\Phi^{\text{obs}^{-1}}\boldsymbol{u}_r^{\text{obs}},\ \boldsymbol{R}_\Phi^{\text{new}} - \boldsymbol{R}_\Phi^{\text{new,obs}}\boldsymbol{R}_\Phi^{\text{obs}^{-1}}\boldsymbol{R}_\Phi^{\text{obs,new}}),\ r = 1,\ldots,R,\\ \boldsymbol{v}_r^{\text{new}}|\boldsymbol{v}_r^{\text{obs}} &\sim \mathcal{N}(\boldsymbol{R}_\Gamma^{\text{new,obs}}\boldsymbol{R}_\Gamma^{\text{obs}^{-1}}\boldsymbol{v}_r^{\text{obs}},\ \boldsymbol{R}_\Gamma^{\text{new}} - \boldsymbol{R}_\Gamma^{\text{new,obs}}\boldsymbol{R}_\Gamma^{\text{obs}^{-1}}\boldsymbol{R}_\Gamma^{\text{obs,new}}),\ r = 1,\ldots,R.\end{aligned} \tag{9}$$

We can obtain the parameters in Equation 9 using posterior samples captured during the K_2 sampling iterations of the MCMC sampling described in Algorithm 1. To estimate the distribution of $\boldsymbol{u}_r^{\text{new}}$ and $\boldsymbol{v}_r^{\text{new}}$, we can use the accumulated estimated values of all spatial and temporal kernel hyperparameters Φ and Γ to evaluate, at each sampling iteration, the covariance matrices $\boldsymbol{R}_\Phi^{\text{obs}}$, $\boldsymbol{R}_\Phi^{\text{new}}$, $\boldsymbol{R}_\Phi^{\text{obs,new}}$, $\boldsymbol{R}_\Gamma^{\text{obs}}$, $\boldsymbol{R}_\Gamma^{\text{new}}$ and $\boldsymbol{R}_\Gamma^{\text{obs,new}}$. From the posterior distributions, we are able to obtain a spatial decomposition sample at new locations $\boldsymbol{u}_r^{\text{new}}$ and a temporal decomposition sample at new time points $\boldsymbol{v}_r^{\text{new}}$ for each of the K_2 sampling iterations. From the $\boldsymbol{u}_r^{\text{new}}$, $\boldsymbol{v}_r^{\text{new}}$ samples and their corresponding $\boldsymbol{w}_r$ at each sampling iteration, we are able to approximate samples of $\boldsymbol{\mathcal{B}}^{\text{new}}$. Lastly, by combining the K_2 sampled $\boldsymbol{\mathcal{B}}^{\text{new}}$ values, we can obtain an estimate of its distribution. After this sampling process, it is fairly simple, following Equation 2, to get the expectation of the response variable $E(\boldsymbol{Y}^{\text{new}}|\boldsymbol{Y}^{\text{obs}})$ using $\boldsymbol{\mathcal{B}}^{\text{new}}$ and the corresponding covariates $\boldsymbol{\mathcal{X}}^{\text{new}}$.

An important aspect of using kriging for predictions is that it is mainly effective when new locations and time points are close to observed records. Thus, BKTR predictions made in relatively close spatial and temporal neighborhoods should be very effective. However, in tasks such as predicting temporal values located very far in the future, the current prediction methodology might yield poor results as kriging relies on nearby observations to make predictions.

This interpolation sampling process is implemented in both **BKTR** packages. The performance and results of this implementation are reviewed extensively on simulated data in Section 5.3 and on real-world data in Section 6.3.

3. BKTR regressor class

The **BKTR** packages provide a `BKTRRegressor` class that encapsulates the core concepts and functionalities of the BKTR algorithm. This class provides a simple interface to fit the BKTR model for a given dataset and makes it possible to visualize fitted coefficients as well as to predict values for new or missing observations. Even though the `BKTRRegressor` class has been designed to be user-friendly, its flexibility and number of parameters need to be carefully considered. Thus, this section is dedicated to explaining the different data inputs, parameter inputs and all the attributes and methods of the `BKTRRegressor` class.

3.1. Input data

For the BKTR algorithm, three data frame inputs need to be provided during the initialization of a `BKTRRegressor` object.

- A data frame `spatial_positions_df` with M rows and $1 + d_s$ columns containing information about the spatial locations. The first column is used to label each spatial

location and the other d_s columns encapsulate the spatial position of each location in d_s dimensions. For example, when we consider a location to be represented by longitude and latitude, we would have $d_s = 2$ and a three-column dataframe. For consistency, the first column containing location labels should contain only unique values and needs to be named `location`.

- A data frame `temporal_positions_df` with N rows and $1 + d_t$ columns containing information about each discrete timestamp. The first column labels each time point and the subsequent d_t columns are used to capture the timestamp's temporal position. Typically, as when using dates as time points, $d_t = 1$ can be used, resulting in a two-columns dataframe. For consistency, the first column containing time point labels should contain only unique value and needs to be named `time`.
- A principal data frame `data_df` with MN rows and $3+P$ columns, containing a location label column, a time point label column, a column containing the response variable and P columns for the covariates. The first column of the data frame needs to be named `location`; it must contain the same values as the `spatial_positions_df`'s `location` column and each value must appear N times. Similarly, the second column of the data frame needs to be named `time`; it must contain the same values as the `temporal_positions_df`'s `time` column and each value must appear M times. In other words, the `data_df` data frame's location and time columns must contain all possible combinations of `spatial_positions_df` locations and `temporal_positions_df` time points. In general, it is preferable but not mandatory that the third column of the data frame contains the response variable data.

It is important to note that `data_df` can also contain missing values in the response variable y column. Those missing values must be properly encoded as `NaN` and represent, in fact, a flattened version of the matrix Ω in Equation 4.

To give us a better idea of what would be a valid shape for `BKTRRegressor` input data, let's take a look at the BIXI data presented in Section 6. To keep the visualization succinct, we will take a subset of two locations ($M = 2$), three time points ($N = 3$) and two covariates ($P = 2$). We can start by looking at a valid `spatial_positions_df`:

```
R> bixi_data <- BixiData$new()
R> ex_locs <- c('7114 - Smith / Peel', '6435 - Victoria Hall')
R> ex_times <- c('2019-04-17', '2019-04-18', '2019-04-19')
R> print(bixi_data\$spatial_positions_df[location %in% ex_locs])

               location latitude longitude
1: 6435 - Victoria Hall 45.48129 -73.60033
2:  7114 - Smith / Peel 45.49284 -73.55642
```

Then look at a valid `temporal_positions_df`:

```
R> print(bixi_data$temporal_positions_df[time %in% ex_times])

         time time_index
1: 2019-04-17          2
2: 2019-04-18          3
```

And lastly, look at the corresponding valid `data_df`:

```
R> print(bixi_data$data_df[
R>   location %in% ex_locs & time %in% ex_times,
R>   c(1:4, 17)
R> ])

                  location       time nb_departure area_park  humidity
1: 6435 - Victoria Hall 2019-04-17    0.2178218 0.2254071 0.1919343
2: 6435 - Victoria Hall 2019-04-18    0.3366337 0.2254071 0.4697535
3: 6435 - Victoria Hall 2019-04-19    0.2178218 0.2254071 0.9350261
4:  7114 - Smith / Peel 2019-04-17    0.7821782 0.0652808 0.1919343
5:  7114 - Smith / Peel 2019-04-18    0.3861386 0.0652808 0.4697535
6:  7114 - Smith / Peel 2019-04-19    0.6534653 0.0652808 0.9350261
```

Another important data-related input for the `BKTRRegressor` is the `formula`. By default, if no `formula` is provided, the third column of `data_df` will be used as the response variable $\boldsymbol{y}$ and the remaining columns will be used as covariates $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_P$. If a `formula` is provided, the model $\boldsymbol{y}$ and $\boldsymbol{X}$ will be extracted from the `data_df` and the `formula`. The `formula` must be in the form of `y ~ x1 + x2 + . .. + xP` and corresponds to a valid R formula (Wilkinson and Rogers 1973). All the terms in the `formula` must correspond to valid column names of the `data_df` data frame. For R users, the `formula` must be a `formula` object. For Python users, the `formula` must be a string that can be parsed by the **Formulaic** library. In both cases, it is important to note that by default, an intercept term is automatically added to the model matrix $\boldsymbol{X}$ and can be removed by adding a `-1` term to the `formula`.

The covariates $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_P$ can describe the spatial and temporal attributes of the observations. By nature, the **BKTR** packages are designed to be able to consider spatial attributes that vary through time (e.g. population density that varies over time) or temporal attributes that vary through space (e.g., temperature that varies through different countries). However, it is quite common to have spatial attributes that are constant through time and temporal attributes that are constant through space like in the BIXI example of Section 6. When this is the case and the data is provided in a compressed manner, we provide a `reshape_covariate_dfs` utility function (see Appendix B) that can help users to obtain a valid `data_df`.

3.2. Input parameters

In addition to data frames, the user must provide sampling parameters in order to initialize a regressor. These parameters include `burn_in_iter` and `sampling_iter`, which are integer inputs representing the number of iterations for the burn-in phase (K_1) and the sampling phase (K_2), respectively. There is also `rank_decomp`, which is an integer input representing the rank of the CP decomposition (R). The default values are 500 iterations for K_1 and K_2, and 10 for the rank decomposition. Two different `Kernel` objects can also be provided (these are further described in Section 4.2), one for the spatial kernel `spatial_kernel` and one for the temporal kernel `temporal_kernel`. The user can also provide values for hyperparameters such as `sigma_r`, which is a float representing the variance of the white noise process (τ^{-1}), and `a_0` and `b_0`, representing the initial values for the shape (α) and rate (β) of the gamma function generating τ. To ensure weak priors for `a_0`, `b_0` and `sigma_r`, we provide default

values of `a_0=1E-6`, `b_0=1E-6`, `sigma_r=1E-2` respectively. The spatial kernel used by default is a Matérn kernel 5/2 with a smoothness factor $\nu = 5$ (see Table 4), while the default temporal kernel is a Squared Exponential (SE) kernel. The `BKTRRegressor` class can also take a boolean value input for the `has_geo_coords` parameter. This parameter, which has a default truthful value, indicates whether we need to apply a Mercator projection (Snyder 1987) to the spatial locations. We usually need to apply a projection when the spatial positions are encoded in longitude and latitude so that we keep the kernel valid. The `geo_coords_scale` parameter is directly associated to the Mercator projection and dictates the scale at which it should be projected. More information about the projection is available in Appendix E.

When using all the default parameters mentioned above, it is very simple to run a BKTR regression on a given dataset. Here is a very simple example of the usage of the `BKTRRegressor` on the full version of the BIXI data mentioned in Section 3.1:

```
R> bktr_regressor <- BKTRRegressor$new(
+     data_df=bixi_data$data_df,
+     spatial_positions_df=bixi_data$spatial_positions_df,
+     temporal_positions_df=bixi_data$temporal_positions_df)
```

Once the regressor is initialized, the user can launch the MCMC sampling process by calling the `mcmc_sampling` method, which stores its results inside the `BKTRRegressor` object. Depending on the number of iterations and the size of the data, this process can take a long time to complete. Thus, we added a progress log showing the current MCMC iteration number, the elapsed time and the current error values. The displayed in-sample errors are the mean absolute error (MAE) and the root mean square error (RMSE) of the response variable. The following code chunk shows the usage of the `mcmc_sampling` method, but with a truncated output for the sake of brevity.

```
R>  bktr_regressor$mcmc_sampling()

* Iter 1     | Elapsed    0.48s | MAE  0.1036 | RMSE  0.1465 *
...
* Iter 1500  | Elapsed    2.83s | MAE  0.0543 | RMSE  0.0734 *
* Iter TOTAL | Elapsed 4528.07s | MAE  0.0525 | RMSE  0.0714 *
```

3.3. Attributes and visualizations

Once a `BKTRRegressor` object has been initialized, the user can access its attributes and methods shown in Table 2. Note that except for the `mcmc_sampling` method, all the other methods and attributes are available only after the MCMC sampling process is completed. When called on a `BKTRRegressor` object, the built-in `summary` and `print` R functions will simply display the `summary` attribute.

