FunC

2.2.1

Generated by Doxygen

1 Main page

FunC (Function Comparator) is a C++ tool for approximating any univariate, pure function (without any side-effects) $f : \mathrm{TIN} \to \mathrm{TOUT}$ with a lookup table (LUT) over a closed interval $[a, b]$. TIN and TOUT must be types with overloads for operator+,-, and there must be a commutative operator*:TIN $\times$ TOUT $\to$ TOUT (so TOUT approximately forms a vector space over the approximate field TIN). We take a LUT as any piecewise approximation of f, so a LUT of f takes the following form.

$$L(x) = \begin{cases} p_0(x) & \text{if } x_0 \leq x < x_1, \\ p_1(x) & \text{if } x_1 \leq x < x_2, \\ \vdots & \vdots \\ p_{N-1}(x) & \text{if } x_{N-1} \leq x \leq x_N, \end{cases}$$

where p_k are usually (but not necessarily) polynomials. FunC can build LUTs where each p_k in the equation above are interpolating polynomials (up to degree 7 with Chebyshev nodes of the first kind or degree 6 with Chebyshev nodes of the second kind), Taylor polynomials (up to degree 7), Pade approximants, or degree 3 Hermite interpolating polynomials. The x_k in the equation above partition $[a, b]$, so $a = x_0 < x_1 < \ldots < x_n = b$. The x_k can form a uniform partition (so $x_k = a + k(b - a)/N$) or be an automatically generated nonuniform partition. The user has no control over the nonuniform partition to ensure the hash only takes 6 FLOPs and zero comparisons. FunC aims to streamline finding a good LUT of f for a user's application. To do so, we measure factors such as

- absolute and relative tolerances for error
- domain usage (i.e. the inputs to f during the runtime of the user's program)
- evaluation frequency (i.e. how much work is being done in between calls to f)

FunC's DirectEvaluation class measures the first two, and a LookupTableGenerator optimizes a LUT's step size according to maximum tolerances for error. Installation details are covered in the Readme.md on FunC's GitHub https://github.com/uofs-simlab/func.

2 Example usage

Before delving into the details of each feature in FunC, we provide several examples of replacing a mathematical function with a LUT. We then generalize this process and summarize it as a workflow diagram. After this, we give a brief overview of the class structure of FunC.

2.1 How to use FunC to replace a mathematical function with a LUT

The following example illustrates how to use FunC to build a cubic LUT for an exponential integral over $[0.01, 2]$ with step size $h = 0.1$.

```
#include <boost/math/special_functions/expint.hpp>
#include <func/func.hpp>
#include <iostream>

template <typename T> T f(T x){ return boost::math::expint(1,x); } // user function
int main(){ // build an approximation of f over [0.01,2] with uniform stepsize h=0.1
  func::FunctionContainer<double> fc {FUNC_SET_F(f,double)};
  func::UniformExactInterpTable<3,double> LUT {fc, {0.01,2.0,0.1}};
  double x; std::cin >> x;
  std::cout << "f(" << x << ")~" << LUT(x) << std::endl; // print piecewise cubic approx.
}
```

We observe the following:

- The LUT is only used once.
- The approximation is built according to a step size. We do not currently know how much error the approximation has.
- It is impossible in principle to know what the user will input, so the subdomain $[0.01, 2]$ is likely insufficient.

In this case, using a LUT is a poor choice because the overhead from building the LUT is not balanced out by repeatedly calling the LUT, the LUT introduces an unknown amount of error, and is only valid over a small subset of the original domain of f. By default, FunC's LUTs do not perform bounds checking because doing so introduces nontrivial slowdown when calling operator(). such, any user input outside the range $[0.01, 2]$ is undefined behavior. It is up to the user to guarantee this undefined behavior is not possible. This is generally done by using a LookupTable container (defined shortly), or making the interval $[a, b]$ larger. We now consider the following improved example using the same f as before.

```
int main(){
  // build an approximation of f over [0.01,2] with uniform stepsize h=0.1
  func::FunctionContainer<double> fc {FUNC_SET_F(f,double)};
  func::FailureProofTable<func::UniformEqSpaceInterpTable<3,double>> lut {fc, {0.01,2.0,0.1}};
  // compute max error of LUT
  func::LookupTableGenerator<double> gen(fc,{});
  std::cout << "error=" << gen.error_of_table(lut,1.0) << "\n";

  // take two numbers from the user: max and nevals
  double max; std::cin >> max; int nevals; std::cin >> nevals;
  // print nevals random numbers in the range [0.01, max].
  std::mt19937 mt(0); std::uniform_real_distribution<double> unif(0.01,0.01+std::abs(max));
  for(int i=0; i<nevals; i++){
    double x = unif(mt);
    if(x < 2.0) std::cout << " f(" << x << ") ~ " << lut(x) << "\n";
    else std::cout << " f(" << x << ") = " << lut(x) << "\n";
  } std::cout << std::endl;
}
```

Sample input and output:

```
error=0.0216333
3 3 # user input
 f(1.78261) ~ 0.0663324
 f(2.53435) = 0.0238136
 f(2.57526) = 0.0225693
```

This example is better suited for a LUT because it can involve numerous repeated applications. The undefined behavior is gone because the FailureProofTable resorts back to f's defining mathematical formula if x is out of the bounds of the UniformExactInterpTable. The LookupTableGenerator provides an estimate of

$$E(L) = \max_{x\in[a,b]} \frac{|f(x) - L(x)|}{a_{\text{tol}} + r_{\text{tol}}|f(x)|},$$

(by default with $a_{\text{tol}} = r_{\text{tol}} = 1$ but these values are adjustable). Whether this is an acceptable amount of error depends on the use case. If the user provides $(a_{\text{tol}}, r_{\text{tol}}) = (1, 0)$, then $E(L)$ is the absolute error of f. Similarly, if $(a_{\text{tol}}, r_{\text{tol}}) = (0, 1)$, then $E(L)$ is the relative error of f. The user can provide any positive values for a_{tol} and r_{tol}, and if they are both nonzero then $E(L) < 1$ does not necessarily imply L satisfies *both* the relative and absolute error tolerances individually. The following code shows an MWE of building a DirectEvaluation and a LUT of a special function. A DirectEvaluation can record every argument passed to its operator() in a histogram which is critical for determining useful bounds a, b for a LUT. A DirectEvaluation can simulate error in a LUT by perturbing its arguments by $r_{\text{tol}}, a_{\text{tol}}$. So, the DirectEvaluation can return rtol*R*f(x)+A*atol where R and A are uniformly distributed random numbers in $[-1, 1]$.

```cpp
/* User function here. Some LUT types require derivatives of the user
 * function, and this is provided though Boost's automatic differentiation
 * library. To use automatic differentiation, the definition of f must be
 * templated, and any function f calls must have overloads for Boost's autodiff_fvar */
#include <boost/math/special_functions/jacobi_elliptic.hpp>
template <typename T> T f(T x){ return boost::math::jacobi_cn(0.5, x); }

/* If FUNC_DEBUG is defined before including func.hpp, then any DirectEvaluation or
 * FailureProofTable will have a histogram that stores each argument passed to their
 * operator() during program runtime */
// #define FUNC_DEBUG
#include <func/func.hpp>
#include <iostream>
int main(){
    /* FUNC_SET_F is a macro required to take advantage of Boost's automatic differentiation.
     * - If f is templated on two types, call as FUNC_SET_F(f,TIN,TOUT)
     * - If f cannot be templated as shown FunctionContainer can be constructed with f<double>
     *   but autodiff will not be available */
    func::FunctionContainer<double> fc {FUNC_SET_F(f,double)};

    /* Arguments to a DirectEvaluation are
     * (FunctionContainer fc, TIN min=0, TIN max=1, uint nbins=10, TOUT aerr=0, TIN rerr=0)
     * where min,max are used as bounds for the histogram */
    func::DirectEvaluation<double> de {fc,0,2,10,1,1};
    /* Call the function on line 7 with T=double and x=1.011.
     * If FUNC_DEBUG is defined, then f(x)(1+R*rerr)+A*aerr is returned instead,
     * where A,R are random numbers sampled from a uniformly distributed random variable over [-1,1] */
    std::cout << de(1.011) << "\n";
    std::cout << de << std::endl; // print histogram to stdout if FUNC_DEBUG is defined

    /* build a LUT of f over [0,2] with a step size of h=0.1. Each subinterval will use degree 3
     * Chebyshev interpolating polynomials */
    func::UniformChebyInterpTable<3,double> lut {fc, {0.0,2.0,0.1}};
    std::cout << lut(1.011) << "\n"; // use a piecewise cubic polynomial to return an approx of f(1.011)

    std::cout << lut << "\n"; // print info about lut
}
```

Instead of building a LUT according to a step size, it is better to build a LUT according to tolerances for error that are as large as possible for the user's purpose.

```cpp
int main(){
    auto tableTol = 1e-2;
    func::FunctionContainer<double> fc {FUNC_SET_F(f,double)};
    LookupTableGenerator<TYPE> gen(func_container,0.1,2.0);
    /* Use a LUT factory to build according to a string. Using a NonUniform LUT */
    auto lut = gen.generate_by_tol("NonUniformChebyInterpTable<3>",tableTol);
}
```

Any member function of the LookupTableGenerator class that returns a std::unique_ptr<LookupTable> (generate_by_step, generate_by_impl_size, generate_by_tol) can take an optional std::string filename. When given a filename, that member function returns the result of generate_by_file(filename) if filename exists. Otherwise, that member function saves its result to filename before returning. As such, the code used to generate a LUT is automatically optimized for future runs of the user's program.

2.2 A general workflow

The following is the general workflow we suggest for replacing a mathematical function with a LUT using FunC.

- The user identifies a mathematical function whose evaluation consumes a substantive proportion of total runtime. Further, the mathematical function itself must be complicated enough to warrant the use of a LUT. For example, it is unlikely that a LUT will speed up an elementary function such as $\sin(x)$, e^x, etc. because those functions have been continually optimized for decades. Good candiates for a LUT include deeply nested function compositions, long summations/products, and special functions. Also, the process of building a LUT is not without expense. FunC must either evaluate the user's function a set number of times or read the LUT from a file (both are potentially slow). So, the user's code must evaluate f a sufficiently large number of times for construction of a LUT to be appropriate.
- The user determines an interval of approximation $[a,b]$ and tolerances for error in their LUT. LUTs over smaller domains and with coarser error tolerances perform faster and use less memory.
- The user should experiment with a wide variety of different LUTs in isolation, each constructed according to the user's tolerances for error. FunC provides the LookupTableGenerator to test each LUT in isolation.
- After determining 1–2 ideal LUTs, the user should still benchmark their code after using those LUTs. Use of a LUT necessarily increases memory usage and that can result in overall slowdown compared to the original code.

We note that the above process does not put any emphasis on the specific approximation used on each subinterval. Again, most LUTs perform similarly because they share much of the same source code. The main determining factor for a LUT's performance is its *order* of convergence. For a user to determine a suitable order for their application, they must experiment with several LUTs in isolation.

3 FunC's class structure

With this workflow in mind, we now present a brief summary of FunC's debugging tools and important classes. To visualize how each of these classes relate to one another, FunC's UML class diagram is provided. Each of FunC's classes are related to LookupTable because they either implement LookupTable, encapsulate a LookupTable implementation, or construct LookupTable implementations.

- Classes implementing the LookupTable interface implement a useful set of functions for approximating a mathematical function with a piecewise function. The most important member function of a LookupTable implementation is its operator() (because it returns approximations of $f(x)$).

