POLITECNICO
MILANO 1863

Master's Degree in Mathematical Engineering
Advanced Programming for Scientific Computing
(8 CFU course project)

Translation into C++ of an R package

"Intervalwise Testing for Functional Data"

Tutor:
Dr. Alessia Pini

Student:
Diego Di Maulo
875397

May 2018

Contents

1 Abstract

The goal of this project is to translate from R into C++ an algorithm that solves intervalwise T^2 testing problems, which has been coded only for checking its theoretical consistency, without caring of its computational efficiency, so that it is of very little use when the dimensions grow, due to the extremely long execution times. As a consequence, our target is also to optimize the code as much as possible, by taking advantage of some advanced programming techniques of C++, such as the `Eigen` library and the OpenMP parallelization. Speeding this code up is crucial also because it relies on Monte Carlo methods, which require the larger possible number of iterations to come up with good results. In the end, we managed to reduce the execution times by two orders of magnitude, thus allowing the use of the algorithm also for any kind of application.

2 Introduction

When performing a statistical test, we have a null hypothesis H_0 and an alternative hypothesis H_1. Usually, a statistical test aims at rejecting the statement in H_0 in favour of the statement in H_1, by providing statistical evidence against H_0. The key quantity to discriminate between H_0 and H_1 is the p-value, a real number between 0 and 1. The lowest the p-value of a test is, the more evidence against H_0 there is. Common thresholds for determining whether a p-value is low or high are 0.10, 0.05, and 0.01 but they can vary according to the nature of the application. However, when we deal with functional data, such statistical tests are not that easy any longer.

Intervalwise testing with functional data is a procedure to find the portions of the domain that cause the null hypothesis H_0 to be rejected. To this extent, Alessia Pini and Prof. Simone Vantini implemented an R package ([1]) that encompasses several functions, the most important of which being `IWT2` (standing for IntervalWise T^2 test). The procedure followed by the `IWT2` algorithm can be divided into these five steps:

1. data pre-processing,
2. computation of the pointwise test statistic on the original data,
3. computation of the pointwise test statistic on the permuted data,
4. intervalwise test statistics,
5. corrections.

The goal of our project is to prosecute Alessia Pini and Simone Vantini's work and to implement their algorithm in C++ in order to make it more efficient and parallelizable. The current R version of `IWT2`, indeed, is only a draft to check whether the procedure studied in theory actually works and it does not rely on a very efficient design, thus resulting pretty slow and almost unusable in most applications.

3 Description

3.1 R version

The original R function, called `IWT2`, accepts a number of parameters defining some options about the algorithm execution and returns a list with some information about the test and its main results. The inputs that the function accepts are:

- `data1`: an $n1 \times p$ matrix, whose rows represent the observations coming from the first population and whose columns represent the points of the domain;
- `data2`: an $n2 \times p$ matrix, whose rows represent the observations coming from the second population and whose columns represent the points of the domain;
- `mu`: a p-dimensional vector representing the mean difference of the two populations under the null hypothesis H_0 at each point of the domain;
- `B`: a positive integer indicating the number of Monte Carlo iterations to perform;
- `paired`: a flag indicating whether the test should be "paired" or "unpaired"[1];
- `dx`: the step size of the domain;
- `recycle`: a flag indicating whether to use the "recycle" or not[2];
- `alt`: a string indicating the test type (or alternative) to perform.

Note that the function accepts `data1`, `data2`, and `mu` also in form of `fd` objects, i.e. in form of functional data objects, a special class of objects defined in the R package `fda`. The step size parameter `dx` is required only if `data1`, `data2`, and `mu` are given as `fd` objects, otherwise it can be ignored. The string `alt` determines the type of statistical test to perform among the following alternatives:

- "two.sided": H_0: $\mu_1 - \mu_2 = \mu_0$ vs H_1: $\mu_1 - \mu_2 \neq \mu_0$,

[1] A test is "paired" if the observations of the two populations are coupled (so $n1$ and $n2$ must be equal).

[2] Typically, this option is activated when the domain is periodic. We will explain it more in detail later on.

– "greater": H_0: $\mu_1 - \mu_2 = \mu_0$ vs H_1: $\mu_1 - \mu_2 > \mu_0$,

– "less": H_0: $\mu_1 - \mu_2 = \mu_0$ vs H_1: $\mu_1 - \mu_2 < \mu_0$.

After receiving the inputs, the function pre-processes the data, it computes the pointwise test statistics both on the original data and on the permuted data (for `B` permutations), it computes the intervalwise test statistics (p^2 values for `B` permutations), and it finally computes the p-values corrections, that is, it corrects the p-values to make them theoretically sound by adjusting them for multiplicity. In the end, the output is a list with the following objects:

– `test`: a string indicating the test type;

– `mu`: the p-dimensional vector that was given in input;

– `unadjusted_pval`: the p-dimensional vector of the pointwise unadjusted p-values;

– `adjusted_pval`: the p-dimensional vector of the pointwise adjusted p-values;

– `pval_matrix`: the $p \times p$ matrix of the intervalwise p-values;

– `data.eval`: an $(n1 + n2) \times p$ matrix containing `data1` and `data2`;

– `ord_labels`: an $(n1 + n2)$-dimensional vector indicating the population to which each observation belongs.

3.2 C++ version

The first step is to translate the existing code from R into C++ in order to be able to enhance its performances, by taking advantage of the `Eigen` library and of some parallelization techniques.

Inputs The inputs in the C++ version are pretty much the same as those in the R version but without the step size `dx` and with a couple of new useful ones:

– `data1`, `data2`: observations of the two populations,

– `mu`: mean difference under H_0,

– `B`: number of Monte Carlo iterations,

– `alt`: test type or alternative,

– `maxrow`: truncation parameter,

– `paired`: flag for "paired" or "unpaired" test,

– `recycle`: flag for using "recycle" (for periodic domains),

– `THREADS`: number of parallel threads to exploit.