The `predict` method is used to estimate the response variable $\boldsymbol{Y}$ and the $\boldsymbol{\mathcal{B}}$ coefficients for new locations and/or time points, which we referred to as interpolation in Section 2.3. The method takes as input a data frame containing the covariates for the new locations and/or time points `new_data_df`, a data frame containing the spatial positions of the new locations `new_spatial_positions_df` and lastly a data frame containing the temporal positions of the new time points `new_temporal_positions_df`. It is possible to provide either

BKTRRegressor Methods	
mcmc_sampling	Launches the MCMC sampling process for a predefined number of iterations and a given set of parameters
predict	Used to estimate the response variable $\boldsymbol{y}$ and the $\boldsymbol{\mathcal{B}}$ coefficients for new locations or time points
get_beta_summary_df	Gets a summary of estimated beta values (mean, stdev, quantiles). Labels can be provided for spatial locations, time points and features. When no labels are given for a dimension, all its betas are shown.
BKTRRegressor Attributes	
summary	Summary of the MCMC regressor object
beta_covariates_summary_df	Data frame summarizing beta covariates per feature (mean, stdev, quantiles)
y_estimates	Data frame for the estimated target variable
imputed_y_estimates	Data frame of the estimated and imputed target variable including missing data (Ω)
beta_estimates	Data frame estimated values for betas
hyperparameters_per_iter_df	Data frame of estimated MCMC hyperparameters (kernels' and τ) values per iteration

Table 2: BKTRRegressor attributes and methods

a new_spatial_positions_df or a new_temporal_positions_df or both. In R, it can also be called via the predict built-in function using the BKTRRegressor object as first argument followed by all the other arguments in the same order as described above. Extensive usage examples of the predict method are provided in Section 5.3 and Section 6.3.

The get_beta_summary_df method is used to get a summary of the estimated $\boldsymbol{\mathcal{B}}$ coefficients. It can take as inputs a list of labels for the spatial locations, a list of time point labels and a list of feature labels. When no labels are given for one of the input, all the $\boldsymbol{\mathcal{B}}$ coefficients for the corresponding dimension are shown. The method returns a data frame containing the mean, standard deviation, quantiles for each of the queried $\boldsymbol{\mathcal{B}}$ coefficient for the sampled posterior distribution. A usage example of the get_beta_summary_df method is provided in Section 6.1.

Multiple visualization functions are also available in the **BKTR** packages. Those visualization functions are shown in Table 3. All functions take a BKTRRegressor object as a first input for which the sampling process must be completed. We can also specify, via the show_figure boolean parameter, to either return a plot object (ggplot2 in R and plotly in Python) or to just display the plot. The plot object can then be used to customize the plot further if needed or to save it as an image. Also, all visualization functions can use the width or the height parameters to customize the size of the plotting area.

Most of the attributes, methods and visualization functions described in this section are further explored and tested in the Section 5 and Section 6.

4. BKTR Kernels

BKTRRegressor Plot Functions	
`plot_temporal_betas`	Plot beta values through time for a given spatial point and a set of feature labels.
`plot_spatial_betas`	Plot beta values through space for a given time point and a set of feature labels.
`plot_covariates_beta_dists`	Plot the distribution of beta estimates grouped by features. A subset of features can be provided to plot only a subset of them.
`plot_hyperparams_dists`	Plot the distribution of τ and kernels' hyperparameters for all sampling iterations. `hyperparameters` allows to plot only a subset of parameters.
`plot_hyperparams_per_iter`	Plot the values of τ and kernels' hyperparameters through sampling iterations (trace plot). The argument `hyperparameters` allows to plot only a subset of parameters.
`plot_y_estimates`	Plot the estimated values for the response variable $\hat{\boldsymbol{y}}_\Omega$ alongside their corresponding observed values $\boldsymbol{y}_\Omega$.

Table 3: Plot functions that can be used on a `BKTRRegressor` object after MCMC sampling

Kernels play a crucial role in Gaussian processes. They are used to model the similarity or dissimilarity between observations. This similarity can then serve to generate a covariance matrix that can be used as a prior distribution for given parameters. The choice of a kernel is very important as it influences properties of the Gaussian process and makes hypotheses about the underlying structure of the function being modelled. For example, a squared exponential kernel (SE) is highly smooth, often referred to as the Gaussian Kernel. Consequently, it assigns a high covariance to input points that are defined as close and low covariance to points that are distant. In contrast, a periodic kernel assigns high covariance to points that are separated by a multiple of a given period, resulting in a GP that can capture periodic patterns in the data (Duvenaud 2014). Thus, the pattern difference between the SE and periodic kernel highlights the importance of choosing a sensible kernel when modelling a function with a GP.

The `spatial_kernel` and `temporal_kernel` objects can come from any kernel class implemented in the `kernels` module of **BKTR** (see Section 4.2). We can say that **BKTR** kernel classes themselves need two main components to be defined: the kernel parameters (see Section 4.1) and the kernel function (explained in Section 4.2). The kernels implemented in **BKTR** take inspiration from the kernels existing in the **GPyTorch** package (Gardner *et al.* 2018) and the **Scikit-learn** package (Pedregosa *et al.* 2011).

4.1. Kernel Parameters

All kernels implemented in **BKTR** have a set of `KernelParameter` parameters that can be optimized during the training process of the MCMC sampling. This set of parameters can be accessed via the `parameters` attribute of a kernel after its initialization. Kernels in general contain sensible default parameters. However, the user can change the behaviour of a given `Kernel`'s parameter by providing a `KernelParameter` object to the related parameter

argument.

The `KernelParameter` class can take multiple arguments at initialization:

- `value`: an initial value used for the parameter during slice sampling or the constant value used for the parameter when it is not optimized
- `is_fixed`: a Boolean argument indicating whether the hyperparameter value is fixed or optimized during sampling
- `lower_bound`: a value that indicates the minimum value that can be taken by the parameter during sampling
- `upper_bound`: a value that indicates the maximum value that can be taken by the parameter during sampling
- `slice_sampling_scale`: a value indicating the scale parameter for the slice sampling algorithm
- `hparam_precision`: a value indicating the precision of the hyperparameter

The `value` argument is the only argument needed to initialize `KernelParameter`. The other arguments have default values if not provided. The `is_fixed` argument is set to false by default, which means that the parameter will necessarily be optimized during the training process. By default, the `lower_bound` and `upper_bound` arguments are set to `1E-03` and `1E+03`, respectively, which means that the parameter will be sampled in a range of $[10^{-3}, 10^{3}]$ during the training process. The `slice_sampling_scale` argument is set to `2` by default, which represents the slice sampling step size during the MCMC sampling process. Finally, the `hparam_precision` argument is set to `1E-3` by default.

4.2. Kernel Classes

BKTR `Kernel` classes are core components encapsulating behaviours and attributes used to compute the covariance matrix during Gaussian processes. They contain information about observations' positions, all the kernel's hyperparameters and a kernel function.

For a kernel to induce a covariance matrix, it needs an initial input encoding the positions of the observations. This position vector is set for a `Kernel` object using the `set_positions_df` method. This method accepts a data frame with a number of rows equals to the number of observations and number of column equals to $1 + K$, where the first column is for labelling each observation and the other columns contain the location information for the K dimensions of each observation.

When using stationary kernels, we can initially calculate a distance between observations $d(x, x')$ and use it to generate the covariance matrix. To ensure the validity of covariance matrices, in **BKTR**, we use the Euclidean distance function:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{K} (x_{ik} - x_{jk})^2}, \tag{10}$$

where x_i and x_j are two given observations' positions from the position vector $\boldsymbol{x}$ and K is the number of dimensions of a given position.

Kernel Class	Optimizable Parameters	Kernel Function
`KernelWhiteNoise` White noise		$k(x, x') = I_{\lvert x \rvert}$, where $\lvert x \rvert$ is the the number of elements in the position vector x.
`KernelSE` Squared exponential	`lengthscale` (ℓ)	$k(x, x'; \ell) = \exp\Big(-\frac{d(x,x')^2}{2\ell^2}\Big)$
`KernelRQ` Rational quadratic	`alpha` (α) `lengthscale` (ℓ)	$k(x, x'; \alpha, \ell) = \Big(1 + \frac{d(x,x')^2}{2\alpha\ell^2}\Big)^{-\alpha}$
`KernelPeriodic` Periodic	`lengthscale` (ℓ) `period_length` (t)	$k(x, x'; \ell, t) = \exp\Big(-\frac{2\sin^2(\frac{\pi d(x,x')}{t})}{\ell^2}\Big)$
`KernelMatern` Matérn	`lengthscale` (ℓ)	$k(x, x'; \ell, \nu) = \begin{cases} \exp(-\frac{D}{\ell}), & \text{if } \nu = 1 \\ (1 + \frac{\sqrt{3}D}{\ell})\exp(-\frac{D}{\ell}), & \text{if } \nu = 3 \\ (1 + \frac{\sqrt{5}D}{\ell} + \frac{5D^2}{3\ell^2})\exp(-\frac{D}{\ell}), & \text{if } \nu = 5 \end{cases}$, where $D = d(x, x')$ and ν is a Matérn kernel input called the `smoothness_factor`. The `smoothness_factor` input can be either `1`, `3` or `5`, which correspond to so called Matérn $\frac{1}{2}$, $\frac{3}{2}$ and $\frac{5}{2}$ kernels respectively.
`KernelAddComposed` Composed via addition		$k(x, x'; \Lambda_a, \Lambda_b) = k_a(x, x'; \Lambda_a) + k_b(x, x'; \Lambda_b)$, where k_a and k_b are two kernel functions and Λ_a and Λ_b their sets of parameters.
`KernelMulComposed` Composed via multiplication		$k(x, x'; \Lambda_a, \Lambda_b) = k_a(x, x'; \Lambda_a) * k_b(x, x'; \Lambda_b)$, where k_a and k_b are two kernel functions and Λ_a and Λ_b their sets of parameters.

Table 4: List of kernel classes implemented in the **BKTR** packages and their respective parameters and equations. The $d(x, x')$ function is the Euclidean distance function applied on the observations' positions. Note that only the parameters that are objects coming from the `KernelParameter` class (parameters that can be sampled) are listed in the Optimizable Parameters column.

With this position information, we can use all the different kernel classes available in the **BKTR** package, which are enumerated in Table 4. Different kernels yield completely different covariance matrices according to their function and parameters. For instance, the `KernelSE` kernel yields a covariance matrix with a smooth decreasing function from the diagonal, whereas the `KernelPeriodic` kernel yields a covariance matrix with a periodic pattern

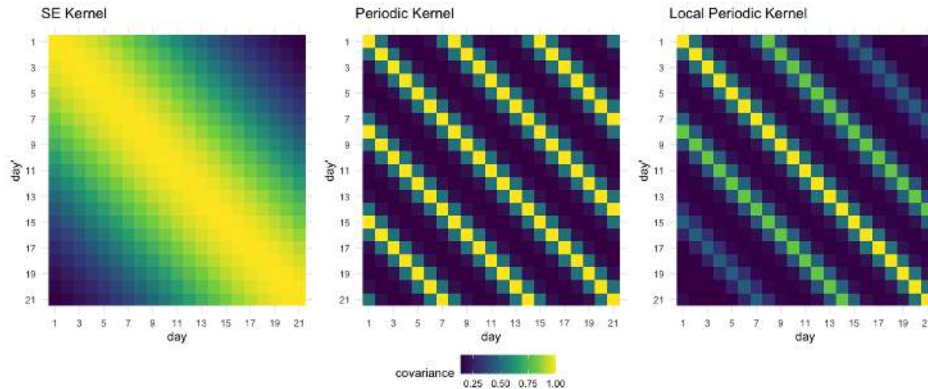


Figure 3: Heatmap plots of the covariance matrix for three different kernels implemented in the **BKTR** package calculated on 21 consecutive days. Presented kernels are, in order, a `KernelSE` with a `lengthscale` of 10, a `KernelPeriodic` with a `period_length` of 7 and a local periodic kernel (`KernelMulComposed`) resulting from the multiplication of two other kernels. The local periodic kernel plot shows that it contains the periodicity of the `KernelPeriodic` kernel and the decay of the `KernelSE` kernel.

from the diagonal. Those different covariance matrices can be visualized using the `plot` method of any implemented `Kernel` class and the two aforementioned kernel examples are shown in Figure 3 (first two subfigures).