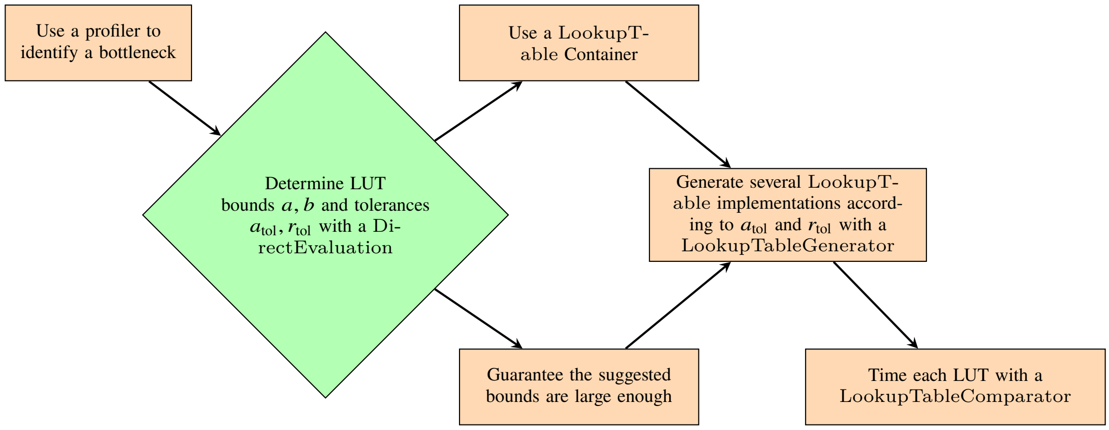


Figure 1 Suggested FunC workflow

- The MetaTable class provides all the mechanisms required to approximate a mathematical function with an array of Polynomial. MetaTable exists to reduce code redundancy and as such is templated on several parameters: the number of polynomial coefficients for each subinterval, TIN,TOUT, and whether the partition of $[a,b]$ is uniform. Currently, every class that *constructs* a piecewise approximation of f inherits from MetaTable.
- The LookupTableGenerator class uses the factory design pattern and provides several member functions for building any supported LUT according to a step size, data size, or tolerances for relative and absolute error.
- The DirectEvaluation class is used for profiling or debugging as per the preprocessor macro FUNC_DEBUG. When this macro is defined, it helps determine useful LUT bounds (by recording each argument it is given before returning $f(x)$) and tolerances for relative and absolute error (by perturbing its outputs).
- The FailureProofTable class passes each of its arguments x to the LUT it encapsulates after checking whether x is within its LUT's bounds. If x is not within the LUT's bounds, then the FailureProofTable computes $f(x)$ using the defining mathematical formula for f. This makes LUTs safe and straightforward to incorporate into existing code, especially if it is impossible/impractical to ensure each argument lies within a LUT's bounds.
- The ArgumentRecord class only exists in FunCif the preprocessor macro FUNC_DEBUG is defined. If FUNC_DEBUG is defined, then every argument passed to a DirectEvaluation and any out of bounds arguments passed to a FailureProofTable are also passed to ArgumentRecord to save in a histogram before computing $f(x)$. When the destructor of an ArgumentRecord is called, it prints its histogram to a provided std::ostream* (but does nothing if the pointer is null).
- The CompositeLookupTable class builds a LUT of f over several pairwise disjoint intervals. This enables users to build a LUT over custom partitions. When performing interval search, a CompositeLookupTable must perform binary search over a sorted tree of endpoints in $O(\log(N))$ time. If binary search fails, $f(x)$ is returned. This class is ideal for piecewise continuous functions and can interact nicely with nonuniform LUTs.
- The LookupTableComparator class can compare the time taken to apply a set of LUTs to a uniformly distributed random vector.

Every class in FunC with an operator() implements LookupTable. Only three classes that implement LookupTable do not also inherit from MetaTable. Of these, FailureProofTable and CompositeLookupTable are what we call *LUT containers*. They resort back to the defining mathematical formula for f if their interval search fails. LUT containers notably slow the LUT they encapsulate only because their interval search is slower. The only other class implementing LookupTable that does not also inherit MetaTable is the DirectEvaluation class. A DirectEvaluation does not provide an approximation of f for any inputs. Rather, it evaluates f using the defining mathematical formula. The DirectEvaluation class is provided for the convenience of debugging and profiling mathematical functions (depending on whether the preprocessor macro FUNC_DEBUG is defined).

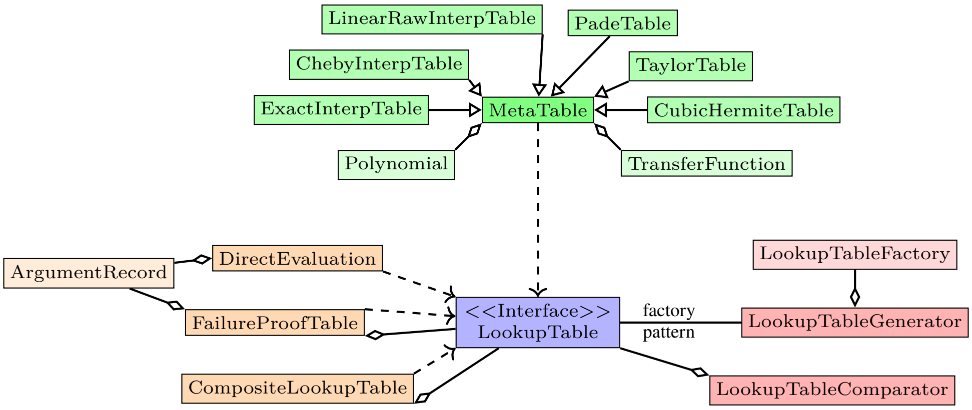


Figure 2 FunC**'s Class Diagram**

Arrow legend:

- Arrows with a triangular tip mean "inherits." For example, ChebyInterpTable inherits MetaTable.
- Arrows with a diamond tip mean "is a member variable of." For example, Polynomial is a member variable of MetaTable.
- Dotted arrows mean "implements the interface." For example, MetaTable implements the interface LookupTable.

4 FunC's LookupTable implementations

The practical differences between any two LUTs is the error in the approximation they use for each subinterval (the p_k from the equation on the main page) and the number of coefficients it uses to represent its p_k. Categorizing every LookupTable implementation based on how they compute their p_k results in the following four distinct families and two special cases. The following table shows each family of LookupTable supported in FunC along with some properties.

Name	Necessarily continuous?	Requires derivative?	NonUniform Support
ChebyInterpTable<n>	No	No	Yes
ExactInterpTable<n>	C^0	No	Yes
TaylorTable<n>	No	Yes	Yes
PadeTable<m,n>	No	Yes	No
CubicHermiteTable	C^1	Yes	Yes
LinearRawInterpTable	C^0	No	No

Notes:

- Every nonuniform LUT requires derivatives of f to construct their nonuniform partition.
- An ExactInterpTable<0> is a piecewise constant LUT, so it is likely not continuous.

Overview of each LUTs

- By default, a ChebyInterpTable<n> takes each p_k to be the polynomial of degree n interpolating f on $n+1$ Chebyshev nodes of the first kind in $[x_k, x_{k+1}]$. This class depends on Armadillo to solve Vandermonde systems. Armadillo only supports matrices with entries in float or double. FunC takes the working precision to be double because that is the most accurate type available to Armadillo. When using a ChebyInterpTable<n> with other types, FunC statically casts TIN or TOUT to double, solves the linear system with Armadillo, and then casts back to the original types.

- An ExactInterpTable<n> takes each p_k to be the polynomial of degree-n interpolating f on $n+1$ Chebyshev nodes of the *second kind* in $[x_k, x_{k+1}]$ (these are equally spaced nodes for $n \in \{0, 1, 2\}$). These LUTs are deemed "exact" because their source code contains a hard-coded symbolic expression for V^{-1}. The coefficients are accurate to the precision of TIN. ExactInterpTable<n> is the only LookupTable implementation that can be built over any types where TOUT forms an approximate vector space over an approximate field TIN.
- A TaylorTable<n> takes each p_k to be a degree n truncated Taylor series of f centered at the midpoint of $[x_k, x_{k+1}]$. This class uses Boost's automatic differentiation library to compute derivatives from the source code defining f.
- A PadeTable<m,n> takes each p_k to be the $[m/n]$ Pade approximant of f centered at the midpoint of $[x_k, x_{k+1}]$. We require $m \geq n > 0$. This class depends on both Armadillo and Boost. As such, the working precision is double.

There are two classes that are not templated on an unsigned integer.

- A CubicHermiteTable takes each p_k to be the cubic Hermite spline over the data

$$\{(x_k, f(x_k)), (x_k, f\prime(x_k)), (x_{k+1}, f(x_{k+1})), (x_{k+1}, f\prime(x_{k+1}))\}.$$

  These LUTs are $C^1[a, b]$ and depend on Boost to compute derivatives.
- A UniformLinearRawInterpTable builds the same p_k as a UniformExactInterpTable<1> with the same parameters, but a UniformLinearRawInterpTable saves memory by only storing $f(x_0), f(x_1), ..., f(x_n)$. Its operator() must then compute the two coefficients of p_k from $f(x_k)$ and $f(x_{k+1})$ before returning $p_k(x)$. For comparison, a UniformExactInterpTable<1> stores both coefficients of each p_k, so it uses approximately twice as much memory as a UniformLinearRawInterpTable with the same parameters. The overhead in a UniformLinearRawInterpTable's operator() is not large. There is no nonuniform variant of UniformLinearRawInterpTable because it does not allow for quick interval search.

We recommend users try to choose a particular family with properties that are conducive to best approximate f. For example, if the LUT must be continuous, then an ExactInterpTable or CubicHermiteTable are best.

5 Lookup Tables over nonuniform partitions of [a,b]

FunC provides two methods for constructing a LUT over a nonuniform partition of $[a, b]$. First, many of FunC's LookupTable implementations have built-in support for a nonuniform partition, but such a construction does not allow for custom partitions of $[a, b]$ from the user. This way, we ensure FunC can hash its nonuniform LUTs in 6 FLOPs and zero comparisons. If a custom partition is required then the other option is to use a CompositeLookuptable, although the resulting LUT will be much slower. Let $L_1, L_2, ..., L_M$ be LUTs of f over the pairwise disjoint intervals $[a_1, b_1], [a_2, b_2], ..., [a_M, b_M]$, respectively (not necessarily partitioning the domain of f). A CompositeTable of $L_1, L_2, ..., L_M$ is a piecewise function of the form

$$C(x) = \begin{cases} L_1(x), & \text{if } a_1 \leq x \leq b_1, \\ L_2(x), & \text{if } a_2 \leq x \leq b_2, \\ \vdots & \vdots \\ L_M(x), & \text{if } a_M \leq x \leq b_M, \\ f(x), & \text{otherwise.} \end{cases}$$

A CompositeLookupTable allows users to build LUTs over arbitrary nonuniform partitions. This is useful if f has discontinuities, f is difficult to accurately approximate on some subset of its domain, or the user's program requires that f is exact at certain other special points (roots, extrema, inflection points, etc). The downside is that this requires $O(\log n)$ comparisons each time the class's operator() is called. We can reduce the relative error in a LUT by building a CompositeLookupTable over f's roots. Doing so with $f(x) = \ln|\Gamma(x)|$ over $[0.1, 3]$,

L = UniformExactInterpTable<3>, and 30 subintervals reduces $E(L)$ with $a_{\text{tol}} = r_{\text{tol}} = 1$ from 1.19805×10^{-4} to 5.7253×10^{-6} (21 times less error). As for the nonuniform LUTs, they tend to perform best when $f\prime$ is largest at its endpoints a, b. For example, the nonuniform LUT will have almost the exact same partition as a uniform LUT for the function $f(x) = e^{x^2}$ because $f\prime(a) = -f\prime(b) \approx 10^{-10}$ is very small. To remedy this issue, we can build a CompositeLookupTable over f's inflection points (extremum of $f\prime$) as shown in the following figure. The constituent nonuniform LUTs use a nontrivial partition of $[-5, 5]$, and the overall composite LUT is 28 times more accurate than a single nonuniform LUT and has the same memory usage the other LUTs. Building a CompositeLookupTable of e^{-10x^2} and including inflection points in the partition of $[-5, 5]$:

```
FunctionContainer<double> func_container{FUNC_SET_F(MyFunction,double)}; auto step = 0.05;
UniformExactInterpTable<3,double>       uniformlut(func_container, {min,max,step});
NonUniformExactInterpTable<3,double> nonuniformlut(func_container, {min,max,step});
/* Build a Composite LUT over the inflection points of exp(-10*x*x) with the same error
 * as uniformlut. Using rtol = atol = 1.0 with E(L) */
LookupTableGenerator<double> gen(func_container, min, max);
auto err = gen.error_of_table(uniformlut); auto a = 0.035;
CompositeLookupTable<double> nonunifcom(func_container, {
  /* {tableKey, left, right, atol, rtol}, */
  {"NonUniformExactInterpTable<3>",min,-1.0/sqrt(2.0*10.0),                     a*err,a*err
    },
  {"NonUniformExactInterpTable<3>",     -1.0/sqrt(2.0*10.0),1.0/sqrt(2.0*10.0),     a*err,a*err
    },
  {"NonUniformExactInterpTable<3>",                          1.0/sqrt(2.0*10.0),max,a*err,a*err
    },
});

/* Verify nonuniformlut and uniformlut are approx. equal and compare with the composite LUT */
std::cout << "Error in uniform LUT:     " << err << std::endl;
std::cout << "Error in nonuniform LUT: " << err << " + "
  << gen.error_of_table(nonuniformlut) - err << std::endl;
std::cout << "Error in nonuniform composite LUT: " << gen.error_of_table(nonunifcom) << std::::
    endl;
std::cout << "Memory usage of uniform LUT: " << uniformlut.size() << std::endl;
std::cout << "Memory usage of nonuniform composite LUT: " << nonunifcom.size() << std::endl;
```