Outputs Many outputs of the R version are actually useless since they contain the same information given as input, so in the C++ version we return only

- `T0`: vector of the T^2 statistics of the original data,
- `pvalue_point`: pointwise unadjusted p-values,
- `pvalue_inter`: intervalwise p-values,
- `pvalue_corr`: pointwise adjusted p-values.

Libraries Needless to say, we need some basic libraries such as `iostream`, `fstream`, `vector`, and `string`, together with the `Eigen/Dense` library (we used the version 3.3.3), which will be of help for many operations on matrices and vectors. Moreover, we also need the libraries `ctime` and `iomanip` since we want to be able to clock and display the algorithm execution times along the way.

Macros We define three binary macro variables, `INFO`, `SHOW`, and `TIME`, for enabling or suppressing the information display of the code flow, the partial results, and the execution times, respectively.

Definitions We want to give the opportunity to easily change the data types of some crucial variables in the implementation phase and in possible future revisions. We may desire to change the data type used for matrices and vectors, especially for the two input matrices `data1` and `data2`, as well as the data type used for the variable indicating the test type (or alternative). For the alternative variable, we use a string in order to be consistent with the original R code and in order to make the code more readable. For matrices and vectors, we use the dynamic-size data types defined by `Eigen` in order to be able to perform important operations efficiently. Moreover, we want all matrices to be stored with the `RowMajor` option because most of the times we consider them row-by-row. Later on, we will see that this option speeds the code up by a factor of almost 3. We now show the code excerpt relative to the topics just discussed above and we will proceed like this in the entire report. We just want to mention that most comments and explanations (which do compare in the actual C++ code) have been cut out from the excerpts that we show throughout the report, so to shorten and relieve the notation.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <Eigen/Dense>
#include "GetPot"
#include <ctime>
#include <iomanip>
```

```
#define INFO true // Print info on the code flow
#define SHOW true // Print info on the partial results
#define TIME true // Print info on the elapsed times

using namespace Eigen;

typedef Array<double, Dynamic, Dynamic, RowMajor> MatrixType;
typedef Array<double, Dynamic,    1> VectorType;
typedef std::string AlterType;  // alternative
```

Data reading Now we need to read the inputs `data1`, `data2`, and `mu` and to store them into suitable variables. For the moment, we assume them to be available in separate text files, together with their dimensions. Later on in this report, we will also illustrate a method that automatically generates these data, thus making the entire algorithm independent from any external file. The four text files are

- `Param.txt`, containing the three dimensions $n1$, $n2$, and p,
- `Data1.txt`, containing the $n1 \times p$ elements of `data1` (by row),
- `Data2.txt`, containing the $n2 \times p$ elements of `data2` (by row),
- `Mean0.txt`, containing the p elements of `mu`.

```
// Read dimensions
int n1, n2, p;
std::ifstream Param("Param.txt", std::ifstream::in);
Param >> n1 >> n2 >> p;
Param.close();

// Read data1
MatrixType data1(MatrixType::Zero(n1,p));
std::ifstream Data1("Data1.txt", std::ifstream::in);
for (int i = 0; i < n1; i++)
for (int j = 0; j < p; j++)
Data1 >> data1(i,j);
Data1.close();

// Read data2
MatrixType data2(MatrixType::Zero(n2,p));
std::ifstream Data2("Data2.txt", std::ifstream::in);
for (int i = 0; i < n2; i++)
```

```
for (int j = 0; j < p; j++)
Data2 >> data2(i,j);
Data2.close();

// Read mu
VectorType mu(VectorType::Zero(p));
std::ifstream Mean0("Mean0.txt", std::ifstream::in);
for (int j = 0; j < p;  j++)
    Mean0 >> mu(j);
Mean0.close();
```

Tilde test We have to mention that the algorithm does not perform the test

$$\mathrm{H}_0 : \mu_1 - \mu_2 = \mu_0 \text{ vs } \mathrm{H}_1 : \mu_1 - \mu_2 \neq \mu_0,$$

but it actually performs

$$\mathrm{H}_0 : \tilde{\mu_1} = \mu_2 \text{ vs } \mathrm{H}_1 : \tilde{\mu_1} \neq \mu_2,$$

where $\tilde{\mu_1} = \mu_1 - \mu_0$. Of course, this holds analogously also for tests of type "greater" and "less". This is performed for many future operations to result much simpler to handle. Hence, as soon as we read the data, we subtract the vector `mu` from each observation of the first population, i.e. from each row of `data1`.

Inputs check This algorithm requires a number of inputs to define some execution options - such as the number of Monte Carlo iterations, the permutations nature, and the domain structure - and we need to check all these inputs. First of all, the matrices `data1` and `data2` must have the same number of columns and - in case of a "paired" test - they also need to have the same number of rows. As already mentioned, the R function accepts `data1` and `data2` both as numerical matrices and as functional data objects; nevertheless, in the latter case, they need to be converted into numerical matrices, so we design the algorithm by assuming `data1` and `data2` to be always numeric matrices. Thanks to this, we no longer need the input parameter `dx`, required only in presence of `fd` objects. The same holds for the vector `mu`, which must be a numeric vector of p components.

Besides, the string `alt` must be equal to one alternative among "two.sided", "greater", and "less", the iterations number `B` must be a positive integer, and the truncation parameter `maxrow` must be a non-negative integer strictly smaller than p. Note that for some variables like p, `B`, and `maxrow`, we prefer the *int* type with respect to the *unsigned* type - which would be more appropriate for them - in order to avoid annoying warnings raised by the compiler (with the `-Wall` option) when comparing *int* and *unsigned* variables and also in order to have safer exit conditions in the for cycles with decreasing dummy variable.

Computation of T0 After all useful information has been loaded and checked, the computations can begin. The function `compute_T2` is devoted to the computation of the T^2 statistic values and its prototype is

`VectorType compute_T2 (const VectorType &, int, const AlterType &).`

If `delta` is the p-dimensional vector of the mean differences of the two populations at each point of the domain (i.e., the column-by-column mean differences between `data1` and `data2`), then - in case of "two.sided" tests - the T^2 value equals the scalar product of `delta` with itself. On the other hand, when performing a test of type "greater" (or "less"), only the positive (or negative) elements of `delta` must be taken into account. For more details about the construction and the rationale underneath such values, we refer to [1].