As in the case of the local periodic kernel, it is sometimes useful to combine different kernels to obtain a composition of different covariance matrices. This feature is implemented in the **BKTR** package with the `KernelAddComposed` class when adding two kernels and `KernelMulComposed` class when multiplying two kernels. We can look at an example of a composed kernel in **BKTR** by creating a local periodic kernel when multiplying a `KernelSE` kernel with a `KernelPeriodic` kernel. The `KernelSE` kernel is used to model the smooth decreasing function from the diagonal and the `KernelPeriodic` kernel is used to model the periodic function from the diagonal. In the BIXI example used by Lei *et al.* (2023), a local periodic kernel was used to model the BIXI stations' demand with a constant period of seven days. We can easily create and visualize this local periodic kernel using the `KernelComposed` class as shown in the following code snippet:

```
R> library(data.table)
R> days_df <- data.table(day=1:21, position=1:21)
R> se_lengthscale <- KernelParameter$new(value=10)
R> per_length <- KernelParameter$new(value=7, is_fixed=TRUE)
R> kernel_periodic <- KernelPeriodic$new(period_length=per_length)
R> kernel_se <- KernelSE$new(lengthscale=se_lengthscale)
R> kernel_local_periodic <- kernel_periodic * kernel_se
R> kernel_local_periodic$set_positions(days_df)
R> kernel_local_periodic$kernel_gen()
R> kernel_local_periodic$plot()
```

The resulting kernel plot is illustrated as the rightmost plot of Figure 3. The `lengthscale` parameter of the SE kernel was set to a value of 10 in this example, and the `period_length`

of the periodic kernel was set to 7 to enhance the visualization of kernel properties.

5. Simulation-based study

To showcase the capabilities of the **BKTR** package, we will first generate simulated datasets with known ground truths and then use our software library to estimate the underlying parameters. This aims to demonstrate the package's capabilities and to illustrate the different insights that can be obtained from the results of **BKTR**. Subsequently, we will show the imputation and interpolation abilities of the **BKTR** packages on the aforementioned simulated dataset. The imputation and interpolation results will also be compared with the results of the Python library implementation.

To simulate data, we use the `simulate_spatiotemporal_data` utility function implemented in the **BKTR** packages to simulate a spatiotemporal dataset with four different data frames: spatiotemporal locations, time points, covariates (including the response variable and an intercept term) and beta coefficients. Details regarding the simulation process and the implementation of the function `simulate_spatiotemporal_data` are given in Appendix C.

In the following, we use two different types of simulated datasets. The former, which we will call the *Smaller* dataset, will hold a $\boldsymbol{\mathcal{B}}$ tensor with 2,400 values having $M = 20$ spatial locations, $N = 30$ time points, two spatial covariates with means $\boldsymbol{\mu}_s = [0, 2]$ and one temporal covariate with mean $\boldsymbol{\mu}_t = [1]$. The other type of simulated dataset, which we will call the *Larger* one, will use a $\boldsymbol{\mathcal{B}}$ with 90,000 values having $M = 100$ spatial locations, $N = 150$ time points, three spatial covariates with means $\boldsymbol{\mu}_s = [0, 2, 4]$ and two temporal covariates with means $\boldsymbol{\mu}_t = [1, 3]$. Both datasets have a noise scale of $\sigma_\epsilon^2 = 1$, a spatial scale of $S_s = 10$, a temporal scale of $S_t = 10$ and spatial data in $d = 2$ dimensions. Spatial and temporal kernel functions used to obtain covariance matrices will vary on a case by case basis, depending on subsections.

5.1. Estimation of the parameters

We start by simulating a *Larger* dataset described above using a spatial Matérn 5/2 Kernel with a lengthscale parameter value $\phi_1^{\text{sim}} = 14$ and a temporal squared exponenential kernel with a lengthscale parameter $\gamma_1^{\text{sim}} = 5$.

```
R> TSR$set_params(seed = 1)
R> matern_lengthscale <- KernelParameter$new(value = 14)
R> se_lengthscale <- KernelParameter$new(value = 5)
R> spatial_kernel <- KernelMatern$new(lengthscale = matern_lengthscale)
R> temporal_kernel <- KernelSE$new(lengthscale = se_lengthscale)
R>
R> # Simulate data
R> simu_data <- simulate_spatiotemporal_data(
+    nb_locations=100,
+    nb_time_points=150,
+    nb_spatial_dimensions=2,
+    spatial_scale=10,
+    time_scale=10,
```

```
+    spatial_covariates_means=c(0, 2, 4),
+    temporal_covariates_means=c(1, 3),
+    spatial_kernel=spatial_kernel,
+    temporal_kernel=temporal_kernel,
+    noise_variance_scale=1)
```

Once the data is simulated, we can fit a BKTR model to it.

```
R> bktr_regressor <- BKTRRegressor$new(
+    data_df = simu_data$data_df,
+    spatial_kernel = KernelMatern$new(),
+    spatial_positions_df = simu_data$spatial_positions_df,
+    temporal_kernel = KernelSE$new(),
+    temporal_positions_df = simu_data$temporal_positions_df,
+    has_geo_coords=FALSE)
R> bktr_regressor$mcmc_sampling()

[1] "Iter 1     | Elapsed Time:     0.60s | MAE:  2.9356 | RMSE:  3.7555"
...
[1] "Iter 1000  | Elapsed Time:     0.38s | MAE:  0.8038 | RMSE:  1.0084"
[1] "Iter TOTAL | Elapsed Time:   658.22s | MAE:  0.7974 | RMSE:  1.0007"
```

The `mcmc_sampling` method prints the results for each iteration; we then truncated the output for brevity. We can print the summary of the BKTR model to get a brief overview of its parameters and to assess quality of the fit.

```
>  summary(bktr_regressor)

======================================================================
                        BKTR Regressor Summary
======================================================================
Formula: y ~ .

Burn-in iterations: 500
Sampling iterations: 500
Rank decomposition: 10
Nb Spatial Locations: 100
Nb Temporal Points: 150
Nb Covariates: 6
======================================================================
In Sample Errors:
  RMSE: 1.001
  MAE: 0.797
Computation time: 658.22s.
======================================================================
-- Spatial Kernel --
Matern 5/2 Kernel
```

```
Parameter(s):
                      Mean   Median       SD  Low2.5p  Up97.5p
lengthscale         13.878   13.904    0.584   12.776   14.964

-- Temporal Kernel --
SE Kernel
Parameter(s):
                      Mean   Median       SD  Low2.5p  Up97.5p
lengthscale          4.603    4.616    0.246    4.038    5.054
======================================================================
Beta Estimates Summary (Aggregated Per Covariates)

                      Mean   Median       SD
Intercept            2.415    2.626    1.695
s_cov_0              2.452    2.638    1.411
s_cov_1             -4.224   -4.171    2.770
s_cov_2             -1.938   -1.952    0.467
t_cov_0             -2.372   -2.613    1.495
t_cov_1              0.071    0.160    0.797
======================================================================
```

The model was able to recover the kernel parameters used to simulate the data. The estimated lengthscale ϕ_1 for the spatial kernel is 13.878 while the underlying value was $\phi_1^{\text{sim}} = 14$. The estimated lengthscale γ_1 for the temporal kernel is 4.603, while the true value was $\gamma_1^{\text{sim}} = 5$. The estimated noise variance is $1.001^2 = 1.002$ while the true value was 1. We can also compare the estimated $\boldsymbol{\mathcal{B}}$ coefficients with their simulated values.

```
R> beta_err <- unlist(abs(
+    bktr_regressor$beta_estimates[, -c(1, 2)]
+    - simu_data$beta_df[, -c(1, 2)]))
R> print(sprintf('Beta RMSE: %.4f', sqrt(mean(beta_err^2))))
R> print(sprintf('Beta MAE: %.4f', mean(abs(beta_err))))
```

```
[1] "Beta RMSE: 0.1426"
[1] "Beta MAE: 0.0840"
```

It is possible to observe that the model was able to find $\boldsymbol{\mathcal{B}}$ coefficients close to the ones used to simulate the data, with a $\text{MAE}_{\boldsymbol{\mathcal{B}}}$ of 0.0840 and a $\text{RMSE}_{\boldsymbol{\mathcal{B}}}$ of 0.1426.

Using the `plot_hyperparams_traceplot` function, we can plot the estimated hyperparameters values (kernels' and τ) through iterations to see how they evolved during the MCMC sampling. This function will plot a subset of the aforementioned hyperparameters if a vector of hyperparameter names is provided as the `hyperparameters` argument. In our case, we were interested in plotting the evolution of all kernel hyperparameters, which we can do using the following code:

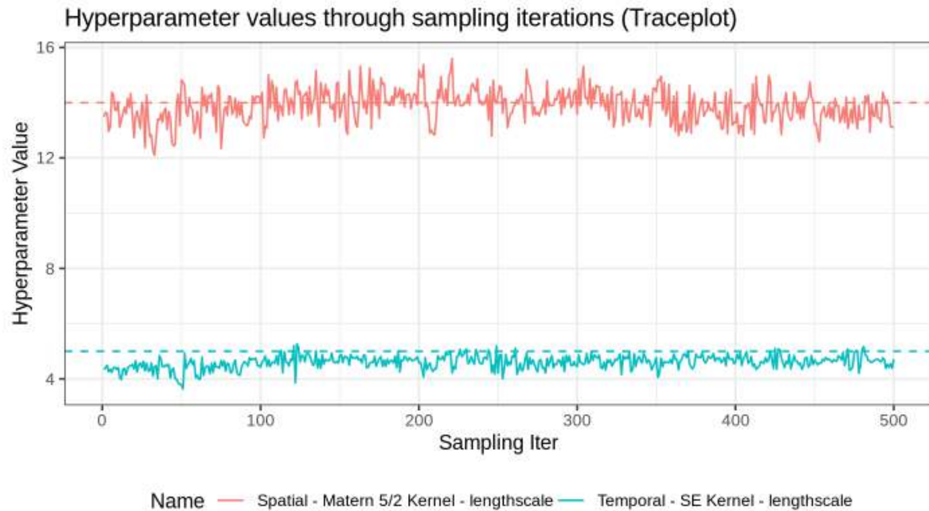


Figure 4: Traceplot of the hyperparameters through sampling iterations for a simulated dataset with 100 spatial locations, 150 temporal points, 6 covariates using a rank decomposition of 10, 500 burn-in iterations and 500 sampling iterations. The underlying values of the hyperparameters that were used for simulation are traced with dashed lines.

```
R> fig <- plot_hyperparams_traceplot(bktr_regressor, c(
+    'Spatial - Matern 5/2 Kernel - lengthscale',
+    'Temporal - SE Kernel - lengthscale'
+    ), show_figure = FALSE)

R> col_1 <- '#f87d76'; col_2 <- '#00bfc4';
R> fig +
+    scale_colour_manual(name = 'Name', values = c(col_1, col_2)) +
+    geom_hline(yintercept = matern_lengthscale$value,
+      linetype='dashed', col = col_1) +
+    geom_hline(yintercept = se_lengthscale$value,
+      linetype = 'dashed', col = col_2)
```

The plot resulting from the `plot_hyperparams_traceplot` function is shown in Figure 4. In the figure, we observe that during the sampling iterations, the posterior distribution of the hyperparameters was hovering around the true values used to simulate the data, which were $\phi_1^{\text{sim}} = 14$ for the spatial kernel's lengthscale and $\gamma_1^{\text{sim}} = 5$ for the temporal kernel's lengthscale. This is a good indication of the strength of the BKTR model, which is able to recover the underlying hyperparameters used to simulate the data. It is also worth noting that it would have been possible to plot the posterior distribution of the hyperparameters using the `plot_hyperparams_dists` function with the same parameters.

