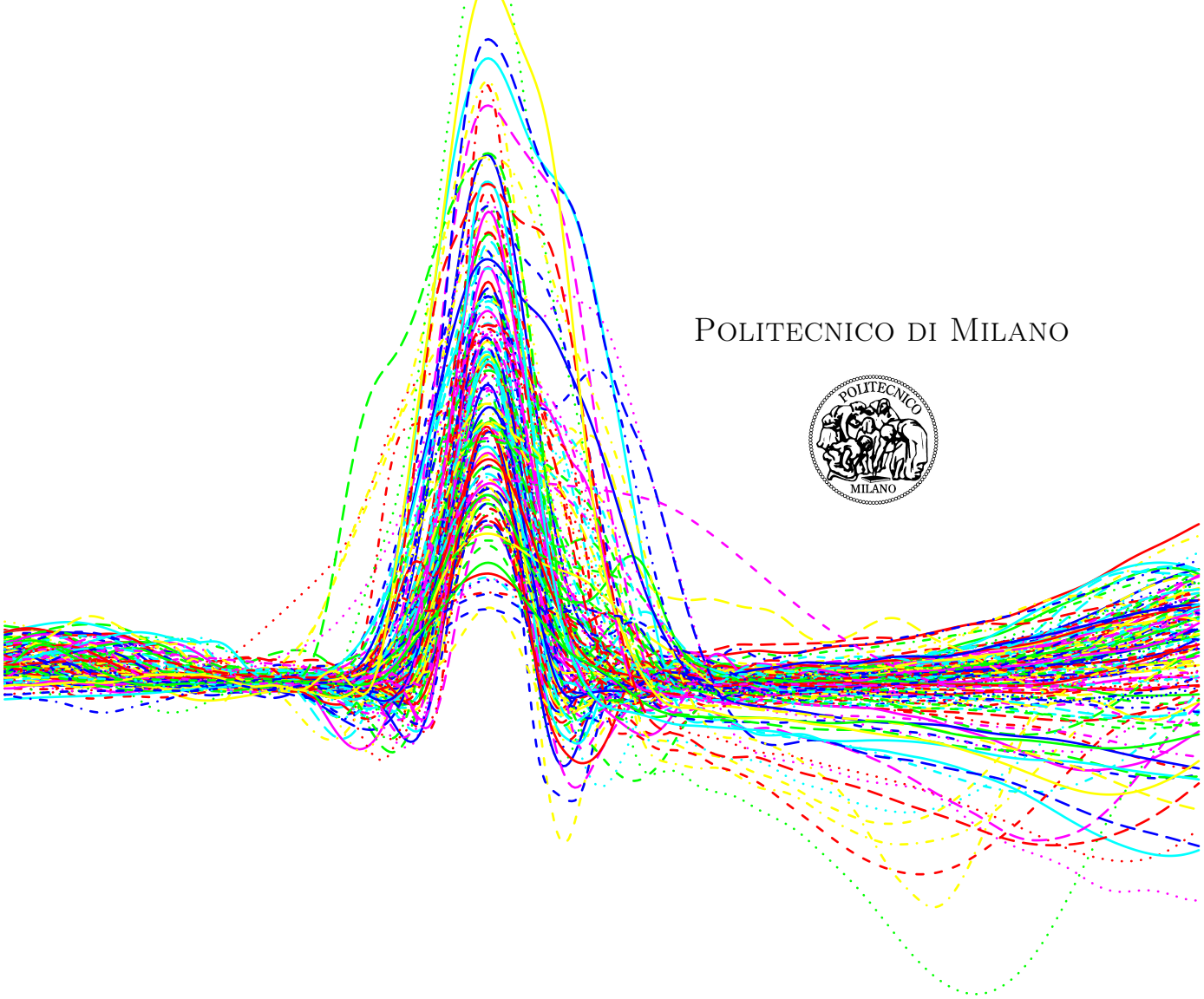

Politecnico di Milano

Libreria HPCS

Misure di profondità per dati funzionali multivariati

Progetto del corso di Programmazione avanzata per il Calcolo Scientifico

Nicholas Tarabelloni
Matr. n. 782187

Anno Accademico 2012-2013

Indice

Capitolo 1

Introduzione

Il presente documento descrive il progetto da me realizzato per il corso di Programmazione Avanzata per il Calcolo Scientifico (PACS), riguardante l'implementazione all'interno di una libreria C++, chiamata HPCS (High-Performance-Computational-Statistics), di strumenti per il calcolo delle *misure di profondità* di dati funzionali multivariati.

Il calcolo delle misure di profondità delle unità statistiche di una popolazione funzionale può essere molto oneroso e quando il numero di campioni della popolazione in esame cresce la complessità, in generale, aumenta con un trend combinatorio. In contesti applicativi in cui si debba considerare un numero di segnali molto grande o quando l'adattamento delle misure di profondità ad una particolare applicazione diventi esso stesso oggetto di studio, tale costo diventa un bottleneck proibitivo. Il principale strmento open-source utilizzato nella statistica e data-analysis a livello accademico, R, non permette di affrontare adeguatamente questa criticità, che diventa anche più seria quando si trattano dati funzionali multivariati.

HPCS è stata creata apposta per affrontare questo problema, beneficiando sia dell'implementazione in un linguaggio compilato, sfruttando liberie robuste ed efficienti (e.g. le boost), sia di una parallelizzazione nativa del calcolo attraverso il protocollo MPI.
Al pari dell'efficienza si è curata la flessibilità delle strutture di HPCS, fattore chiave per le applicazioni di data-analysis, attraverso un uso intensivo del parser GetPot, che permette l'inizializzazione della maggior parte degli oggetti tramite file, riducendo così al minimo la necessità di ricompilare il sorgente per modificare i parametri del calcolo.
L'adattabilità di HPCS all'ambiente di utilizzo è garantita dall'uso di CMake, che permette di localizzare automaticamente le dipendenze (e.g. librerie MPI e boost) e di generare Makefile nativi.
Insieme al sorgente, la libreria è accompagnata da una documentazione che riassume le caratteristiche delle classi, mostrandone i metodi e la collaborazione. Essa è prodotta sfruttando il software Doxygen.

In qualità di progetto open-source e visto il possibile utilizzo in diversi contesti applicativi, HPCS è pensata per potere essere distribuita ed estesa, inglobando contributi di statistica computazionale ad alte prestazioni da parte di altri sviluppatori. A tal proposito è stato creato un apposito repository di GitHub (https://github.com/larvatus/HPCS.git) che la ospita e dalla quale si può scaricare o clonare.

Nel seguito verrà presentato sia il contesto statistico delle misure di profondità, sia la loro implementazione all'interno di HPCS. Al termine verranno mostrati alcuni esempi di utilizzo delle funzionalità messe a disposizione.

Capitolo 2

Misure di profondità di dati funzionali multivariati

2.1 Dati funzionali multivariati

Si consideri un processo stocastico multivariato $\mathbf{X}$ con legge $\mathbb{P}$ a valori nello spazio delle funzioni $\mathcal{C}^0(I;\mathbf{R}^s)$, con $s>0$ e $I\subset\mathbb{R}$ intervallo compatto.
Una popolazione di dati funzionali multivariati è un insieme

$$F=(\mathbf{f}_1,\ \mathbf{f}_2,\ \ldots,\ \mathbf{f}_N),\quad \mathbf{f}_i=(f_{i;1},f_{i;2},\ldots,f_{i;s}):I\longrightarrow\mathbb{R}^s,\quad \mathbf{f}_i\in\mathcal{C}^0(I;\mathbb{R}^s)\quad\forall i,$$

in cui le $\mathbf{f}_i$ sono le realizzazioni del processo $\mathbf{X}$.

Nei contesti applicativi all'insieme F corrisponde la versione discreta F_h:

$$F_h=(\mathbf{f}_1,\mathbf{f}_2,\ldots,\mathbf{f}_N),\quad \mathbf{f}_i=(f_{i;1},f_{i;2},\ldots,f_{i;s}),\qquad f_{i;j}\in\mathbb{R}^P,$$

in cui le $\mathbf{f}_i$ vengono determinate come campionamenti, fitti a piacere, delle funzioni di partenza, cui segue una fase di smoothing per ricostruire il segnale originario e preparare i dati all'analisi statistica.

2.2 Misure di profondità

La misura di profondità statistica di un dato funzionale è un indice di natura non-parametrica che esprime la *centralità* di quell'osservazione rispetto ad un fissato dataset funzionale o alla distribuzione di una popolazione. Essa può essere utilizzata per ordinare il dataset a partire dai segnali meno centrali (o profondi), verso quelli più centrali, e permette:

1. di costruire *boxplot funzionali*, cioè l'analogo funzionale dei boxplot tradizionali, con cui visualizzare la dispersione del dataset e identificare eventuali *outlier* (si vedano, ad esempio, [SG11], [SG12]);
2. di estendere al contesto funzionale le statistiche d'ordine e poter applicare i metodi non-parametrici univariati basati su di esse (si veda, ad esempio, [IP13]).

Il concetto di profondità di un dato proviene dal contesto dell'analisi statistica multivariata, rispetto alla quale rappresenta un possibile modo di ordinare i dati vettoriali ambientati in spazi euclidei multidimensionali.

In [LPJ07] e [LPR09] viene estesa questa misura al caso funzionale, esaminando la relazione della nuova definizione con quelle del caso vettoriale, insieme alle sue proprietà asintotiche.