```
VectorType compute_T2 (const VectorType & delta, int p,
                       const AlterType & alt) {

VectorType T(delta);

if (!alt.compare("greater")) {
    for (int j = 0; j < p; j++)
        if (delta(j) < 0) T(j) = 0;
}
else if (!alt.compare("less")) {
    for (int j = 0; j < p; j++)
        if (delta(j) > 0) T(j) = 0;
}

return T * T;
}
```

Pointwise p-values The next step is to compute the traditional pointwise p-values, which we compute by taking advantage of the so-called "two-population" permutation tests. Assuming a dataset to consist of observations coming from two different populations (as it is in our case), a permutation test aims at estimating the distribution of the test statistic under H_0, by randomizing the labels of the observations and by computing the value of the test statistic under each new random configuration. By comparing the T^2 values of the permutation tests with T0, we can estimate the p-value at each point of the domain. The more random permutations we generate, the more precise the Monte Carlo estimation results. The default permutations number `B` is just 1000, but in applications it can assume much larger values. So, we generate the $B \times p$ matrix `T_perm` to store the T^2 values of the `B` permutation tests conducted at each of the p points.

Paired and unpaired tests We have to distinguish between the two different ways of permuting the observations of the two populations. In case of a "paired" test, the numerosity of both populations must be the same (i.e., it must hold $n1 = n2 = n$) and the permutations are performed by generating a random binary sequence of n values and by "virtually" exchanging the observations of the first population with those of the second population when

the random sequence is equal to 1. We use the expression "virtually" because, in practice, the observations are not really exchanged (it would be a waste of time and memory), we just take note of the observations that have been selected, by means of an *int* vector called `indices`. Then, we just need to compute the T^2 values, by taking into account that some observations have been relabelled.

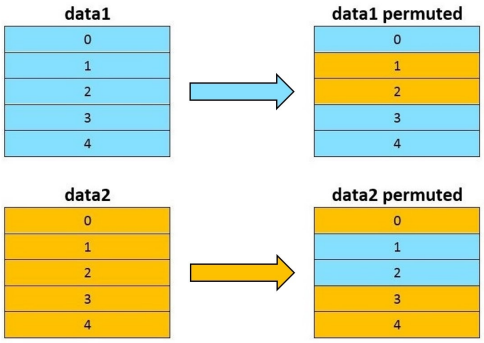
```
// Declare vector
std::vector<int> indices(n1);

// Generate random binary sequence
for (int i = 0; i < n1; i++)
    indices[i] = rand() % 2;
```

On the other hand, if the test is "unpaired", there are no requirements on the populations numerosity, and the sampling is simpler: we just randomly reorder the $n1 + n2$ observations of both populations.

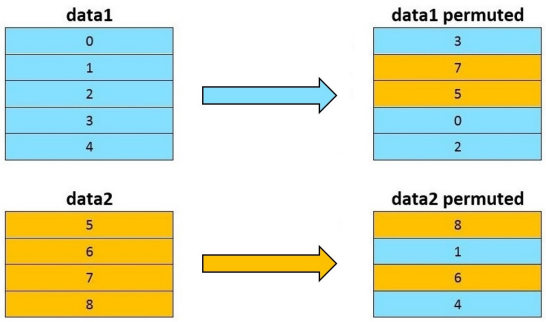
```
// Declare variables
std::vector<int> indices(n1+n2);
bool ok;
int k;

// Sample integers from 1 to n1+n2
for (int i = 0; i < (n1+n2); i++) {
    indices[i] = rand() % (n1+n2);
    if (i > 0) {
        ok = false;
        while (ok == false) {
            k = 0;
            while (indices[i] != indices[k]) k++;
            if (k < i) indices[i] = rand() % (n1+n2);
            else ok = true;
        }
    }
}
```

In the figure below, we illustrate an example of how the two samplings work. The example on the left shows the "paired" sampling resulting from the vector `indices` $= (0, 1, 1, 0, 0)$, indeed we see that only the second and third element are exchanged; the example on the right shows the "unpaired" sampling resulting from the vector `indices` $= (3, 7, 5, 0, 2, 8, 1, 6, 4)$, indeed the observations are just reordered.

Regardless of whether the test is "paired" or "unpaired", we have to generate a random sampling and consequently compute the resulting mean differences `B` times. After that, we just need to find - for each point of the domain - the fraction of permutation test statistics T^2 that are greater than or equal to the corresponding test statistic T0. This fractions



(a) Example of "paired" permutation.



(b) Example of "unpaired" permutation.

correspond to the pointwise p-values. In order to perform that, we create the p-dimensional vector `count` to tally up how many times the T^2 values are higher than or equal to T0 at each point. Note that, although the vector `count` assumes only integer non-negative values, we prefer to implement it as a vector of *double* in order to be able to carry out some operations more easily in the following sections. Besides, its implementation as vector of *unsigned* does not even bring relevant performance improvements, so we do not believe there is need to go for that.

```
// Initialize variables
MatrixType T_perm(MatrixType::Zero(B,p));
VectorType count(VectorType::Zero(p));

// Permutation tests
for (int b = 0; b < B; b++) {
    double temp1, temp2;
    VectorType delta(VectorType::Zero(p));

    // Generate random sample...

    // Compute mean differences...

