Application Manual NDBI

Inhaltsverzeichnis

1 Overview

This manual describes the C++ and Python code and the data that are necessary to reproduce figures 2 to 7 of [1]. The code and the data are not maintained. They are frozen at the level close to the date of the last change of [1].

Warning: The code and data were tested only under Windows 10 (Professional edition). Running the code on a different operation system (e.g. LINUX) might require a format change for the data and other text files.

2 Getting started

2.1 Installations

Use Window 10 (Professional edition). Windows 11 should work also, but was not tested. Unzip the download of GitHub. Its root directory is NDBI/. The calculations were performed on an AMD Rhyzen processor. Extremely tiny deviations might occur when using another processor due to a different implementation of the floating point arithmetic.

2.1.1 Python

If you are unfamiliar with python, consult google and learn how to install python, python libraries with the pip utility and how to start python scripts. Be sure that python v3.10.0 or higher is installed.

The hardware requirements are those of python.

Verify that the following python libraries are installed or install them with pip:

- `numpy`
- `scipy`
- `matplotlib`

2.1.2 C++

You don't have to use Visual Studio. You can create figures 2 to 7 by running the delivered executable NDBI/EADSy.exe. The name EADSy arises, since NDBI is a subset of a fork of the larger project EADSy (**E**stimation **A**lgorithms for **D**ynamical **Sy**stems).

Installation of Visual Studio helps however, if you want to have a look at the C++ code. Use Microsoft Visual Studio C++ 17.0.2 or higher. Observe the hardware requirements. The installation is self explaining and ran always smoothly since Visual Studio 1.x ;). After installation double click the file `NDBI/NDBI.vcxproj`. Then the project opens. In case of an error or unfamiliarity with Visual Studio consult the Microsoft pages. These are almost always helpful.

The project must be compiled first. The reproduction of the C++ calculations is described in the section “C++ code” below.

2.2 Use Case: Create Figures 2 and 3 (Data Examples)

Run the script `NDBI/python/runDataExamplePlot.py`. The script simply reads the calculated derivative data from the folder structure described in the section “Folder structure” and creates the figures in the folder python/graphs of this folder. The creation of the derivative data is

described in the section “C++ code”.

2.3 Use Case: Create Figures 4 and 5 (Aggregated Comparisons)

Run the script `NDBI/python/runSummaryPlots.py`. The behaviour of the script is as in the previous subsection. It uses scripts from python/aggregation. The acronyms res and ref in the code stand for result data calculated with the various methods and reference data, which contain the theoretical result obtained by numerically differentiation the noiseless state input data.

2.4 Use Case: Create Figure 6 (Lotka Volterra Data)

Run the script `NDBI/python/runLotkaVolterraPlots.py`. The behaviour of the script is as in the previous subsection.

2.5 Use Case: Create Figure 7 (Higher Order Derivatives)

Run the script `NDBI/python/plotHigherOrderDeriv.py`. The script is short and self explaining.

2.6. Use Case: Calculate Derivatives from Data with the C++ Code

An example to run the C++ calculation from a windows command line is

```
cd ../../../
EADSy.exe data/3D/SKIP_SCAN/DIFFERENTIATION_BY_INTEGRATION/1Configuration.cfg
```

The first line navigates to the directors where EADSy.exe resides. The second line starts EADSy.exe with a single command line parameter, which points to the configuration file `1Configuration.cfg` that is used for the calculation. This file is read by CEADSyConfig.cpp, which is described in the C++ section. The quantities, which can be configured are defined in CEADSyConfig.h and are described in the next subsections.

If you want to know, where a specific configurable parameter is used in the project, right click on the getter function `CEADSyConfig.h` in Visual Studio, e.g. `CEADSyConfig::getOperationMode()`. Then a pop up menu opens. There choose the item, which shows the callers of this function.

More information on how the configuration works is given in the description of the C++ code below.