2.2.1 Il caso univariato (BD)

Si consideri la collezione di funzioni univariate

$$F = (f_1,\ f_2,\ \ldots, f_N)\,, \quad f_i : I \longrightarrow \mathbb{R}, \quad f_i \in \mathcal{C}^0\,(I;\mathbb{R}) \quad \forall i,$$

Sia $j \in \mathbb{N},\ \ 2 \leq j \leq N$ e un elemento f del campione F.
Si indica con

$$\begin{aligned} A_j(f) &= A\big(f;\ f_{i_1}, f_{i_2}, \ldots, f_{i_j}\big) \\ &= \left\{ t \in I : \min_{r=i_1,\ldots,i_j} f_r(t) \leq f(t) \leq \max_{p=i_1,\ldots,i_j} f_p(t) \right\} \end{aligned}$$

il sottoinsieme di I in cui il grafico di f è contenuto nell'inviluppo della j-pla di segnali $f_{i_1}, f_{i_2}, \ldots, f_{i_j}$.
Indicando con λ la misura di Lebesgue sull'intervallo I,

$$\lambda_r\left(A_j(f)\right) = \frac{\lambda\left(A_j(f)\right)}{\lambda\left(I\right)}$$

è la misura di $A_j(f)$ relativa a I.
Allora si può dare la seguente definizione:

Definizione 2.1. Sia $J \in \mathbb{N}$ un valore fissato, con $2 \leq J \leq N$, si definisce la misura di profondità (univariata) di ciascuna curva della collezione F:

$$\mathrm{BD}_N^{(j)}(f) = \binom{N}{j}^{-1} \sum_{1 \leq i_1 < i_2 < \ldots < i_j \leq N} \lambda_r\left(A(f; f_{i_1}, f_{i_2}, \ldots, f_{i_j})\right), \tag{2.1}$$

$$\mathrm{BD}_N^{J}(f) = \sum_{j=2}^{J} BD_N^{(j)}(f) \tag{2.2}$$

Osservazione 2.1. La Def. 2.1 corrisponde a una modifica dell'iniziale definizione di misura di profondità per un dato funzionale univariato:

$$\mathrm{BD}_N^{(j)}(f) = \binom{N}{j}^{-1} \sum_{1 \leq i_1 < i_2 < \ldots < i_j \leq N} \mathbb{I}\big\{G(f) \subseteq B(f_{i_1}, \ldots, f_{i_j})\big\}, \tag{2.3}$$

$$\mathrm{BD}_N^{J}(f) = \sum_{j=2}^{J} BD_N^{(j)}(f) \tag{2.4}$$

dove $G(f)$ indica il grafico di f e $B(f_{i_1}, \ldots, f_{i_j})$ l'inviluppo dei grafici delle $\big(f_{i_1}, \ldots, f_{i_j}\big)$.

Come si può notare, nella definizione originaria il contributo alla banda di profondità è dato dall'appartenenza o meno del grafico di f all'inviluppo $B(f_{i_1}, \ldots, f_{i_j})$ per *tutto* il dominio I. Questo ha come generale conseguenza la comparsa di *code* nel dataset, cioè gruppi di curve cui vengono attribuiti valori bassi di profondità molto simili tra loro.
Nella Def. 2.1, invece, la condizione di appartenenza stretta all'inviluppo B viene rilassata, considerando come contributo la proporzione di dominio I in cui il grafico di f si trova nell'inviluppo B.

2.2.2 Il caso multivariato (MBD)

Una possibile estensione delle misure di profondità di dati funzionali al caso multivariato è riportata in [IP13].

Definizione 2.2. Sia $\mathbf{F}$ un processo stocastico con legge $\mathbb{P}$ a valori nello spazio $C^0(I;\mathbb{R}^s)$ delle funzioni continue $\mathbf{f} = (f_1, \ldots, f_s) : I \longrightarrow \mathbb{R}^s$, $J \in \mathbb{N}$. Definiamo misura di profondità multivariata la quantità:

$$\mathrm{MBD}^J_{P_{\mathbf{F}}}(\mathbf{f}) = \sum_{k=1}^{s} p_k \; BD^J_{P_{F_k}}(f_k), \quad p_k > 0 \;\; \forall k = 1, \ldots, s, \quad \sum_{k=1}^{s} p_k = 1 \tag{2.5}$$

dove viene utilizzata la definizione

$$\mathrm{BD}^J_{P_{F_k}}(f_k) = \sum_{j=2}^{J} \mathbb{P}\left\{G(f_k) \subset B\left(F_{1,k}, F_{2,k}, \ldots, F_{j,k}\right)\right\} \tag{2.6}$$

in cui $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_j$ sono copie indipendenti del processo stocastico $\mathbf{F}$.

Questa è chiaramente solo una delle possibili estensioni del concetto di banda di profontià al caso multivariato funzionale, ma ha il vantaggio di permettere di considerare in vari modi l'influenza di ciascuna componente. In generale, tuttavia, non esiste una regola universale per la scelta dei pesi $\{p_k\}_{k=1}^s$, che dipende dal problema in esame. In particolare i pesi dovrebbero incorporare le informazioni a priori sul fenomeno, le eventuali relazioni di interdipendenza e la struttura di correlazione tra le componenti. Così, la libertà nella scelta dei pesi causa una mancanza di unicità nella definizione (2.5), ma assicura una grande adattabilità al problema reale in studio.

Se $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N$ sono copie indipendenti del processo stocastico $\mathbf{F}$, è possibile introdurre la versione campionaria di (2.5), utile per condurre analisi descrittive ed inferenziali su un insieme di dati funzionali multivariati $\mathbf{f}_1, \ldots, \mathbf{f}_N$ generati dal processo $\mathbf{F}$.

Definizione 2.3. Per ogni $\mathbf{f}$ nel campione multivariato F, calcoliamo la sua misura di profondità come:

$$\mathrm{MBD}^J_N = \sum_{k=1}^{s} p_k BD^J_{N,k}\left(f_k\right), \tag{2.7}$$

in cui a sua volta si definisce

$$\mathrm{BD}^J_{N,k}\left(f_k\right) = \sum_{j=2}^{J} \binom{N}{j}^{-1} \sum_{1 \le i_1 < i_2 < \ldots < i_j \le N} \lambda_r\left\{A(f_k; f_{i_1,k}, \ldots, f_{i_j,k})\right\}, \tag{2.8}$$

dove, analogamente al caso delle bande di profondità univariate, si è indicato:

$$\lambda_r\left(A\right) = \frac{\lambda\left(A\right)}{\lambda(I)}, \quad \forall A \subseteq I \quad \text{Lebesgue misurabile}$$

$$A\big(f;\; f_{i_1}, f_{i_2}, \ldots, f_{i_j}\big) = \left\{t \in I : \min_{r=i_1,\ldots,i_j} f_r(t) \le f(t) \le \max_{p=i_1,\ldots,i_j} f_p(t)\right\}$$

Osservazione 2.2. Anche nel caso multidimensionale la Def. 2.3 rappresenta una versione modificata della banda di profondità, nella quale si considera il contributo dato dalla frazione di dominio I in cui i segnali stanno negli inviluppi.

Capitolo 3

HPCS

Nel seguito viene descritta l'implementazione in HPCS delle strutture per il calcolo delle misure di profondità di dati funzionali univariati e multivariati, come definite nel Cap. 2.

Dal punto di vista macroscopico il codice è organizzato in una struttura modulare e gerarchica. In una prima directory `source` vengono raccolti i sorgenti `.hpp` e `.cpp` in cui sono state ripartite ed implementate le singole funzionalità. Nella cartella `synthetic` sono riportati i dataset creati artificialmente che possono essere usati nei test e nello studio delle misure di profondità. I sorgenti dei test, invece, sono collocati nella directory `test`.

3.1 Strumenti di sviluppo

Lo sviluppo di HPCS ha richiesto l'utilizzo di alcuni strumenti specifici per semplificare e sistematizzare la gestione del codice, quali `CMake`, `git`, e `Doxygen`.
`CMake` è un software multipiattaforma che permette di configurare e generare `Makefile` nativi al sistema in uso, utilizzando compilatori e percorsi delle librerie locali, rendendo molto più facile la diffusione del codice tra gli sviluppatori.
`Git` è un sistema avanzato di versioning molto diffuso ed è utile per tenere traccia della storia di un progetto e gestire in maniera apposita sia contributi dall'esterno che fusioni di rami di sviluppo differenti del singolo programmatore.
`Doxygen` è un utile strumento per generare facilmente la documentazione associata ad una libreria. Una volta introdotti nel sorgente dei commenti preceduti da opportune keyword, esso può essere lanciato per leggere il codice e generare, tra le altre cose, l'elenco delle classi implementate, i loro diagrammi di collaborazione e la descrizione di metodi ed attributi.

Nel seguito riportiamo alcune note importanti riguardanti l'integrazione di questi strumenti nell'utilizzo e sviluppo di HPCS .

3.1.1 CMake

Il funzionamento di `CMake` è basato sull'utilizzo di opportuni files `CMakeLists.txt` contenuti nelle directory del progetto.
Nel file `CMakeLists.txt` della directory principale viene impostata la dipendenza dalla versione di `CMake` richiesta, il nome del progetto corrente e si estende il path di compilazione includendo le directory dei sorgenti. Oltre a questo vengono cercate le librerie esterne utilizzate dai sorgenti, che sono:

- `boost`, di cui vengono usati in maniera capillare gli shared pointers, robusti rispetto ai problemi di scoping dei puntatori raw, e le `ublas`, che sono il fondamento della rappresentazione matriciale dei dataset;
- `MPI`, che viene utilizzato per la parallelizzazione del codice;
- `Doxygen`, che viene utilizzato per generare la documentazione.

Per usare `CMake` e creare dei `Makefile` nativi con cui compilare il codice sorgente è necessario creare una directory di `build`, in cui il codice verrà compilato, e lanciare `CMake` nel seguente modo

```
cd <build_dir>
cmake <source_dir>
```

`CMake` esaminerà le dipendenze delle librerie esterne e imposterà variabili di path e obiettivi specifici secondo quando richiesto nel file `CMakeLists.txt`. `CMake` procederà anche ad esaminare le sottocartelle `source` e `test` e, grazie alla presenza in esse di altri file `CMakeLists.txt` che definiscono le ulteriori sottocartelle e obiettivi, esaminerà in maniera ricorsiva tutto l'albero della directory.