To visualize the proximity of the fitted model's response variable values $\hat{\boldsymbol{y}}_\Omega$ to the observed reponse variable $\boldsymbol{y}_\Omega$, we can use the `plot_y_estimates` function of BKTR.

```
R> plot_y_estimates(bktr_regressor, fig_title = NULL)
```

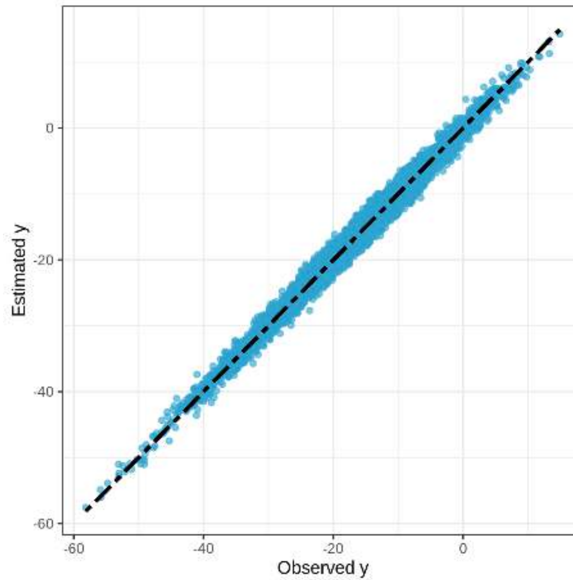


Figure 5: Scatter plot of estimated response variable vs. observed values in a simulated dataset with 100 spatial locations, 150 time points, 6 covariates using a rank decomposition of 10, 500 burn-in iterations and 500 sampling iterations.

When modelling the response variable effectively, the `plot_y_estimates` function should show points that closely align with a diagonal reference line, representing an ideal prediction. In Figure 5, the output of the function applied to the simulated data reveals estimated values that closely match the true $\boldsymbol{y}_\Omega$ values.

5.2. Imputation

In this section, we will examine the effectiveness of the **BKTR** packages at imputation. By imputation, we mean being able to find the best estimate for response values that were missing from the dataset. For our testing purposes, we again simulate *Larger* datasets. Then, from these datasets, we remove at random a certain percentage of response variable values and replace them with `NaN`. We use three scenarios of missing value rates: 10%, 50% and 90%.

In Section 5.1 we obtained results that were extremely close to the ground truth. This can be seen through the fact that the $\text{RMSE}_{\boldsymbol{Y}}$ stayed at around 1, which was the value of the noise scale S_ϵ. One factor that could have helped **BKTR** to capture the underlying structure of the data so well might have been that the kernels used during simulation were very smooth. These kernels resulted in spatial and temporal covariance matrices with high values, making the synthetic data highly correlated in space and time. Thus, to verify the effect of the kernel parameters on the beta convergence and the imputation method, we will test the imputation implementation on two different lengthscale value scenarios. The first scenario will use lengthscale values of 3 for both kernels, $\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 3$, and the second scenario will use a value of $\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 6$. Both of these scenarios are coupled with each missing percentage scenario, analyzing a total of 6 different settings. For all different settings, we simulate 10 new datasets, on which we fit $K_1 = 500$ burn-in iterations, $K_2 = 500$ sampling iterations and a rank decomposition $R = 10$.

Simulation Settings	Missing	Lang.	Performance Metrics	
			$MAE_{\mathcal{B}}/RMSE_{\mathcal{B}}$	$MAE_{\boldsymbol{Y}}/RMSE_{\boldsymbol{Y}}$
$\phi_1^{sim} = 3$ $\gamma_1^{sim} = 3$	10%	R	0.76±0.21/1.21±0.31	0.87±0.02/1.10±0.03
		Python	0.83±0.33/1.38±0.66	0.87±0.02/1.10±0.03
	50%	R	0.67±0.14/1.04±0.22	0.93±0.03/1.17±0.0
		Python	0.72±0.11/1.17±0.17	0.91±0.02/1.16±0.04
	90%	R	1.02±0.20/1.50±0.33	2.47±0.39/3.54±0.61
		Python	0.95±0.08/1.43±0.14	2.27±0.26/3.31±0.52
$\phi_1^{sim} = 6$ $\gamma_1^{sim} = 6$	10%	R	0.23±0.05/0.38±0.09	0.81±0.02/1.02±0.02
		Python	0.18±0.04/0.28±0.07	0.81±0.01/1.02±0.02
	50%	R	0.20±0.06/0.31±0.11	0.83±0.01/1.05±0.01
		Python	0.18±0.03/0.28±0.07	0.83±0.01/1.04±0.01
	90%	R	0.40±0.07/0.60±0.13	1.24±0.10/1.68±0.19
		Python	0.39±0.06/0.60±0.10	1.23±0.10/1.65±0.17

Table 5: **BKTR** imputation performance comparison on simulated data. Mean ± standard deviation of the MAE and RMSE for $\boldsymbol{Y}$ and $\mathcal{B}$ computed across 10 distinct simulated datasets for different simulation scenarios.

After the completion of the MCMC sampling, we use the `imputed_y_estimates` attribute to get the imputed values and compare them with equivalent initial values that were removed to get the $MAE_{\boldsymbol{Y}}$ and $RMSE_{\boldsymbol{Y}}$. We also look at the impact of missing values on the evaluation of underlying beta values in each scenario by calculating $MAE_{\mathcal{B}}$ and $RMSE_{\mathcal{B}}$. We compare the results obtained with the **BKTR** package implemented in R with the results of the **pyBKTR** package implemented in Python. The results of this experiment are shown in Table 5.

We observe that estimation of missing $\boldsymbol{Y}$ values seems much more accurate when using kernel parameters creating matrices with higher correlations. Moreover, it emerges that reaching 90% of missing values, in all scenarios, is related to an important increase in $MAE_{\boldsymbol{Y}}$ and $RMSE_{\boldsymbol{Y}}$. Nonetheless, it is quite impressive to see that the $RMSE_{\boldsymbol{Y}}$ still has an average value of 1.05 even when 50% of $\boldsymbol{Y}$ values are missing for lengthscale values of 6. The results in R and Python are very similar, indicating that the calculation implementations correspond well in both languages.

5.3. Interpolation

Since interpolation has not been explored in the original BKTR paper (Lei *et al.* 2023), we show some results results on the two types of datasets, *Smaller* and *Larger* for both the R and the Python packages. For each dataset, we also use two different values (3 and 6) for both the spatial kernel lengthscale ϕ_1^{sim} and the temporal kernel lengthscale γ_1^{sim}. For each of the four aforementioned scenarios, we simulate 10 different datasets, from which we randomly set aside four spatial locations and six time points during the **BKTR** training phase. We fit the regression on $K_1 = 500$ burn-in iterations and $K_2 = 500$ sampling iterations with a rank decomposition of 10. Then, we use the predict method to try to find the $\boldsymbol{Y}$ and $\mathcal{B}$ values of the locations that were set aside. Subsequently, we calculate the error between the initial data and the set-aside-data via the $MAE_{\mathcal{B}}$, $RMSE_{\mathcal{B}}$, $MAE_{\boldsymbol{Y}}$ and $RMSE_{\boldsymbol{Y}}$ which we show in Table 6. From this table, we can observe again that using kernels that have higher correlation

Dataset	Simulation Settings	Lang.	Performance Metrics	
			$\text{MAE}_{\mathcal{B}}/\text{RMSE}_{\mathcal{B}}$	$\text{MAE}_{\boldsymbol{Y}}/\text{RMSE}_{\boldsymbol{Y}}$
Smaller $M = 20$ $N = 30$ $P = 4$	$\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 3$	R	0.81±0.19/1.08±0.27	1.79±0.69/2.48±1.03
		Python	0.76±0.19/1.03±0.27	1.68±0.50/2.35±0.97
	$\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 6$	R	0.51±0.11/0.71±0.19	1.16±0.12/1.49±0.16
		Python	0.52±0.10/0.71±0.19	1.11±0.13/1.42±0.21
Larger $M = 100$ $N = 150$ $P = 6$	$\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 3$	R	1.23±0.73/2.26±1.96	2.18±0.58/3.20±0.93
		Python	1.26±0.64/2.13±1.24	2.60±1.23/3.93±2.17
	$\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 6$	R	0.27±0.08/0.44±0.14	1.00±0.13/1.27±0.17
		Python	0.25±0.08/0.40±0.15	0.96±0.18/1.23±0.29

Table 6: BKTR interpolation performance comparison on simulated data. Mean ± standard deviation of the MAE and RMSE for $\boldsymbol{Y}$ and $\mathcal{B}$ computed across 10 distinct simulated datasets for different simulation scenarios.

Interpolated Portion	Performance Metrics	
	$\text{MAE}_{\mathcal{B}}/\text{RMSE}_{\mathcal{B}}$	$\text{MAE}_{\boldsymbol{Y}}/\text{RMSE}_{\boldsymbol{Y}}$
$\boldsymbol{Y}^1$ and $\mathcal{B}^1$; $M_{\text{new}} = 10 \times N_{\text{obs}} = 130$	0.24±0.06/0.38±0.10	1.05±0.14/1.34±0.22
$\boldsymbol{Y}^2$ and $\mathcal{B}^2$; $M_{\text{obs}} = 90 \times N_{\text{new}} = 20$	0.22±0.03/0.35±0.06	0.81±0.01/1.01±0.01
$\boldsymbol{Y}^3$ and $\mathcal{B}^3$; $M_{\text{new}} = 10 \times N_{\text{new}} = 20$	0.24±0.06/0.37±0.10	1.02±0.14/1.31±0.20
Total $\boldsymbol{Y}^{\text{new}}$ and $\mathcal{B}^{\text{new}}$	0.23±0.04/0.37±0.07	0.92±0.07/1.18±0.11

Table 7: **BKTR** interpolation performance breakdown on the different portions of the predicted data. Mean ± standard deviation of the MAE and RMSE for $\boldsymbol{Y}$ and $\mathcal{B}$ computed across 10 distinct simulated datasets for different interpolation portions.

$\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 6$ yields results with a lower error in all scenarios.

Another task we want to perform with simulated data is to examine the errors of the different interpolation segments presented in Figure 2 (right). Therefore, we investigated the performance of BKTR for interpolation across $\boldsymbol{Y}^1, \boldsymbol{Y}^2, \boldsymbol{Y}^3$, as well as the $\text{RMSE}_{\mathcal{B}}$ $\text{MAE}_{\mathcal{B}}$ errors on $\mathcal{B}^{\text{new}}$ across $\mathcal{B}^1, \mathcal{B}^2, \mathcal{B}^3$. We do so with a scenario that uses a *Larger* simulated dataset and simulation kernel lengthscale values of $\phi_1^{\text{sim}} = \gamma_1^{\text{sim}} = 6$. We set aside 10 locations and 20 time points during regression and use interpolation on them afterward. This translates to having $M^{\text{obs}} = 90$ observed locations and $M^{\text{new}} = 10$ new locations. For the time points, the equivalent sizes are $N^{\text{obs}} = 130$ and $N^{\text{new}} = 20$. We fit the BKTR regression onto the synthetic data using $K_1 = 1000$, $K_2 = 500$ and $R = 10$. We repeat the simulation, regression and interpolation exercise 10 times; the results are shown in Table 7.