Output:

```
Error in uniform LUT:     1.05229e-06
Error in nonuniform LUT: 1.05229e-06 + -3.83676e-14
Error in nonuniform composite LUT: 3.52733e-08
Memory usage of uniform LUT: 6432
Memory usage of nonuniform composite LUT: 6496
```

The following figure shows how long it takes to apply each LUT from the previous figure to a random vector of length 1 000 000. We see that the CompositeLookupTable's operator() is about 12 times slower than individual LUTs. The CompositeLookupTable's improvement in accuracy comes at a cost. Average time to apply the operator() of each LUT from the previous figure ten times to a random vector of size 1 000 000

```
----------------------------------------------------------------------------
Table input and output types: d -> d
Number of trials performed: 10
Number of evaluations used: 1 000 000
----------------------------------------------------------------------------
| LookupTable:       NonUniformExactInterpTable<3> -5 5 0.05 200
| Memory usage (B): 6432
| Timings:           Min 0.00310616s Max 0.00320847s Mean 0.00313021s
----------------------------------------------------------------------------
| LookupTable:       UniformExactInterpTable<3> -5 5 0.05 200
| Memory usage (B): 6432
| Timings:           Min 0.00259926s Max 0.00672951s Mean 0.0030255s
----------------------------------------------------------------------------
| LookupTable:       CompositeLookupTable -5 5 0.0154212 200
| Memory usage (B): 6496
| Timings:           Min 0.0371598s Max 0.0372521s Mean 0.0371988s
----------------------------------------------------------------------------
```

6 Note on templates

One can theoretically use LUTs with any types such that TOUT forms an approximate vector space over TIN (addition and scalar multiplication accurate to machine epsilon exists). This generality is possible for the ExactInterpTable because the solution to its Vandermonde system is hard-coded. We cannot currently offer this level of generality for PadeTables and ChebyInterpTables because they depend on Armadillo to solve linear systems of equations (possibly also LUTs requiring derivatives because it is difficult to work out the theory in that case). Armadillo only supports matrices with entries in float or double (which is typical for high-performance linear algebra libraries). The generality allowed by FunC's templates is typical of header-only libraries, but FunC is not a header-only library. Template values TIN=TOUT=float and TIN =TOUT =double are explicitly instantiated for the LookupTableFactory. These are then compiled into a dynamic library. This way, the user can link their project with libfunc.so (avoiding the increase in compile time from templates) if they use LUTs with the two most common numeric types. Doing this has resulted in a 1.75 times speedup compared to headers only when compiling all the example code on our GitHub. This test does not include the time taken to compile libfunc.so, but the user need only compile libfunc.so once anyways. Linking with a dynamic library makes it much easier to quickly experiment with different LUTs and debug the user code. If user code instantiates other values of TIN and TOUT, then those LUTs are compiled from scratch with those template values at the same time as the user's code. Similarly, the rest of FunC's code is header-only.

7 Topic Documentation

7.1 Topics

Here is a list of all topics with brief descriptions:

7.2 Polynomial Based LookupTable implementations

Classes

- class ChebyInterpTable< N, TIN, TOUT, GT >

  LUT using degree 1 to 7 polynomial interpolation over Chebyshev nodes on each subinterval.
- class CubicHermiteTable< TIN, TOUT, GT >

  A LUT using cubic splines on each subinterval.
- class ExactInterpTable< N, TIN, TOUT, GT >

  Interpolation over Chebyshev nodes of the second kind. The inverse Vandermonde matrix is hard-coded. This class allows for full type generality, but numerical output is not quite as good as ChebyInterpTable for $n > 4$ because Armadillo does iterative refinement.
- class LinearRawInterpTable< TIN, TOUT, GT >

  Linear Interpolation LUT where coefficients are computed when calling operator(). Uses approx 50% less memory than an equivalent UniformExactInterpTable<1> but the hash involves an additional subtraction.
- class MetaTable< N, TIN, TOUT, GT >

  MetaTable handles any piecewise polynomial based interpolation. Highly templated.
- class PadeTable< M, N, TIN, TOUT, GT >

  LUT using [M/N] pade approximants.
- class TaylorTable< N, TIN, TOUT, GT >

  LUT using degree 1 to 7 truncated Taylor series.

7.2.1 Detailed Description

This group of classes implement MetaTable. So, the underlying approximation methods involve polynomials in some way

7.3 LUT Utilities

Files

- file cxx17utils.hpp

  Beta features for building multivariate LUTs via currying. Each feature requires C++17.

Classes

- class ArgumentRecord< TIN >

  A class used internally by FunC. Wraps a vector of unsigned int that act as a histogram for recording the usage of a function's domain.
- class CompositeLookupTable< TIN, TOUT >

  Approximate a single 1*D function with M LUTs over pairwise disjoint subintervals of its domain.* CompositeLookupTable *works well for functions with disconnected domains, or unused regions, or regions with difficult to approximate behaviour.*
- class DirectEvaluation< TIN, TOUT >

  Wrap a std::function *and optionally plot that function's domain usage with an* ArgumentRecord *(builds a histogram). To determine useful LUT bounds, users should replace their mathematical function with this class and compile with* -DFUNC_DEBUG.
- class FailureProofTable< LUT_TYPE >

  A wrapper for any implementation of LookupTable L. The operator()(x) *ensures x is within the bounds of L before returning $L(x)$. Returns $f(x)$ for out of bounds arguments. If* FUNC_DEBUG *is defined then out of bounds arguments are recorded in a histogram.*
- class LookupTableComparator< TIN, TOUT >

  Compare the average time taken to call the operator() *of any LookupTable implementation.*
- class LookupTableFactory< TIN, TOUT >

  Factory design patter for LookupTable implementations.
- class RngInterface< POINT_TYPE >

  Abstract interface for classes that can generate random numbers.
- class StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >

  An implementation of RngInterface, intended to be used with the generators/ distributions defined in std::random. Only LookupTableComparator uses this class, and it's for sampling random arguments.
- class Timer

  Starts a timer when created. Stops when stop() *is called and returns the duration in seconds with* duration().
- class LookupTableGenerator< TIN, TOUT, TERR >

  Generate a FunC LookupTable from a given name and one of the following: stepsize, tolerance, memory size limit, or filename with generate_by_step, generate_by_tol, generate_by_impl_size, and generate_by_file respectively. This class is also equipped to compute the error in a LUT built with any given stepsize and plot a LUT against its exact function.

7.3.1 Detailed Description

Each of these classes perform an operation on LUTs. This includes a factory design pattern, profiling, argument recording, computing the error of a LUT, and bounds checking.

8 Class Documentation

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

8.2 ArgumentRecord< TIN > Class Template Reference

A class used internally by FunC. Wraps a vector of unsigned int that act as a histogram for recording the usage of a function's domain.

#include <ArgumentRecord.hpp>

Public Member Functions

- ArgumentRecord (TIN min, TIN max, unsigned int histSize, std::ostream *streamer)
- ~ArgumentRecord ()
- ArgumentRecord (nlohmann::json jsonStats)
- void record_arg (TIN x)
- template<typename T>
  std::string to_string_with_precision (const T val, const int n=std::numeric_limits< T >::max_digits10) const
- std::string **ith_interval** (unsigned int i, const int n=3) const
- std::string to_string () const
- void print_json (nlohmann::json &jsonStats) const
- TIN **min_arg** () const
- TIN **max_arg** () const
- unsigned int **total_recorded** () const
- unsigned int index_of_peak () const
- unsigned int peak () const
- unsigned int num_out_of_bounds () const
- TIN max_recorded () const
- TIN **min_recorded** () const

8.2.1 Detailed Description

template<typename TIN>
class func::ArgumentRecord< TIN >

A class used internally by FunC. Wraps a vector of unsigned int that act as a histogram for recording the usage of a function's domain.

When constructed with a std::ostream, this will print to that ostream when the destructor is called. It is okay for provided arguments to be out of bounds. In this case, the max or min arg recorded will be updated, but the histogram's bounds will not grow dynamically.

Note

Code from this class is only included in FunC if the -DFUNC_DEBUG flag is specified at compile time and DirectEvaluation and FailureProofTable are the only classes that might use an ArgumentRecord.

An ArgumentRecord is designed to be a private member variable.

Recording arguments is threadsafe.

Todo Implement functions to_json & from_json

The histogram will never have that many buckets so we could likely get away with simply making every one of this class's member variables threadprivate.

Perhaps we should use boost histogram instead but that would add an additional dependency

8.2.2 Constructor & Destructor Documentation

ArgumentRecord() [1/2]

```
template<typename TIN>
ArgumentRecord (
        TIN min,
        TIN max,
        unsigned int histSize,
        std::ostream * streamer) [inline]
```

Arguments min and max determine histogram bounds, affecting the output in the printed histogram. histSize is the number of buckets. When provided an ostream != nullptr, this class will output *this to ostream when destructed

~ArgumentRecord()

```
template<typename TIN>
~ArgumentRecord () [inline]
```

Print to the optional argument mp_streamer if one was provided at the time of construction

ArgumentRecord() [2/2]

```
template<typename TIN>
ArgumentRecord (
        nlohmann::json jsonStats) [inline]
```

Rebuild our argument record

Note

This assumes the encapsulating LookupTable provided a valid json object

Todo Maybe optionally provide an ostream?

8.2.3 Member Function Documentation

index_of_peak()

```
template<typename TIN>
unsigned int index_of_peak () const [inline]
```

Return the index of the bucket with the largest count.

max_recorded()

```
template<typename TIN>
TIN max_recorded () const [inline]
```

Return the extreme args to help the user decide what bounds to use for their LUTs.

num_out_of_bounds()

```
template<typename TIN>
unsigned int num_out_of_bounds () const   [inline]
```

Return the number of elements outside the histogram's range.

peak()

```
template<typename TIN>
unsigned int peak () const   [inline]
```

Return the count from the bucket with the largest count.

print_json()

```
template<typename TIN>
void print_json (
                nlohmann::json & jsonStats) const   [inline]
```

Print each field in this class to the given ostream.

record_arg()

```
template<typename TIN>
void record_arg (
                TIN x)   [inline]
```

Place x in the histogram. Mimic pipeline parallelism for any statistics with only one instance

to_string()

```
template<typename TIN>
std::string to_string () const   [inline]
```

Make a string representation of the histogram.

to_string_with_precision()

```
template<typename TIN>
template<typename T>
std::string to_string_with_precision (
                const T val,
                const int n = std::numeric_limits<T>::max_digits10) const   [inline]
```

std::to_string(1e-7) == "0" which is unacceptable so we'll use this code from this SO post https://stackoverflow.com/questions/16605967/set-precision-of-stdto-string-when-converting-floating-point-values Default is the max possible precision by so users can choose how they'll round the answer on their own

The documentation for this class was generated from the following file:

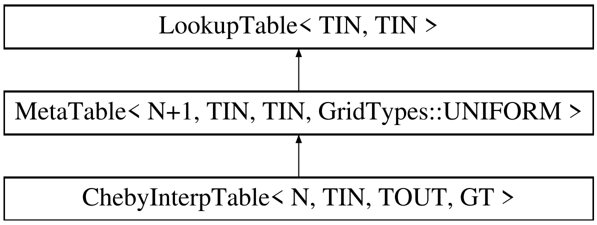
- ArgumentRecord.hpp

8.3 ChebyInterpTable< N, TIN, TOUT, GT > Class Template Reference

LUT using degree 1 to 7 polynomial interpolation over Chebyshev nodes on each subinterval.