2.6.1 OPERATION_MODE

The available operation modes are defined in CEADSyConfig.h as

```
struct OperationMode
{
    FLAG AS_IS = "AS_IS";
    FLAG NOISE_SCAN = "NOISE_SCAN";
    FLAG SKIP_SCAN = "SKIP_SCAN";
};
const OperationMode m_OperationModesAvailable;
std::string getOperationMode() { return m_OperationMode; };
```

The struct defines the available Operation modes. AS_IS is used for the processing of a single file,

without adding noise or skipping data points. It is used for the real world data in `data/experimental/`. The meaning of the next two modes is obvious from [1]. They are used to create the data base for the use case "Create Figures 4 and 5" mentioned above. The line behind the struct instantiates this struct. The operation mode can be accessed by the mehod defined in the last line of the code snippet above. Below that line accessor methods for the various parameters of the noise and the skip scan are defined.

2.6.2 DIFFERENTIATION_METHOD

The available operation modes are defined in CEADSyConfig.h as below. The meaning of these flags becomes clear from the paper [1].

```
struct DifferentiationMethod
{
    FLAG FINITE_DIFF = "FINITE_DIFF";
    FLAG TANGENT_REGRESSION = "TANGENT_REGRESSION";
    FLAG FRIEDRICHS_MOLLIFIER_WEAK = "FRIEDRICHS_MOLLIFIER_WEAK";
    FLAG FRIEDRICHS_MOLLIFIER_NUMERIC = "FRIEDRICHS_MOLLIFIER_NUMERIC";
    FLAG SPECTRAL_LOWPASS = "SPECTRAL_LOWPASS";
    FLAG DIFFERENTIATION_BY_INTEGRATION = "DIFFERENTIATION_BY_INTEGRATION";
};
```

3 Folder Structure, Data Format and Python Code

The data format is as follows. The first row contains the number of independent variables, which is always 1 for NDBI. The independent variable is time. The other rows contain the data, where the first column is time and the other columns represent the dependent variable x1, x2, ...of the system.

The 3 subfolders of the root folder `NDBI/` are described in the next subsections.

3.1 NDBI/code

This folder contains the C++ code, which is described in the next section.

3.2 NDBI/data and subfolders

These folders contain the input and output data for all calculations described here. `NDBI/data` contains 6 subfolders. `2D, 3D, Driven3D, Hamiltonian4D` and `Jerks` contain the ODE solution data or input data described in appendix B of [1] (2023). as *.DAT files, where * is a placeholder for the name of the ODE system. Also theoretical derivatives for comparison with the calculations of [1] are contained as *.DAT_DERIVED files. The folder `experimental` contains the Lotka-Volterra data. Files in theses folders that do not end on .DAT or .DAT_DERIVED should be ignored.

Each of the folders mentioned in the previous paragraph contains 2 subfolders `NOISE_SCAN` and `SKIP_SCAN`. Each of these 2 contains 6 subfolders for the differentiation methods used in the paper. There reside the results of the noise and skip scan of the corresponding differentiation method. The file naming is as follows -

Noise scan:

`NDBI/data/2D/NOISE_SCAN/SPECTRAL_LOWPASS/STATE_FHN_0.03000000.PREP_DAT` contains the estimated state data using spectral lowpass filtering for the FitzHugh-Nagumo (FHN) model. 3% noise (scale is described in [1]) were added to the input data

NDBI/data/2D/FHN.DAT before the differentiation processing starts.

Skip scan:

NDBI/data/Jerks/SKIP_SCAN/FRIEDRICHS_MOLLIFIER_WEAK/DERIV_Jerk_1_20.PREP_DAT contains the estimated derivative using weak differentiation with the Friedrichs mollifier for the Jerk_1 ODE. 20 data points are skipped after each used data point of the file NDBI/data/Jerks/Jerk_1.DAT and 5% of noise were added to all numbers of skipped points ranging from 0 to 20 (increment: 2) before the differentiation processing starts.

