Theoretica
Software Structure & Specification

M. Isgrò

Abstract—The project structure of the Theoretica mathematical library is introduced with respect to software design considerations, providing a general overview of its internal organization and its ongoing development.

Keywords—*scientific computing, software design, project structure*

Contents

1	Introduction	1
2	Directory Structure	1
3	Modules	1
4	Error Reporting	2
5	Programming Paradigm	2
6	Coding Standard	3
7	API Design	3
8	Testing	3
9	Benchmarks	4
10	Continuous Integration	4
11	Documentation	4
12	Future Development	4
	References	5

1. Introduction

Theoretica is a numerical and automatic header-only mathematical library in C++ for scientific and graphical applications. It is general purpose, with some specific fields of ongoing specialization. The purpose of the library is to provide access to powerful numerical methods and algorithms while keeping an elegant and simple interface, empowering researchers, students and developers. The following sections describe how it is organized with respect to project structure, workflow and design choices. Best practices for scientific computing have been taken into account during the development of the library [1][2].

2. Directory Structure

The library, publicly available online at GitHub[1], has a structured directory organization. The main folder contains text files with general information, first of all the `README.md` file which introduces the library. The `LICENSE` file and the `Makefile` for building the library are also at this level. The `BIBLIOGRAPHY` file contains a collection of material used for the development and research of the project, while `CODING_STANDARD` contains directives regarding writing code. The `CONTRIBUTING` file contains instructions for new contributors on how to participate to the project. The remaining files are organized in five different directories: `src`, `test`, `benchmark`, `examples` and `docs`:

1. The `src` folder contains the implementation files, consisting in header files with `.h` extension which contain the actual code for the library's functionalities. The code is subdivided into modules, as explained in section 3.

2. The `test` folder contains the test units for the library, which are implemented in different `.cpp` files, one for each module, with the naming convention "`test_module_name.cpp`". Test units are implemented using the Chebyshev[2] library, as introduced in paragraph 8. Unlike most libraries, Theoretica needs to test approximations and numerical methods for their accuracy, being a mathematical library, and Chebyshev provides facilities specifically for this task.

3. The `benchmark` folder contains benchmarking code, which is used to measure the performance of critical functions. These benchmarks also use the Chebyshev library, which has specific features for measuring the runtime of C++ functions. This component is further illustrated in paragraph 9.

4. The `examples` folder contains example programs which use the library's functionalities. These examples address common problems and showcase the library's usage. With the addition of new features or when a new common application to a problem is identified, a corresponding example program may be added.

5. The `docs` folder contains documentation files for the library. The documentation is periodically updated and generated using Doxygen, building on the documentation written in the source code, using Doxygen's features. It is good practice to document functions immediately when they are implemented, making it possible for both users and developers to understand the new code. The `docs` folder additionally contains a `txt` folder which is used to store text files such as the bibliography and the coding standard.

This directory structure ensures that different kinds of tasks and problems are compartmentalized.

3. Modules

Theoretica's implementation (stored in the `src` directory) is divided between many different modules, which correspond to different fields or subfields of interest. All of the library's functions and structures are contained in the `theoretica` namespace, aliased as `th`. The currently implemented modules are:

1. The `algebra` module defines vectors (`vec<Type,N>`) and matrices (`mat<Type,N,K>`) and linear algebra operations, as well as distances and norms. The linear algebra code is written in the `algebra.h` header, which uses templates to abstract from the specific data structures and is confined to the `algebra` namespace. Both statically and dynamically allocated vectors and matrices are implemented, using the case `N = 0` in template parameters to identify dynamically allocated structures. The `algebra_types.h` header defines common specialized classes of vectors and matrices, depending on size (`N = 2,3,4`) and data type (`Type = real,complex`).