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

PROJECT(BandDepth)

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})

SET( CMAKE_BUILD_TYPE ${CMAKE_CXX_FLAGS_DEBUG})

#Require Boost for this project
FIND_PACKAGE(Boost)
INCLUDE_DIRECTORIES(${Boost_INCLUDE_DIR})

# Require MPI for this project:
FIND_PACKAGE(MPI REQUIRED)
SET( CMAKE_CXX_COMPILE_FLAGS ${CMAKE_CXX_COMPILE_FLAGS} ${MPI_COMPILE_FLAGS})
SET( CMAKE_CXX_LINK_FLAGS ${CMAKE_CXX_LINK_FLAGS} ${MPI_LINK_FLAGS})
INCLUDE_DIRECTORIES(${MPI_INCLUDE_PATH})

#Require Doxygen for documentation
FIND_PACKAGE(Doxygen)
IF (DOXYGEN_FOUND)
SET(DOXYGEN_INPUT ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in )
SET(DOXYGEN_OUTPUT ${CMAKE_BINARY_DIR}/doc)
ADD_CUSTOM_COMMAND(OUTPUT ${DOXYGEN_OUTPUT}
   COMMAND ${CMAKE_COMMAND} -E echo ${CMAKE_BINARY_DIR}
   COMMAND ${CMAKE_COMMAND} -E echo_append "Building API Documentation..."
   COMMAND ${DOXYGEN_EXECUTABLE} ${DOXYGEN_INPUT}
   COMMAND ${CMAKE_COMMAND} -E echo "Done."
   WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
   DEPENDS ${DOXYGEN_INPUT}
  )

ADD_CUSTOM_TARGET(doc DEPENDS ${DOXYGEN_OUTPUT})
ENDIF (DOXYGEN_FOUND)

# Adding test subdirectory
ADD_SUBDIRECTORY(source)
ADD_SUBDIRECTORY(test)
```

Nelle cartelle di test i files `CMakeLists.txt` definiscono gli obiettivi relativi ai sorgenti dei test e creano i collegamenti simbolici ai files di appoggio contenuti in `test` che vengono usati durante l'esecuzione. Questi sono principalmente files `data` usati da `GetPot` e script `gnuplot` per la visualizzazione dei risultati.

```
EXECUTE_PROCESS( COMMAND ln -s ${CMAKE_CURRENT_SOURCE_DIR}/data ${CMAKE_CURRENT_BINARY_DIR}/data
      )
```

```
EXECUTE_PROCESS( COMMAND ln -s ${CMAKE_CURRENT_SOURCE_DIR}/multiDepthMeasure.plot ${
    CMAKE_CURRENT_BINARY_DIR}/multiDepthMeasure.plot )
EXECUTE_PROCESS( COMMAND ln -s ${CMAKE_CURRENT_SOURCE_DIR}/depthMeasure.plot ${
    CMAKE_CURRENT_BINARY_DIR}/depthMeasure.plot )

# A first executable
ADD_EXECUTABLE(main_depthMeasureAll.exe main_depthMeasureAll )
TARGET_LINK_LIBRARIES(main_depthMeasureAll.exe ${MPI_LIBRARIES} mbd)

ADD_EXECUTABLE(main_depthMeasureRef.exe main_depthMeasureRef )
TARGET_LINK_LIBRARIES(main_depthMeasureRef.exe ${MPI_LIBRARIES} mbd)

ADD_EXECUTABLE(main_multiDepthMeasureAll.exe main_multiDepthMeasureAll )
TARGET_LINK_LIBRARIES(main_multiDepthMeasureAll.exe ${MPI_LIBRARIES} mbd)
EXECUTE_PROCESS( COMMAND ln -s ${CMAKE_CURRENT_SOURCE_DIR}/weightsAll.dat ${
    CMAKE_CURRENT_BINARY_DIR}/weightsAll.w )

ADD_EXECUTABLE(main_multiDepthMeasureRef.exe main_multiDepthMeasureRef )
TARGET_LINK_LIBRARIES(main_multiDepthMeasureRef.exe ${MPI_LIBRARIES} mbd)
EXECUTE_PROCESS( COMMAND ln -s ${CMAKE_CURRENT_SOURCE_DIR}/weightsRef.dat ${
    CMAKE_CURRENT_BINARY_DIR}/weightsRef.w )
```

Nella directory `source` viene aggiunta la creazione di una libreria, `mbd`, a partire dai sorgenti contenuti.

```
ADD_LIBRARY( mbd bandDepth bandDepthData bandDepthRef combinations mpi_utility dataSet )
```

3.1.2 Git

È possibile scaricare `HPCS` e collaborare alla sua estensione attraverso `git` clonando il codice dal repository GitHub corrispondente:

```
git clone https://github.com/larvatus/HPCS.git
```

La directory principale contiene anche un file `.gitignore`, in cui vengono elencate le estensioni dei files e le sottocartelle che `git` escluderà dal versioning. I files esclusi sono le versioni temporanee `*.~` eventualmente generate dagli editor, la cartella di `build` e le cartelle `doc`, `html` e `latex` che possono essere create accidentalmente da una configurazione non ideale di `Doxygen`.

3.1.3 Doxygen

Per potere utilizzare `Doxygen` e creare la documentazione di `HPCS` è necessario il file `Doxyfile.in`, contenuto nella directory principale, che definisce alcune variabili di configurazione importanti quali le cartelle in cui cercare i sorgenti, l'estensione dei files da considerare e la directory in cui creare la documentazione.

```
PROJECT_NAME           = HPCS
OUTPUT_DIRECTORY       = ./build/doc
INPUT                  = ./source ./test/
FILE_PATTERNS          = *.cpp *.hpp
SEARCH_ENGINE          = YES
HAVE_DOT               = YES
COLLABORATION_GRAPH    = YES
HIDE_UNDOC_RELATIONS   = NO
```

Per creare la documentazione, dopo aver lanciato `CMake`, è sufficiente eseguire:

```
cd <build_dir>
make doc
```

Al termine dell'esecuzione verrà creata una directory `build/doc` che conterrà la documentazione LaTeX e `html`.

3.2 Implementazione

In questa sezione viene descritta l'implementazione vera e propria del calcolo delle misure di profondità.

Si consideri la definizione di misura di profondità, ad esempio univariata, riportata nel Capitolo 2.

$$\mathbf{U}\begin{cases} \mathrm{BD}_N^{(j)}(f) = \binom{N}{j}^{-1} \sum_{1\le i_1<i_2<\ldots<i_j\le N} \lambda_r\left(A(f; f_{i_1}, f_{i_2}, \ldots, f_{i_j})\right) \\ \mathrm{BD}_N^{J}(f) = \sum_{j=2}^{J} BD_N^{(j)}(f) \end{cases} \tag{3.1}$$

Il calcolo di $\mathrm{BD}_N^J(f)$ è stato suddiviso macroscopicamente in tre passi:

(U1) Acquisizione del dataset.

(U2) Calcolo delle bande di profondità $\mathrm{BD}_N^{(j)},\ j = 1, \ldots, J$.

(U3) Sintesi e calcolo delle misure di profondità BD_N^J.

Considerando le misure di profondità multivariate

$$\mathbf{M}\begin{cases} \mathrm{BD}_{N,k}^{j}(f_k) = \sum_{j=2}^{j} \binom{N}{j}^{-1} \sum_{1\le i_1<i_2<\ldots<i_j\le N} \lambda_r\left\{A(f_k; f_{i_1,k}, \ldots, f_{i_j,k})\right\} \\ \mathrm{BD}_N^{J}(f) = \sum_{j=2}^{J} BD_N^{(j)}(f), \\ \mathrm{MBD}_N^{J} = \sum_{k=1}^{s} p_k BD_{N,k}^{J}(f_k), \end{cases} \tag{3.2}$$

ai tre passi precedenti, nei quali si calcolano le profondità delle singole componenti, si aggiunge un quarto passo per il calcolo della misura di profondità globale del dato come media pesata con i pesi $\{p_k\}$ delle profondità di tutte le componenti.

(M1) Acquisizione del dataset.

(M2) Calcolo delle bande di profondità $BD_N^{(j)},\ j = 1, \ldots, J$.

(M3) Sintesi e calcolo delle misure di profondità $\mathrm{BD}_{N,k}^J$ di ogni componente del segnale multivariato.

(M4) Sintesi e calcolo delle misure di profondità multivariate MBD_N^J di ogni segnale.

Osservazione 3.1. Oltre al calcolo classico delle misure di profondità, svolto considerando ogni funzione del dataset e misurandone la profondità rispetto a tutte le altre, in `HPCS` si è voluto fornire l'implementazione di un altro metodo, basato sulla suddivisione del dataset in due parti: una di riferimento (reference-set), che raccoglie i segnali rispetto ai quali calcolare le profondità, e un'altra di test (test-set) che contiene i dati di cui si vuole calcolare la profondità rispetto al reference set.
Questa procedura è molto utile per avere una misura non-parametrica efficace della dissimilarità tra segnali appartenenti a differenti sottopopolazioni di uno stesso campione. In particolare i dati della prima sottpopolazione possono costituire il reference-set, che rappresenta un metro di misura con cui confrontare i segnali della seconda popolazione (che formerà il test-set).

Nel seguito la procedura standard, in cui i dati sono confrontati in maniera combinatoria con tutti gli altri, viene indicata con la keyword *all*, mentre la procedura che separa il test-set dal training-set è indicata con la keyword *reference*.

A partire dalla suddivisione in passi del calcolo delle profondità e dati gli obiettivi ricorrenti, si è suddivisa l'implementazione in maniera da incapsulare i diversi passi in alcuni oggetti corrispondenti:

- **(U1,M1)** `--→ DataSet`
- **(U2,M2)** `--→ BandDepth, BandDepthRef`
- **(U3,M3)** `--→ DepthMeasure`
- **(M4)** `--→ MultiDepthMeasure`

Nelle quali si è prestata attenzione a permettere il calcolo sia con la procedura *all* che *reference*.

Nel seguito verranno forniti alcuni dettagli sull'implementazione di queste classi.

3.2.1 Data Set

Le classe `DataSet` è la struttura base che rappresenta il dataset funzionale. La sua interfaccia pubblica dovrà permettere una costruzione flessibile, che si adatti alle varie forme nelle quali i dati possano essere disponibili nel codice, e contemporaneamente la sua implementazione dovrà garantire una grande efficienza, sia rispetto alla memorizzazione dei dati, sia rispetto all'accesso dall'esterno, che presumibilmente avverrà molto frequentemente.
In vista del calcolo delle bande di profondità rispetto ad una sottopopolazione di riferimento, oltre al concetto di di dataset funzionale normale si è implementato il concetto di dataset sudddiviso in *livelli*, ossia una partizione in sottoinsiemi corrispondenti alle sottopopolazioni. La classe corrispondente è `DataSetLevelled`.

`DataSet`

La classe `DataSet` memorizza i dati sfruttando le funzionalità messe a disposizione dal modulo `ublas` delle librerie `boost`. Nelle `ublas` si trovano implementati molti oggetti tipici delle librerie di algebra lineare, tra cui matrici, vettori e proxies. La loro robustezza, l'efficienza di gestione e la presenza di molte funzioni specializzate per l'accesso a righe o colonne o loro insiemi ne fanno un ambiente ideale da cui trarre la struttura dati alla base della classe `DataSet`, che è una `matrix< Real >`.
L'interfaccia pubblica di `DataSet` permette l'utilizzo di diversi costruttori, che richiedono di specificare il numero di campioni nel dataset, `nbSamples`, il numero di punti dei segnali, `nbPts`, e i dati veri e propri.
Essi verranno salvati in uno `shared_pointer` ad un oggetto `matrix` con `nbSamples` righe e `nbPts` colonne.