A noise scan for all .DAT files in the folder data/Jerks/ can be started by double clicking the file data/Jerks/NOISE_SCAN/startEADSY_NOISE_SCAN.bat. The batch command text tells you, where the configurations for EADSy.exe reside. Similarly a skip scan for all .DAT files in the folder data/2D/ can be started by double clicking the file data/2D/SKIP_SCAN/startEADSY_SKIP_SCAN.bat

3.3 NDBI/python and subfolders

The python folders contain the scripts mentioned in the use cases above.

The NDBI/python/aggregation folder contains classes to aggregate the derivatives calculated with the skip scan and the noise scan (DERIV*.PREP_DAT files) and to compare them with the theoretical derivatives in the *.DAT_DERIVED files. The usage of the files in this folder becomes clear from the script NDBI/python/runSummaryPlots.py.

The NDBI/python/bock folder contains scripts for the bock algorithm that is used for the creation of figure 6 by the script NDBI/python/runLotkaVolterraPlots.py,which explains the usage of the files in that folder.

The NDBI/python/graphs folder contains scripts to create the figures in that directory and the output graphics. The usage of the python scripts in that folder becomes clear from the scripts mentioned in the use case described above.

4 C++ Code

For the skip and the noise scans C++ and the “pipes and filters” [3] pattern was used (see Chapter Design decisions). The pipes and filters pattern arranges operations on data like in an assembly line of industrial production. Concerning code documentation I am *not* in the camp of extreme programmers like Kent Beck [4], which say that the code is the documentation. However a well written source code makes up IMHO more than 50% of the documentation needed and the documentation aside from the code should be very concise and well thought over. The reason is that documentation and source code comments are not checked by the compiler, the linker and the tests. They can be arbitrarily wrong and completely misleading. The compiler and test checked code however tells you **always exactly** what the system does. Thus the documentation should mainly confine itself to the relation between classes and packages (i.e. folders).

4.1 Main file code/EADSy.cpp and classes code/util/.cpp*

EADSy.cpp contains the main function, where the startup method is called, which initializes the logging in util/CLog.cpp and reads the configuration with util/CEADSyConfig.cpp. The main function handles also exceptions like util/CEADSyException and the general error in (...) . An example is data/Configuration4All.cfg. There the wiring of the pipeline is described and parameters for the various calculation algorithms are given.

The class `pipeline/PipelineOperator` is instantiated in `main`. In the `run` method of the latter class instantiation of the pipeline and all the processing in the pipeline is performed.

The classes `util/CFileReader` and `util/CFileWriter` are used to perform text file io during the pipeline operation. The class `util/CSubDirFilter` helps in reading several input files simultaneously. Typical input files are state data corresponding to ordinary differential equation solutions (*.DAT files).

4.2 The classes in code/pipeline

4.2.1 The class CPipelineElement

This class is the base class for pipeline elements. Its derived classes perform all the calculations in the `next()` method.

4.2.2 The class CPullEnd

This class is derived from `CPipelineElement`. Its `next()` method calls the predecessers `next()` method, whicht in turn calls the predecessor of thatpredescessor `next()` method. In these `next()` methods all the processing is performed. An example is in the next subsubsection.

4.2.3 The class CPipelineOperator

This class creates the pipeline and runs it. For an understanding of the structure and operation of the pipeline, it is instructive to consider its creation. In the following line the pipeline is created and assigned to an instance of CPullEnd as

```
CPullEnd pullEnd(createPipelineFromConfig(&pSource, &pNoiseAdder));
```

The following method shows how the pipeline is wired up. First an instace of `CDataSource` is instantiated. This class reads the *.DAT files. It is a, possibly indirect, predecessor of all other pipeline elements and the direct predecessor of an instance of `CNoiseAdder`. An instance of this class is then the predecessor of a pipeline element that differentiates the data (e.g. an instance of `CFiniteDiff`). If in the configuration `EstimationMethod=NONE` is set, then no estimation of an ODE model or its parameter is performed.

```
CPipelineElement* CPipelineOperator::createPipelineFromConfig(CDataSource** ppS,
CNoiseAdder** ppNAD)
{
    CPipelineElement* pLastPE = *ppS = new CDataSource(0);
    pLastPE = createNoiseAdder(pLastPE, ppNAD);
    pLastPE = createDifferentiationMethod(pLastPE);
    pLastPE = createEstimationMethod(pLastPE);
    return new CDataSink(pLastPE);
}
```