#include <ChebyInterpTable.hpp>

Inheritance diagram for ChebyInterpTable< N, TIN, TOUT, GT >:



Public Member Functions

- **ChebyInterpTable** (const MetaTable< N+1, TIN, TOUT, GT > &L)
- **ChebyInterpTable** (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())

Public Member Functions inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TIN, GT > & operator= (MetaTable< N, TIN, TIN, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TIN **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TIN, GT > & operator+= (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator-=** (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TIN, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TIN operator() (TIN x) const override
- TIN diff (unsigned int s, TIN x) const

  Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TIN, GT > &L) noexcept

    Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- std::string **m_name**

    name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

    bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

    fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

    > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

    order of accuracy of implementation
- std::size_t **m_dataSize**

    size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

    = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

    length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

    < holds polynomials coefficients

8.3.1 Detailed Description

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
class func::ChebyInterpTable< N, TIN, TOUT, GT >

LUT using degree 1 to 7 polynomial interpolation over Chebyshev nodes on each subinterval.

Chebyshev nodes are a partition of the interval $[a, b]$ such that

$$t_s = (a+b)/2 + (b-a)\cos\left(\frac{2s-1}{2n}\pi\right)/2, \quad s = 1, \ldots, n.$$

Example usage

```
// return x^9
template <typename T>
```

```
T foo(T x){ return (x*x*x)*(x*x*x)*(x*x*x); }

int main(){
  auto min = -1.0; auto max = 1.0; auto step = 0.001;
  UniformChebyInterpTable<1,double> L1({FUNC_SET_F(foo,double)}, {min, max, step}); // degree 1
  UniformChebyInterpTable<2,double> L2({FUNC_SET_F(foo,double)}, {min, max, step}); // degree 2
  UniformChebyInterpTable<3,double> L3({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformChebyInterpTable<4,double> L4({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformChebyInterpTable<5,double> L5({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformChebyInterpTable<6,double> L6({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformChebyInterpTable<7,double> L7({FUNC_SET_F(foo,double)}, {min, max, step});
  // similar for nonuniform case:
  NonUniformChebyInterpTable<1,double> L7({FUNC_SET_F(foo,double)}, {min, max, step}); // deg 1 nonuniform
  NonUniformChebyInterpTable<4,double> L7({FUNC_SET_F(foo,double)}, {min, max, step});
  double val = look(0.87354);
  // The following preserves the root of $f$ at $x=0$.
  UniformChebyInterpTable<2,double> L2({FUNC_SET_F(foo,double)}, {min, max, step, {{0, 0, 0.0}}};
}
```

Note

The template implementation is only registered in the factory for $N = 1, 2, 3, 4, 5, 6, 7$ but users could construct this class with larger N if they wish. We make no promises on convergence/error in this case.

ChebyTable only works if we can cast both TOUT and TIN to double. This requirement exists because Armadillo Mat<T>'s is_supported_elem_type<T> will only let us do arithmetic with float or double (not even long double!). You might think "generic types is what arma::field is made for" but I can't get that class to work for our purposes.

This is currently the only LookupTableImplementation using the special_points field in LookupTableParameters – a vector of 3-tuples:

$$(x_1, s_1, f^{(s_1)}(x_1)), \ldots, (x_n, s_n, f^{(s_n)}(x_n)).$$

Then, Chebyshev nodes are replaced with the nearest nodes x_k in this list, and f (or its derivate) is exact at those nodes. Doing this can reduce the relative error in f and/or its derivatives.

Warning

Attempting to make f's derivatives exact at some given special points can result in a singular Vandermonde matrix. We don't have a way to handle this issue at the moment.

The documentation for this class was generated from the following file:

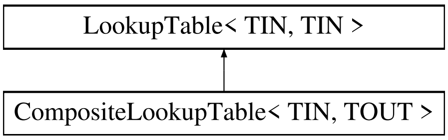
- ChebyInterpTable.hpp

8.4 CompositeLookupTable< TIN, TOUT > Class Template Reference

Approximate a single 1D function with M LUTs over pairwise disjoint subintervals of its domain. CompositeLookupTable works well for functions with disconnected domains, or unused regions, or regions with difficult to approximate behaviour.

#include <CompositeLookupTable.hpp>

Inheritance diagram for CompositeLookupTable< TIN, TOUT >:



Public Member Functions

- CompositeLookupTable (const FunctionContainer< TIN, TOUT > &func_container, const std::vector< std::tuple< std::string, TIN, TIN, TIN > > &name_l_r_steps)
- CompositeLookupTable (const FunctionContainer< TIN, TOUT > &func_container, const std::vector< std::tuple< std::string, TIN, TIN, TIN, TIN > > &name_l_r_atol_rtols)
- TOUT operator() (TIN x) const final
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- std::shared_ptr< LookupTable< TIN, TOUT > > get_table (TIN x)

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

8.4.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::CompositeLookupTable< TIN, TOUT >

Approximate a single 1D function with M LUTs over pairwise disjoint subintervals of its domain. CompositeLookupTable works well for functions with disconnected domains, or unused regions, or regions with difficult to approximate behaviour.

This is implemented with a std::map of M shared_ptr<LookupTable>. The operator()(\tin{} x) of a CompositeLookupTable calls map::upper_bound(x) to perform a binary search over the right endpoint of each LUT. So, the hash is $O(\log M)$. The operator() caches the most recently used LUT and skips the binary search when repeatedly evaluating from the same LUT's range.

This class works as a naive non-uniform lookup table.

Usage example:

```
std::vector<std::tuple<std::string,TYPE,TYPE,TYPE» v{
  // {tableKey, left, right, step},
  {"UniformExactInterpTable<3>",MIN_ARG,std::exp(7.7/13.0287),STEP},
  {"UniformExactInterpTable<3>",std::exp(7.7/13.0287),MAX_ARG,STEP}
};
FunctionContainer<TYPE> func_container{FUNC_SET_F(MyFunction,TYPE)};
CompositeLookupTable<TYPE> C(func_container, v);
std::cout « "C(0.01) = " « C(0.01) « std::endl;
```

Note

Constructs a LookupTableGenerator to construct each LUT.

Each member function is marked const.

Evaluate by using parentheses, just like a function.

The number of subintervals is the sum of each encapsulated LUT's subintervals, which is much greater than n

Todo Implement to/from_json. We can use the unique_ptr<LookupTable> version of from_json in LookupTableFactory to build each member LUT easily

8.4.2 Constructor & Destructor Documentation

CompositeLookupTable() [1/2]

```
template<typename TIN, typename TOUT = TIN>
CompositeLookupTable (
        const FunctionContainer< TIN, TOUT > & func_container,
        const std::vector< std::tuple< std::string, TIN, TIN, TIN > > & name_l_r_steps)   [inline]
```

Construct a CompositeLookupTable from a FunctionContainer and a vector of 4-tuples: n LUT names, n step sizes, n lower bounds, and n upper bounds.

CompositeLookupTable() [2/2]

```
template<typename TIN, typename TOUT = TIN>
CompositeLookupTable (
        const FunctionContainer< TIN, TOUT > & func_container,
        const std::vector< std::tuple< std::string, TIN, TIN, TIN, TIN > > & name_l_r_atol_rtols)   [inline]
```

Construct a CompositeLookupTable from a FunctionContainer and a vector of 5-tuples: n LUT names, n lower bounds, n upper bounds, n absolute tolerances and n relative tolerances

8.4.3 Member Function Documentation

bounds_of_subinterval()

```
template<typename TIN, typename TOUT = TIN>
std::pair< TIN, TIN > bounds_of_subinterval (
                    unsigned int intervalNumber) const   [inline], [final], [virtual]
```

Iterate through each LUT, counting each of its subintervals until we arrive upon the intervalNumber-th subinterval

Implements LookupTable< TIN, TIN >.

get_table()

```
template<typename TIN, typename TOUT = TIN>
std::shared_ptr< LookupTable< TIN, TOUT > > get_table (
                    TIN x)   [inline]
```

Get the closest LUT such that this argument is less than its upper bound.

Note

x might be outside the return LUT's bounds if the given function is not approximated with a single LUT over its entire domain

max_arg()

```
template<typename TIN, typename TOUT = TIN>
TIN max_arg () const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

min_arg()

```
template<typename TIN, typename TOUT = TIN>
TIN min_arg () const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

name()

```
template<typename TIN, typename TOUT = TIN>
std::string name () const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

num_subintervals()

template<typename TIN, typename TOUT = TIN>
unsigned int num_subintervals () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

operator()()

template<typename TIN, typename TOUT = TIN>
TOUT operator() (
TIN x) const [inline], [final], [virtual]

Perform binary search for a LUT with the appropriate bounds and returns LUT(x). Resorts to the original function if arguments are out of bounds. Caches the most recently used LUT.

Implements LookupTable< TIN, TIN >.

order()

template<typename TIN, typename TOUT = TIN>
unsigned int order () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

print_json()

template<typename TIN, typename TOUT = TIN>
void print_json (
std::ostream & out) const [inline], [final], [virtual]

Note

Every class implementing LookupTable should call their implementation of nlohmann's to_json from print↩_json

Implements LookupTable< TIN, TIN >.

size()

template<typename TIN, typename TOUT = TIN>
std::size_t size () const [inline], [final], [virtual]

Sum the sizes of each LookupTable

Implements LookupTable< TIN, TIN >.

step_size()

template<typename TIN, typename TOUT = TIN>
TIN step_size () const [inline], [final], [virtual]

Return the min step size of each LookupTable

Implements LookupTable< TIN, TIN >.

The documentation for this class was generated from the following file:

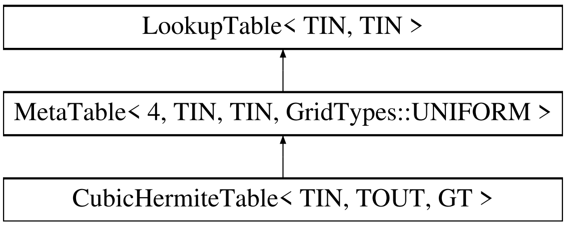
- CompositeLookupTable.hpp

8.5 CubicHermiteTable< TIN, TOUT, GT > Class Template Reference

A LUT using cubic splines on each subinterval.

#include <CubicHermiteTable.hpp>

Inheritance diagram for CubicHermiteTable< TIN, TOUT, GT >:



Public Member Functions

- **CubicHermiteTable** (const MetaTable< 4, TIN, TOUT, GT > &L)
- **CubicHermiteTable** (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())

Public Member Functions inherited from MetaTable< 4, TIN, TIN, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TIN, GT > & operator= (MetaTable< N, TIN, TIN, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TIN **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TIN, GT > & operator+= (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator-=** (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TIN, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TIN operator() (TIN x) const override
- TIN diff (unsigned int s, TIN x) const

    Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< 4, TIN, TIN, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TIN, GT > &L) noexcept

  Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< 4, TIN, TIN, GridTypes::UNIFORM >

- std::string **m_name**

  name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

  bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

  fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

  > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

  order of accuracy of implementation
- std::size_t **m_dataSize**

  size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

  = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

  length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

  < holds polynomials coefficients

8.5.1 Detailed Description

template<typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
class func::CubicHermiteTable< TIN, TOUT, GT >

A LUT using cubic splines on each subinterval.