```
class DataSet{
typedef boost::numeric::ublas::matrix< Real > data_Type;
typedef boost::shared_ptr< data_Type > dataPtr_Type;

// Constructors
DataSet( const UInt & nbSamples, const UInt & nbPts );
DataSet( const Real * data, const UInt & nbSamples, const UInt & nPts );
DataSet( const std::vector< Real > & data, const UInt & nbSamples, const UInt & nbPts );
DataSet( const data_Type & data );
DataSet( const dataPtr_Type & dataPtr );

protected:
dataPtr_Type M_data;
};
```

L'accesso agli elementi del dataset è possibile grazie all'overload dell'operatore `()`:

```
Real operator()( const UInt & sample, const UInt & pt ) const;
```

Le altre funzionalità della classe sono fornite da setters & getters, un metodo `ShowMe` e uno per scrivere i dati su file. Particolarmente utile è il setter

```
void readData( const std::string & filename );
```

che permette di impostare i dati contenuti nella struttura a partire dal contenuto di un file esterno.

`DataSetLevelled`

La classe `DataSetLevelled` implementa il concetto astratto, ma comune in ambito statistico, di data set con livelli, cioè gruppi in cui i campioni sono raggruppati in base a qualche criterio. Questa struttura è alla base del calcolo delle misure di profondità di un dataset rispetto ad un campione di riferimento. In particolare, considerando due livelli, un primo definirà la parte di campione da cui verrà costruita la sotto-popolazione di riferimento (reference set), mentre l'altro definirà la parte di campione di cui calcolare le profondità rispetto al riferimento (test set).

La classe, derivata da `DataSet` è pensata per gestire un numero qualunque di livelli, che vengono determinati specificando gli ID dei dati relativi. Visto che a priori il numero di livelli è sconosciuto, che i singoli ID dei dati corrispondenti ai vari livelli possono avere una disposizione irregolare nel dataset, e soprattutto vista la possibile necessità di fare ricerche negli insiemi di ID di un dato gruppo, si è scelto di usare le `std::map` per rappresentare gli insiemi ordinati di ID dei gruppi.

```
class DataSetLevelled : public DataSet
{
public:
typedef std::set< UInt > IDContainer_Type;
typedef std::vector< IDContainer_Type > levelsContainer_Type;
typedef boost::shared_ptr< levelsContainer_Type > levelsContainerPtr_Type;

DataSetLevelled( const UInt & nbSamples, const UInt & nbPts, const UInt & nbLevels );

private:
levelsContainerPtr_Type M_levelsPtr;
};
```

I costruttori di `DataSetLevelled` riprendono quelli di `DataSet`, aggiungendo la variabile `nbLevels`. Per brevità ne è stato riportato solo uno.

Il setup dei livelli viene effettuato sfruttando i setters disponibili in due modi: è possibile passare alla classe direttamente un oggetto `levelsContainerPtr_Type`, oppure è possibile specificare sottoinsiemi contigui del dataset che costituiscono i vari gruppi.

```
public:
 void setLevels( const levelsContainerPtr_Type & levelsPtr );
 void setLevels( const std::vector< UInt > & linearExtrema );
 void setLevelsFromExtrema( const std::string & levelsExtremaFilename );
```

Gli estremi dei livelli (`linearExtrema`) sono gli ID che definiscono il dato livello. Così il vettore di estremi $(0, 10, 20)$ identificherà i due livelli $L_0 = (0, \ldots, 9)$ e $L_1 = (10, \ldots, 20)$. L'impostazione di tali estremi è possibile sia passando direttamente il vettore degli ID estremi a `setLevels`, sia specificando a `setLevelsFromExtrema` un file esterno che li contiene. Completano la classe i setters & getters delle nuove variabili e l'override del metodo `showMe` della classe base.

3.2.2 Band Depth

Le classi deputate al calcolo dei contributi $\mathrm{BD}_N^{(j)}$ sono `BandDepth<UInt _J>` e `BandDepthRef<UInt _J>`, template rispetto al parametro intero `_J` che definisce la dimensione della tupla considerata nella costruzione del sottoinsieme $A_j(f) = A\big(f;\ f_{i_1}, f_{i_2}, \ldots, f_{i_j}\big)$. `BandDepth<_J>` corrisponde al calcolo dei contributi nel caso con normale, mentre `BandDepthRef<_J>` corrisponde al calcolo rispetto ad un sottoinsieme di riferimento.

Entrambe le classi derivano da una classe base, `BandDepthBase<BDPolicy _policy >`, template rispetto ad una policy `BDPolicy`, definita come tipo enumerativo, che esprime se il calcolo debba essere standard o rispetto al sottoinsieme di riferimento.
`BandDepth<_J>` deriva dalla specializzazione `BandDepthBase<All>`, mentre `BandDepthRef<_J>` deriva da `BandDepthBase<Reference>`.

```
typedef unsigned int UInt;

enum BDPolicy { Reference, All };

template < BDPolicy _policy >
class BandDepthBase
{...};

template < UInt _J >
class BandDepth : public BandDepthBase< All >
{...};

template < UInt _J >
class BandDepthRef : public BandDepthBase< Reference >
{...};
```

Nota 3.2. La scelta, presa in fase di design del codice, di definire entrambe le classi `BandDepth<_J>` e `BandDepthRef<_J>` template rispetto al parametro `_J` ha due motivi: da una parte si vuole imporre un rigido controllo direttamente a compile-time sul valore di questo parametro, cruciale per l'implementazione. Dall'altro questa scelta permette di specializzare i metodi responsabili del calcolo effettivo delle profondità, che nel caso di `_J=2` possono sfruttare un algoritmo più semplice e molto più veloce rispetto a quello standard.

Nota 3.3. La classe base `BandDepthBase<_policy>` viene completamente specializzata per i valori di `_policy=All` e `_policy=Reference` per fornire ai tipi derivati un'interfaccia pubblica utilizzabile in maniera polimorfica che tenga conto della diversa gesione esterna tra il caso `All` e `Reference`.
Essendo classi base con interfaccia pubblica virtuale, costituita principalmente da setters & getters e dal metodo `computeBDs()`, il codice duplicato è minimo.
La presenza di queste classi base, indipendenti dal parametro `_J`, ha anche il vantaggio di permettere l'istanziazione di oggetti `BandDepth<_J>` e `BandDepthRef<_J>` attraverso un'apposita factory che, insieme all'uso di `GetPot`, incrementa la flessibilità e la gestibilità a run-time del codice.

Inizializzazione: `BandDepthData` **e** `BandDepthRefData`

La gestione delle informazioni necessarie al setup degli oggetti `BandDepth<_J>` e `BandDepthRef<_J>` è esterna e viene demandata alle classi `BandDepthData` e `BandDepthRefData`. La classe `BandDepthData` contiene variabili che esprimono:

- il numero di segnali nel dataset e numero di punti per segnale;
- l'eventuale file di input da cui leggere il dataset;
- l'offset destro e sinistro nella lettura del dataset;

- il livello di verbosità;
- il valore di `_J`
- il file di output nel quale verranno stampate le profondià determinate.

La classe `BandDepthRefData`, derivata da `BandDepthData`, è pensata per contenere i dati degli oggetti che calcolano le bande di profondità rispetto ad una sottopopolazione di riferimento contenuta nel dataset. Come spiegato nella sezione riguardante la classe `DataSetLevelled`, la creazione di un reference-set è possibile dividendo il dataset in livelli ed estraendo i segnali di riferimento da un particolare livello (che di default è il livello con ID $= 0$).

La gestione dei livelli richiede ad un oggetto `BandDepthRefData` di disporre di altre informazioni oltre a quelle della classe base:

- il numero degli elementi del reference set, che deve essere minore della cardinalità del livello corrispondente.
- il numero di elementi del test-set, che viene determinato sottraendo ai segnali totali il numero di elementi del reference-set.
- un'ventuale stringa contenente il nome del file che riporta gli estremi dei livelli considerati.

Gli oggetti `BandDepthData` e `BandDepthRefData` sono costruibili specificando direttamente il valore delle variabili che essi contengono, oppure passando un oggetto `GetPot` e la sezione di cui fare il parsing.