From this table, we observe, via all error metrics, that the interpolation struggles more doing interpolation for new locations ($\boldsymbol{Y}^1$ and $\mathcal{B}^1$) than for new time points ($\boldsymbol{Y}^2$ and $\mathcal{B}^2$). Given that time points span only one dimension, compared to the two dimensions of spatial points, we are led to believe that the covariance pattern between observations is easier to estimate for new time points than new locations. It also seems as though the observations at new locations and new timestamps, $\boldsymbol{Y}^3$ and $\mathcal{B}^3$, have error values very similar to those of new locations, suggesting that the majority of the errors in this example come from predicting values for unseen locations.

6. Experimental study

This section aims to test the capacity of the **BKTR** package on real world data. We achieve this task using the same bike sharing demand dataset that was used by Lei *et al.* (2023). A considerable portion of this dataset was initially gathered by Wang *et al.* (2021), who used information from multiple sources to create a feature-rich set of data points.

This dataset contains five distinct data frames:

- `bixi_station_departures` (the response variable)
- `bixi_temporal_features` (the temporal covariates)
- `bixi_spatial_features` (the spatial covariates)
- `bixi_spatial_locations` (the location of each spatial point)
- `bixi_temporal_locations` (the location of each temporal point)

The `station_departures` data frame comes from BIXI Montréal (2023), which is a bike sharing company based in Montreal, Canada. The response variable is the total number of daily bike departures, for each station operated by BIXI during its 2019 season. The database contains $M = 587$ rows each representing one station and $N = 196$ columns representing different days from April 15 to October 27, 2019. The value of each data point in this data frame represents the total number of bike departures for a station on a given day. The `bixi_temporal_features` data includes daily meteorological covariates that vary through time such as temperature, precipitation, humidity, etc. This information was collected from the Environment and Climate Change Canada Historical Climate Data website. Temporal features also include data regarding whether each date was a holiday or not, according to the Quebec government. The `station_features` data frame includes data related to the location of each bike-sharing station such as local population (taken from the 2016 Canada census data at a dissemination block level), walk score (Walk Score 2023) and a number of other local features collected by DMTI Spatial Inc. (2019) such as the number of universities, metro stations, and so on. The data frame `temporal_locations` simply represents the position (`time_index`) of each of the $N = 196$ days relative to each other. Since there are no missing days, it simply translates to a range from 0 to 195 associated to the order of each date. The last data frame, `spatial_locations`, encodes the position for each of the $M = 587$ bike stations with geographic coordinates (e.g. latitude and longitude).

All data frames are available, as is, in the **BKTR** library. A normalized version of all datasets can also be accessed through the `BixiData` class of the `examples` module in both the **BKTR** packages.

6.1. Analysis

In this section we fit a `BKTRRegressor` to the `BixiData`, and then interpret the results using different properties and visualizations that the **BKTR** package offers.

We begin by fitting the number of bike departures using three covariates: the mean temperature, the total precipitation in mm and the total park area. To accomplish this task, we will pass the formula `nb_departure ~ 1 + mean_temp_c + area_park + walkscore` to the

initialization of the BKTRRegressor object. We will then run an MCMC sampling for 1500 iterations, including $K_1 = 1000$ burn-in iterations and $K_2 = 500$ sampling iterations, a rank decomposition of 8, a spatial Matérn 5/2 kernel and a locally periodic temporal kernel. To initialize a BKTRRegressor object with the aforementioned parameters, we can simply run the following code chunk (output omitted for brevity).

```
R> TSR$set_params(seed = 1, fp_type = 'float32', fp_device = 'cuda')
R> bixi_data <- BixiData$new()
R> <- KernelParameter$new(value = 7, is_fixed = TRUE)
R> k_local_periodic <- KernelSE$new() * KernelPeriodic$new(
+    period_length = KernelParameter$new(value = 7, is_fixed = TRUE))
R> bktr_regressor <- BKTRRegressor$new(
+    formula = nb_departure ~ 1 + mean_temp_c + area_park + total_precip_mm,
+    data_df = bixi_data$data_df,
+    spatial_positions_df = bixi_data$spatial_positions_df,
+    temporal_positions_df = bixi_data$temporal_positions_df,
+    rank = 8,
+    spatial_kernel = KernelMatern$new(smoothness_factor = 5),
+    temporal_kernel = kernel_local_periodic,
+    burn_in_iter = 1000,
+    sampling_iter = 500)
R> bktr_regressor$mcmc_sampling()
```

After sampling completion, we can obtain the summary of the BKTRRegressor object.

```
> summary(bktr_regressor)

=======================================================================
                        BKTR Regressor Summary
=======================================================================
Formula: nb_departure ~ 1 + mean_temp_c + area_park + total_precip_mm

Burn-in iterations: 1000
Sampling iterations: 500
Rank decomposition: 8
Nb Spatial Locations: 587
Nb Temporal Points: 196
Nb Covariates: 4
=======================================================================
In Sample Errors:
  RMSE: 0.072
  MAE: 0.053
Computation time: 1235.50s.
=======================================================================
-- Spatial Kernel --
Matern 5/2 Kernel
Parameter(s):
```

```
                       Mean   Median       SD  Low2.5p  Up97.5p
lengthscale          21.128   20.877    1.401   18.658   23.738

-- Temporal Kernel --
Composed Kernel (Mul)
  SE Kernel
  Parameter(s):
                         Mean   Median       SD  Low2.5p  Up97.5p
  lengthscale           6.448    6.437    0.114    6.252    6.685
  *
  Periodic Kernel
  Parameter(s):
                         Mean   Median       SD  Low2.5p  Up97.5p
  lengthscale           0.941    0.942    0.020    0.899    0.979
  period length          Fixed Value: 7.000
======================================================================
Beta Estimates Summary (Aggregated Per Covariates)

                       Mean   Median       SD
Intercept             0.447    0.376    0.306
mean_temp_c          -0.011   -0.008    0.042
area_park            -0.005   -0.011    0.185
total_precip_mm      -0.260   -0.214    0.189
======================================================================
```

The displayed model summary is again divided into four different sections. In the kernel section, we see that even the kernel resulting from the multiplication of two kernels shows a parameter summary in a very comprehensive fashion. In the beta coefficients estimates section, we observe that the `total_precip_mm` has the lowest coefficient estimate, which means that it has, on average, the most negative effect on the number of bike departures. The summary reveals that it took 1235.5 seconds to complete MCMC sampling and fit over 460k coefficients. It is worth noting that GPU acceleration played a significant role in handling this extensive dataset scenario, as running the same code once on our system with `fp_type='cpu'` took approximately 9.5h hours to complete (34,224s).

Another way to visualize some aspects of the fitted model is to plot the coefficient estimates of covariates through time for a given spatial point. This can be done by calling the `plot_temporal_betas` function on a `BKTRRegressor` object.

```
R> plot_temporal_betas(
+    bktr_regressor,
+    plot_feature_labels = c('mean_temp_c', 'area_park', 'total_precip_mm'),
+    spatial_point_label = '7114 - Smith / Peel')
```

The result of this call is shown in Figure 6. This plot helps us visualize the non-stationarity of the coefficients via the evolution of the influence of the covariates through time. For instance, we can see that the influence of the mean temperature on the number of bike departures is lower during the initial and final months of the season compared to the midsummer months. For this station, precipitation negatively impacts departures during summer.

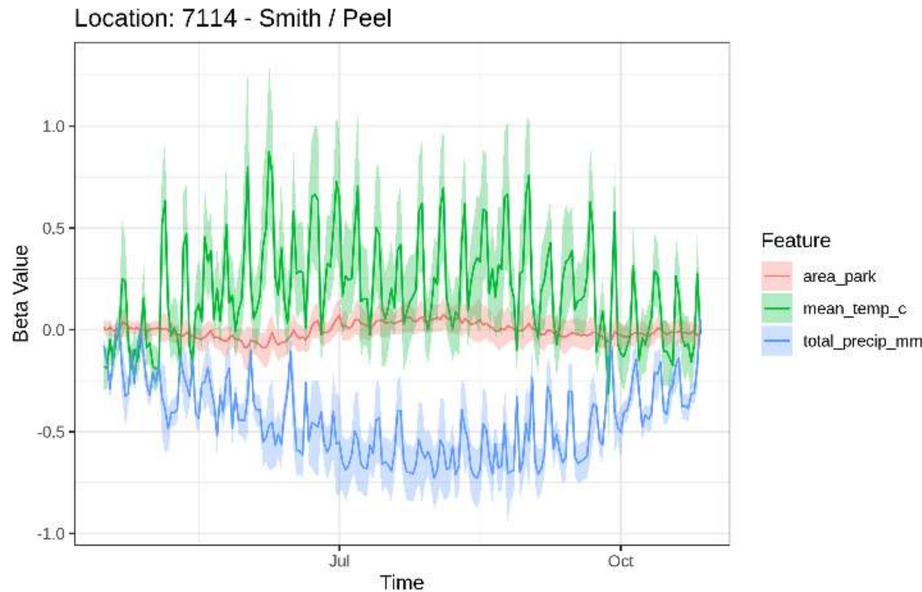


Figure 6: Result of the `plot_temporal_betas` function to plot the coefficient estimates of the covariates through time for a given spatial location.

Similarly, we can plot the coefficient estimates of the covariates through space for a given time point. This can be done similarly by calling the `plot_spatial_betas` function.

```
R> plot_spatial_betas(
+     bktr_regressor,
+     plot_feature_labels = c('mean_temp_c', 'area_park', 'total_precip_mm'),
+     temporal_point_label = '2019-07-19',
+     nb_cols = 3)
```

The result of this function call is shown in Figure 7. Again, this plot helps us to better visualize non-stationarity with the evolution of the influence of the covariates through space. We can observe that the precipitation has a more negative influence on the number of bike departures in the centre of the city than in the periphery of the city.

We can demonstrate the proximity of **BKTR**'s estimated response variables to the actual values by utilizing the `plot_y_estimates` function, employing the same code as presented in Section 5.1. The outcomes of this function are visualized in Figure 8. These plots show estimated values that, while slightly further from the diagonal compared to those in Figure 5, still display a relevant correlation between estimated and observed response variables.

We can see that the **BKTR** package provides a very easy way to fit local regression models onto a given dataset and that it also provides useful and user-friendly integrated functions to help visualize and understand the fitted model.

6.2. Imputation Example

In this section, we will explore how to use **BKTR** to estimate the values for missing data points in the response variable $\boldsymbol{Y}$ of the BIXI dataset. Imputation for **BKTR** can only be

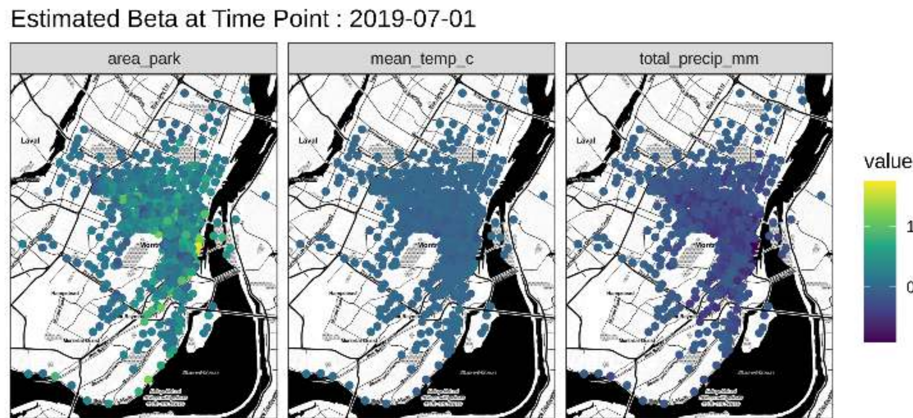


Figure 7: Result of the `plot_spatial_betas` function to plot the coefficient estimates of the covariates through space for a given time point.

