Translation into C++ of the R package "Intervalwise Testing for Functional Data"

Advanced Programming for Scientific Computing (8 CFU)

Tutor: **Dr. Alessia Pini** Student: **Diego Di Maulo**

May 2018

POLITECNICO
MILANO 1863

Introduction

R package: **Intervalwise T^2 Testing for Functional Data**.

Main function: **IWT2** (plus minor functions to plot the results).

IWT2 algorithm:

- coded by Dr. Alessia Pini (with Prof. Simone Vantini),
- draft to check algorithm correctness,
- not optimized, with some memory wastes,
- inefficient and useless in most applications.

The goals are to implement it **in C++**, to make it **more efficient**, to make it **parallel** and to interface it with the other functions of the R package.

Description

R version

The inputs accepted by **IWT2** are:

- `data1`: observations of the first population,
- `data2`: observations of the second population,
- `mu`: mean difference of the two populations under H_0,
- `B`: number of Monte Carlo iterations,
- `paired`: flag for "paired" or "unpaired" test,
- `dx`: domain step size,
- `recycle`: flag for using "recycle" (for periodic domains),
- `alt`: test type or alternative.

Description

R version

The possible test types are:

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

Search for the domain portions that cause H_0 to be rejected:

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

Description

R version

The outputs returned by **IWT2** are:

- `test`: test type,
- `mu`: mean difference of populations under H_0 (same as input),
- `unadjusted_pval`: pointwise unadjusted p-values,
- `adjusted_pval`: pointwise adjusted p-values,
- `pval_matrix`: intervalwise p-values,
- `data.eval`: `data1` and `data2` juxtaposed,
- `ord_labels`: population to which the observations belong.

Description

C++ version

The inputs in C++ are:

- `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.

Description

C++ version

The outputs in C++ are only:

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

Description

C++ version

Libraries

- `iostream`, `fstream`
- `vector`, `string`
- `Eigen/Dense` (version 3.3.3)
- `ctime`, `iomanip`

Macros

- `INFO` (code flow), `SHOW` (partial results), `TIME` (elapsed times)

Typedefs

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

Description

Data reading

For now, we assume the inputs to be available in the text files:

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

Later on, we will also see a method to generate them automatically.

Description

Tilde test

We do not really 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 we actually perform the **tilde test**

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

where $\tilde{\mu}_1 = \mu_1 - \mu_0$.

- We only care of the difference $\mu_1 - \mu_2$ w.r.t. μ_0.
- Many operations will result much simpler.

Description

Inputs check

The **inputs data** must satisfy some constraints:

- `data1` and `data2`:
  - same number of columns,
  - same number of rows (only in "paired" tests),
- `mu`:
  - same number of elements as the number of columns of `data1`.

In R, the variables `data1`, `data2`, and `mu` are accepted in many formats, while in C++ they are accepted only as numeric variables.

Hence, the step size parameter `dx` is not needed any more.

Description

Inputs check

The **input parameters** must satisfy some constraints:

- `alt`: one among “two.sided”, “greater”, and “less”,
- `B`: positive,
- `maxrow`: between 0 and $p - 1$ (included).

We prefer *int* instead of *unsigned* for many variables so to

- avoid annoying warnings (if compiling with `-Wall`),
- rely on safer exit conditions when cycling on decreasing dummy variables.

Description

T0 computation

Let `delta` be the column-by-column mean differences vector,

```
for (int j = 0; j < p; j++)
  delta(j) = data1.col(j).mean() - data2.col(j).mean();
```

then the $\mathbf{T}^2$ **test statistic** is

- `(delta) * (delta)` in "two.sided" tests,
- $\texttt{(delta)}^+ * \texttt{(delta)}^+$ in "greater" tests,
- $\texttt{(delta)}^- * \texttt{(delta)}^-$ in "less" tests.

Description

Pointwise p-values computation

Pointwise p-values computed via **"two-pop" permutation tests**:

- we randomize the labels, i.e., we **virtually** exchange some observations between the two populations,
- we compute the T^2 statistics under the new configuration,
- we compare the T^2 statistics with the original T0 statistics,
- we estimate the **pointwise p-values**.