2. The `autodiff` module contains code for forward-mode automatic differentiation. This feature is implemented for real functions of real variable in the `dual.h` and `dual_functions.h` files, while multivariate differentiation is implemented in `multidual.h` and `multidual_functions.h`. These classes are implementations of dual algebras with $\varepsilon^2 = 0$. Second order automatic differentiation, the case of $\varepsilon^3 = 0$, is implemented in the `dual2.h` and `dual2_functions.h` headers. Using these structures and functions, the `autodiff.h` header implements common differential operators such as gradient, divergence,

[1]github.com/chaotic-society/theoretica

[2]github.com/chaotic-society/chebyshev

Laplacian and curl.

3. The `calculus` module implements common numerical methods for real calculus problems, in particular, integrals, derivatives and differential equations. Common integral quadrature methods are implemented in the `integral.h` header, while *finite difference* approximation of derivatives is implemented in the `deriv.h` header file. Ordinary differential equation methods are implemented in the `ode.h` header, with the `ode_solution` class containing the discrete solution of a differential equation.

4. The `complex` module contains complex number structures, including complex numbers in algebraic form (`complex.h`) and exponential form (`phasor.h`) and quaternions (`quat.h`). The available types are `complex<Type>`, `quat<Type>`, `phasor<Type>` and `bicomplex<Type>`. Common complex functions of complex variable are implemented in `complex_analysis.h`

5. The `core` module contains functions and structures of general interest, such as real function approximations (`real_analysis.h`), mathematical constants and library parameters (`constants.h`) which also defines the `real` type for real variables, error handling with `errno` and `th::math_exception` (`error.h`), special functions (`special.h`) and operations on generic datasets (`dataset.h`).

6. The `interpolation` module implements polynomial (`polynomial.h`) and spline interpolation functions (`splines.h`), including a `spline` class for natural cubic splines.

7. The `optimization` module contains numerical methods for minimization (`extrema.h`) and root finding (`roots.h`) of univariate and multivariate (`multi_extrema.h`, `multi_roots.h`) real functions, leveraging the `autodiff` module's capabilities to automatically compute gradients.

8. The `polynomial` module implements a `polynomial<Type>` class for polynomials with `Type` coefficients (`polynomial.h`) and common orthogonal bases of polynomials (`ortho_polyn.h`).

9. The `pseudorandom` module implements functions and classes for the generation and usage of pseudorandom numbers. Several modern PRNGs are implemented, such as *xoshiro256++*, *wyrand* and *splitmix64* algorithms, as well as more consolidated ones (`pseudorandom.h`). The `PRNG` class provides simple, sequential access to pseudorandom generators. The `quasirandom.h` header also provides quasirandom sequences using Weyl's sequence. The `montecarlo.h` implements common Monte Carlo methods for integral approximation. Finally, the `sampling.h` header file provides methods for distribution sampling.

10. The `statistics` module contains functions for descriptive and inferential statistics. The `statistics.h` header implements common statistical functions such as numerically stable and accurate calculation of mean, variance, Gaussian moments, covariance and p-value. The `distributions.h` header contains most common probability distribution functions. The `histogram.h` header implements a `histogram` class for easy construction of histograms from data and running statistics. The `regression.h` header, with the `regression` namespace, contains functions for linear regressions, including a `linear_model` class which makes it simple to fit data points to a line. Lastly, the `errorprop.h` header provides automatic propagation of statistical errors from experimental datasets using automatic differentiation, as well as Monte Carlo error estimation.

11. The `signal` module currently implements the Fast Fourier Transform inside the `fft.h` header, and is planned to include other signal processing algorithms in the future.

4. Error Reporting

Errors are reported throughout the library by calling the `TH_MATH_ERROR` macro, which makes it immediate and easy to report an error during execution. The macro sets `errno` and throws a custom `th::math_exception`, depending on compile options. Exceptions may be turned on by defining `THEORETICA_THROW_EXCEPTIONS`, while exceptions may be used as the only error reporting method by defining `THEORETICA_EXCEPTIONS_ONLY`. Error codes as defined in `MATH_ERRCODE` overlap with the standard C definitions for `errno`. The `th::math_exception` class gives access to additional information regarding the error, such as a text description and an additional real value associated with the error. This code exemplifies a common case of error checking, where the first argument to `TH_MATH_ERROR` is the function (with all namespaces included except for `th`), the second argument is the additional real value (such as a length or a parameter) and the third argument is the error code:

```

if(v1.size() != v2.size()) {

    TH_MATH_ERROR(
        "algebra::dot", v1.size() INVALID_ARGUMENT);

    return Type(nan());
}
```

Code 1. Example of error reporting

The specific way to report the error is automatically chosen by the macro, with respect to `#defines`. A macro is also used in place of a function call to include additional information, such as the file and code line at which the error occurred.