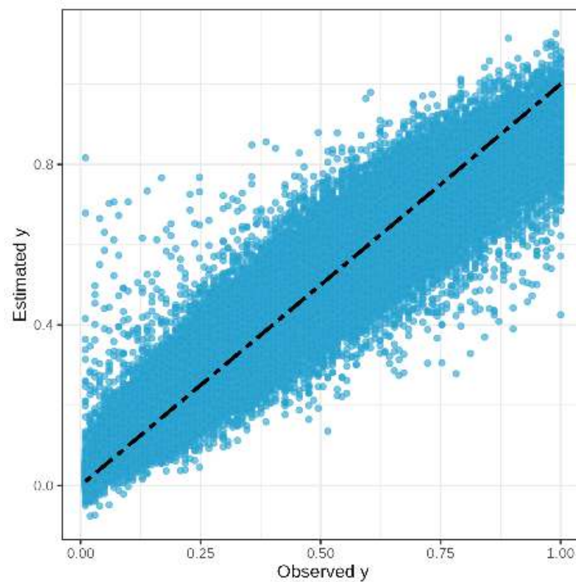


Figure 8: Scatter plot of **BKTR** estimated response variable vs. observed values in the BIXI dataset.

done at the response variable level, which means that it is not currently possible to estimate missing covariate values. It is possible to observe that there are already some missing values in the `data_df` data frame of the BIXI dataset for the response variable column `nb_departure`, as the next code snippet illustrates.

```
R> y_is_na <- is.na(bixi_data$data_df$nb_departure)
R> nb_y_na <- sum(y_is_na)
R> sprintf(
```

```
+   'There is %.d missing `nb_departure` values representing ~%.2f%%',
+   nb_y_na,
+   nb_y_na / length(y_is_na) * 100))
```

```
"There is 14940 missing `nb_departure` values representing ~12.99%"
```

This shows us that around 13% of the response variable values are already missing from our dataset. Let us take a look at the first three missing values of the data frame in order to find the location and moment these values were situated.

```
R> bixi_data$data_df[which(y_is_na)[1:3], 1:3]
```

```
                                          location       time nb_departure
1: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-04-22           NA
2: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-05-08           NA
3: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-05-16           NA
```

Knowing that the location *10002 - Métro Charlevoix (Centre / Charlevoix)* is missing multiple values, we can effortlessly use the `imputed_y_estimates` attribute from the fitted regressor instance of Section 6.1 to estimate those data points.

```
R> bktr_regressor$imputed_y_estimates[which(y_is_na)[1:3]]
```

```
                                          location       time     y_est
1: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-04-22 0.7535655
2: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-05-08 1.0123121
3: 10002 - Métro Charlevoix (Centre / Charlevoix) 2019-05-16 1.0687331
```

6.3. Interpolation Example

In the BIXI case, interpolation can prove to be highly valuable in estimating the number of departures at newly planned locations. This estimation could help significantly with allocation planning for the number of docks needed at a given new location. Furthermore, when there is complete absence of data for a given period (e.g. due to a reporting data centre shortage with a duration of two days), we could simply use interpolation to gather an estimate of the number of departure per station during that period. To test the interpolation capabilities of **BKTR**, we arbitrarily select three bike stations and two contiguous days in the dataset. We then temporarily remove those days and stations from the initial dataset and set them aside. Next, we fit the regressor on the remaining data, and lastly, we use the `predict` method on the set-aside data with the goal of estimating the number of departures on those days at those locations.

We start by setting aside the following three locations *4002 - Graham / Wicksteed*, *7079 - Notre-Dame / Gauvin* and *6236 - Laurier / de Bordeaux*. We also set aside two time points equivalent to the dates *2019-05-01* and *2019-05-02*. We then separate the three main BIXI data frames into two portions, which are the observed data and the new data.

```
R> library(data.table)
R> TSR$set_params(seed=0, fp_type='float32')
R> bixi_data <- BixiData$new()
R> data_df <- bixi_data$data_df
R> spa_df <- bixi_data$spatial_positions_df
R> tem_df <- bixi_data$temporal_positions_df
R> # New locations and times
R> new_s <- c('4002 - Graham / Wicksteed',
+    '7079 - Notre-Dame / Gauvin',
+    '6236 - Laurier / de Bordeaux')
R> new_t <- c('2019-05-01', '2019-05-02')
R> # Cast to IDate to match implicit cast of data.table
R> new_t <- as.IDate(new_t)
R> # Get obs data
R> obs_s <- setdiff(unlist(spa_df$location), new_s)
R> obs_t <- setdiff(unlist(tem_df$time), new_t)
R> obs_data_df <- data_df[data_df[, .I[
+    location %in% obs_s & time %in% obs_t]], ]
R> obs_spa_df <- spa_df[spa_df[, .I[location %in% obs_s]], ]
R> obs_tem_df <- tem_df[tem_df[,.I[time %in% obs_t]], ]
R> # Get new data
R> new_data_df <- data_df[data_df[, .I[
+    location %in% new_s | time %in% new_t]], ]
R> new_spa_df <- spa_df[spa_df[, .I[location %in% new_s]], ]
R> new_tem_df <- tem_df[tem_df[, .I[time %in% new_t]], ]
```

Subsequently, we train the BKTR model on the observed data to predict the new unobserved datasets. As a last step, we compare the interpolation results with the real observed response values when they are not missing with the usual error metrics.

```
R> # Train and predict
R> bktr_regressor <- BKTRRegressor$new(
+    data_df = obs_data_df,
+    spatial_positions_df = obs_spa_df,
+    temporal_positions_df = obs_tem_df,
+    #... other parameters like section 6.1)
R> preds <- bktr_regressor$predict(
+    new_data_df, new_spa_df, new_tem_df)
R> # Sort data for comparison and remove na values
R> new_data_df <- data_df[
+    data_df[, .I[location %in% new_s | time %in% new_t]],
+    c('location', 'time', 'nb_departure')]
R> pred_y_df <- preds$new_y_df
R> setkey(new_data_df, location, time)
R> setkey(pred_y_df, location, time)
R> non_na_indices <- which(!is.na(new_data_df$nb_departure))
R> # Compare predictions
```

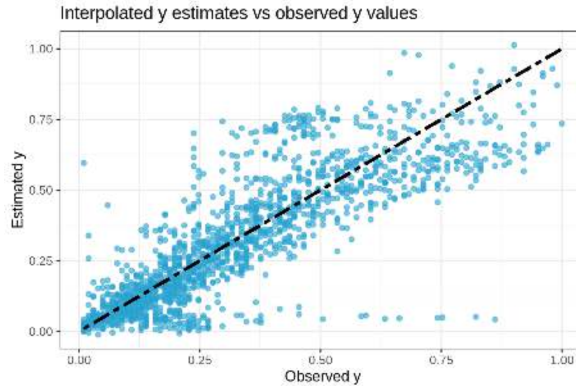


Figure 9: Scatter plot of **BKTR** interpolated response variable vs. observed values in the BIXI dataset.

```
R> y_err <- (new_data_df$nb_departure[non_na_indices]
+     - pred_y_df$y_est[non_na_indices])
R> sprintf('Predicting %d y values || MAE: %.4f || RMSE: %.4f',
+     length(non_na_indices), mean(abs(y_err)), sqrt(mean(y_err ^ 2)))

[1] "Predicting 1664 y values || MAE: 0.0878 || RMSE: 0.1278"
```

We perceive that once the data is sorted and split into training and prediction, the other commands related to predictions are fairly easy to use. Also, since the $\boldsymbol{Y}$ values had some missing data points, we needed to remove them from the evaluated vector to calculate the error metrics properly. We obtain values of $\text{RMSE}_{\boldsymbol{Y}} = 0.088$ and a $\text{MAE}_{\boldsymbol{Y}} = 0.128$, which are higher than the in-sample errors, but still showcase decent prediction capabilities on a real dataset. In Figure 9, we compare the obtained interpolated $\boldsymbol{Y}$ data points with their respective true underlying values. The results show the capacity of **BKTR** to estimate values at time points or locations that were never observed before.

7. Summary and discussion

The **BKTR** (Bayesian Kernelized Tensor Regression) packages presented through this work introduce a compelling tool for local spatiotemporal regression analysis in both R and Python. This framework offers a user-friendly endpoint, while also focusing on flexibility and computational performance. Leveraging a library specific to tensor , such as Falbel and Luraschi (2023), enables this package to be extremely efficient and to also harness the computing speed of dedicated floating point operation hardware such as GPUs. The sensible default values and choices used in **BKTR** allows for a low-friction starting point for the majority of users, while enabling complex fine-tuning for the more advanced ones through composed kernels and access to multiple regression parameters. This library, implemented in two of the most prominent statistical programming languages, makes the work realized by Lei *et al.* (2023) available to the research community. The **BKTR** packages also bring to life a new feature, the capacity to do predictions on unobserved time points and locations, which is called interpolation. Through extensive testing, we demonstrate the important capabilities of **BKTR** in

regression, imputation and interpolation. Since it is one of the very first packages enabling local regressions to fit coefficients for every location and time point combinations of very large datasets, we think that this breakthrough will greatly improve the modelling process of vast number of spatiotemporal analyses.

We aim to continuously improve and expand the functionalities of the **BKTR** packages that we presented herein. We will do so by actively monitoring user issues and requests that will be provided to each package's respective GitHub. By developing the package in both languages, we also commit the development of features in a way that will be accessible for R and Python developers. Since both packages are based on **torch** and **pytorch**, we will keep a close eye on both packages' advancements to be able to provide our users with the latest enhancements in tensor computation.

Computational details

Results in this paper were obtained using Google Colab instances with a type of shape that was *High-Ram* and a V100 GPU. Thus, the calculations were done with an 8-core Xeon processor, 25.5GB of total CPU RAM and 16GB of GPU Ram. We also used the default R and Python versions in Colab, which were R 4.3.1 with **torch** 0.11.0 and Python 3.10.6 with **pytorch** 1.12.1.

In all examples shown in the paper, we used an `fp_type` of `'float32'` and `'cuda'` as an `fp_device`. We specifically looked into the influence of the device and floating point format used during tensor operations and details are provided in Appendix D. The results show that there is no significant difference in the parameter estimation precision when using different floating point format or device. However, it is interesting to observe that the use of `'float32'` over `'float64'` leads to important computational speed improvements when we compare the mean of sampling runtime, with an improvement of 36% on the CPU and a lesser uptick of 5% on the GPU. It is also possible to perceive that using the GPU improves the execution speed for both floating point formats (223% using `'float64'`and 138% using `'float32'`).

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015). "**TensorFlow**: Large-Scale Machine Learning on Heterogeneous Systems." Software available from tensorflow.org, URL `https://www.tensorflow.org/`.

Baddeley A, Turner R (2005). "**spatstat**: An R Package for Analyzing Spatial Point Patterns." *Journal of Statistical Software*, **12**(6), 1–42. `doi:10.18637/jss.v012.i06`.

Bakar KS, Kokic P, Jin H (2016). "Hierarchical Spatially Varying Coefficient and Temporal Dynamic Process Models using **spTDyn**." *Journal of Statistical Computation and Simulation*, **86**(4), 820–840. URL `https://doi.org/10.1080/00949655.2015.1038267`.

Bakar KS, Sahu SK (2015). "spTimer: Spatio-Temporal Bayesian Modeling Using R." *Journal of Statistical Software*, **63**(15), 1–32. doi:10.18637/jss.v063.i15. URL https://www.jstatsoft.org/index.php/jss/article/view/v063i15.

BIXI Montréal (2023). "Open Data." Accessed: 2023-07-11, URL https://bixi.com/en/open-data.

Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "**Stan**: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. doi:10.18637/jss.v076.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v076i01.