    T_perm.row(b) = compute_T2(delta, p, alt);
    for (int j = 0; j < p; j++)
        if (T_perm(b,j) >= T0(j))
            count(j) += 1;
}

// Compute p-values
VectorType pvalue_point(count/B);
```

Intervalwise p-values We now enter the most demanding part of the algorithm, which is particularly hard from a computational point of view because we have to find a p-value for every possible subinterval of the domain. So, for each subinterval, we have to take the sum of the permutation test statistics previously computed at all its points, compare this sum with the sum of the corresponding T0 values, and count how often the former sum is greater than or equal to the latter one. After performing this comparison for all `B` permutations, i.e. for all `B` rows of the matrix `T_perm`, we estimate the intervalwise p-value of each subinterval by dividing the resulting total count by `B`.

In order to store these p-values, we create the $p \times p$ matrix `pvalue_inter`, whose row i contains the p-values of the subintervals consisting of $p - i$ consecutive points and whose column j contains the p-values of the subintervals that start from point j. Thus, the element (i, j) of `pvalue_inter` is the p-value of the subinterval consisting of $p-i$ points starting from point j.

Recycle In the algorithm version with "recycle", we end up with a full squared matrix since we can build subintervals of any length starting from any point due to the periodicity of the domain (see Figure 2). Conversely, in the version without "recycle", we end up with a lower triangular matrix since the number of feasible subintervals decreases when their length increases: the subintervals of p points can only start from point 0, the subintervals of $p - 1$ points can only start from point 0 or 1, and so on (see Figure 3). Because of this, the values in the upper-triangular part of the matrix are not defined and they always remain equal to 0 (they will be set equal to `NaN` after interfacing the algorithm with R, so to simplify future graphical features). Again, for a more consistent discussion on that we refer to [1].

	0	1	2	3	4	5
0	012345	123450	234501	345012	450123	501234
1	01234	12345	23450	34501	45012	50123
2	0123	1234	2345	3450	4501	5012
3	012	123	234	345	450	501
4	01	12	23	34	45	50
5	0	1	2	3	4	5

Figure 2: Subintervals and their points (with "recycle").

At this point, the original R code creates the $2p$-dimensional vector T02 and the $p \times 2p$ matrix `T_perm2`, by juxtaposing T0 and `T_perm` to themselves, so to simplify some future block-operations on them. This duplication is actually useless and it produces a significant waste of memory. For instance, supposing to run an algorithm with `B` $= 100.000$ iterations on a domain with $p = 1.000$ points, the matrix `T_perm` would have $p^2 = 10^8$ elements and

0	0 1 2 3 4 5	———	———	———	———	———
1	0 1 2 3 4	1 2 3 4 5	———	———	———	———
2	0 1 2 3	1 2 3 4	2 3 4 5	———	———	———
3	0 1 2	1 2 3	2 3 4	3 4 5	———	———
4	0 1	1 2	2 3	3 4	4 5	———
5	0	1	2	3	4	5
	0	1	2	3	4	5

Figure 3: Subintervals and their points (without "recycle").

its dimension would be about 800 MB. We want to avoid this waste, so we rely only on the already existing variables without performing any duplication, by carrying out some tricky operations on the matrix indices.

In the code excerpt below, we see how the code cycles on all possible subintervals: the outer for cycle considers the subintervals lengths, from the shortest one to the longest one; the inner for cycle considers the subintervals starting points, from the first one to the last one. Then, the p-values are computed by counting how often T^2 is greater than or equal to T0.

```
// Cycle on subinterval length
for (int i = p - 2; i >= maxrow; i--) {

    int len (p - i);
    unsigned cont;
    double T0_temp, T_temp;

    // Cycle on subinterval starting point
    for (int j = 0; j < p; j++) {

        if (j + len > p)
            T0_temp = T0.tail(p-j).sum()
                    + T0.head(len-(p-j)).sum();
        else
            T0_temp = T0.segment(j,len).sum();

        cont = 0;

        // Cycle on permutations
        for (int b = 0; b < B; b++) {
```

```
            if (j + len > p)
                T_temp = T_perm.row(b).tail(p-j).sum()
                       + T_perm.row(b).head(len-(p-j)).sum();
            else
                T_temp = T_perm.row(b).segment(j,len).sum();

            if (T_temp >= T0_temp)
                cont++;
        }

    // Compute p-values
    pvalue_inter(i,j) = cont / double(B);