The instance of `CDataSink` is then passed to the constructor of `CPullEnd`. The innocent and also somehow nonsensical looking method call

```
delete pPull->next();
```

does the following. It calls the next method of its predecessor, so that the chain of predecessors is called through the pipeline until one arrives at the instance of `CDataSource`. This reads a .DAT file and returns it in some data structures described below to its calling class `CNoiseAdder`. This

class adds the noise according to the configuration parameters and returns the data to an instance of `CFiniteDiff`, which performs the differentiation and returns the data to an instance of `CDataSink`, which outputs the state data and the calculated derivative to text files STATE*-PREP_DAT and DERIV*.PREP_DAT . Now it should be clear that by a call of the pullends `next()` method the raw data are read are pulled through the pipeline and the various calculations are performed on the data ***like in an assembly line of industrial production***.

4.3 The folders in code/pipeline

The description is confined to the classes used for the NDBI-paper.

4.3.1 pipeline/io

This directory contains the classes `CDataSink` and `CDataSource` mentioned above. They in turn use the classes `util/CFileReader` and `util/CFileWriter`.

4.3.2 pipeline/datastructures

Here the data structures for the protocol of the pipeline (i.e. data accessible to all constituents of the pipeline) are contained. The wrapper that contains all the data structures is `CPipelineData`. It aggregates `CDEData`, which contains the input data from the *.DAT files. Possibly some noise is added after reading by `CNoiseAdder` or the number of data points is reduced as a consequence of a skip scan. `CResultData`, which is also aggregated by `CPipelineData` contains results like estestimated parameters of the ODE. It also aggregates the differential equation in the class `pipeline/de/CDE` with ist active terms. The estimated state and derivative data by the various differentiation methods are stored in the `std::vector<CDEData*> m_PreparedDEData` member variable of `CPipelineData`.

4.3.3 pipeline/de

Here de stands for differential equations. The class `CDE` mentioned in the previous paragraph aggregates `CDETerm`. A term represents a summand on the right hand side of an ordinary differential equation in standard from. Its derived class `CDEMonomeTerm` represents a monome of an arbitrary degree in an arbitrary amount of variables. An instance of CDE can be created with `CDEFactory`, which can generate a polynomial right hand side of an ODE with an arbitrary number of variables up to an arbitrary order.

4.3.4 pipeline/datapreparation

This folder contains classes that perform the various differentiation methods mentioned in the paper. The names of the classes guide to the differentiation method.

The subfolder `datapreparation/integration` contain classes that perform NDBI. The class `CHermiteUnivariatePolynomial` represents a Hermite function in one variable. The class `CMultivariateHermiteSeries` contains all products of Hermite functions in m variables up to order n. The class `CDifferentiationByIntegration` does the differentiation by integration with the help of the latter two classes.

4.3.5 pipeline/estimation/parameters

The class CLeaveOneOutEstimator estimates parameter for an ODE by statistical jackkniving, i.e.

Linear regressions for the ODE parameters based on N data points are performed N times and each of the points is left out once. The average of these N estimates is used as the estimate of the parameter.

The equation system is represented by `parameters/equationsystems/CLotkaVolterra`, which is derived from `CEquationSystem` in the same folder.

4.4 The folders in code/numerics

The description is confined to the classes used for the NDBI-paper.

4.4.1 numerics/linalg

There the classes `CCMVector` and `CCMMatrix` reside, which represent vectors and matrixes in a chunk of **c**ontiguous **m**emory. For ease of computation the class `CCMDataManager` exists, which allows its derived classes (e.g. `CCMVector`) to access the private variables of `CCMMatrix`. `CCMLUDecomposer`, also derived from `CCMDataManager`, does LU decomposition and is programmed according to [5]. In the `linalg/solver` folder `CCholeskySolver` and `CGivensNES` (NES = normal equation solver) reside, which are derived from `CLinearEquationSolver` for exchangeability of the various linear equation solving algorithms. The solvers are also programmed according to [5].