```
class BandDepthData
{
public:
  typedef GetPot data_Type;
  BandDepthData( const UInt & nbPz, const UInt & nbPts, ..., const UInt & J, ... );
  BandDepthData( const data_Type & dataFile, const std::string & section );
};

class BandDepthRefData : public BandDepthData
{
 BandDepthRefData( const UInt & nbPz, const UInt & nbPts, ... );
 BandDepthRefData( const data_Type & dataFile, const std::string & section );
};
```

La separazione tra costruzione e setup di `BandDepth<_J>` e `BandDepthRef<_J>` ha il vantaggio di rendere più semplice l'inizializzazione degli oggetti, che sono costruibili, rispettivamente, ricevendo variabili `BandDepthData` e `BandDepthRefData`, per copia o attraverso `shared_ptr`.

```
template < UInt _J >
class BandDepth : public BandDepthBase< All >
{
public:
    BandDepth( const bdData_Type & bdData );
};

template < UInt _J >
class BandDepthRef : public BandDepthBase< Reference >
{
public:
  BandDepthRef( const bdRefData_Type & bdRefData );
};
```

Setup di reference/test set per `BandDepthRef<_J>`

La classe `BandDepthRef<_J>` ha un'interfaccia pubblica differente da quella di `BandDepth<_J>`, dovendo fornire delle funzionalità per il setup di reference-set e test-set. Questo è anche il motivo per cui ese derivano da classi `BandDepthBase<_policy>` specializzate diversamente.

Come anticipato, la separazione in sottoinsiemi di riferimento e di test viene effettuata sfruttando il dataset a livelli `DataSetLevelled`. A tal proposito `BandDepthRef<_J>` mette a disposizione due metodi principali, la cui azione combinata (e ordinata) permette di impostare la divisione in sottopopolazioni.

```
template < UInt _J >
class BandDepthRef : public BandDepthBase< Reference >
{
public:
 void addToReferenceSet( const UInt & levelID, const UInt & size, const UInt & seed = 1 );
 void setTestSet();
 void clearReferenceSet();
};
```

`addToReferenceSet` permette di aggiungere un numero pari a `size` di elementi, provenienti dal livello `levelID`, all'insieme di riferimento. Se `size` è minore del numero di elementi del livello specificato, essi vengono estratti sfruttando un generatore di numeri pseudo-casuali inizializzato con `seed`.
Una volta che all'insieme di riferimento sono stati aggiunti tutti i segnali desiderati (che comunque devono essere in numero minore a quelli espressi nell'oggetto `BandDepthRefData`), si può finalizzarne la costruzione e creare l'insieme di test attraverso `setTestSet()`, che vi aggiunge tutti i restanti elementi del dataset.
Il reference-set può essere resettato con `clearReferenceSet()`.

Il metodo `computeBDs()`

Il metodo più importante dell'interfaccia pubblica di `BandDepth<_J>` e `BandDepthRef<_J>` è `computeBDs()`, con cui vengono effettivamente calcolati i contributi $\mathrm{BD}_N^{(j)}$, e che rappresenta il punto computazionalmente più oneroso del calcolo delle misure di profondità, presentando una complessità combinatoria rispetto all'aumento del parametro `_J`. Come anticipato nella Nota 3.3, esso viene specializzato rispetto al valore di `_J` poichè nel caso `_J`$= 2$ è possibile sfruttare un algoritmo molto più efficiente, proposto da Genton in [SGN12].
Nel caso di `_J`> 2 il calcolo di BD_N^J segue la definizione

$$\mathrm{BD}_N^{(j)}(f) = \binom{N}{j}^{-1} \sum_{1 \le i_1 < i_2 < \ldots < i_j \le N} \lambda_r \left(A(f; f_{i_1}, f_{i_2}, \ldots, f_{i_j}) \right)$$

Per ogni segnale nel dataset, per ogni punto del dominio viene calcolato il numero di combinazioni dei valori delle restanti funzioni che includono il valore del segnale corrente, sommando sui punti del dominio I e standardizzando infine per la sua misura.
L'idea dell'algoritmo di Genton, avanzato per `_J`$= 2$, è quella di ordinare i segnali ad ogni istante di tempo fissato e sfruttare le informazioni deducibili dai ranghi. Considerando il valore in $f_i(t_0)$ del segnale i-esimo al tempo t_0, dato il suo rango, $k(t_0)$, rispetto ai valori delle altre funzioni, allora esso sarà maggiore di $n_i(t_0) = k(t_0) - 1$ valori e minore di $m_i(t_0) = N - k(t_0)$, dove N indica il numero di funzioni nel dataset.
Nella formula che definisce BD_N^2

$$\mathrm{BD}_N^2(f_i) = \binom{N}{2}^{-1} \sum_{1 \le i_1 < i_2 < \le N} \lambda_r \left\{ A(f_i; f_{i_1}, f_{i_2}) \right\}$$

il sottoinsieme $A(f_i; f_{i_1}, f_{i_2}) \subset I$ degli istanti per cui $f_i(t) \in (\min\{f_{i_1}(t), f_{i_2}(t)\}, \max\{f_{i_1}(t), f_{i_2}(t)\})$ conterrà t_0 solo per un numero limitato delle possibili coppie del campione, in particolare $n_i(t_0) \cdot m_i(t_0)$.

Si ha dunque a disposizione la formula esatta:

$$\mathrm{BD}_N^2(f_i) = \binom{N}{2}^{-1} \int_I \frac{n_i(t) \cdot m_i(t)}{\lambda(I)} \, \mathrm{d}t$$

La strategia appena vista può, in linea di principio, essere estesa anche per alcuni valori di `_J`> 2, determinando delle formule esplicite analoghe a quella per `_J`$= 2$, e questo sarà parte di una successiva estensione di `HPCS`. Si noti come anche in questo caso la scelta di rendere le classi `BandDepth<_J>` e `BandDepthRef<_J>` template rispetto al parametro `_J` per poi specializzare il metodo `computeBDs()` sia appropriata.

Parallelizzazione di `computeBDs()`

Rappresentando il metodo che richiede maggior sforzo computazionale, in particolare quando `_J`$\neq 2$ oppure se il numero di elementi del dataset è grande, in `computeBDs()` il carico di lavoro viene partizionato tra tutti i processi disponibili. La strategia di parallelizzazione è uguale per `BandDepth<_J>` e `BandDepthRef<_J>` nei casi di `_J`> 2, ossia quando l'algoritmo per il calcolo delle profondità implementa la definizione. Nel caso di `_J`$= 2$, cioè quando viene utilizzato l'algoritmo di Genton, la strategia viene opportunamente modificata.
Nel caso `_J`$\neq 2$ si deve considerare che:

1. per il calcolo di un singolo termine $\mathrm{BD}_N^{(j)}(f_i)$ è necessario confrontare il segnale i-esimo con tutti i restanti segnali da processare, quindi la struttura dati che li raccoglie deve essere accessibile da tutti i processi. Per diminuire la comunicazione con granularità bassa, ma a discapito della memoria impiegata da ogni processo, il dataset viene letto durante il setup dell'oggetto `BandDepth<_J>` o `BandDepthRef<_J>` da tutti i processi.
2. l'obiettivo di scalabilità interessante non riguarda il valore di `_J`, che influenza la complessità del calcolo ma è pensato per rimanere comunque basso (nella pratica, visto lo sforzo richiesto, si utilizza `_J`$= 2, 3, 4$), piuttosto riguarda il numero di segnali da analizzare. Quindi in presenza di dataset di dimensioni anche molto differenti, fissando il valore di `_J`, idealmente si può trovare il numero di processi minimo affinchè il calcolo richieda un tempo minore di una soglia desiderata per ognuno di essi.

In seguito a queste considerazioni si è pensato di suddividere il numero di segnali da processare (cioè tutti nel caso normale e quelli del test-set nel caso con riferimento) tra i processi a disposizione in maniera bilanciata, attribuendo eventuali residui al master. Ogni processore calcola la profondità dei segnali corrispondenti e al termine un broadcast collettivo allinea le informazioni di tutti i processi.

Nel caso di `_J`$= 2$, a causa dell'algoritmo utilizzato, la strategia che porta ad un'esecuzione meno complessa è differente: nel caso di `BandDepth<_J>` i processi si suddividono inizialmente il dataset rispetto all'asse temporale, e ognuno di essi produce una struttura che riporta l'ordinamento del dataset per ogni istante di tempo. Al termine, una comunicazione collettiva trasmette a tutti i processi le strutture costruite, che vengono poi assemblate. Nel passo seguente il dataset viene partizionato rispetto al numero di segnali e viene calcolata la profondità di ciascuno, con successiva comunicazione collettiva finale per scambiare le informazioni di ciascun processo.
Nel caso di `BandDepthRef<_J>`, invece, è più conveniente dal punto di vista implementativo partizionare unicamente il dataset temporale, e costruire in ogni processo l'ordinamento del chunk di reference set corrispondente, con cui poi testare tutti gli elementi del test-set. Al termine viene svolta una riduzione collettiva tra tutti i processi.

Utilizzo tramite `BDFactory<BDPolicy _policy>`

L'istanziazione di un oggetto `BandDepth<_J>` o `BandDepthRef<_J>` richiede la conoscenza del valore del parametro `_J` a compile-time, mentre è preferibile che questa informazione sia esprimibile a run-time, e

impostabile esternamente attraverso un opportuno file data letto da `GetPot`, così da avere la massima flessibilità nell'utilizzo del codice. La struttura gerarchica di queste classi, che derivano da `BandDepthBase<_policy>`, classe template indipendente dal valore di `_J`, permette di conciliare le due esigenze costruendo un'apposita *abstract factory* (si veda ad esempio [Ale01]), con cui delegare a run-time la costruzione dell'oggetto richiesto. La factory `BDFactory<BDPolicy _policy>` deriva da una classe base che implementa una factory generica, e il costruttore viene specializzato per i valori `_policy=All` e `_policy=true`, visto che le classi `BandDepth<_J>` e `BandDepthRef<_J>`hanno in comune una classe padre che ha dovuto necessariamente subire una differente specializzazione dell'interfaccia pubblica:

```
template < class ProductType, typename KeyType, typename CreatorType >
class Factory
{
  public:
      bool registerProduct( const KeyType & key, const CreatorType & rule );
      bool unregisterProduct( const KeyType & key );
      product_Type * create( const KeyType & key );
};

template < BDPolicy _policy >
class BDFactory  : public Factory< BandDepthBase<_policy>, UInt, CreationRulePtrWrapper<
    BandDepthBase< _policy > > >
{};

template <>
BDFactory< All >::
BDFactory() : Factory< BandDepthBase<All>, UInt,  CreationRulePtrWrapper<BandDepthBase< All > >
    >()
{}

template <>
BDFactory< Reference >::
BDFactory() : Factory< BandDepthBase< Reference >, UInt,  CreationRulePtrWrapper< BandDepthBase<
    Reference > > >()
{}
```

Le regole di creazione dei prodotti nella factory sono implementate in una semplice gerarchia di funtori `struct`, il che, se emergesse la necessità, permetterebbe di adattare la regola allo specifico oggetto creato:

```
template < typename _returnType >
struct CreationRule
{
    CreationRule(){}
    virtual _returnType * operator()() const
    {
          return new _returnType();
    }
};

struct
CreateBDAll2 : public CreationRule< BandDepthBase< All > >
{
  CreateBDAll2(){}

  BandDepthBase< All > * operator()() const
  {
     return new BandDepth< 2 >();
  }
};

struct
CreateBDRef2 : public CreationRule< BandDepthBase< Reference > >
{
   BandDepthBase< Reference > * operator()() const
   {
      return new BandDepthRef< 2 >();
    }
};
```

Sono fornite le implementazioni per i principali valori di _J utilizzabili, _J$= 2, 3, 4, 5$. Poichè nella sua dichiarazione la factory BDFactory<_policy> deve specificare un solo tipo di creatore, viene disegnato un semplice wrapper, usato come Creator_Type in BDFactory<_policy>, che accettando uno shared_ptr ad una CreationRule permette di usare in maniera polimorfica le regole di creazione, adattando la struttura delle regole di creazione dei prodotti con l'esigenza di univocità del tipo creatore della factory.

```
template < typename _returnType >
class CreationRulePtrWrapper
{
public:
      CreationRulePtrWrapper( const creationRulePtr_Type & rulePtr );

      _returnType * operator()() const {  return (*(this->M_creationRulePtr))(); }
};
```

Nelle specializzazioni del costruttore della BDFactory<_policy> per _policy=All e _policy=true vengono automaticamente registrati i prodotti con _J$= 2, 3, 4, 5$.

3.2.3 Depth Measure

Il calcolo delle misure di profondità univariate $\mathrm{BD}_N^J(f)$ viene svolto tramite la classe DepthMeasure<_J,_policy>. Considerando la definizione,

$$\mathrm{BD}_N^J(f) = \sum_{j=2}^{J} \mathrm{BD}_N^{(j)}(f)$$

essa si deve interfacciare con le classi BandDepth<_J> e BandDepthRef<_J> per calcolare i contributi $\mathrm{BD}_N^{(j)}(f)$, e accumularne i valori per creare le misure di profondità univariate.

Dal punto di vista implementativo, essa è una classe template con due parametri, _J e la policy di tipo BDPolicy che specifica se si sfrutta la procedura normale oppure la suddivisione in test-set/training-set. DepthMeasure<_J,_policy> deriva da una classe base DepthMeasureBase<_policy>, indipendente da _J, che fornisce un'interfaccia pubblica essenziale utile per usare gli oggetti DepthMeasure<_J,_policy> in maniera polimorfica. Essi possono essere costruiti ricevendo uno shared pointer ad un oggetto BandDepthData o BandDepthRefData, a seconda del valore di _policy.
La conformità tra valore di _policy e oggetto corrispondente tra BandDepthData e BandDepthRefData è garantita dai seguenti typedef, che permettono di acquisire questa informazione direttamente dalle classi BandDepthBase<_policy>.

```
template < UInt _J, BDPolicy _policy >
class DepthMeasure : public DepthMeasureBase< _policy >
{
public:
 typedef BandDepthBase< _policy > bdBase_Type;
 typedef boost::shared_ptr< bdBase_Type > bdBasePtr_Type;
 typedef typename bdBase_Type::bdData_Type bdData_Type;
 typedef boost::shared_ptr< bdData_Type > bdDataPtr_Type;
};
```

I principali metodi pubblici di DepthMeasure<_J,_policy> sono computeDepths() e computeRanks(). Mentre computeDepths() è responsabile del calcolo delle profondità, computeRanks() permette di calcolare i ranghi corrisopndenti all'ordinamento indotto sul dataset dal calcolo delle profondità. I dati con profondità più bassa, e quindi meno *centrali* nel campione, avranno un rango basso, mentre i dati con profondità alta,

cioè i più centrali, avranno un rango alto.

Il metodo `computeDepths()`

Il metodo che calcola le profondità dei dati deve creare progressivamente oggetti di tipo `BandDepth<_J>` o `BandDepthRef<_J>` con valori di `_J` crescenti fino a quello dell'oggetto `DepthMeasure<_J,_policy>` corrente, inizializzarli e chiamare i loro metodi `computeBDs()`.
Gli oggetti delle classi `BandDepth<_J>` e `BandDepthRef<_J>` creati in `computeDepths()`, tuttavia, devono subire inizializzazioni differenti che non possono essere ricondotte ad un'unica struttura comune, dovendo esplicitamente usare nel secondo caso i metodi per il setup di reference-set e test-set. Per questo motivo il metodo `computeDepths()` deve avere definizioni differenti nei due casi. Poichè la classe `DepthMeasure<_J,_policy>` ha due parametri template non è possibile specializzare parzialmente rispetto al valore di `_policy` solo `computeDepths()`, senza specializzare l'intera classe, cosa da evitare essendo piuttosto corposa.
Di conseguenza si è pensato di creare un nuovo oggetto, `computeImplement<_J,_policy>`, a cui affidare l'implementazione del metodo `computeDepths()`. Visto che questo metodo dipende sia dalla `_policy` che da `_J`, `computeImplement` dovrà essere una classe template con gli stessi due parametri, ma la sua specializzazione nei due casi `All` e `Reference` sarà molto più agevole, essendo una classe di limitata estensione.

```
template < UInt _J, BDPolicy _policy >
class computeImplement
{
public:
   computeImplement( const bdDataPtr_Type & bdDataPtr );
   void compute();
};
```

Utilizzo tramite `DMFactory<BDPolicy _policy>`

In maniera del tutto analoga al caso di `BandDepth<_J>` e `BandDepthRef<_J>`, anche `DepthMeasure<_J,_policy>` può essere utilizzata attraverso una factory apposita, `DMFactory<BDPolicy _policy>`, derivata dal tipo generico di abstract factory di `HPCS`. La sua definizione e la struttura ricalcano quanto visto per `BDFactory<_policy>`.

```
template < BDPolicy _policy >
struct CreateDM2 : public CreationRule< DepthMeasureBase< _policy > >
{
   CreateDM2(){}

  DepthMeasureBase< _policy > * operator()() const
  {
    return new DepthMeasure< 2, _policy >();
  }
};

template < BDPolicy _policy >
class DMFactory
:
public Factory< DepthMeasureBase< _policy >, UInt, CreationRulePtrWrapper< DepthMeasureBase<
    _policy > > >
{};
```

3.2.4 Multivariate Depth Measure

Nel caso di dataset funzionale multivariato le misure di profondità corrispondenti sono determinate a partire dalle profondità univariate delle componenti attraverso una media con opportuni pesi:

$$\mathrm{MBD}_N^J = \sum_{k=1}^{s} p_k BD_{N,k}^J(f_k),$$

Il loro calcolo è possibile attraverso la classe `MultiDepthMeasure<_J,_policy>`, template rispetto ai parametri `_J` e `_policy`, derivata dalla classe base corrispondente `MultiDepthMeasureBase<_policy>`.

```
template < BDPolicy _policy >
class MultiDepthMeasureBase
{...};

template < UInt _J, BDPolicy _policy >
class MultiDepthMeasure : public MultiDepthMeasureBase< _policy >
{...};
```

La presenza di più componenti nei dati funzionali è gestita abbinando ad un costruttore `void` il metodo `addDimension`, che permette di aggiungere una dimensione a quelle già presenti.

```
void addDimension( const getPot_Type & dataFile, const std::string & section );
```

Non dovendo interagire con le altre dimensioni in maniera diretta, ognuna di esse è trattata come un dataset funzionale a sé stante di cui calcolare le profondità univariate. In seguito esse vengono combinate per dar luogo alle MBD_N^J. Questo permette di riciclare le strutture usate per processare i dataset univariati già presenti nel codice. Inoltre, le varie dimensioni sono univocamente caratterizzabili attraverso i corrispondenti oggetti di dati, `BandDepthData` o `BandDepthRefData`, che vengono salvati in una lista di shared pointer (per limitare i duplicati), memorizzata come membro privato della classe. Il metodo `addDimension` agisce proprio aumentando il contatore interno delle dimensioni e creando, a partire dalla variabile `GetPotricevuta`, un oggetto data da aggiungendo alla lista. Quando sarà necessario costruire l'oggetto `DepthMeasure<_J,_policy>` corrispondente alla dimensione desiderata, verrà selezionato dalla lista l'oggetto data relativo.

Un'altra funzionalità importante dell'interfaccia pubblica di `MultiDepthMeasure<_J,_policy>` è deputata all'impostazione dei pesi da usare nel calcolo effettivo delle profondità multivariate. È possibile specificare tali pesi sfruttando dei setters che li ricevano sotto forma di un contenitore (tali setters sono template rispetto al contenitore o ai suoi iteratori), oppure usando un setter che riceva un oggetto `GetPot` da cui dedurre il file esterno in cui essi sono salvati, per poi poterli leggere.

I metodi più importanti di questa classe sono `computeMultiDepths()` e `computeMultiRanks()`, omologhi dei metodi `computeDepths()` e `computeRanks()` della classe `DepthMeasure<_J,_policy>`.