```
// CubicHermiteTable requires the user's mathematical function is templated
template <typename T>
T foo(T x){ return x; }

int main(){
  double min = 0.0, max = 10.0, step = 0.0001;
  UniformCubicHermiteTable<double>    L({FUNC_SET_F(foo,double)}, {min, max, step}); // uniform partition
  NonUniformCubicHermiteTable<double> L({FUNC_SET_F(foo,double)}, {min, max, step}); // nonuniform partition
  auto val = L(0.87354);
}
```

Note

Each member function is declared const

The documentation for this class was generated from the following file:

- CubicHermiteTable.hpp

8.6 curriedLUT< N, TIN, TOUT, classname > Struct Template Reference

#include <cxx17utils.hpp>

Public Types

- using **type** = classname<TIN, typename curriedLUT<N-1,TIN,TOUT,classname>::type>

8.6.1 Detailed Description

template<unsigned int N, typename TIN, typename TOUT, template< typename... > class classname>
struct func::curriedLUT< N, TIN, TOUT, classname >

BETA FEATURE: Convenience functions for defining LUTs of LUTs

8.6.2 Member Typedef Documentation

type

template<unsigned int N, typename TIN, typename TOUT, template< typename... > class classname>
using type = classname<TIN, typename curriedLUT<N-1,TIN,TOUT,classname>::type>

The documentation for this struct was generated from the following file:

- cxx17utils.hpp

8.7 curriedLUT< 0, TIN, TOUT, classname > Struct Template Reference

#include <cxx17utils.hpp>

Public Types

- using **type** = classname<TIN,TOUT>

8.7.1 Detailed Description

template<typename TIN, typename TOUT, template< typename... > class classname>
struct func::curriedLUT< 0, TIN, TOUT, classname >

BETA FEATURE: Base case: zero currying is equivalent to the LUT itself

8.7.2 Member Typedef Documentation

type

```
template<typename TIN, typename TOUT, template< typename... > class classname>
using type = classname<TIN,TOUT>
```

The documentation for this struct was generated from the following file:

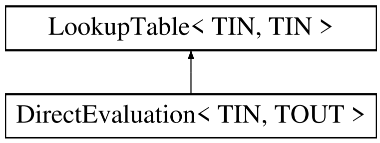
- cxx17utils.hpp

8.8 DirectEvaluation< TIN, TOUT > Class Template Reference

Wrap a std::function and optionally plot that function's domain usage with an ArgumentRecord (builds a histogram). To determine useful LUT bounds, users should replace their mathematical function with this class and compile with -DFUNC_DEBUG.

#include <DirectEvaluation.hpp>

Inheritance diagram for DirectEvaluation< TIN, TOUT >:



Public Member Functions

- DirectEvaluation (const FunctionContainer< TIN, TOUT > &func_container, TIN min=0.0, TIN max=1.0, unsigned int histSize=10, TOUT aerr=0.0, TIN rerr=0.0, std::ostream *streamer=nullptr)
- TOUT operator() (TIN x) const final

    Evaluate the underlying std::function *and optionally record the argument x.*
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

8.8.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::DirectEvaluation< TIN, TOUT >

Wrap a std::function and optionally plot that function's domain usage with an ArgumentRecord (builds a histogram). To determine useful LUT bounds, users should replace their mathematical function with this class and compile with -DFUNC_DEBUG.

Usage example:

```
DirectEvaluation<double> de({MyFunction},0,10);
double fx = de(0.87354);
// sim code calling de goes here
de.print_details(std::cout); // prints max/min recorded args if FUNC_DEBUG is defined
```

\notes When compiled with -DFUNC_DEBUG, the ArgumentRecord uses min and max constructor arguments to construct a histogram's bounds. This histogram record arguments passed to the DirectEvaluation and there is no issue if sampled arguments are out of bounds.

Note

View the histogram with print_details(), or construct DirectEvaluation with a pointer to ostream and get output upon destruction.

8.8.2 Constructor & Destructor Documentation

DirectEvaluation()

```
template<typename TIN, typename TOUT = TIN>
DirectEvaluation (
        const FunctionContainer< TIN, TOUT > & func_container,
        TIN min = 0.0,
        TIN max = 1.0,
        unsigned int histSize = 10,
        TOUT aerr = 0.0,
        TIN rerr = 0.0,
        std::ostream * streamer = nullptr)   [inline]
```

Simply store the first member of func_container and pass each argument to it whenever operator() is called. Optionally set up argument recording if FUNC_DEBUG is defined used at compile time

8.8.3 Member Function Documentation

bounds_of_subinterval()

```
template<typename TIN, typename TOUT = TIN>
std::pair< TIN, TIN > bounds_of_subinterval (
        unsigned int intervalNumber) const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

max_arg()

template<typename TIN, typename TOUT = TIN>
TIN max_arg () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

min_arg()

template<typename TIN, typename TOUT = TIN>
TIN min_arg () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

name()

template<typename TIN, typename TOUT = TIN>
std::string name () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

num_subintervals()

template<typename TIN, typename TOUT = TIN>
unsigned int num_subintervals () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

operator()()

template<typename TIN, typename TOUT = TIN>
TOUT operator() (
TIN x) const [inline], [final], [virtual]

Evaluate the underlying std::function and optionally record the argument x.

Implements LookupTable< TIN, TIN >.

order()

template<typename TIN, typename TOUT = TIN>
unsigned int order () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

print_json()

```
template<typename TIN, typename TOUT = TIN>
void print_json (
                std::ostream & out) const   [inline], [final], [virtual]
```

Note

Every class implementing LookupTable should call their implementation of nlohmann's to_json from print↩ _json

Implements LookupTable< TIN, TIN >.

size()

```
template<typename TIN, typename TOUT = TIN>
std::size_t size () const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

step_size()

```
template<typename TIN, typename TOUT = TIN>
TIN step_size () const   [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

The documentation for this class was generated from the following file:

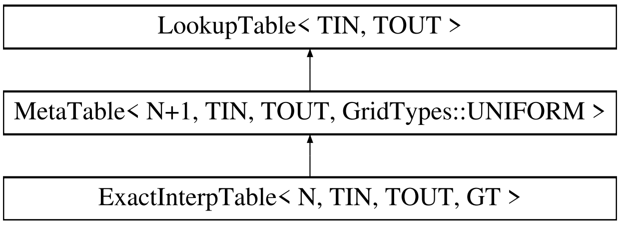
- DirectEvaluation.hpp

8.9 ExactInterpTable< N, TIN, TOUT, GT > Class Template Reference

Interpolation over Chebyshev nodes of the second kind. The inverse Vandermonde matrix is hard-coded. This class allows for full type generality, but numerical output is not quite as good as ChebyInterpTable for $n > 4$ because Armadillo does iterative refinement.

#include <ExactInterpTable.hpp>

Inheritance diagram for ExactInterpTable< N, TIN, TOUT, GT >:



Public Member Functions

- **ExactInterpTable** (const MetaTable< N+1, TIN, TOUT, GT > &L)
- **ExactInterpTable** (const FunctionContainer< TIN, TOUT > &fun_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())

Public Member Functions inherited from MetaTable< N+1, TIN, TOUT, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TOUT, GT > & operator= (MetaTable< N, TIN, TOUT, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TOUT **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TOUT, GT > & operator+= (const MetaTable< N, TIN, TOUT, GT > &other)
- MetaTable< N, TIN, TOUT, GT > & **operator-=** (const MetaTable< N, TIN, TOUT, GT > &other)
- MetaTable< N, TIN, TOUT, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TOUT, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TOUT operator() (TIN x) const override
- TOUT diff (unsigned int s, TIN x) const

    Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TOUT >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TOUT >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< N+1, TIN, TOUT, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TOUT, GT > &L) noexcept

    Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< N+1, TIN, TOUT, GridTypes::UNIFORM >

- std::string **m_name**

    name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

    bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

    fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

    > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

    order of accuracy of implementation
- std::size_t **m_dataSize**

    size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

    = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

    length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

    < holds polynomials coefficients

8.9.1 Detailed Description

template<unsigned int N, typename TIN, typename TOUT, GridTypes GT = GridTypes::UNIFORM>
class func::ExactInterpTable< N, TIN, TOUT, GT >

Interpolation over Chebyshev nodes of the second kind. The inverse Vandermonde matrix is hard-coded. This class allows for full type generality, but numerical output is not quite as good as ChebyInterpTable for $n > 4$ because Armadillo does iterative refinement.

```
// return x^9
template <typename T>
T foo(T x){ return (x*x*x)*(x*x*x)*(x*x*x); }

int main(){
  double min = 0.0, max = 10.0, step = 0.0001;
  UniformExactInterpTable<1,double>    L({FUNC_SET_F(foo,double)}, {min, max, step}); // uniform partition
  UniformExactInterpTable<4,double>    L({FUNC_SET_F(foo,double)}, {min, max, step}); // degree 4
  NonUniformExactInterpTable<1,double> L({FUNC_SET_F(foo,double)}, {min, max, step}); // nonuniform partition
  auto val = L(0.87354);
}
```

Note

> Each polynomial coefficient is computed when the constructor is called and looked up every time its operator() is called.
>
> Each member function is declared const

The documentation for this class was generated from the following file:

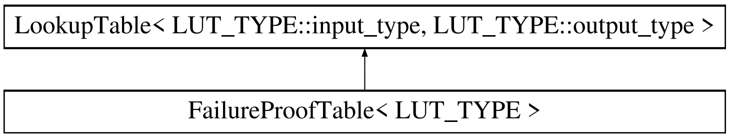
- ExactInterpTable.hpp

8.10 FailureProofTable< LUT_TYPE > Class Template Reference

A wrapper for any implementation of LookupTable L. The $\mathrm{operator}()(\mathrm{x})$ ensures x is within the bounds of L before returning $L(x)$. Returns $f(x)$ for out of bounds arguments. If FUNC_DEBUG is defined then out of bounds arguments are recorded in a histogram.

#include <FailureProofTable.hpp>

Inheritance diagram for FailureProofTable< LUT_TYPE >:



Public Member Functions

- FailureProofTable (const FunctionContainer< TIN, TOUT > &fc, const LookupTableParameters< TIN > &par, TIN histMin=1, TIN histMax=0, unsigned int histSize=10, std::ostream *streamer=nullptr)
- TOUT operator() (TIN x) const final
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final

Public Member Functions inherited from LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >

- using **input_type**
- using **output_type**

8.10.1 Detailed Description

template<class LUT_TYPE>
class func::FailureProofTable< LUT_TYPE >

A wrapper for any implementation of LookupTable L. The $\mathrm{operator}()(\mathrm{x})$ ensures x is within the bounds of L before returning $L(x)$. Returns $f(x)$ for out of bounds arguments. If FUNC_DEBUG is defined then out of bounds arguments are recorded in a histogram.

Template Parameters

LUT_TYPE	is a specific implementation of LookupTable (eg. ChebyInterpTable<3,double>)

Usage example:

```
// Build a UniformChebyInterpTable<3,double> with the arguments {0,10,0.0001}
FailureProofTable<UniformChebyInterpTable<3,double>> failsafe(
  {MyFunction},{0,10,0.0001}
);
double val1 = failsafe(0.87354);
double val2 = failsafe(100);
```

Note

Each member function is marked const

User can optionally call the constructor with arguments for ArgumentRecord to improve binning (better tracking the max & min arguments)

Todo This class will support to_json but not from_json because it needs a FunctionContainer. Add another constructor to build this class from a FunctionContainer and a filename

8.10.2 Constructor & Destructor Documentation

FailureProofTable()

```
template<class LUT_TYPE>
FailureProofTable (
        const FunctionContainer< TIN, TOUT > & fc,
        const LookupTableParameters< TIN > & par,
        TIN histMin = 1,
        TIN histMax = 0,
        unsigned int histSize = 10,
        std::ostream * streamer = nullptr)   [inline]
```

Build our own LUT_TYPE. This constructor will works if the template argument LUT_TYPE is specific enough

8.10.3 Member Function Documentation

bounds_of_subinterval()

```
template<class LUT_TYPE>
std::pair< TIN, TIN > bounds_of_subinterval (
        unsigned int intervalNumber) const   [inline], [final], [virtual]
```

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

max_arg()

```
template<class LUT_TYPE>
TIN max_arg () const   [inline], [final], [virtual]
```

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

min_arg()

template<class LUT_TYPE>
TIN min_arg () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

name()

template<class LUT_TYPE>
std::string name () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

num_subintervals()

template<class LUT_TYPE>
unsigned int num_subintervals () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

operator()()

template<class LUT_TYPE>
TOUT operator() (
TIN x) const [inline], [final], [virtual]

if x isn't in the LUT's bounds, then return f(x)

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

order()

template<class LUT_TYPE>
unsigned int order () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

print_json()

template<class LUT_TYPE>
void print_json (
std::ostream & out) const [inline], [final], [virtual]

Note

Every class implementing LookupTable should call their implementation of nlohmann's to_json from print_json

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

size()

template<class LUT_TYPE>
std::size_t size () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

step_size()

template<class LUT_TYPE>
TIN step_size () const [inline], [final], [virtual]

Implements LookupTable< LUT_TYPE::input_type, LUT_TYPE::output_type >.