    }
}
```

Note that in the code above the dummy variable j cycles on all points of the domain, from 0 to $p-1$ but this happens only in the "recycled" version. In the version without "recycle", in facts, j cannot proceed until the last point $p-1$ due to the non-periodicity of the domain so that the exit condition is $j \leq i$, instead of $j < p$, and so that both if-else statements skip directly to the else-clause because $j + \texttt{len} = j + (p-i) = p + (j-i)$ is never strictly greater than p.

Truncation parameter The truncation parameter `maxrow` is a non-negative integer indicating until which row of the matrix `pvalue_inter` we should go on with the computations. It can assume values from 0 to $p-1$, included, and its default value is 0, which means that we can proceed until the first row of `pvalue_inter`, i.e. until the subinterval that coincides with the entire domain. On the other hand, if `maxrow` is equal to another value, then the computations must stop at the `maxrow`-th row of `pvalue_inter`, i.e., when the length of the subintervals is $p - \texttt{maxrow}$. Note that if `maxrow` is equal to $p-1$ the computation of the intervalwise p-values does not even start (since the exit condition of the first cycle is never verified), so we basically take into account only the subintervals of length $p-(p-1) = 1$, i.e., only the single points.

Corrections Now we want to adjust (or correct) the pointwise p-values in order to obtain more robust results. The adjusted (or corrected) p-value at a given point is the maximum among its unadjusted pointwise p-value and the intervalwise p-values of all subintervals that contain that point. The subintervals containing a point, say, j are those whose p-value in the matrix `pvalue_inter` lies in the "upper-left triangle" whose lower vertex is identified by the j-th element of the last row. The two figures below should help to understand such configuration: the highlighted cells of the table represent the so-called "upper-left triangle" build upon point 3. The first figure (Figure 4) refers to the algorithm version with "recycle",

in facts, the triangle continues on the upper-right side of the table, by passing from the first column to the last one, as if the domain were circular. The second figure (Figure 5) refers to the version without "recycle", in facts, the triangle is cut on the first column and it does not continue on the last column. As always, for further details see [1].

0	**012345**	**123450**	**234501**	**345012**	**450123**	**501234**
1	**01234**	**12345**	**23450**	**34501**	45012	**50123**
2	**0123**	**1234**	**2345**	**3450**	4501	5012
3	012	**123**	**234**	**345**	450	501
4	01	12	**23**	**34**	45	50
5	0	1	2	**3**	4	5
	0	1	2	**3**	4	5

Figure 4: Subintervals involved in the correction of point 3 (with "recycle").

0	**012345**	———	———	———	———	———
1	**01234**	**12345**	———	———	———	———
2	**0123**	**1234**	**2345**	———	———	———
3	012	**123**	**234**	**345**	———	———
4	01	12	**23**	**34**	45	———
5	0	1	2	**3**	4	5
	0	1	2	**3**	4	5

Figure 5: Subintervals involved in the correction of point 3 (without "recycle").

Also here, as well as in the intervalwise p-values section, the R code duplicates the matrix `T_perm` (when the `recycle` option is activated) and, again, we avoid such waste of time and memory by playing somehow with the matrix indices. Besides, the R code also generates a $p \times p$ matrix to compute the adjusted p-values but this is actually useless since in the end we only need a single row of it, the one of index `maxrow`. To overcome such inefficiency, we implement a method that relies just on a p-dimensional vector and that, instead of using p new elements every time, overwrites always the same ones.

The code below is the implementation of the corrections method. In case the "recycle" is not activated, the dummy variable j starts from the diagonal element i (and not from the last element $p-1$) because `pvalue_inter` is a lower triangular matrix and half of its elements

are not defined. Moreover, the value `pvalue_corr(p-1)` (saved in the variable `temp_last` for convenience) does not need to be considered due to the non-periodicity of the domain.

```
// Cycle on subintervals length
for (int i = maxrow; i < p; i++) {

    if (recycle==true) {

        // Cycle on points
        for (int j = p - 1; j >= 0; j--) {

            if (j==p-1) temp_last = pvalue_corr(p-1);

            if (i==maxrow)
                pvalue_corr(j) = pvalue_inter(i,j);
            else if (j==0)
                pvalue_corr(j) = max(pvalue_inter(i,j),
                                     max(temp_last,
                                         pvalue_corr(j)));
            else
                pvalue_corr(j) = max(pvalue_inter(i,j),
                                     max(pvalue_corr(j-1),
                                         pvalue_corr(j)));
        }
    }
    else {
        // Cycle on points
        for (int j = i; j >= 0; j--) {

            if (j==0)
                pvalue_corr(j) = max(pvalue_inter(i,j),
                                         pvalue_corr(j));
            else if (i==j)
                pvalue_corr(j) = max(pvalue_inter(i,j),
                                         pvalue_corr(j-1));
            else
                pvalue_corr(j) = max(pvalue_inter(i,j),
                                     max(pvalue_corr(j-1),
                                         pvalue_corr(j)));
        }
    }
}
```

3.3 Parallelization

The next step is that of parallelizing the algorithm. In order to achieve that, we take advantage of the OpenMP language. We prefer it to the most common one, MPI, because OpenMP is more appropriate for our problem. In particular, the intervalwise T^2 testing algorithm starts from two large matrices, `data1` and `data2`, and it performs several operations on both matrices in their entirety, as any Monte Carlo procedure. In MPI, we would have to broadcast all values of both matrices from the master to all nodes, while in OpenMP we can avoid this because we work in shared memory (more details on OpenMP can be found in [3]).

The more demanding sections are the pointwise and the intervalwise p-values computations. Concerning the pointwise p-values computation, we can easily parallelize it by equally splitting the `B` permutations among the threads. Concerning the intervalwise p-values computation, which is much more time-consuming than the pointwise one, it is wiser to parallelize the for-loop that cycles over the rows of `T_perm`. All what we need to do in order to perform an OpenMP parallelization is add the instruction `#pragma omp parallel for` just before the beginning of the for-cycle that we want to parallelize and, possibly, we can also specify some additional options. In the pointwise p-values computation, we take advantage of the `schedule(static)` option, so that the `B` permutations are divided among the threads into equal-sized chunks (or as equal-sized as possible, in case `B` is not evenly divisible by the number of threads). Conversely, in the intervalwise p-values computation, we take advantage of the `schedule(dynamic)` option: the rows of `T_perm` are - ideally - put in a work queue and every thread, as soon as it is ready, receives the first row of the queue and begins the computations on that. The reason why we use two different schedules is that in the first case all iterations have the same complexity, while in the second case the iterations become longer and longer as we proceed from the row $p-2$ to the row `maxrow`, because the first iterations handle short subintervals (of length 2, 3, 4, and so forth) while the last iterations handle long subintervals (of length up to $p-2$, $p-1$, p).

3.4 Tests

Now that the algorithm has been completely implemented in C++, before interfacing it with R, we want to test it.

Inputs Considered the rather large number of input parameters that we have to pass to the function, we take advantage of the library `GetPot`. The data matrices `data1` and `data2` and the mean vector `mu` can be either generated within the algorithm or assumed to be stored in text files. We have already discussed earlier about how to read the data from external text files, so we now just illustrate how they can be generated directly by the algorithm. In order to do that, we just need to provide the data dimensions $n1$, $n2$, and p from the command line, then the algorithm does everything else: it creates the random matrices `data1` and `data2` by taking advantage of the function `MatrixType random_matrix`, which returns an $n \times p$ matrix whose values in the first half of the domain are jittered around the value `m1` and

whose values in the second half are jittered around the value `m2`, and it also generates the vector `mu` by taking advantage of the function `VectorType constant_vector`, which returns a p-dimensional vector equal to `m1` in the first half of the domain and equal to `m2` in the second half. For `m1` and `m2` we select specific values that provide an informative enough output. Since they do not play an important role from the implementation point of view, we do not let them be modified from the command line for simplicity.