Chang W (2021). ***R6**: Encapsulated Classes with Reference Semantics*. R package version 2.5.1, URL https://CRAN.R-project.org/package=R6.

DMTI Spatial Inc (2019). "Enhanced Point of Interest (DMTI)." URL https://www.dmtispatial.com.

Dowle M, Srinivasan A (2023). ***data.table**: Extension of 'data.frame'*. R package version 1.14.8, URL https://CRAN.R-project.org/package=data.table.

Duvenaud D (2014). *Automatic Model Construction with Gaussian Processes*. Ph.D. thesis, University of Cambridge.

Falbel D, Luraschi J (2023). ***torch**: Tensors and Neural Networks with 'GPU' Acceleration*. R package version 0.9.1, URL https://CRAN.R-project.org/package=torch.

Finley AO, Banerjee S, EGelfand A (2015). "**spBayes** for Large Univariate and Multivariate Point-Referenced Spatio-Temporal Data Models." *Journal of Statistical Software*, **63**(13), 1–28. URL https://www.jstatsoft.org/article/view/v063i13.

Gamerman D, Lopes HF, Salazar E (2008). "Spatial Dynamic Factor Analysis." *Bayesian Analysis*, **3**(4), 759–792. doi:10.1214/08-BA329. URL https://doi.org/10.1214/08-BA329.

Gardner J, Pleiss G, Weinberger KQ, Bindel D, Wilson AG (2018). "Gpytorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration." *Advances in neural information processing systems*, **31**.

Gelfand AE, Kim HJ, Sirmans CF, Banerjee S (2003). "Spatial Modeling With Spatially Varying Coefficient Processes." *Journal of the American Statistical Association*, **98**(462), 387–396. doi:10.1198/016214503000170. https://doi.org/10.1198/016214503000170, URL https://doi.org/10.1198/016214503000170.

Geman S, Geman D (1984). "Geman, D.: Stochastic Relaxation, Gibbs Distribution, and the Bayesian Restoration of Images. IEEE Trans. Pattern Anal. Mach. Intell. PAMI-6(6), 721-741." *IEEE Trans. Pattern Anal. Mach. Intell.*, **6**, 721–741. doi:10.1109/TPAMI.1984.4767596.

Gräler B, Pebesma E, Heuvelink G (2016). "Spatio-Temporal Interpolation Using **gstat**." *The R Journal*, **8**, 204–218. URL https://journal.r-project.org/archive/2016/RJ-2016-014/index.html.

Khatri C, Rao CR (1968). "Solutions to Some Functional Equations and their Applications to Characterization of Probability Distributions." *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 167–180.

Kolda TG, Bader BW (2009). "Tensor Decompositions and Applications." *SIAM review*, **51**(3), 455–500.

Lei M, Labbe A, Sun L (2023). "Scalable Spatiotemporally Varying Coefficient Modelling with Bayesian Kernelized Tensor Regression." arXiv:2109.00046.

Lindgren F, Rue H (2015). "Bayesian Spatial Modelling with **R-INLA**." *Journal of Statistical Software*, **63**(19), 1–25. doi:10.18637/jss.v063.i19. URL https://www.jstatsoft.org/index.php/jss/article/view/v063i19.

Lunn DJ, Thomas A, Best N, Spiegelhalter D (2000). "**WinBUGS** – A Bayesian Modelling Framework: Concepts, Structure, and Extensibility." *Statistics and Computing*, **10**(4), 325–337. ISSN 0960-3174. doi:10.1023/A:1008929526011. URL https://doi.org/10.1023/A:1008929526011.

Lustiger H (2022). *Design Patterns in R*. Tidylab, Auckland, New Zealand. URL https://github.com/tidylab/R6P.

Millo G, Piras G (2012). "**splm**: Spatial Panel Data Models in R." *Journal of Statistical Software*, **47**(1), 1–38. URL https://www.jstatsoft.org/v47/i01/.

Neal RM (2003). "Slice sampling." *The Annals of Statistics*, **31**(3), 705 – 767. doi:10.1214/aos/1056562461. URL https://doi.org/10.1214/aos/1056562461.

Nychka D, Furrer R, Paige J, Sain S (2021). "**fields**: Tools for Spatial Data." R package version 15.2, URL https://github.com/dnychka/fieldsRPackage.

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019). "**PyTorch**: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pebesma E (2012). "**spacetime**: Spatio-Temporal Data in R." *Journal of Statistical Software*, **51**(7), 1–30. URL https://www.jstatsoft.org/v51/i07/.

Pebesma EJ (2004). "Multivariable Geostatistics in S: the **gstat** Package." *Computers Geosciences*, **30**, 683–691.

Pebesma EJ, Bivand R (2005). "Classes and Methods for Spatial Data in R." *R News*, **5**(2), 9–13. URL https://CRAN.R-project.org/doc/Rnews/.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

Plotly Technologies Inc (2015). "Collaborative data science." URL https://plot.ly.

Plummer M (2003). "**JAGS**: A Program for Analysis of Bayesian Graphical Models using Gibbs Sampling." *3rd International Workshop on Distributed Statistical Computing (DSC 2003); Vienna, Austria*, **124**.

Plummer M (2023). ***rjags**: Bayesian Graphical Models using MCMC*. R package version 4-14, URL https://CRAN.R-project.org/package=rjags.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rue H, Martino S, Chopin N (2009). "Approximate Bayesian Inference for Latent Gaussian Models by Using Integrated Nested Laplace Approximations." *Journal of the Royal Statistical Society Series B: Statistical Methodology*, **71**(2), 319–392.

Sigrist F, Künsch HR, Stahel WA (2015). "**spate**: An R Package for Spatio-Temporal Modeling with a Stochastic Advection-Diffusion Process." *Journal of Statistical Software*, **63**(14), 1–23. doi:10.1111/rssb.12061.

Snyder JP (1987). *Map Projections–A Working Manual*, volume 1395. US Government Printing Office.

Stan Development Team (2023). "**RStan**: the R Interface to Stan." R package version 2.21.8, URL https://mc-stan.org/.

Sturtz S, Ligges U, Gelman A (2005). "**R2WinBUGS**: A Package for Running **WinBUGS** from R." *Journal of Statistical Software*, **12**(3), 1–16. URL http://www.jstatsoft.org.

Takeuchi K, Kashima H, Ueda N (2017). "Autoregressive Tensor Factorization for Spatio-Temporal Predictions." In *2017 IEEE International Conference on Data Mining (ICDM)*, pp. 1105–1110. doi:10.1109/ICDM.2017.146.

The Pandas Development Team (2022). "**Pandas** 1.4.2." doi:10.5281/zenodo.6408044. URL https://doi.org/10.5281/zenodo.6408044.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Fourth edition. Springer, New York. ISBN 0-387-95457-0, URL https://www.stats.ox.ac.uk/pub/MASS4/.

Walk Score (2023). "Walk Score Methodology." Accessed: 2023-07-11, URL https://www.walkscore.com/methodology.shtml.

Wang X, Cheng Z, Trépanier M, Sun L (2021). "Modeling Bike-Sharing Demand Using a Regression Model With Spatially Varying Coefficients." *Journal of Transport Geography*, **93**, 103059. ISSN 0966-6923. doi:https://doi.org/10.1016/j.jtrangeo.2021.103059. URL https://www.sciencedirect.com/science/article/pii/S0966692321001125.

Wardrop M (2022). "**Formulaic**: An implementation of Wilkinson formulas." URL https://matthewwardrop.github.io/formulaic.

Wickham H (2016). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org.

Wilkinson GN, Rogers CE (1973). "Symbolic Description of Factorial Models for Analysis of Variance." *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **22**(3), 392–399. ISSN 00359254, 14679876. URL http://www.jstor.org/stable/2346786.

A. Tensor context

In both **BKTR** packages, we created modules called `tensor_ops` containing an object named `TSR`. The goal of `TSR` is to centralize all tensor operations and properties in one object. This way, if in the future we needed to fix an operation or wanted to integrate another tensor operation backend like **tensorflow** (Abadi *et al.* 2015), we could do it at a central location. As we wanted to be able to use kernels and the `BKTRRegressor` independently, we decided to set the tensor environment directly with the `TSR` object via the `set_params` method.

One of the few issues that we encountered with this implementation was that we were unable to use the equivalent of Python's `classmethods` and class properties with the **R6** package. Therefore, we decided to opt for a singleton design pattern to keep only one `TSR` instance with one setting alive at a time inside the R environment. To use this design pattern, we found that the **R6P** package (Lustiger 2022) was a sound implementation of it and thus decided to add it to our dependencies.

B. Covariates reshaping

With spatiotemporal data, it is common to observe datasets where the spatial covariates do not vary through time and where the temporal covariates do not vary in space. This is notably the case for the BIXI data of Section 6. When such a case arises, we often see that the dataset is expressed in a much more compressed manner. When facing this setup, we can reformulate `data_df` into three matrices. The first matrix is the response variable we call `y_df`, which can be of shape $M \times N$, the spatial covariates matrix that we can call `spatial_df` with p_s spatial features and M locations with a shape of $M \times p_s$, and lastly, the temporal covariates with p_t temporal features and N time points with a shape of $N \times p_t$. Using these three data frames, we can create a `data_df` containing $MN \times (1 + p_s + p_t)$ and achieve that with few operations, repeating the matrices a certain number of times and stacking them together. However, it is possible to achieve this more easily using the `reshape_covariate_dfs` utility functions that we implemented. This function takes the three above-mentioned data frames as arguments and another argument called `y_column_name` denoting the desired column name for the response variable in the `y_df` data frame. With all these parameters provided, the `reshape_covariate_dfs` function will return a single data frame containing the information of the 3 data frames combined. Let us see an example of this function in action using the BIXI dataset.

```
R> spatial_df <- bixi_data$spatial_features_df
R> temporal_df <- bixi_data$temporal_features_df
R> y_df <- bixi_data$departure_df
R> p_s <- ncol(spatial_df) - 1 # Not counting index column
R> p_t <- ncol(temporal_df) - 1
R> sprintf('Response M=%d and N=%d', nrow(y_df), ncol(y_df))
R> sprintf('Spatial features M=%d x p_s=%d', nrow(spatial_df), p_s)
R> sprintf('Temporal features N=%d x p_t=%d', nrow(temporal_df), p_t)
R> data_df <- reshape_covariate_dfs(spatial_df, temporal_df,
+    y_df, 'nb_departure')
R> sprintf('Should obtain MN=%d x P=%d', nrow(spatial_df) *
```

```
+    nrow(temporal_df), 1 + p_s + p_t)
R> sprintf('Reshaped MN=%d x P=%d', nrow(data_df), ncol(data_df) - 2)

[1] "Response M=587 and N=197"
[1] "Spatial features M=587 x p_s=13"
[1] "Temporal features N=196 x p_t=5"
[1] "Should see MN=115052 x P=19"
[1] "Reshaped MN=115052 x P=19"
```

C. Simulation Study

To simulate data and test **BKTR** we used a utility function that we implemented in **BKTR** named `simulate_spatiotemporal_data`.

The function generates M spatial locations in a d_s dimension Euclidean space with each dimension being in a range of $[0, S_s]$, where S_s is the scale parameter of the spatial dimensions. It also generates N sequential time points that are in a range of $[0, S_t]$, where S_t is the time scale parameter. Resulting time points therefore have a time resolution of $S_t/(N-1)$.