The documentation for this class was generated from the following file:

- FailureProofTable.hpp

8.11 FuncMutex Class Reference

Public Member Functions

- void **lock** ()
- void **unlock** ()

The documentation for this class was generated from the following file:

- ArgumentRecord.hpp

8.12 FuncScopedLock Class Reference

Public Member Functions

- **FuncScopedLock** (FuncMutex &mutex)

The documentation for this class was generated from the following file:

- ArgumentRecord.hpp

8.13 FunctionContainer< TIN, TOUT > Class Template Reference

Wrapper for std::function<TOUT(TIN)> and some optional std::functions of Boost's automatic differentiation type.

#include <FunctionContainer.hpp>

Public Member Functions

- template<unsigned int N>
  fun_type< N >::type get_nth_func () const

  return the Boost autodiff function that automatically differentiates the user's function N times.
- **FunctionContainer** (std::function< TOUT(TIN)> fun)
- **FunctionContainer** (std::function< TOUT(TIN)> fun, std::function< adVar< TOUT, 1 >(adVar< TIN, 1 >)> fun1, std::function< adVar< TOUT, 2 >(adVar< TIN, 2 >)> fun2, std::function< adVar< TOUT, 3 >(adVar< TIN, 3 >)> fun3, std::function< adVar< TOUT, 4 >(adVar< TIN, 4 >)> fun4, std::function< adVar< TOUT, 5 >(adVar< TIN, 5 >)> fun5, std::function< adVar< TOUT, 6 >(adVar< TIN, 6 >)> fun6, std::function< adVar< TOUT, 7 >(adVar< TIN, 7 >)> fun7)

  Users should not call this constructor without wrapping their argument with the macro FUNC_SET_F(...). *See the example usage for FunctionContainer.*

Public Attributes

- std::function< TOUT(TIN)> **standard_fun**
- std::function< adVar< TOUT, 1 >(adVar< TIN, 1 >)> **autodiff1_fun**
- std::function< adVar< TOUT, 2 >(adVar< TIN, 2 >)> **autodiff2_fun**
- std::function< adVar< TOUT, 3 >(adVar< TIN, 3 >)> **autodiff3_fun**
- std::function< adVar< TOUT, 4 >(adVar< TIN, 4 >)> **autodiff4_fun**
- std::function< adVar< TOUT, 5 >(adVar< TIN, 5 >)> **autodiff5_fun**
- std::function< adVar< TOUT, 6 >(adVar< TIN, 6 >)> **autodiff6_fun**
- std::function< adVar< TOUT, 7 >(adVar< TIN, 7 >)> **autodiff7_fun**

8.13.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::FunctionContainer< TIN, TOUT >

Wrapper for std::function<TOUT(TIN)> and some optional std::functions of Boost's automatic differentiation type.

Used to pass mathematical functions to FunC's LUTs.

Note

Only the LUTs that need derivatives (Taylor, Hermite, Pade, and every NonUniformLUT) need Boost's ad↩
Var[1-7] functions.

The automatic differentiation requires the mathematical function is templated on some abstract type

Autodiff was introduced in Boost 1.71. This class reduces to a simple std::function wrapper if Boost is not available or is too old.

Most of the machinery is necessary to use get_nth_func<N> which returns the ith order autodiff functions based on an index

Copy and paste the following example code into a new file and rename the example to your own function with :s/foo/new_name/g in Vim

```
#include FunctionContainer.hpp
template <typename T>
T foo(T x){ return x; }

#define TYPE double

int main(){
  FunctionContainer<TYPE> foo_container {FUNC_SET_F(foo,TYPE)};

  // Use this version if it's inconvenient to template your function:
  FunctionContainer<TYPE> foo_container2 {foo<TYPE>};
  // The only downside is that you can't generate LUTs that need
  // automatic differentiation. There will be a runtime exception if you try
  return 0;
}
```

8.13.2 Member Function Documentation

get_nth_func()

template<typename TIN, typename TOUT = TIN>
template<unsigned int N>
fun_type< N >::type get_nth_func () const [inline]

return the Boost autodiff function that automatically differentiates the user's function N times.

Note

Get the nth differentiable template instantiation of the given function by calling the function container as func_container->template get_nth_func<N>()

Only supports 7 or fewer derivatives.

Implemented with 9 different overloads of fun_type<N>

The documentation for this class was generated from the following file:

- FunctionContainer.hpp

8.14 ImplTimer< TIN, TOUT > Struct Template Reference

Helper struct: takes a LookupTable and attaches a set of timings to it.

#include <LookupTableComparator.hpp>

Public Member Functions

- **ImplTimer** (LookupTable< TIN, TOUT > ∗inImpl)
- void **append_runtime** (double time)
- void **compute_timing_stats** ()
- void **print_timing_stats** (std::ostream &out)

Public Attributes

- LookupTable< TIN, TOUT > ∗ **impl**
- std::vector< double > **evaluationTimes**
- double **maxTime**
- double **minTime**
- double **meanTime**

8.14.1 Detailed Description

template<typename TIN, typename TOUT>
struct func::ImplTimer< TIN, TOUT >

Helper struct: takes a LookupTable and attaches a set of timings to it.

The documentation for this struct was generated from the following file:

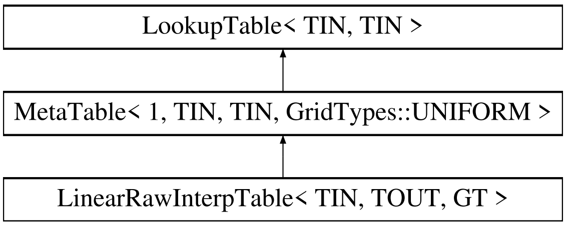
- LookupTableComparator.hpp

8.15 LinearRawInterpTable< TIN, TOUT, GT > Class Template Reference

Linear Interpolation LUT where coefficients are computed when calling operator(). Uses approx 50% less memory than an equivalent UniformExactInterpTable<1> but the hash involves an additional subtraction.

#include <LinearRawInterpTable.hpp>

Inheritance diagram for LinearRawInterpTable< TIN, TOUT, GT >:



Public Member Functions

- **LinearRawInterpTable** (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())
- TOUT operator() (TIN x) const override

Public Member Functions inherited from MetaTable< 1, TIN, TIN, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TIN, GT > & operator= (MetaTable< N, TIN, TIN, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TIN **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TIN, GT > & operator+= (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator-=** (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TIN, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TIN operator() (TIN x) const override
- TIN diff (unsigned int s, TIN x) const

  Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< 1, TIN, TIN, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TIN, GT > &L) noexcept
  Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< 1, TIN, TIN, GridTypes::UNIFORM >

- std::string **m_name**
  name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**
  bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**
  fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**
  > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**
  order of accuracy of implementation
- std::size_t **m_dataSize**
  size of relevant data for impl evaluation
- unsigned int **m_numIntervals**
  = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**
  length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction
  < holds polynomials coefficients

8.15.1 Detailed Description

template<typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
class func::LinearRawInterpTable< TIN, TOUT, GT >

Linear Interpolation LUT where coefficients are computed when calling operator(). Uses approx 50% less memory than an equivalent UniformExactInterpTable<1> but the hash involves an additional subtraction.

```
// LinearRawInterpTable does not benefit from templated functions because
// there is no nonuniform variant
double foo(double x){ return x; }

int main(){
  double min = 0.0, max = 10.0, step = 0.0001;
  UniformLinearRawInterpTable<double> L({foo}, {min, max, step});
  auto val = L(0.87354);
}
```

Note

Each member function is marked const

Evaluate by using parentheses, just like a function

Does not have a nonuniform variant and it's not obvious how to make this LookupTable implementation nonuniform unless we make the operator() far slower (basically defeating the purpose of this LUT type e.g. lookup breakpoints from m_grid?)

8.15.2 Member Function Documentation

operator()()

template<typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TOUT operator() (
TIN x) const [inline], [override], [virtual]

This operator() is different from MetaTable's provided Horner's method because we must compute the two coefficients of the linear interpolating polynomial

Todo is there a way to make this work with nonuniform grids in a way that works with our model?

Implements LookupTable< TIN, TIN >.

The documentation for this class was generated from the following file:

- LinearRawInterpTable.hpp

8.16 LookupTable< TIN, TOUT > Class Template Reference

Abstract interface for representing an approximation to a user provided mathematical function.

#include <LookupTable.hpp>

Public Types

- using **input_type** = TIN
- using **output_type** = TOUT

Public Member Functions

- LookupTable ()=default
- virtual TOUT operator() (TIN x) const =0
- virtual std::string **name** () const =0
- virtual TIN **min_arg** () const =0
- virtual TIN **max_arg** () const =0
- virtual unsigned int **order** () const =0
- virtual std::size_t size () const =0
- virtual unsigned int **num_subintervals** () const =0
- virtual TIN step_size () const =0
- virtual std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const =0
- virtual void print_json (std::ostream &out) const =0

8.16.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::LookupTable< TIN, TOUT >

Abstract interface for representing an approximation to a user provided mathematical function.

LookupTable possesses no member variables, or runnable code. Implementations of this class handle actual data (reading, writing, hashing, etc). The LookupTable interface is necessary for our LookupTableFactory, LookupTableGenerator, and LookupTableComparator.

Warning

{ We make no promises about checking array bounds (as this notably reduces performance) }

8.16.2 Constructor & Destructor Documentation

LookupTable()

template<typename TIN, typename TOUT = TIN>
LookupTable () [default]

Every implementation of LookupTable will have a constructor that looks like this: LookupTable(const Function↩ Container<TIN,TOUT>& func_container, const LookupTableParameters<TIN>& par) { }

8.16.3 Member Function Documentation

bounds_of_subinterval()

```
template<typename TIN, typename TOUT = TIN>
virtual std::pair< TIN, TIN > bounds_of_subinterval (
            unsigned int intervalNumber) const [pure virtual]
```

Implemented in CompositeLookupTable< TIN, TOUT >.

operator()()

```
template<typename TIN, typename TOUT = TIN>
virtual TOUT operator() (
            TIN x) const [pure virtual]
```

Implemented in CompositeLookupTable< TIN, TOUT >, DirectEvaluation< TIN, TOUT >, FailureProofTable< LUT_TYPE >, LinearRawInterpTable< TIN, TOUT, GT >, LinearRawInterpTable< TIN, TOUT, GridTypes::UNIFORM >, MetaTable< N, TIN, TOUT, GT >, MetaTable< 1, TIN, TIN, GridTypes::UNIFORM >, MetaTable< 4, TIN, TIN, GridTypes::UNI MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >, MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >, MetaTable< N+1, TIN, TOUT, GridTypes::UNIFORM >, PadeTable< M, N, TIN, TOUT, GT >, and PadeTable< M, N, TIN, TOU

print_json()

```
template<typename TIN, typename TOUT = TIN>
virtual void print_json (
            std::ostream & out) const [pure virtual]
```

Note

> Every class implementing LookupTable should call their implementation of nlohmann's to_json from print←_json

Implemented in CompositeLookupTable< TIN, TOUT >, DirectEvaluation< TIN, TOUT >, FailureProofTable< LUT_TYPE >, MetaTable< N, TIN, TOUT, GT >, MetaTable< 1, TIN, TIN, GridTypes::UNIFORM >, MetaTable< 4, TIN, TIN, GridTypes::UNI MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >, MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >, and MetaTable< N+1, TIN, TOUT, GridTypes::UNIFORM >.

size()

```
template<typename TIN, typename TOUT = TIN>
virtual std::size_t size () const [pure virtual]
```

Implemented in CompositeLookupTable< TIN, TOUT >.

step_size()

```
template<typename TIN, typename TOUT = TIN>
virtual TIN step_size () const [pure virtual]
```

Implemented in CompositeLookupTable< TIN, TOUT >.

The documentation for this class was generated from the following file:

- LookupTable.hpp

8.17 LookupTableComparator< TIN, TOUT > Class Template Reference

Compare the average time taken to call the operator() of any LookupTable implementation.

#include <LookupTableComparator.hpp>

Public Member Functions

- LookupTableComparator (ImplContainer< TIN, TOUT > &inImpl, TIN minArg, TIN maxArg, unsigned int nEvals=100000, unsigned int seed=2017, std::unique_ptr< RngInterface< TIN > > inRng=nullptr)
- void run_timings (int nRuns=1)
- void compute_statistics ()
- void sort_timings (Sorter type=Sorter::MEAN)
- void print_summary (std::ostream &)
- void print_json (std::ostream &)
- void **print_csv_header** (std::ostream &)
- void print_csv (std::ostream &, Sorter type=Sorter::NONE)
- std::vector< double > **fastest_times** ()
- std::vector< double > **slowest_times** ()

8.17.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::LookupTableComparator< TIN, TOUT >

Compare the average time taken to call the operator() of any LookupTable implementation.