```
void computeMultiDepths();
void computeRanks();
```

Il primo metodo, che calcola le misure di profondità multivariate, prende in considerazione sequenzialmente ogni dimensione e crea il corrispondente oggetto `DepthMeasure<_J,_policy>`, di cui chiama il metodo `computeDepths()` dopo averne fatto il setup attraverso l'oggetto data corrispondente a quella dimensione. Sempre iterativamente, le profondità vengono poi sommate con il peso apposito a quelle già ottenute. Il metodo `computeMultiRanks()` attribuisce i ranghi agli elementi del dataset in maniera concorde alla loro loro profondità.

Osservazione 3.4. Mentre nella classe `DepthMeasure<_J,_policy>` si è dovuta distingurere l'implementazione del metodo `computeDepths()` tra il caso `All` e `Reference`, qui non è necessario, poiché gli oggetti `DepthMeasure<_J,_policy>` stessi, creati nel metodo `computeDepths()`, hanno un'interfaccia pubblica identica.

Utilizzo tramite `MDMFactory<BDPolicy _policy>`

Analogamente alle classi precedenti, anche questa sfrutta la propria classe base indipendente dal parametro template `_J`, `DepthMeasureBase<_policy>`, per realizzare una factory che renda molto flessibile l'utilizzo degli oggetti di tipo `MultiDepthMeasure<_J,_policy>` all'interno di files sorgente manipolabili esternamente con `GetPot`.

Dunque, analogamente ai casi precedenti, si è implementata la classe `MDMFactory<BDPolicy _policy>`:

```
template < BDPolicy _policy >
struct CreateMultiDM2 : public CreationRule< MultiDepthMeasureBase< _policy > >
{
  CreateMultiDM2(){}

  MultiDepthMeasureBase< _policy > * operator()() const
  {
    return new MultiDepthMeasure< 2, _policy >();
  }
};

template < BDPolicy _policy >
class MultiDepthMeasureFactory
:
public Factory< MultiDepthMeasureBase< _policy >, UInt, CreationRulePtrWrapper<
    MultiDepthMeasureBase< _policy > > >
{...};
```

3.2.5 Oggetti ausiliari

Nell'implementazione del codice sono stati usati alcuni oggetti appartenenti a classi minori che incapsulano delle funzioni ricorrenti.

mpiUtility

Gli oggetti della classe `mpiUtility` forniscono informazioni sull'esecuzione parallela del codice. I principali metodi permettono di dedurre il rank del processo corrente, il numero di threads in esecuzione, l'ID del processo master, di stampare col processo master specifiche stringhe e di calcolare il tempo trascorso durante l'esecuzione di un processo esterno (sfruttando `MPI_Wtime`).
Gli oggetti `mpiUtility`, semplici ma utilissimi durante l'esecuzione parallela, vengono inclusi come membri privati o protetti da classi i cui metodi prevedono una parallelizzazione del calcolo.

extendedSort<T>

Una classe template rispetto al tipo `T` che effettua l'ordinamento in senso crescente degli elementi di un contenitore calcolando nel contempo i ranghi rispetto alla relazione d'ordine di tipo `less` disponibile per `T`. Per fare ciò utilizza le strutture dati e gli algoritmi della standard library.

combinationFactory

Questa classe fornisce un'interfaccia per il calcolo delle $\binom{N}{k}$ combinazioni di N interi presi a gruppi di k. L'utilizzatore può decidere se usarla facendo produrre le combinazioni in maniera sequenziale, partendo da una combinazione originaria e generando su richiesta la successiva aggiornando la precedente, oppure facendo calcolare tutte le combinazioni salvandole poi in memoria e restituendole su richiesta.

3.3 Test

3.3.1 Data set disponibili

Al fine di utilizzare i test o creare dei propri esempi è possibile sfruttare alcuni dataset multivariati disponibili insieme al sorgente di HPCS, generati artificialmente sfruttando R. Essi si trovano nella directory `synthetic`, e sono:

- AmplitudeSine, in cui i segnali generati sono sinusoidali bivariati ($s = 2$) con una variabilità congiunta localizzata unicamente nella loro ampiezza,
- `PhaseSine`, costituito da segnali sinusoidali bivariati con una variabilità congiunta localizzata unicamente nella loro fase,
- `PhaseAmplitudeSine`, in cui i segnali bivariati presentano sia una variabilità di fase che di ampiezza,
- `towerSine`, in cui i segnali sinusoidali bivariati non hanno variabilità naturale, ma sono generati a partire da una funzione di riferimento che viene traslata in maniera incrementale verso l'alto, così da produrre dei grafici impilati sulla verticale senza intersezioni.
- `outliersSine`, costituito da segnali bivariati con variabilità naturale sia di ampiezza che di fase, ai quali si sono aggiunti degli outliers in ampiezza generati ad hoc in maniera tale da allontanarsi velocemente ed improvvisamente dalla banda centrale dei dati.

3.3.2 Test disponibili

I test implementati in HPCS si trovano nella directory `test`. Per compilarli è sufficiente posizionarsi nella directory di `build` prescelta e dare i comandi:

```
cmake <source_dir>
make -j2
```

dove all'argomendo `-j2` si può sostituire il numero desiderato di processori. Durante la configurazione verranno linkati nelle cartelle i files richiesti dall'esecuzione degli esempi, tra cui i files `data` letti da `GetPot`, i files contenenti i pesi, i files dei livelli e alcuni script `GnuPlot` per visualizzare il risultato.
Di seguito vengono elencate le sotto-directories contenenti i test.

`dataSet`

In questa directory si trovano due sorgenti, `main_dataSet.cpp` e `main_dataSetLevelled.cpp` che mostrano, rispettivamente, come utilizzare le classi `DataSet` e `DataSetLevelled` per creare un dataset di dati funzionali. In entrambi viene fatto vedere sia come impostare i dati sfruttando un file esterno, sia come generarli nel `main` e passarli agli oggetti corrispondenti.

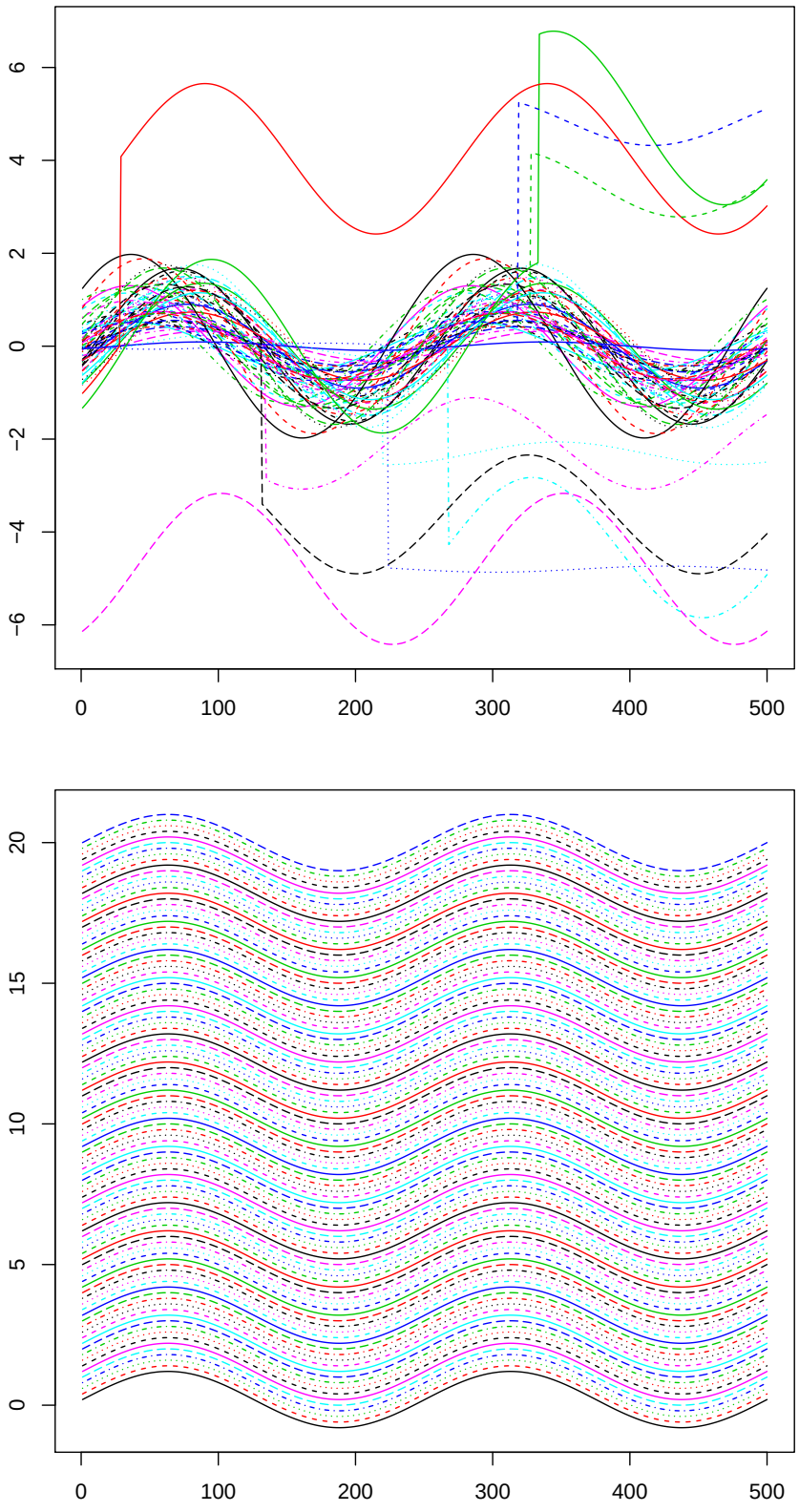

Figura 3.1: Plot delle prime componenti dei dati di `outliersSine`, in alto, e `towerSine`, in basso.

factory

In questa directory si trovano due esempi, `main_bandDepth.cpp` e `main_bandDepthRef.cpp`, che mostrano come utilizzare la factory `BDFactory<_policy>` per generare, rispettivamente, degli oggetti `BandDepth<_J>` e `BandDepthRef<_J>`, impostando contemporaneamente i corrispondenti oggetti di dati.

genton

In questa directory si trovano due test `main_gentonAll.cpp` e `main_gentonRef.cpp`, un file `user_fun.hpp` e un file `user_fun.cpp`. Scopo di questo test è assicurarsi che le profondità calcolate per `_J`$= 2$ attraverso l'algoritmo di Genton e quelle standard siano uguali. Visto però che le classi `BandDepth<_J>` e `BandDepthRef<_J>` implementano l'algoritmo semplificato, in `user_fun.hpp` e `user_fun.cpp` sono state dichiarate delle classi aggiuntive che calcolino le profondità di un dataset con l'algoritmo standard nel caso `_J`$= 2$.

Tali classi sono `BDDirectComputation`, `BDAllDirectComputation` e `BDRefDirectComputation`.