The covariates are simulated using an intercept term, p_s spatial covariates and p_t temporal covariates.

$$\begin{aligned} \boldsymbol{\mathcal{X}}(:,:,p=1) &= 1 && \text{(intercept)}, \\ \boldsymbol{\mathcal{X}}(:,:,p=2,\ldots,1+p_s) &= 1_N^\top \otimes \boldsymbol{x}_s^p, \text{ with } \boldsymbol{x_s^p} \sim \mathcal{N}(\boldsymbol{\mu}_s^p, \boldsymbol{I}_M) && \text{(spatial covariates)}, \\ \boldsymbol{\mathcal{X}}(:,:,p=2+p_s,\ldots,P) &= \boldsymbol{x}_t^p \otimes 1_M^\top, \text{ with } \boldsymbol{x_t^p} \sim \mathcal{N}(\boldsymbol{\mu}_t^p, \boldsymbol{I}_N) && \text{(temporal covariates)}, \end{aligned} \tag{11}$$

where $P = 1 + p_s + p_t$ is the total number of covariates, and $\boldsymbol{\mu}_s^p$ and $\boldsymbol{\mu}_t^p$ are vectors of length M and N, respectively.

The $\boldsymbol{\beta}$ values are generated from a multivariate normal distribution:

$$\text{vec}(\boldsymbol{B}_{(3)}) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{K}_t^{\text{sim}} \otimes \boldsymbol{K}_s^{\text{sim}} \otimes \boldsymbol{\Lambda}_w^{-1}), \tag{12}$$

with $\boldsymbol{K}_s^{\text{sim}}$ and $\boldsymbol{K}_t^{\text{sim}}$ being covariance matrices generated by the spatial and temporal kernels, respectively, and $\boldsymbol{\Lambda_W}$ is generated from a Wishart distribution $\mathcal{W}(\boldsymbol{I}_P, P)$ with $\boldsymbol{I}_P$ being an identity matrix of size $P \times P$. Both mentioned kernels are input parameters for the `simulate_spatiotemporal_data` function.

The response variable $\boldsymbol{Y}$, for its part, is created from the product of the covariates and the $\boldsymbol{\mathcal{B}}$ values to which we add an error term $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}_{MN}, \sigma_\epsilon^2\ \boldsymbol{I}_{MN})$, where σ_ϵ^2 is the scale parameter of the error term and $\boldsymbol{I}_{MN}$ is an identity matrix of size $MN \times MN$.

In summary, the `simulate_spatiotemporal_data` function input parameters are

- M: `nb_spatial_locations` (number of spatial locations)
- N: `nb_time_points` (number of time points)
- d_s: `nb_spatial_dimensions` (number of spatial dimensions in the Euclidean space)
- S_s: `spatial_scale` (scale of the spatial dimensions)

- S_t: `time_scale` (scale of the time dimension)
- $\boldsymbol{\mu}_s$: `spatial_covariates_means` (mean vector of the spatial covariates)
- $\boldsymbol{\mu}_t$: `temporal_covariates_means` (mean vector of the temporal covariates)
- $k_s^{\text{sim}}(s_m, s_{m'}; \Phi^{\text{sim}})$: `spatial_kernel` (spatial kernel generating $\boldsymbol{K}_s^{\text{sim}}$)
- $k_t^{\text{sim}}(t_n, t_{n'}; \Gamma^{\text{sim}})$: `temporal_kernel` (temporal kernel generating $\boldsymbol{K}_t^{\text{sim}}$)
- σ_ϵ: `noise_scale` (scale parameter for the added noise)

The `simulate_spatiotemporal_data` function returns a list (dictionary in Python) containing four data frames: `data_df`, `spatial_locations_df`, `time_points_df` and `beta_df`. The first three data frames can directly be passed to the initialization of a `BKTRRegressor`. The `beta_df` data frame represents the true $\boldsymbol{\mathcal{B}}$ values and has a shape of $MN \times (2 + P)$ (including two columns for the labels). The dataset generated has spatial covariates that are independent of the spatial locations and temporal covariates that are independent of the time points. Therefore, we used a process similar to the `reshape_covariate_dfs` utility function (see Appendix B) to create a valid `data_df` data frame.

It is important to note that because of the double Kronecker product used in the Equation 12, the covariance matrix that will be generated will use a sizeable amount of memory since its shape will be $MNP \times MNP$. For this reason, we decided to sample from a matrix normal distribution instead of generating the whole covariance matrix. The used matrix normal distribution is defined as follows:

$$\text{vec}(\boldsymbol{B}_{(3)}) \sim \mathcal{MN}(\mathbf{0}_{N\times MP}, \boldsymbol{K}_t^{\text{sim}}, \boldsymbol{K}_s^{\text{sim}} \otimes \boldsymbol{\Lambda}_w^{-1}), \tag{13}$$

From this distribution we can significantly reduce memory usage by sampling a tensor of size $N \times MP$ from MNP independent $\mathcal{N}(0, 1)$ distributions. The generated tensor can then be multiplied by the Cholesky decomposition of $\boldsymbol{K}_t^{\text{sim}}$ and be followed by a matrix multiplication with the Cholesky decomposition of $\boldsymbol{K}_s^{\text{sim}} \otimes \boldsymbol{\Lambda}_w^{-1}$.

D. Influence of device and floating point format

In this section, we look into the influence of the device and floating point format used during tensor operations. The goal is to be able to select a default type of floating point format and calculation device when using **BKTR**.

As some reader might have noticed, we have used a (particular) `TSR` component in all our operations thus far. `TSR` is a wrapper containing all used tensor operations functions. This object allows us to set the seed for our operations via the `seed` argument of the `set_params` method. In addition to the `seed` argument, the `set_params` method can also receive information about where the calculation should be done (on which device) and using which type of floating point format. This information can be passed via the `fp_device` and `fp_type` parameters of the `set_params` method. The available values for `fp_device` at the moment are `'cpu'` and `'cuda'`, where `'cpu'` uses the computer's central processing unit (CPU) and `'cuda'` uses the graphics processing unit (GPU) if there is one on the system being used. For the floating point format, there are also two different values that can be passed to the

fp_device	fp_type	Performance Metrics		
		$\text{MAE}_{\boldsymbol{\mathcal{B}}}/\text{RMSE}_{\boldsymbol{\mathcal{B}}}$	$\text{MAE}_{\boldsymbol{Y}}/\text{RMSE}_{\boldsymbol{Y}}$	Time (s)
'cpu'	'float64'	0.078±0.01/0.125±0.02	0.792±0.00/0.992±0.00	713±9
	'float32'	0.080±0.01/0.126±0.02	0.787±0.01/0.987±0.01	525±11
'cuda'	'float64'	0.075±0.01/0.125±0.03	0.793±0.00/0.994±0.01	221±10
	'float32'	0.080±0.02/0.123±0.03	0.787±0.01/0.986±0.01	210±9

Table 8: **BKTR** regression fitting performance comparison on simulated data using different processing device (`fp_device`) and floating point format (`fp_type`).

`fp_type` parameter, which are `'float32'` for using a single precision number format and `'float64'` to use double precision.

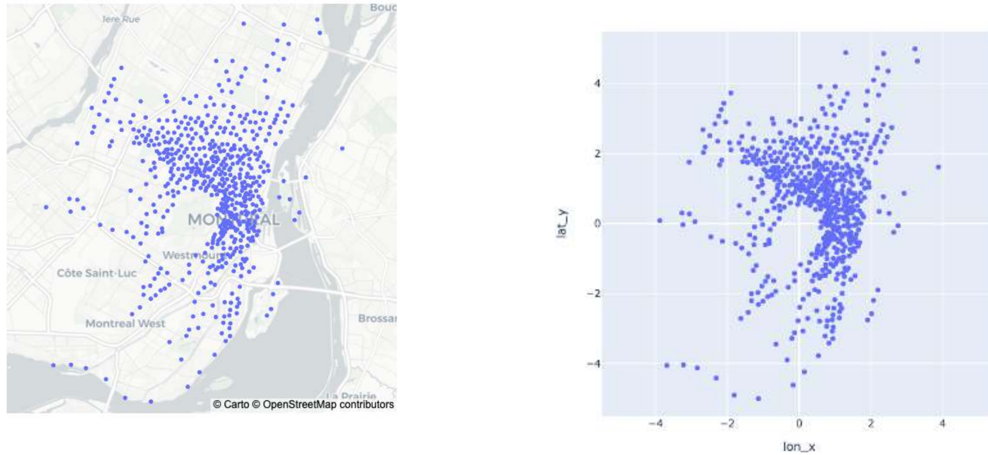
In some software packages or applications, using single precision instead of double precision can degrade the precision of the results obtained, while providing major computational speed upticks. When using a sizeable amount of data, running matrix and tensor operations on the GPU instead of the CPU can usually provide great computational speed gains. Therefore, we test the four possible combinations of device and floating point format on simulated data to assess their performance. For each combination, we use 10 new different *Larger* datasets (as mentioned in 5), on which we fit the **BKTR** package using 500 burn-in iterations, 500 sampling iterations and a rank decomposition of 10. The results obtained are shown in Table 8.

The results show that there is no significant difference in the parameter estimation precision when using a different floating point format or device. However, it is interesting to observe that the use of `'float32'` over `'float64'` leads to important computational speed improvements when we compare the mean of sampling runtime, with an improvement of 36% on the CPU and a lesser uptick of 5% on the GPU. Looking at the influence of the device, it is also possible to perceive that using the GPU improves the execution speed for both floating point formats (223% using `'float64'`and 138% using `'float32'`). Thus, in this paper, we use an `fp_type` of `'float32'` and `'cuda'` as an `fp_device`.

E. Spatial coordinates projection

By default, a vast quantity of geographic tools use longitude and latitude measurements. Since these values represent angles on earth from the meridian and equator, we need to transform them into coordinates that can be projected in a Euclidean space. This is to ensure that we keep valid kernels for the MCMC sampling process. One of the simple forms of projection used on a multitude of maps is the Mercator projection (Snyder 1987). This type of projection makes it possible to transform longitude and latitude in a 2D space by projecting them on a plane. By doing so, the coordinates become simple x and y coordinates on a map that can be drawn on paper. However, this comes with some limitations. To transform a sphere into a plane, the Mercator projection greatly distorts the size of areas that are far from the equator. To be able to do this cylindrical projection, the projection also needs to have a cutting point, which is usually the meridian. This means that even if two points are close but located on different sides of the meridian, they will appear to be extremely far apart on a Mercator projection.

The above-mentioned limitations can have important impacts on some datasets. However,



(a) Plotly OpenStreetMap scatterbox plot on latitude and longitude

(b) Mercator projection with a scale factor of $S^p = 10$

Figure 10: Visualization comparison between Plotly OpenStreetMap scatter box plot and **BKTR** Mercator's projection for BIXI spatial locations.

when using coordinates that are very close to each other (e.g. at a city level) as in Section 6, these limitations become negligible. Therefore, we implemented a means to seamlessly transform by default, with a Mercator projection, all longitude and latitude coordinates provided to the `BKTRRegressor` class. When projecting the coordinates in a 2D plane, we allow the user to choose a given scale at which to perform the projection. This scale S^p is used to transform all provided coordinates into a square with a domain for the X and Y coordinates of $[-S^p/2, S^p/2]$. Knowing the scale in which the data was projected can help the user in choosing sensible values for kernel lengthscale afterward. To help visualize the projection, we compare an example of the Mercator projected data for the BIXI dataset with a plot of the respective geographic coordinates (longitude, latitude) from Plotly Technologies Inc. (2015) using Open Street Map. The comparison is shown is shown in Figure 10 for a scaling parameter of $S^p = 10$.

The figure shows that the position property of the points are similar in both cases, but in the **BKTR** mercator's projection it is now located on an X and Y axis with a range of 10.

Affiliation:

Julien Lanthier, Aurélie Labbe
Department of Decision Sciences
HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec), Canada H3T 2A7
E-mail: julien.lanthier@hec.ca, aurelie.labbe@hec.ca

Mengying Lei, Lijun Sun

Department of Civil Engineering
McGill University
817 Sherbrooke Street West, Macdonald Engineering Building
Montréal (Québec), Canada H3A 0C3
E-mail: mengying.lei@mail.mcgill.ca, lijun.sun@mcgill.ca