For example usage, see any file in the examples directory.

Note

This class takes ownership of the vector of LUT implementations it is constructed with

Points are randomly sampled, and by default uses a std::uniform_real_distribution<TIN> with the std↩ ::mt19937 variant of the std::mersenne_twister_engine. This can be changed by passing in a different StdRng

Todo LookupTableComparator's constructor should accept any callable type

8.17.2 Constructor & Destructor Documentation

LookupTableComparator()

```
template<typename TIN, typename TOUT>
LookupTableComparator (
        ImplContainer< TIN, TOUT > & inImpl,
        TIN minArg,
        TIN maxArg,
        unsigned int nEvals = 100000,
        unsigned int seed = 2017,
        std::unique_ptr< RngInterface< TIN > > inRng = nullptr) [inline]
```

Prepare to run several timings for each LUT in the vector inImpl

8.17.3 Member Function Documentation

compute_statistics()

```
template<typename TIN, typename TOUT = TIN>
void compute_statistics ( )   [inline]
```

Compute fastest and slowest times

print_csv()

```
template<typename TIN, typename TOUT>
void print_csv (
                std::ostream & out,
                Sorter type = Sorter::NONE)   [inline]
```

Output a space separated listing of timing results. Does not print a final newline if type != Sorter::NONE which is helpful for plotting timing results with external programs (e.g. Python)

print_json()

```
template<typename TIN, typename TOUT>
void print_json (
                std::ostream & out)   [inline]
```

Print out the raw timings for each LookupTable

print_summary()

```
template<typename TIN, typename TOUT>
void print_summary (
                std::ostream & out)   [inline]
```

Print out the computed statistics for each LookupTable (no raw timings are displayed)

run_timings()

```
template<typename TIN, typename TOUT = TIN>
void run_timings (
                int nRuns = 1)   [inline]
```

Run timings with different set of random arguments

sort_timings()

```
template<typename TIN, typename TOUT>
void sort_timings (
                Sorter type = Sorter::MEAN)   [inline]
```

Sort the vector of implementations (m_implTimers) based on their max Sorter::WORST, mean Sorter::MEAN, or min Sorter::BEST times

The documentation for this class was generated from the following file:

- LookupTableComparator.hpp

8.18 LookupTableGenerator< TIN, TOUT, TERR >::LookupTableErrorFunctor Struct Reference

Public Member Functions

- **LookupTableErrorFunctor** (const LookupTable< TIN, TOUT > ∗impl, const std::function< TOUT(TIN)> fun, TERR relTol)
- TERR **operator()** (const TERR &x)

The documentation for this struct was generated from the following file:

- LookupTableGenerator.hpp

8.19 LookupTableFactory< TIN, TOUT > Class Template Reference

Factory design patter for LookupTable implementations.

#include <LookupTableFactory.hpp>

Public Types

- using registry_t

  This map type holds the registry.

Public Member Functions

- LookupTableFactory ()
- std::unique_ptr< LookupTable< TIN, TOUT > > create (std::string string_name, const FunctionContainer< TIN, TOUT > &fc, const LookupTableParameters< TIN > &args, const nlohmann::json &json↩ Stats=nlohmann::json())
- std::vector< std::string > get_registered_keys ()

8.19.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
class func::LookupTableFactory< TIN, TOUT >

Factory design patter for LookupTable implementations.

Note

> LookupTableFactory<TIN,TOUT>::create(str_name, fc, par) generates LookupTable types derived from LookupTable<TIN,TOUT>
>
> Add new LookupTable implementations to the registry by adding their names to the ::initialize() member function

8.19.2 Member Typedef Documentation

registry_t

template<typename TIN, typename TOUT = TIN>
using registry_t

Initial value:

```
std::map<std::string, std::function<LookupTable<TIN, TOUT> *(const FunctionContainer<TIN, TOUT>&, const
  LookupTableParameters<TIN>&, const nlohmann::json& jsonStats)»
```

This map type holds the registry.

8.19.3 Constructor & Destructor Documentation

LookupTableFactory()

```
template<typename TIN, typename TOUT = TIN>
LookupTableFactory () [inline]
```

Constructor initializes registry, default destructor.

8.19.4 Member Function Documentation

create()

```
template<typename TIN, typename TOUT>
std::unique_ptr< LookupTable< TIN, TOUT > > create (
                std::string name,
                const FunctionContainer< TIN, TOUT > & fc,
                const LookupTableParameters< TIN > & args,
                const nlohmann::json & jsonStats = nlohmann::json())
```

Create a lookup table from

- string_name - Stringified table type
- fc - FunctionContainer holding the function that the table evaluates
- args - Additional arguments needed for construcing the table

Create a new lookup table. Throw exception asking for an unregistered table.

get_registered_keys()

```
template<typename TIN, typename TOUT>
std::vector< std::string > get_registered_keys ()
```

Return a container of the keys for table types that have been registered

Return a vector of the keys that have been registered

The documentation for this class was generated from the following file:

- LookupTableFactory.hpp

8.20 LookupTableGenerator< TIN, TOUT, TERR > Class Template Reference

Generate a FunC LookupTable from a given name and one of the following: stepsize, tolerance, memory size limit, or filename with generate_by_step, generate_by_tol, generate_by_impl_size, and generate_by_file respectively. This class is also equipped to compute the error in a LUT built with any given stepsize and plot a LUT against its exact function.

`#include <LookupTableGenerator.hpp>`

Classes

- struct LookupTableErrorFunctor
- struct OptimalStepSizeFunctor

Public Member Functions

- **LookupTableGenerator** (const FunctionContainer< TIN, TOUT > &fc, const LookupTableParameters< TIN > &par)
- **LookupTableGenerator** (const FunctionContainer< TIN, TOUT > &fc, TIN minArg, TIN maxArg)
- std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_file (std::string filename, std::string tableKey="")
- std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_step (std::string tableKey, TIN stepSize, std::string filename="")
- std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_tol (std::string tableKey, TIN a_tol, TIN r_tol, std::string filename="")
- std::unique_ptr< LookupTable< TIN, TOUT > > **generate_by_tol** (std::string tableKey, TIN desiredErr, std::string filename="")
- std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_impl_size (std::string tableKey, unsigned long desiredSize, std::string filename="")
- long double error_at_step_size (std::string tableKey, TIN stepSize, TIN relTol=static_cast< TIN >(1.0))
- long double **error_of_table** (const LookupTable< TIN, TOUT > &L, TIN relTol=static_cast< TIN >(1.0))
- void plot_implementation_at_step_size (std::string tableKey, TIN table_step, TIN plot_step)
- TIN **min_arg** ()
- TIN **max_arg** ()

8.20.1 Detailed Description

template<typename TIN, typename TOUT = TIN, typename TERR = long double>
class func::LookupTableGenerator< TIN, TOUT, TERR >

Generate a FunC LookupTable from a given name and one of the following: stepsize, tolerance, memory size limit, or filename with generate_by_step, generate_by_tol, generate_by_impl_size, and generate_by_file respectively. This class is also equipped to compute the error in a LUT built with any given stepsize and plot a LUT against its exact function.

Note

> If generate_by_XXX is given a nonempty filename then it will generate a table once and save that output to to filename. Future runs will build the LUT from filename instead of generating that LUT from scratch.
>
> filenames are relative to the cwd unless users provide an absolute path.
>
> Many architectures (including Apples arm chips) typedef long double as double (truly horrendous)
>
> If Boost is not available then users can only use this class to build LUTs by file or by step.
>
> LookupTableGenerator is header only because it is templated on the error precision TERR. We MUST be able to cast TERR to TIN and vice versa. Ideally TERR satisfies: $\sqrt(\epsilon_{\text{TERR}}) <= \epsilon_{\text{TOUT}}$.

Todo Newton's iterations are currently unused because sometimes it'll try building a LUT so large it'll kill mortal computers. There must be a way to use it, but I'm not sure how.

8.20.2 Member Function Documentation

error_at_step_size()

```
template<typename TIN, typename TOUT, typename TERR>
long double error_at_step_size (
        std::string tableKey,
        TIN stepSize,
        TIN relTol = static_cast<TIN>(1.0))
```

Return the approx error in tableKey at stepSize

- relTol is a parameter which determines how much effect small f(x) values have on the error calculation

generate_by_file()

```
template<typename TIN, typename TOUT = TIN, typename TERR = long double>
std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_file (
        std::string filename,
        std::string tableKey = "") [inline]
```

A wrapper for the LookupTableFactory<std::string> which builds tables from a file tableKey arg only exists as a sanity check (it's pointless otherwise)

generate_by_impl_size()

```
template<typename TIN, typename TOUT, typename TERR>
std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_impl_size (
        std::string tableKey,
        unsigned long desiredSize,
        std::string filename = "")
```

Generate a table takes up desiredSize bytes

generate_by_step()

```
template<typename TIN, typename TOUT = TIN, typename TERR = long double>
std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_step (
        std::string tableKey,
        TIN stepSize,
        std::string filename = "") [inline]
```

A wrapper for the LookupTableFactory

generate_by_tol()

```
template<typename TIN, typename TOUT, typename TERR>
std::unique_ptr< LookupTable< TIN, TOUT > > generate_by_tol (
        std::string tableKey,
        TIN a_tol,
        TIN r_tol,
        std::string filename = "")
```

Generate a table that has the largest possible stepsize such that the error is less than desiredErr

plot_implementation_at_step_size()

```
template<typename TIN, typename TOUT, typename TERR>
void plot_implementation_at_step_size (
                    std::string tableKey,
                    TIN table_step,
                    TIN plot_step)
```

compare tableKey to the original function at stepSize

The documentation for this class was generated from the following file:

- LookupTableGenerator.hpp

8.21 LookupTableParameters< TIN, TOUT > Struct Template Reference

A struct containing data necessary/useful for constructing a LUT.

#include <LookupTable.hpp>

Public Member Functions

- **LookupTableParameters** (TIN min, TIN max, TIN step, std::vector< std::tuple< TIN, unsigned, TOUT > > pts)
- **LookupTableParameters** (TIN min, TIN max, TIN step)

Public Attributes

- TIN **minArg**
- TIN **maxArg**
- TIN **stepSize**
- std::vector< std::tuple< TIN, unsigned, TOUT > > **special_points**

    special_points (roots, critical points, inflection points)

8.21.1 Detailed Description

template<typename TIN, typename TOUT = TIN>
struct func::LookupTableParameters< TIN, TOUT >

A struct containing data necessary/useful for constructing a LUT.

The documentation for this struct was generated from the following file:

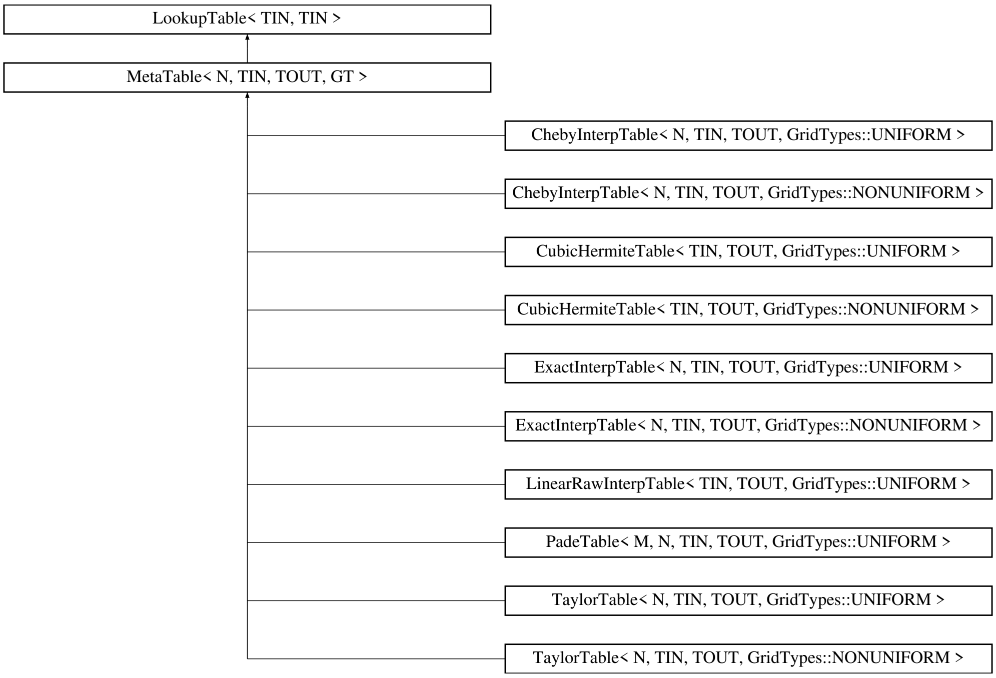
- LookupTable.hpp

8.22 MetaTable< N, TIN, TOUT, GT > Class Template Reference

MetaTable handles any piecewise *polynomial* based interpolation. Highly templated.