```
class BDDirectComputation
{...};

class BDAllDirectComputation : public BDDirectComputation
{...};

class BDRefDirectComputation : public BDDirectComputation
{...}
```

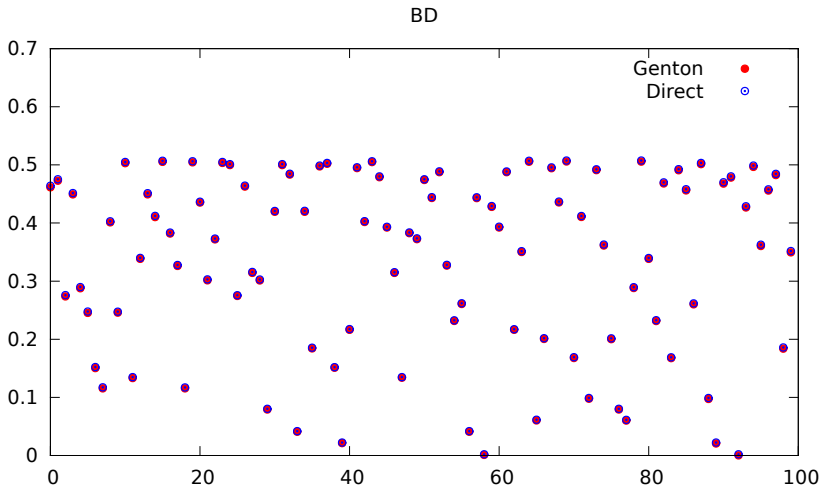

Figura 3.2: Risultato del test `main_gentonAll.cpp`

.

depthMeasure

Nella directory `depthMeasure` sono presenti quattro programmi che mostrano come utilizzare gli oggetti `DepthMeasure<_J,_policy>` e `MultiDepthMeasure<_J,_policy>` nei casi di policies `All` e `Reference`, impostando i parametri del calcolo attraverso un opportuno file data, letto da `GetPot`. Al termine dell'esecuzione vengono stampati i valori delle profondità cercate nei files specificati, che a loro volta vengono processati da

GnuPlot per produrre un grafico qualitativo.
Nella directory si trovano anche i files che riportano i pesi da utilizzare nel caso di profondità multivariate. Con pesi uniformi ed utilizzando il dataset bivariato di amplitudeSine si ottiene il risultato di figura 3.3.

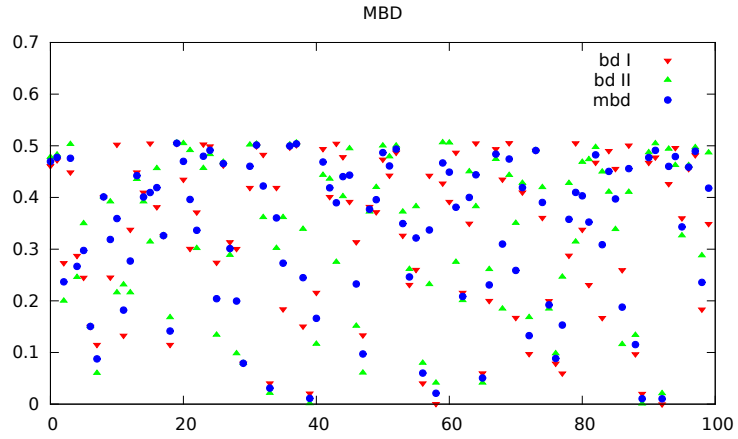

Figura 3.3: Output di main_multiDepthMeasureAll.cpp in corrispondenza del dataset amplitudeSine e pesi uniformi. Si noti come i valori delle profondità multivariate siano la media delle profondità univariate nelle singole componenti.

outliers

La directory outliers contiene due test riguardanti il dataset outliersSine (rappresentato in figura 3.1), che è stato costruito appositamente per mostrare il comportamento delle misure di profondità rispetto ad outliers morfologici. In entrambi i test, main_outliersAll.cpp e main_outliersRef.cpp, vengono calcolate le misure di profondità multivariate del campione rispetto alle corrispondenti procedure sia per _J= 2, sia per _J= 3. Dalla figura 3.4 si nota come i dieci outliers, che nella struttura dati occupano le ultime posizioni, abbiamo una profondità decisamente inferiore rispetto agli altri dati per entrambi i valori di _J, il che fa capire come le misure di profondità possano aiutare nei processi di outliers detection.

tower

Nella directory tower sono presenti due test per il calcolo delle profondità del corrispondente dataset, towersSine, sia nel caso di profondità normali, sia nel caso con riferimento. Come anticipato il dataset tower è costituito da segnali ottenuti traslando in maniera incrementale verso l'alto una stessa funzione sinusoidale, dando luogo a una struttura con grafici impilati. Come conseguenza, le profondità possono essere dedotte a priori, almeno nel caso All, e saranno crescenti man mano che i segnali si avvicineranno al *centro* del dataset. Il test serve appunto per confrontare le aspettative con il risultato del calcolo. Riportiamo in figura 3.5 le profondità del dataset determinate nel caso All e per _J= $2, 3, 4$.

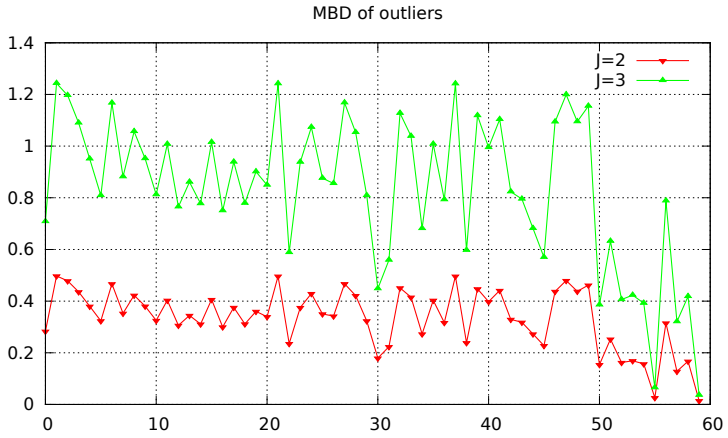

Figura 3.4: Output di `main_outliersAll.cpp` in corrispondenza del dataset `outliers` e pesi uniformi. Si noti come i valori delle profondità multivariate siano nettamente più bassi per gli outliers artificiali, indicizzati nelle ultime posizioni del data set, indicando correttamente la loro bassa centralità rispetto al dataset.

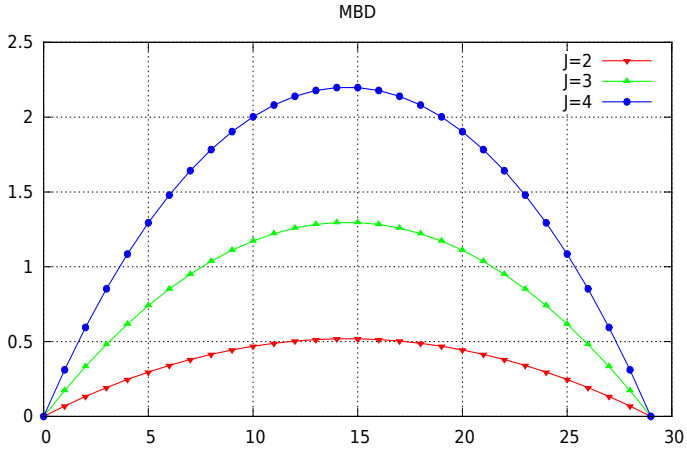

Figura 3.5: Output del test `main_towerAll.cpp` in corrispondenza del dataset `towersSine`, pesi uniformi e `_J`$= 2, 3, 4$. Si noti come i valori delle profondità multivariate presentino un andamento parabolico, con minimi nei valori estremi del dataset e massimi in corrispondenza della regione centrale. La dominanza dei profili per valori di `_J` crescenti è una conseguenza della definizione di misura di profondità.

Bibliografia

[Ale01] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70431-5.

[IP13] F. Ieva e A. M. Paganoni. "Depth Measures for Multivariate Functional Data". In: *Communications in Statistics - Theory and Methods* 42.7 (2013), pp. 1265–1276.

[LPJ07] S. López-Pintado e R. Juan. "Depth-based inference for functional data". In: *Computational Statistics and Data Analysis* 51.10 (2007), pp. 4957 –4968.

[LPR09] S. López-Pintado e J. Romo. "On the Concept of Depth for Functional Data". In: *Journal of the American Statistical Association* 104.486 (2009), pp. 718–734.

[SG11] Y. Sun e M. G. Genton. "Functional Boxplots". In: *Journal of Computational and Graphical Statistics* 20.2 (2011), pp. 316–334.

[SG12] Y. Sun e M. G. Genton. "Adjusted functional boxplots for spatio-temporal data visualization and outlier detection". In: *Environmetrics* 23.1 (2012), pp. 54–64.

[SGN12] Y. Sun, M. G. Genton e D. W. Nychka. "Exact fast computation of band depth for large functional datasets: How quickly can one million curves be ranked?" In: *Stat* 1.1 (2012), pp. 68–74. ISSN: 2049-1573.