```
MatrixType random_matrix (int n, int p, double m1, double m2) {

    MatrixType data(MatrixType::Random(n,p));

    data.block(0, 0,           n, floor(p/2.)) += m1;
    data.block(0, floor(p/2.), n,  ceil(p/2.)) += m2;

    return data;
}

VectorType constant_vector (int p, double m1, double m2) {

    VectorType mu(VectorType::Zero(p));

    mu << VectorType::Constant(floor(p/2.),m1),
          VectorType::Constant(ceil(p/2.),m2);

    return mu;
}
```

Output Below, we show an output example. Note that, before compiling and launching the program, we have to load the `Eigen` library and indicate the number of threads to reserve.

```
$ module load eigen
$ export OMP_NUM_THREADS=1
$ make
$ ./main n1=50 n2=50 p=10

*** Inputs check ***
data1 and data2: OK
mu: OK
B: OK
maxrow: OK
alternative: OK
THREADS: OK
```

```
*** T0 ***
0.038 0.001 0.004  0.01 0.006 4.297 4.413 3.538 3.783 3.876

*** Pointwise p-values ***
0.082 0.764 0.583 0.433 0.493     0     0     0     0     0

*** Interval-wise p-values ***
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
0.489     0     0     0     0     0     0     0     0     0
0.426 0.823     0     0     0     0     0     0     0     0
0.344 0.779 0.707     0     0     0     0     0     0     0
0.208 0.817 0.623 0.566     0     0     0     0     0     0
0.082 0.764 0.583 0.433 0.493     0     0     0     0     0

*** Corrections ***
0.489 0.823 0.823 0.823 0.823     0     0     0     0     0

*** Elapsed time ***
Read:        1 ms
T0:          0 ms
Point:     252 ms
Inter:      25 ms
Corr:        0 ms
TOTAL:     278 ms
```

3.5 R interface

At this point, the algorithm is ready to be interfaced with R. In order to take advantage of a C++ parallel function from the R environment, we need the packages `RcppEigen` and `omp` and we have to add a easy few instructions to make the function available in R (for more information on how to interface a function from C++ to R and much more, see [2]). Finally, we want the function to return a `List` variable, called `RES`, with all useful results: the T0 statistics, the pointwise p-values, the intervalwise p-values, and the corrected p-values. Now we just need to execute `sourceCpp('IWT2OMP.cpp')` in the R to have the algorithm loaded into the R environment as a regular function of prototype

`List IWT2OMP (int, AlterType, int, bool, bool, int).`

```
#include <RcppEigen.h>
#include <omp.h>
// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::plugins(openmp)]]

// Includes...
// Macros...
// Typedefs...

//[[Rcpp::export]]
List IWT2OMP (int B = 1000,
              AlterType alt = "two.sided",
              int maxrow = 0,
              bool paired = false,
              bool recycle = false,
              int THREADS = 1) {

    // Algorithm...

    List RES;
    RES["T0"] = T0;
    RES["point"] = pvalue_point;
    RES["inter"] = pvalue_inter;
    RES["corre"] = pvalue_corr;

