Chebyshev
Software Structure & Specification

M. Isgrò

Abstract—The Chebyshev Testing Framework is introduced with respect to its features and software design choices. Its different modules and their implementation is also illustrated.

Keywords—*scientific computing, software testing, test units*

Contents

1. Introduction

The Chebyshev Testing Framework is a header-only testing library in C++, specialized in precision testing for software in scientific computing. It has been developed in conjunction with the Theoretica math library to be able to test its features in depth. All of the framework's functions are defined inside the `chebyshev` namespace, with additional namespaces for different modules. The framework is named after Pafnuty Chebyshev for his notable contributions to approximation theory and mathematics at large.

2. Scope

Chebyshev has been developed to test scientific software, numerical algorithms in particular. This is made possible by estimating error integrals of approximated functions. In addition to precision testing, Chebyshev also implements benchmarks and error checking of functions, through dedicated modules. Although it has been developed to test Theoretica, its implementation is general enough to be helpful in testing generic scientific software in C++. Chebyshev is thus an attempt at providing the scientific computing community with a unit testing framework for C++ code.

3. Modules

Chebyshev is subdivided in three different modules, depending on the type of testing, with two additional auxiliary modules:

1. The `prec` module is dedicated to testing the precision of approximations and is implemented in the `prec.h` header file. The main concern are real functions of real variable, but the implementation is generalized to arbitrary types and functions using templates.

2. The `benchmark` module is used to test the performance of functions and is implemented in the `benchmark.h` header file. It takes a function and runs it numerous times to compute its mean runtime. This feature may be used to benchmark performance critical code.

3. The `err` module tests the correct reporting of errors, either through exceptions or by setting `errno`. This is meant to check that when a function is called with certain arguments, such as values outside of its domain, it reports an error in the correct way. This module is implemented in the `err.h` header file.

4. The `output` module handles the printing of test results to the terminal and output files. It provides several different output formats which cover both pretty printing (`fancy`, `barebone`, `simple`) and output formats for different mediums (for example `csv` and `latex`).

5. The `random` module provides functions to generate random numbers and random samples for testing purposes.

Each module has its own `<module>::setup` and `<module>::terminate` functions which are used to setup the environment for testing and terminating (and printing out) the executed test units. It is advised to write a single program to test some module or functionality and to use one Chebyshev module at a time (e.g. write a precision test unit, with a distinct program to benchmark performance). The three modules have a `<module>::state` structure which contains options for the module and default values.

4. Precision Testing

The precision of numerical methods and approximations in general may be estimated by using the `prec` module, which implements two general types of tests: `prec::estimate` and `prec::equals`. The `prec::equals` function is used to test the equality of two evaluated expressions of any given type, potentially up to a given tolerance, as is needed for floating point numbers. `prec::equals` works by taking in two values of the same type x_1, x_2 and applying a `distance` function d on them, checking that the distance is smaller then the tolerance ϵ. The test is considered to be passed if:

$$d(x_1, x_2) \leq \epsilon \tag{1}$$

The distance function has signature `d:(T, T)` $\to$ `Real`, where `Real` is an alias for a floating point type, such as `long double`, and `T` is the type of the evaluated expression. The type `T` does not need to be a real number, it may be a complex number or even a matrix or vector, or more structured data type. It is only necessary that the distance function is modified accordingly, taking in a couple of elements of type `T` and returning a real number. This function is thus dedicated to checking equivalence between two values of any given type. On the other hand, one may need to test an approximation on a more fundamental level, for example a real function needs to be tested on its domain, not at a single point. The `prec::estimate` function is the solution to this problem. The approximation f' and an exact function f to compare it to are taken in as arguments, as well as a tolerance ϵ, an estimator $\mathbb{E}$ and a fail function $\mathcal{F}$. Note that in most applications, you do not need to specify an estimator or a fail function, as the library provides default arguments for common test cases. The estimator is used to compute estimates of error integrals over the domain (additional arguments are the domain to test and the number of iterations):

$$\epsilon_{max} = \max_{x \in \Omega} |f'(x) - f(x)|$$

$$\epsilon_{mean} = \frac{1}{\mu(\Omega)} \int_{\Omega} |f'(x) - f(x)| dx$$

$$\epsilon_{rms} = \frac{1}{\mu(\Omega)}\sqrt{\int_\Omega |f'(x) - f(x)|^2 dx}$$

$$\epsilon_{rel} = \frac{\int_\Omega |f'(x) - f(x)| dx}{\int_\Omega |f(x)| dx}$$

The estimator is a function which takes in the exact and approximate functions and the other parameters such as the number of iterations and the domain. The function signature of the exact and approximate functions is `f:Args... → R` (where variadic templates are used to take an arbitrary number of arguments), while that of the estimator (simplified to show relevant parameters) is:

$$\mathbb{E} : (f', f, \Omega) \to \{\hat{\epsilon}_i\}$$

where $\hat{\epsilon}_i$ are the different estimates of the error integrals. In practice, the results of an estimator are stored inside the `estimate_result` structure. After the errors have been estimated, the fail function is called to determine whether the test was passed or not. The signature of a fail function is $\mathcal{F} : (\{\hat{\epsilon}_i\}, \epsilon) \to$ `bool` (where ϵ is the tolerance). If the boolean result of the fail function is true, the test has failed (the fail function answers the final question *did the test fail?*). The overall expression which determines the passing of a precision test is thus:

$$\neg\, \mathcal{F}(\mathbb{E}(f', f, \Omega), \epsilon) \tag{2}$$

Common fail functions may consist in failure when a chosen error integral estimate is bigger than the tolerance:

$$\mathcal{F} \equiv (\hat{\epsilon}_i > \epsilon)$$

An additional implementation consideration is to make the test fail when the error estimate is NaN (since a simple comparison with the tolerance may fail to signal an error). So far, the exact form of the estimator has not been discussed. This is because the estimator may take any form, estimating errors on very different kinds of functions and types (such as matrices or vectors). The two most common classes of estimators are *deterministic* and *stochastic* estimators. Deterministic estimators use an integral quadrature method to approximate the error integrals, for example using the trapezoid or Simpson's method with a high number of points. These methods are deterministic in nature (given the same input, they give the same output) and may give quite accurate estimates, but are increasingly difficult to use on multidimensional domains or really complex domains. On the other hand, stochastic estimators use random number generators and statistical estimators to estimate error integrals. For example, Monte Carlo methods may be used to estimate error integrals. This approach is preferable when the domain of approximation is too high dimensional to check deterministically. Stochastic estimates have statistical variance as they depend on the sampling of a probability distribution. Built-in estimators are provided in the `prec::estimator` namespace. Since `prec::equals` and `prec::estimate` may take a great number of parameters, the `equation_options` and `estimate_options` structures can be used to fine tune the parameters and run many tests using the same options, as well as define custom options which are passed to the estimator through a `(string,Real)` key-value map.

4.1. Property testing

When precision testing, it is usually necessary to have a reference function with higher accuracy to compare the approximation against. If the function under test must theoretically satisfy certain properties or equations, they may be exploited to get rid of the need for another function to compare to. The `prec::property` namespace does exactly this, providing precision estimation functions such as `involution`, `identity`, `idempotence` and `homogeneous`, which correspond, in the same order, to the following equations:

$$f(f(x)) = x \qquad f(f(x)) = f(x)$$
$$f(x) = x \qquad f(x) = 0$$

By calling these functions, only the approximation is taken as a parameter, and the error is estimated against these equations.

5. Benchmarks

Benchmarks are used to measure the average runtime of a given function over a certain domain or random sample distribution. This is implemented by the `benchmark::benchmark` function, which takes in the function to test `f:Args...→R` and runs it over a sample input set Ω, which is generated through a generating function `g:uint→Args...` (where `uint` is an unsigned integer type used as an index inside the generator). Alternatively, the benchmark may be run on a sample set directly passed by argument. The function is run over m runs (different runs are run using the same input set) made up of n iterations (the size of the input set), measuring the mean value of the runtime over the $m \cdot n$ evaluations. An internal counter of type `R` declared `volatile` is used, when `R` is a type which admits a sum operator and a copy constructor, to ensure that the evaluation of the function is not optimized away.

6. Error checking

The `err` module is used to check that functions correctly report errors, for example modifying `errno` or throwing an exception. It provides three different classes of test cases: assertions, `errno` checking and exception checking. Assertions check that a given expression evaluates to true and are available through the `err::assert` function. Assertions are often similar in nature to the `prec::equals` directive, the key difference being that assertions test the truth of an expression, without taking into considerations a tolerance (which is fundamental for floating point values). Assertions are useful when a certain expression, not necessarily mathematical, must be true at runtime. Many functions, such as mathematical functions in the standard library, set `errno`, using it to report errors. The `err::check_errno` functions makes it possible to check that when a given function is executed with the given input, `errno` is set to a specific value (or alternatively to multiple flags). It is also possible to use an `InputGenerator` function which takes an unsigned integer value and returns an input to the function. The `err::check_exception` function makes it possible to check that a given function throws an exception when called with certain arguments. It is also possible to check that the function throws a specific type of exception. Like for `errno`, it is also possible to use an `InputGenerator`.

7. Conclusion

Chebyshev is a testing library tailored to the needs of the scientific computing community, and new features are continuously added to improve its capabilities at testing scientific software. Some considerable additions under development are *numerical stability analysis* and *complexity analysis*.