We create the matrix `T_perm` to store the T^2 statistics:

- `B` rows (permutations),
- p columns (domain points).

Description

Paired tests

In the **paired tests**, it must hold $n1 = n2 = n$.

- We generate a **random binary sequence** of n elements,
- we **virtually** exchange the observations between the two populations when the sequence has value 1.

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

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

Description

Unpaired tests

- We generate a **random sample** of 1, 2, 3, ..., $n1 + n2$,
- we **virtually** reorder the observations of both populations.

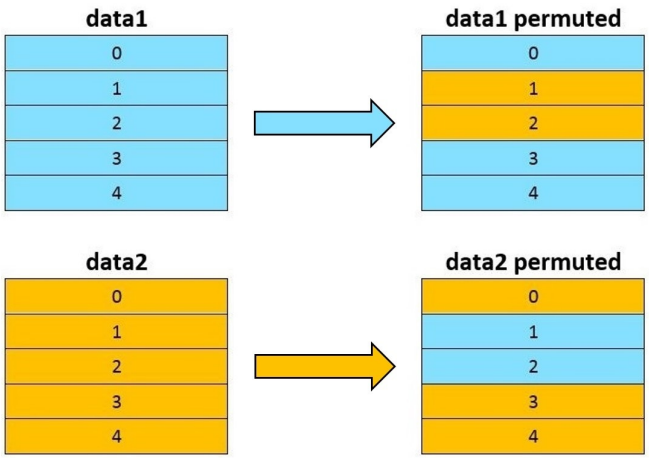
```
// Declare indices vector and auxiliary variables
std::vector<int> indices(n1+n2);
bool ok; int k;

// Generate random sample
for (int i = 0; i < (n1+n2); i++) {
  indices[i] = rand() % (n1+n2);
  if (i > 0) {
    ok = false;
    while (ok == false) { // value not chosen yet
      k = 0;
      while (indices[i] != indices[k]) k++;
      if (k < i) indices[i] = rand() % (n1+n2);
      else ok = true;
} } }
```

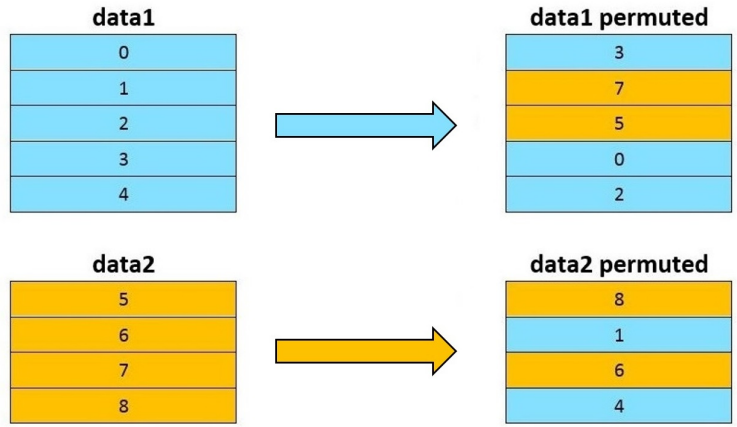
Description

Paired and unpaired tests

$\texttt{indices} = (0, 1, 1, 0, 0)$



$\texttt{indices} = (3, 7, 5, 0, 2, 8, 1, 6, 4)$



Virtually: we do not really exchange the observations in C++. Conversely, in the R version, at each iteration a new copy of `data1` and `data2` is created and the rows are really exchanged.

Description

Pointwise p-values computation

Pointwise p-values computation:

- we generate a random sample,
- we compute the mean differences,
- we compute the ratio of $T^2 \geq T0$.

We create the vector `count` to tally up how many times $T^2 \geq T0$ and we estimate the p-values with the ratio `count`/`B`.

- Although it is just a counter, we implement `count` as `std::vector<double>` in order to ease some operations.
- Besides, even its implementation as `std::vector<unsigned>` brings no performance improvements.

Description

Intervalwise p-values computation

For each subinterval,

- we sum the T^2 and T0 of all its points (as S^2 and S0),
- we compare S^2 and S0,
- we estimate the **intervalwise p-values**.