    return RES;
}
```

4 Results

After having coded everything and having ensured the correctness of the computations, we want to analyse the algorithm performances, by inspecting the impact of each parameter on the total execution time and by measuring how long each section takes. Then, we also want to check whether there is a real performance improvement in our C++ version of the algorithm with respect to the original one, which is the main goal of this project.

4.1 Parameters

To understand how each parameter influences the total execution time, we run the algorithm several times by varying their values. If not differently specified, we use the following default values for the parameters:

- $n = 100$,
- $p = 200$,
- `B` $= 10000$,
- `maxrow` $= 0$,
- `paired` $=$ `true`,
- `recycle` $=$ `true`,
- `alt` $=$ `"two.sided"`,
- `THREADS` $= 1$,

and we change their values one by one in order to measure the variation in the total execution time. In the tables below, we illustrate some interesting results.

The populations numerosity $n1 = n2 = n$ does not affect the execution times, in facts, the algorithm takes almost the same time independently from the value of n. This happens because the only computational difference when n increases is that the column-by-column means involve a greater number of elements, but the global complexity does not change. Actually, by increasing n more and more (up to, say, 1000) the execution times do begin to

slightly raise, but that is just due to the fact that reading larger and larger matrices from text files becomes demanding. On the other hand, the domain dimension p has a strong impact on the performances. By increasing p, in facts, we increase the number of points, so the number of subintervals, and adding a subinterval means adding a lot of operations, even repeated `B` times in some cases. As a consequence, a linear increase of p gives rise to an exponential increase of the total execution times.

n	p	Time (seconds)
50	100	5
100	100	5
200	100	5
50	200	47
100	200	47
200	200	47
50	500	408
100	500	409
200	500	410

Table 1: Execution times when varying n and p.

Closely related to p, the truncation parameter `maxrow` also affects the execution time in an (approximatively) exponential way; nevertheless, conversely to p, it reduces the global execution time since it makes the algorithm neglect a number of subintervals (and, in particular, the longer and most demanding ones), thus significantly lowering the iterations of the intervalwise p-values computation.

`maxrow`	Time (seconds)
0	15
20	12
100	5

Table 2: Execution times when varying `maxrow`.

As we expect, the number of Monte Carlo iterations `B` linearly influences the execution times since it indicates how many times the algorithm should randomly permute the observations and compute their T^2 value. 1000 iterations are probably too few for most applications, while 10000 are already a pretty good value. We could go even further and iterate much more times but this would be of little use unless we want to have an extremely accurate p-values in correspondence of the threshold values.

Regarding the two flag parameters `paired` and `recycle`, the former one does not impact on the execution time since it just defines the sampling method to use and both methods are

B	p	Time (seconds)
1000	200	2
5000	200	17
10000	200	62
1000	500	20
5000	500	101
10000	500	406

Table 3: Execution times when varying `B` and p.

almost equivalent (or, anyway, their computational difference is definitely negligible); on the other hand, the parameter `recycle` has a significant impact: when it is `false`, the code is averagely 3 times faster than when it is `true`. This happens because, in the version without `recycle`, the algorithm takes into account a much lower number of subintervals (almost half of them) and, in particular, it neglects most of the longer ones. More precisely, it takes into account all p subintervals of length 1, $p-1$ subintervals of length 2, $p-2$ subintervals of length 3, and so on, until taking into account only 1 subinterval of length p (see Figure 3).

`paired`	`recycle`	Time (seconds)
`true`	`true`	20
`false`	`true`	20
`true`	`false`	7
`false`	`false`	7

Table 4: Execution times when varying `paired` and `recycle`.

Finally, the test type does not affect the performance since the only difference concerns the computation of T^2: if the test is of type "greater" or "less", some components of the vector `delta` must be set equal to 0 in order to be ignored.

`alternative`	p	Time (seconds)
`"two.sided"`	200	15
`"greater"`	200	15
`"less"`	200	15
`"two.sided"`	500	412
`"greater"`	500	403
`"less"`	500	407

Table 5: Execution times when varying `alternative` and p.

4.2 Sections

By analysing the execution times of the various sections of the algorithm, we find out that the majority of the time (more than 95%) is required for the intervalwise p-values computation, whereas only a very small percentage is dedicated to the pointwise p-values computation (less than 1%) and to the data reading (up to 5%). We do not provide any additional table because they would be of little interest and they are slightly dependent on the machine, above all the data reading time. We just mention that the execution times of the data reading and of the pointwise p-values computation strongly depend on the domain cardinality p, while the intervalwise p-values computations is affected also by the truncation `maxrow`.

4.3 R vs C++

We now compare the execution times of the original R function with those of our C++ version since the aim of this project is exactly to obtain a faster and more efficient algorithm.

After running both algorithms many times[3] by varying all input parameters, we observe that the C++ version generally takes 20 times less than the R version: even without taking advantage of parallel computing, the new algorithm requires only 5% of the time with respect to the old one. In the table below, we list the results of some bilateral paired two-population tests with $n1 = n2 = 100$ and `maxrow` $= 0$ running on a single thread.

p	`B`	`recycle`	**C++ (seconds)**	**R (minutes)**	**Ratio**
500	1000	`true`	20	9	**4%**
500	5000	`true`	101	48	**3%**
500	10000	`true`	406	100	**7%**
200	10000	`true`	17	10	**3%**
200	10000	`false`	9.6	2	**8%**

4.4 Parallelization

We can finally take advantage of the OpenMP parallelization and exploit many threads. After running our algorithm on the *Gigat* queue of the cluster located at Dipartimento di Matematica of Politecnico di Milano, we observe not only that the execution times have reduced to the 5% when using one thread, but also that they significantly decrease when exploiting more threads.