4.4.2 numerics/dft

For calculations of the spectral derivative `CBluesteinFilter` is used. It uses `CBluesteinFFT`, which in turn uses `CCooleyTukeyFFT`. The latter two classes implement the Bluestein and the radix 2 Cooley-Tukey FFT algorithms [6].

4.4.3 numerics/stochastic/randomnumbers

This directory contains some random number generators, which are coded according to [7,8] and the algorithm becomes clear from the name of the class.

5 Design and other decisions

5.1 Usage of C++

If one uses C++ in current times, this has to be justified, since Python is nowadays the *lingua franca* of scientific computing. The typing paradigm of Python is duck typing or type inference: if something behaves like a duck, its type is inferred as a duck. Later a type system including inheritance was implemented in Python, which requires the import of some libraries. The usage of this type system made the Python code slower by a factor 1000 compared to the C++ code. The call of a simple lookup function in an inheritance hierarchy that *only* returns a simple boolean value required more processor time than the entire linear algebra work. This is unbearable, since typing is recommended, if one develops a larger code base. But also as mentioned in the application manual in the GitHub project OliverStrebel/pyLISS the linear algebra routines in Python were slower by a factor 40 than the corresponding ones in C++ Thus C++ was used for the lengthier calculations.

5.2 Usage of the pipes and filters pattern

Since in the skip scans and noise scans many similar data files are processed, we are certainly in the problem frame transformation [9]. The recommended pattern in this problem frame is pipes and filters [3,9].

6 Testing

Unit testing was avoided, since this results typically in a situation which is nowadays frequently called "code-hostage-taking" by unit tests, i.e. the code becomes hard to change, since tests are tightly coupled with the details of the code. This view is supported by the fact, that unit tests introduce a situation that resembles the code smell of parallel inheritance hierarchies [10].

Instead the use cases mentioned above can serve as test cases for a regression test, if one has checked in the data folder together with its subfolders to the Git or SVN source code control system. The test case fails, if the outputs files of the various programs change. The change is shown by the source code control system by setting a red cross on that file. The red cross is propagated up in the folder hierarchy till the data folder, so that one sees in one glance if something failed. The correctness of the code can also be ensured by the correctness of the calculations as presented by the figures. As a consequence the user of NDBI can perform only regression tests.

The correctness of parts of the code was ensured by testing at the *functional level* (different from unit tests!) with an own fork of CPP_UNIT in a different project named EADSyTest, which is not delivered. *Thus only regression testing is delivered with the NDBI fork of EADSy.*

7 References

[1] O. Strebel, Numerical Differentiation by Integration (NDBI) in the Context of Ordinary Differential Equation Estimation Problems (to be published).

[2] Use case: https://en.wikipedia.org/wiki/Use_case

[3] Frank Buschmann et. al.: Pattern Oriented Software Architecture. Wiley 1996

[4] Kent Beck: Extreme Programming Explained. Addison Wesley 2000.

[5] G. Golub et al.: Matrix Computations. Johns Hopkins University Press Baltimore 1986.

[6] https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

https://en.wikipedia.org/wiki/Chirp_Z-transform#Bluestein's_algorithm

[7] W. H. Press et al.: Numerical Recipes in Fortran. Cambridge University Press 1992.

[8] Paul Glasserman: Monte Carlo Methods in Financial Engineering. Springer (Heidelberg) 2003.

[9] Problem Frames: Rebecca Wirfs-Brock on her homepage https://www.wirfs-brock.com/PDFs/Intro_Problem_Frames.pdf and Frank Buschmann in the Conference „Object-Oriented Programming“ Munich (Germany) 2015.

[10] M. Fowler, Refactoring, Addison Wesley 1999.