We create the matrix `pvalue_inter`:

- p rows (cardinalities: subintervals with $p - i$ points),
- p columns (starting points: subintervals starting from j).

Description

Recycle

- Using "recycle" means assuming a **periodic/cyclic domain**.
- We have always p subintervals for any length.
- The matrix `pvalue_inter` is full.

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

Description

No recycle

- Not using "recycle" means assuming an **acyclic domain**.
- We have less and less subintervals as their length increases.
- The matrix `pvalue_inter` is lower-triangular.
- The upper-triangular part is equal to 0 (C++) or `NaN` (R).

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

Description

Intervalwise p-values computation

If the **truncation** parameter `maxrow` is positive, the cycle stops at the subintervals with $p - \texttt{maxrow}$ points, without continuing with the larger ones.

Duplication: the R version duplicates the vector `T0` and the matrix `T_perm` to ease some operations, thus producing a significant waste of memory (if $\texttt{B} = 100.000$ and $p = 1.000$, the matrix `T_perm` is around 800 MB). In C++, we rely only on already existing variables.

Description

Corrections

In order to obtain theoretically sound results, we apply some **corrections** to the p-values computed so far, by combining pointwise and intervalwise p-values.

H_0 should be rejected at a point not only if the pointwise p-value at that point is low, but if also the intervalwise p-values of all subintervals to which that point belongs are low.

The **corrected p-value** at a point is the maximum among

- its pointwise p-value,
- the intervalwise p-values of all subintervals with that point.

Description

Corrections with and without recycle

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

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

Description

Corrections

Duplication: the R version duplicates the $p \times p$ matrix `pvalue_inter` so to simplify some operations with periodic domains but - in so doing - it causes a significant memory waste. In the C++ version, we avoid that by playing with some indices.

Overwrite: the R version generates a $p \times p$ matrix to perform the corrections but - in the end - it needs only the `maxrow`-th row of it. In C++, we only create a p-dimensional vector and we overwrite it at each iteration instead of wasting p new elements every time.

Parallelization

OpenMP is more appropriate for Monte Carlo methods since it works in shared memory.
MPI would need to broadcast both matrices to all slave nodes.

In the **pointwise computation**, we parallelize the `B` permutations

- as equally as possible,
- "statically" (same complexity for each permutation).

In the **intervalwise computation**, we parallelize the subintervals cardinality

- "dynamically" (increasing complexity for increasing cardinality).

Results

Parameters: n, p

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

Results

Parameters: `maxrow`, `B`

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

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

Results

Parameters: `paired`, `recycle`, `alt`

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

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

Results

Sections

The time needed by each section of the algorithm depends on some data and problem dimensions.

- **Read**: $\sim 5\%$ (increases with n)
- **T0**: $\sim 0\%$ (increases with p)
- **Point**: $\sim 1\%$ (increases with p)
- **Interval**: $\sim 95\%$ (increases with `B`, decreases with `maxrow`)
- **Correct**: $\sim 0\%$ (increases with p)

Results

R vs C++

C++ version averagely takes **20 times less** than R version...