In the figure below, we report the execution times of a sample problem solved with an increasing number of threads, from 1 to 20. We are satisfied with the parallelization since the execution times significantly drop until six threads (from 7 to 1.5 minutes), then they still decrease but more slightly.

All in all, our new version of the algorithm can provide the results of intervalwise T^2 tests for functional data in less than 1% of the time than the original algorithm.

[3] On a laptop with Windows 8.1, with an i7-5500U CPU @ 2.40 GHz, and with 8GB of RAM.

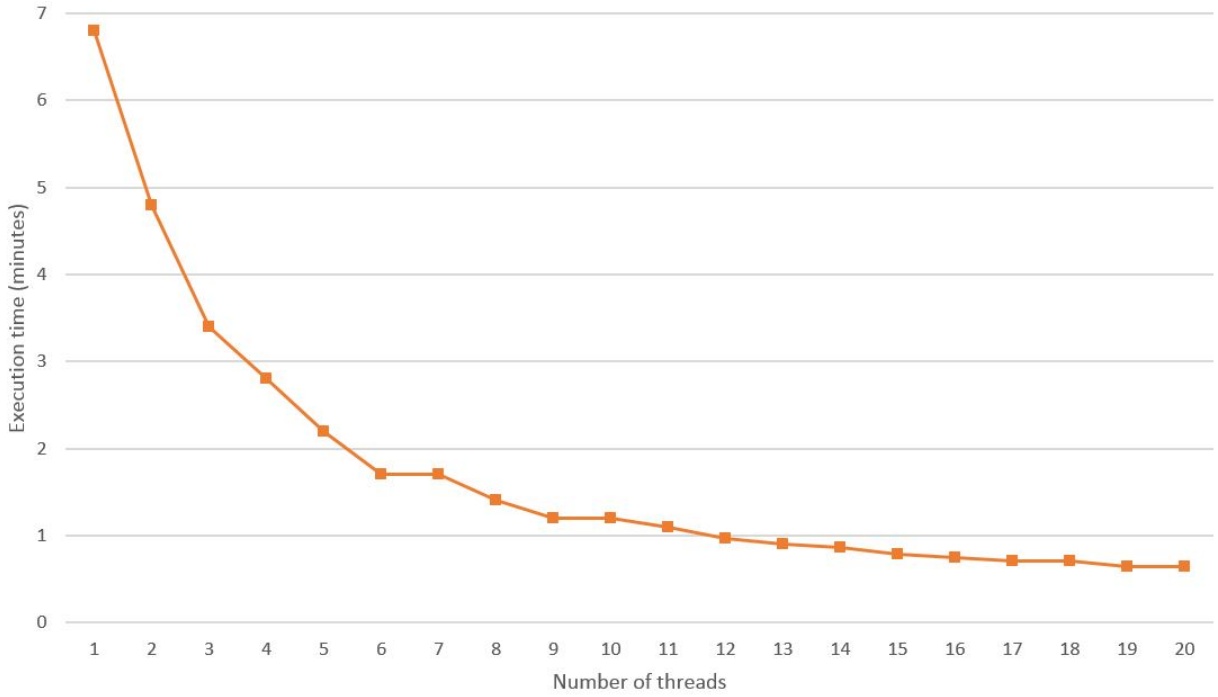


Figure 6: Execution times (in minutes) of a sample problem with increasing threads.

5 Conclusions

We started from a very naive implementation in R of the `IWT2` algorithm, which tackles intervalwise T^2 tests for functional data, and we ended up with a more robust and much faster implementation of that in C++. Apart from translating the code from one language into the other, we enhanced many aspects of it: we wrote it in a more linear and easier to read way, we optimized matrix operations, we eliminated unnecessary variables, we avoided useless duplications of some large matrices, and we parallelized the most demanding for-cycles. The new version of the algorithm averagely takes just 5% of the time with respect to the original one; thus, a problem that used to take 3 hours will now take just 9 minutes and, by exploiting a handful of threads, it can be solved even in just 2 minutes.

Regarding possible extensions of the code, we mention that the same procedure can be adapted to more general domains than the unidimensional and equispaced one considered up to now. For instance, one may want to test functions defined on bidimensional domains, that is, on rectangular grids of points instead that on sequences of points. This may be rather challenging because the subintervals would become subdomains and their numerosity would be really large and it would increase very much by increasing the domain cardinality p; on the other hand, such an extension would be also highly interesting from the applications point of view since it would be possible to solve problems not only in $\mathbb{R}$ but also in $\mathbb{R}^2$, such as problems with spatial functions. Besides, the current version of the algorithm always assumes the domain points to be equidistant; it may be useful to relax such hypothesis and give the users the possibility to analyse domains of different nature.

Another possible improvement concerns the intervalwise p-values computation. As we know, this is the most time-consuming procedure of the entire algorithm so we would be glad to find a way to carry it out faster. The main target of intervalwise testing is to spot the domain portions where the null hypothesis H_0 should be rejected, i.e., to identify the points of the domain whose adjusted p-value lies below a given threshold. Since the adjusted p-value of a point is defined as the maximum among its pointwise p-value and the p-values of all subintervals containing that point, it is useless to go on computing the intervalwise p-values of those subintervals whose points have a p-value greater than the threshold. As a matter of facts, most of the times, we do not care of the exact p-value but we just want to know whether it lies above the threshold or not. For example, if we get an unadjusted p-value equal to 1 for three consecutive points, say, 5, 6, and 7, it is useless to compute the intervalwise p-value of the subintervals (5, 6), (6, 7), and (5, 6, 7) since the adjusted p-value

of the points 5, 6, and 7 will always be 1. As a consequence, in some cases skipping some intervalwise p-values computations may save a little time.

Besides, we could also enlarge the range of test types. For now, the algorithm reckons just on two-population tests but there are also other types of test that could be worth including, such as one-population tests and multi-population tests.

Finally, up to now we have never mentioned the seed for the random generation of the permutation sequences. We did not implement this possibility because we do not believe it is crucial for this algorithm and also because setting the seed while working in multi-thread with OpenMP is not straightforward. Anyway, the implementation of this feature could be appreciated by those researchers who want to make their study fully reproducible.

6 Tutorial

The algorithm has been implemented so that it can be compiled and executed both directly from command line and from the R environment. In order to run the algorithm from command line we need the file `main.cpp`, its `Makefile`, and the header file of `GetPot`. In order to run it from R, we just need the file `IWT2OMP.cpp`. Note that `main.cpp` has a lot of explanatory comments while `IWT2OMP.cpp` does not because both files are almost identical, apart from few lines of interface with R.

From command line In order to compile and launch the algorithm from the command line, we need the library `Eigen` loaded (version 3.3.3 or successive is recommended) and the GetPot header file (version 2.0 or successive), which must be in the same folder where `main.cpp` is. Moreover, if we want to take advantage of parallel computing, we should communicate the number of threads to reserve for OpenMP parallel operations. Then, we just need to compile with `make` and launch with `./main`. We can also specify the values of some parameters, for example $n1 = n2 = 50$ and $p = 10$.

```
$ module load eigen
$ export OMP_NUM_THREADS=1
$ make
$ ./main n1=50 n2=50 p=10
```

From R In order to load the function `IWT2OMP` into R, we need the packages `Rcpp` and `RcppEigen` (version 0.12.13 and 0.3.3.3 or later, respectively) and the so-called `Rtools`, which can be downloaded either from the CRAN website or directly from R after launching the program: if the software does not find the required `Rtools`, it automatically asks to install them. Then, we just have to type the following commands.

```
library(Rcpp)
Sys.setenv("PKG_CXXFLAGS"="-std=c++11 -Wall -pedantic -fopenmp")
sourceCpp("IWT2OMP.cpp")
```

References

[1] Alessia Pini and Simone Vantini. Interval-wise testing for functional data. *MOX-Report*, 2015.

[2] R Core Team. Writing R Extensions. *The Comprehensive R Archive Network*, 2017.

[3] Ruud Van Der Paas. An Overview of OpenMP. *International Workshop of OpenMP*, 2010.