5. Programming Paradigm

The programming paradigm of Theoretica is mostly functional, with the addition of object-oriented features when convenient. The most advanced and specific techniques of OOP are generally avoided for complexity and performance reasons. Data structures are often employed to represent the mathematical objects manipulated by functions. The following snippet shows this idea in action:

```

dvec2 f(dvec2 v) {
    return sqrt(v[0] * v[0] + v[1] * v[1]);
}

complex z = complex(1.0, 1.0);
vec2 v = { Re(z), -Im(z) };
mat2 A = {{1, 2}, {3, 4}};

real d = det(A) * divergence(f)(v);
```

Code 2. Different aspects of the programming paradigm

Encapsulation is used when access to the fields of a class may have consequences of the validity of the state represented by the class itself. For example, the `complex<Type>` class has two fields `a,b` of type `Type` and directly modifying them has the intended result. On the other hand, directly modifying fields in the `PRNG` class may radically alter its functioning, so its fields are made private and encapsulated. Numerical methods are generally implemented following functional programming's key principles whenever possible: *pure functions* with *deterministic output* and *no side effects*. If a given function is called with the **same inputs**, it will give the **same outputs** every time, without modifying the global state of the program. In addition to pure functions, when a certain common use case uses multiple algorithms in a particular way, such as linear regression, an object encapsulating the needed behavior may be implemented.

6. Coding Standard

The whole library follows closely the coding standard to make uniform stylistic decisions. All functions and classes must be declared inside the `theoretica` namespace (with potential sub-namespaces), no global variables except constants are used. **Snake case** is used throughout the code for functions (e.g. `normalize_z_score`) and classes (e.g. `linear_model`), while **camel case** is used for templates (e.g. `RealFunction`). Constants are written full upper case as is standard in C and C++ (e.g. `ROOT_APPROX_TOL`). As the library is header-only, **all functions except constructors** inside headers should be declared `inline`. Never use `using namespace` at global scope and do not use the `goto` directive. The following code snippet showcases common stylistic choices:

```

template<typename Vector>
inline real sum_pairwise(
    const Vector& X, size_t begin = 0,
    size_t end = 0, size_t base_size = 128) {

    if(end == 0)
        end = X.size();

    real sum = 0;

    // Base case with given size (defaults to 128)
    if((end - begin) <= base_size) {

        for (size_t i = begin; i < end; ++i)
            sum += X[i];

    } else {

        // Recursive sum of two halves
        const size_t m = (end - begin) / 2;
        const size_t cutoff = begin + m;

        sum = sum_pairwise(X, begin, cutoff, base_size)
            + sum_pairwise(X, cutoff, end, base_size);
    }

    return sum;
}
```

Code 3. Code style example

Code should generally be documented to explain the reasons behind different choices of implementation. Care should also be taken into naming variables and functions to avoid ambiguity.

7. API Design

The user interface of the library is designed to be easy to use and **direct**, while also leaving the possibility to more advanced users to tweak the underlying execution of the algorithms. This is accomplished by providing **general purpose methods** for a class of problems (take, for example, integral quadrature with the `integral` functions) as well as specific implementations of algorithms (in this case, `integral_laguerre` or `integral_romberg_tol`) which offer more control over execution and use a specific method. Classes are implemented to **simplify access** to certain features of common use. For example, a user may fit data to a line using the `ols_linear` or `wls_linear` functions and then compute the error using `ols_linear_error`, but may also simply instantiate the class `linear_model` passing the data points and related errors, letting the constructor handle all of the underlying algorithms to construct a data structure which holds all relevant information, such as coefficients of the regression, error estimates and p-value:

```

std::vector<real> X;
std::vector<real> Y;
real stdev_X = 1;

// Fill in X and Y

auto result = regression::linear_model(X, Y, stdev_X);
```