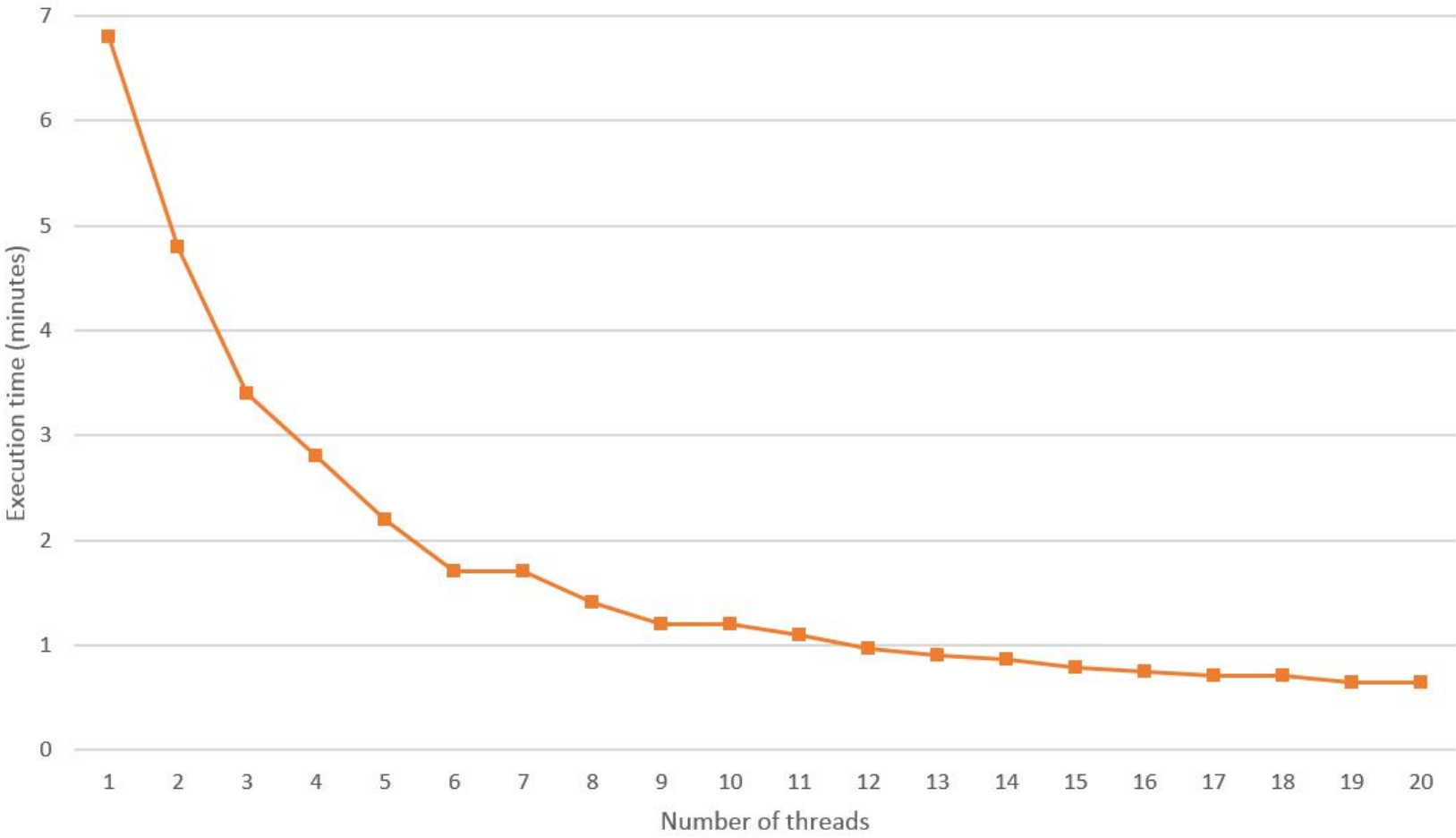
p	B	`recycle`	C++ (min)	R (min)	**Ratio**
500	1000	`true`	0.33	9	**4%**
500	5000	`true`	1.68	48	**3%**
500	10000	`true`	6.77	100	**7%**
200	10000	`true`	0.28	10	**3%**
200	10000	`false`	0.16	2	**8%**

...even without parallelization!

Results

Parallelization

R version: $>$ 3 hours **C++ version**: $<$ 2 minutes



Possible extensions

- **Domain**:
  - 2-dimensional domains,
  - non-equidistant points.
- **Intervalwise p-values computation**:
  - stop when a p-value exceeds a threshold.
- **Test types**:
  - one-population tests,
  - multi-population tests.
- Possibility to set **seed** for random generations.

Acknowledgements

I believe that this project will be of great use because now the **IWT2** algorithm is much faster than before. Due to its original too long execution times, it could not be used very much in the past, but now it can be run in **less than 1% of the time than before**.

I would like to thank everyone who helped me in this project during this long year and, in particular,

- Alessia Pini,
- Aymeric Stamm,
- Carlo De Falco,
- Luca Paglieri.