Code 4. Using `linear_model` to fit data to a line

The constructor for the class handles the different cases of no uncertainty, constant uncertainty or variable uncertainties on X and Y, calling the relevant functions. With the addition of stream and string cast operators, it is immediate to know the result of calculation or to write to a file. For example, the `histogram` class makes it extremely easy to build a histogram from a dataset using default parameters and stream it to a text file for plotting. Namespaces nested inside `th` are not always used, but are chosen to group functions belonging to the same subject or use case, especially when name clashes may happen. In this case, precedence is given to the most commonly used method.

8. Testing

Test units are stored in the `test` directory and are implemented in individual `.cpp` files with the naming convention `test_module_name.cpp`. The testing code uses Chebyshev, a testing library built explicitly to test Theoretica. Chebyshev is subdivided in three different modules:

1. The `prec` module measures the precision of functions by computing relevant error estimates with respect to an exact function.

2. The `err` module checks that functions correctly report errors when called with critical arguments.

3. The `benchmark` module measures the performance of functions by computing their average runtime over many different runs and inputs.

Test units mainly use the `prec` module, checking that the error bounds of the library's approximations are below a chosen threshold (generally 10^{-8} on the maximum, mean or relative error). The specific "fail_function" may be changed by passing it as argument or changing the modules parameters (`prec::state.defaultFailFunction`). This module evaluates a function f_{approx} many times over a given interval and uses the trapezoid method to approximate many error integrals such as the following:

$$\varepsilon_{mean} = \frac{1}{\mu(\Omega)} \int_\Omega |f(x) - f_{approx}(x)| dx \tag{1}$$

$$\varepsilon_{RMS} = \frac{1}{\mu(\Omega)} \sqrt{\int_\Omega |f(x) - f_{approx}(x)|^2 dx} \tag{2}$$

Chebyshev is straightforward to setup and use. This code snippet provides basic usage of the library:

```

#include "theoretica.h"
#include <cmath>
#include "chebyshev/prec.h"

using namespace chebyshev;
using namespace theoretica;


int main(int argc, char const *argv[]) {

    output::settings.outputFiles = { "example.csv" };
    prec::settings.defaultIterations = 1E+06;

  prec::setup("module-name", argc, argv);

```

```
        auto R_opt = prec::estimate_options<real, real
      >(
            prec::interval(-1E+06, 1E+06),
            prec::estimator::quadrature1D()
    );

    prec::estimate(
      "th::sqrt(real)",
      CAST_LAMBDA(th::sqrt, real),
      CAST_LAMBDA(std::sqrt, real),
      Rplus_opt
    );

    prec::equals(
            "th::powf", th::powf(2, 0.5), th::SQRT2
        );

  prec::terminate();
}
```

Code 5. Example usage of chebyshev::prec

The prec module is initialized using the function `prec::setup` with the name of the module being tested as argument. After this call, the library is ready to be used and `prec::estimate` is employed to register a function to be tested over a given interval. The `REAL_LAMBDA` macro is used to narrow a function to be a real function of real parameter and is needed in the case of multiple overloads. The `prec::equals` function is called to individually test equalities, like in the case of `th::square` being tested here. The list contains pairs of inputs and expected outputs of the function. At last, the `prec::terminate` function is called to signal the end of the test units and the registered actions are consequently executed, printing out the results both on standard output and on a file `test/test_module_name.csv` in CSV format. The parameters of the module can be modified by accessing the structure `prec::state`.

9. Benchmarks

Benchmarks are stored inside the `benchmark` folder and are implemented in different `.cpp` files. Like for test units, benchmarks use Chebyshev's specific functionalities to measure the performance of critical code. Like for test units, Chebyshev is setup and terminated using the respective functions.

```

#include "theoretica.h"
#include "chebyshev/benchmark.h"

using namespace chebyshev;
using namespace theoretica;
using namespace benchmark;


int main(int argc, char const *argv[]) {

    benchmark::setup("real_analysis", argc, argv);

        output::settings.outputFiles = {
            "benchmark.csv"
        };

        auto R_opt = benchmark_options<real>(
            10, 100000,
            generator::uniform1D(-1E+06, +1E+06)
        );

        benchmark::benchmark(
            "th::sqrt",
            CAST_LAMBDA(th::sqrt, real),
            R_opt
        );

    benchmark::terminate();
}
```

Code 6. Example benchmark code

The `BENCHMARK` macro is used to call a benchmark request by only specifying a real function and the interval of interest, resulting in a benchmark with random uniform generation over the interval. For a more general benchmark request, the `benchmark::benchmark` function is used, passing as always the name of the function, a function or lambda to run and a new argument which is a function that generates or provides the sample to run the function on. The `chebyshev::uniform_generator` function returns a uniform generator in the given interval and is commonly used to test real functions of real variable. Once the `benchmark::terminate` function is called, the benchmarks are executed and the results are printed to standard output and to a file `benchmark/benchmark_name.csv` in CSV format.

10. Continuous Integration

To continuously ensure the quality and correctness of the implementation, continuous integration (CI) is used. When a new commit is made on the online repo, all of the library's test units and example programs are built on three different systems: Linux, Windows and MacOS. This also ensures cross-platform portability of the code. Benchmarking code is also run on a single instance for every commit, as it is more resource and time intensive. The source code is also periodically scanned by an external tool to ensure high code quality.

11. Documentation

The documentation for the library, stored locally in the `docs` folder and online at chaotic-society.github.io/theoretica, is automatically generated using Doxygen from the documentation written alongside the code. Doxygen works by parsing the comments in front of a function, class or generic object marked by a triple slash `///`. Several commands can be used to specify additional information, such as the `@param` command which specifies a parameter for a function and the `@return` command which specifies what the function returns. LaTeX may be used to display formulas in the documentation by using `\f$` to enclose them. In addition, `@tparam` may be used to describe template parameters. The following code shows the usage of Doxygen comments to document a function:

```

/// Compute the real exponential.
///
/// @param x A real number
/// @return The exponential of x
///
/// The exponential is computed as
/// \f$e^{floor(x)} \cdot e^{fract(x)}\f$,
/// where \f$e^{floor(x)} = pow(e, floor(x))\f$
/// and \f$e^{fract(x)}\f$
/// is approximated using Taylor series on [0, 0.5]
inline real exp(real x) {
    // ...
}
```

Code 7. Example documentation of a function

The comments after commands such as `@param` are used as additional description for the function, while those before them are the short description. The `@note` and `@warning` commands may be used encase notices inside a yellow or red box for important information. The `@see` command may be used to specify a related function, such as the case of aliases. Classes and namespaces may be documented by using the `@class` and `@namespace` commands. Every implementation file should also include a `@file` command at its start to explain the content of the header. To make sure that the documentation is always up-to-date, the online version is updated automatically at each commit, using a CI/CD pipeline.

12. Future Development

Theoretica is nearing a stable release by extensively testing the library and providing fundamental features in scientific computing. Once a stable release is reached, further development will focus on general improvements to existing modules as well as new features, both general purpose functionalities of wide applicability in science and specific functions in areas of specialization. In particular, methods for advanced numerical linear algebra, the Finite Element Method and Finite Difference Method, machine learning algorithms and statistical models will be researched and implemented. The subject of PINNs (Physics-Informed Neural Networks) will be explored as well as time series analysis methods.

References

[1] G. Wilson, D. A. Aruliah, C. T. Brown, *et al.*, "Best Practices for Scientific Computing", en, *PLoS Biology*, vol. 12, no. 1, e1001745, Jan. 2014, arXiv:1210.0530 [cs], ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1001745. [Online]. Available: http://arxiv.org/abs/1210.0530 (visited on 05/27/2024).

[2] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, *Good Enough Practices in Scientific Computing*, en, arXiv:1609.00037 [cs], Oct. 2016. [Online]. Available: http://arxiv.org/abs/1609.00037 (visited on 05/27/2024).