#include <MetaTable.hpp>

Inheritance diagram for MetaTable< N, TIN, TOUT, GT >:



Public Member Functions

- MetaTable ()=default
- MetaTable (const MetaTable< N, TIN, TOUT, GT > &L)
- MetaTable< N, TIN, TOUT, GT > & operator= (MetaTable< N, TIN, TOUT, GT > L)
- MetaTable (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TOUT **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TOUT, GT > & operator+= (const MetaTable< N, TIN, TOUT, GT > &other)
- MetaTable< N, TIN, TOUT, GT > & **operator-=** (const MetaTable< N, TIN, TOUT, GT > &other)
- MetaTable< N, TIN, TOUT, GT > & **operator*=** (const TIN &a)

- MetaTable< N, TIN, TOUT, GT > & **operator/=** (const TIN &a)
- template<GridTypes GT1, typename std::enable_if< GT1==GridTypes::UNIFORM, bool >::type = true>
  std::pair< unsigned int, TIN > hash (TIN x) const
- template<GridTypes GT1, typename std::enable_if< GT1==GridTypes::NONUNIFORM, bool >::type = true>
  std::pair< unsigned int, TIN > hash (TIN x) const
- TOUT operator() (TIN x) const override
- template<typename... TIN2>
  auto operator() (TIN x, TIN2 ... args) const

  If LUT coefficients (of type TOUT) are callable, then apply each to args *as soon as possible.*
- TOUT diff (unsigned int s, TIN x) const

  Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.
- template<typename... TIN2>
  auto **diff** (unsigned int s, TIN x, TIN2... args) const

  Same pattern as the variadic operator() but for the sth derivative.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Protected Member Functions

- void swap (MetaTable< N, TIN, TOUT, GT > &L) noexcept

  Copy-swap pattern, necessary for operator=

Protected Attributes

- std::string **m_name**

  name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

  bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

  fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

  > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

  order of accuracy of implementation
- std::size_t **m_dataSize**

  size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

  = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

  length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

  < holds polynomials coefficients

Friends

- template<unsigned int N1, typename TIN1, typename TOUT1, GridTypes GT1, typename std::enable_if< std::is_constructible< nlohmann::json, TIN1 >::value &&std::is_constructible< nlohmann::json, TOUT1 >::value, bool >::type>
  void from_json (const nlohmann::json &jsonStats, MetaTable< N1, TIN1, TOUT1, GT1 > &lut)

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

8.22.1 Detailed Description

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
class func::MetaTable< N, TIN, TOUT, GT >

MetaTable handles any piecewise *polynomial* based interpolation. Highly templated.

Note

If (max-min)/stepsize is not an integer then the actual maximum allowable argument m_tableMaxArg is std::ceil(m_stepSize_inv*(m_maxArg-m_minArg))) (which is slightly larger than the given max)

If (max-min)/stepsize *is* an integer, then $\mathrm{hash(max)=index\_max+1}$. To remedy this, every implementation must have an extra (unnecessary in most cases) entry in m_table

If TIN approximates a field and TOUT approximates a vector space over TIN, then MetaTable<N,TIN,TOUT,GT> approximates a vector space over TIN. MetaTable provides methods for addition/subtraction & scalar multiplication/division. This feature makes the N-D LUTs possible.

Provides functions to read/write LUTs from a JSON file by implementing nlohmann's to/from_json.

Template Parameters

N,unsigned	int: is the number of coefficients required to represent each polynomial. So, N = deg(p_k) + 1 where p_k is the polynomial approximating f over interval k
TIN,typename	is the type of the inputs passed to f.
TOUT,typename	is the type that f outputs (and the type of the coefficients of each p_k).
GT,GridType	is the type of partition that this LUT uses. The options are • UNIFORM: Every subinterval is the same length so the hash takes 4 FLOPs & zero comparisons; however, unnecessary subintervals might be needed • NONUNIFORM: Use a TransferFunction to create a nonuniform partition of [a,b] with an O(1) hash.

8.22.2 Constructor & Destructor Documentation

MetaTable() [1/3]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
MetaTable () [default]

Using a std::unique_ptr member variables implicitly deletes the default move ctor so we must explicitly ask for the default move ctor

MetaTable() [2/3]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
MetaTable (
    const MetaTable< N, TIN, TOUT, GT > & L) [inline]

deepcopy constructor

MetaTable() [3/3]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
MetaTable (
    const FunctionContainer< TIN, TOUT > & func_container,
    const LookupTableParameters< TIN > & par,
    const nlohmann::json & jsonStats) [inline]

Use a json file to set every generic member variable

8.22.3 Member Function Documentation

bounds_of_subinterval()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
std::pair< TIN, TIN > bounds_of_subinterval (
    unsigned int intervalNumber) const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

diff()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TOUT diff (
    unsigned int s,
    TIN x) const [inline]

Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Todo Make this function virtual and override in PadeTable and LinearRawInterpTables

hash() [1/2]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
template<GridTypes GT1, typename std::enable_if< GT1==GridTypes::UNIFORM, bool >::type = true>
std::pair< unsigned int, TIN > hash (
    TIN x) const [inline]

Find which polynomial p_k to evaluate. Also, each $p_k : [0,1] -> R$ so we must set $dx = (x - x_k)/(x_{k+1} - x_k)$

hash() [2/2]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
template<GridTypes GT1, typename std::enable_if< GT1==GridTypes::NONUNIFORM, bool >::type = true>
std::pair< unsigned int, TIN > hash (
TIN x) const [inline]

The polynomials for nonuniform LUTs map $[x_k, x_{k+1}] -> R$ so we don't have to preprocess x. Calls m_transfer←
Function.inverse(x), using 6 FLOPs and 4 std::array<TIN,4> access

max_arg()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TIN max_arg () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

min_arg()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TIN min_arg () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

name()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
std::string name () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

num_subintervals()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
unsigned int num_subintervals () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

operator()() [1/2]

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TOUT operator() (
TIN x) const [inline], [override], [virtual]

Find the subinterval $[x_k, x_{k+1})$ that x belongs to, fetch the coefficients of the polynomial $p_k(x)$ from $\mathrm{m\_table[k]}$, and use Horner's method to compute $p_k(x)$.

Todo PadeTable & LinearRawInterpTable must override this operator. Maybe operator() will be faster if each implementation provides their own operator() and diff(). If the vtable isn't optimized out then perchance removing the use of virtual will remove that overhead.

surely this could use openmp simd...

Implements LookupTable< TIN, TIN >.

Reimplemented in PadeTable< M, N, TIN, TOUT, GT >, and PadeTable< M, N, TIN, TOUT, GridTypes::UNIFORM >.

operator()() [2/2]

```
template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
template<typename... TIN2>
auto operator() (
        TIN x,
        TIN2 ... args) const [inline]
```

If LUT coefficients (of type TOUT) are callable, then apply each to args as soon as possible.

Note

We must use an auto return type because there's no straightforward way to statically deduce the return type ahead of time.

This function does not perform well with template expressions (e.g. LUTs of arma::mat) so use the other operator() in that case.

This variadic operator() cannot be in the LUT interface because it must be templated.

operator+=()

```
template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
MetaTable< N, TIN, TOUT, GT > & operator+= (
        const MetaTable< N, TIN, TOUT, GT > & other) [inline]
```

LUTs form a vector space over TIN if TOUT forms a vector space over TIN Note these operations could be heavily optimized with template expressions but that doesn't work with the auto keyword (currectly used with operator()(...)) and that'd be a loooot of work...

Maybe we could implement polynomial using an existing linear algebra library with fast (possibly template expression based) operator+ and operator∗?

operator=()

```
template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
MetaTable< N, TIN, TOUT, GT > & operator= (
        MetaTable< N, TIN, TOUT, GT > L) [inline]
```

Implemented with the copy-swap pattern

order()

```
template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
unsigned int order () const [inline], [final], [virtual]
```

Implements LookupTable< TIN, TIN >.

print_json()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
void print_json (
    std::ostream & out) const [inline], [final], [virtual]

Note

Every class implementing LookupTable should call their implementation of nlohmann's to_json from print↩ _json

Implements LookupTable< TIN, TIN >.

size()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
std::size_t size () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

step_size()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
TIN step_size () const [inline], [final], [virtual]

Implements LookupTable< TIN, TIN >.

swap()

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
void swap (
    MetaTable< N, TIN, TOUT, GT > & L) [inline], [protected], [noexcept]

Copy-swap pattern, necessary for operator=

Postcondition

L is overwritten by the data in this* and this* loses access to its data.

8.22.4 Friends And Related Symbol Documentation

from_json

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
template<unsigned int N1, typename TIN1, typename TOUT1, GridTypes GT1, typename std::enable_if< std::is_↩ constructible< nlohmann::json, TIN1 >::value &&std::is_constructible< nlohmann::json, TOUT1 >::value, bool >::type>
void from_json (
    const nlohmann::json & jsonStats,
    MetaTable< N1, TIN1, TOUT1, GT1 > & lut) [friend]

This declaration gives the nlohmann json functions access to every member variable. This *must* use SFINAE because we want people to construct LUTs over arbitrary types regardless of whether those types have implemented to/from json

8.22.5 Member Data Documentation

m_transferFunction

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
___attribute___ ((aligned)) std TransferFunction<TIN> m_transferFunction [protected]

< holds polynomials coefficients

used to make nonuniform grids (default constructable)

The documentation for this class was generated from the following file:

- MetaTable.hpp

8.23 nth_differentiable< TIN, TOUT, N > Struct Template Reference

These structs provide an indexed typedef for Boost's autodiff_fvar type.

`#include <FunctionContainer.hpp>`

Public Types

- using **type** = std::function<adVar<TOUT,N>(adVar<TIN,N>)>

8.23.1 Detailed Description

template<typename TIN, typename TOUT, unsigned int N>
struct func::nth_differentiable< TIN, TOUT, N >

These structs provide an indexed typedef for Boost's autodiff_fvar type.

The advantage of using nth_differentiable instead of a normal typedef is that we can define the function overloads for FunctionContainer::get_nth_func (the return type must be specified with an index).

Note

Does not exist in FunC if Boost's autodiff is not available

8.23.2 Member Typedef Documentation

type

```
template<typename TIN, typename TOUT, unsigned int N>
using type = std::function<adVar<TOUT,N>(adVar<TIN,N>)>
```

The documentation for this struct was generated from the following file:

- FunctionContainer.hpp

8.24 nth_differentiable< TIN, TOUT, 0 > Struct Template Reference

Base case for nth_differentiable when N=0.

#include <FunctionContainer.hpp>

Public Types

- using **type** = std::function<TOUT(TIN)>

8.24.1 Detailed Description

template<typename TIN, typename TOUT>
struct func::nth_differentiable< TIN, TOUT, 0 >

Base case for nth_differentiable when N=0.

8.24.2 Member Typedef Documentation

type

```
template<typename TIN, typename TOUT>
using type = std::function<TOUT(TIN)>
```

The documentation for this struct was generated from the following file:

- FunctionContainer.hpp

8.25 LookupTableGenerator< TIN, TOUT, TERR >::OptimalStepSizeFunctor Struct Reference

Public Member Functions

- **OptimalStepSizeFunctor** (LookupTableGenerator< TIN, TOUT, TERR > &parent, std::string tableKey, TERR relTol, TERR desiredErr)
- TIN **operator()** (const TIN &stepSize)
- TIN error_of_table (const LookupTable< TIN, TOUT > *impl)

8.25.1 Member Function Documentation

error_of_table()

```
template<typename TIN, typename TOUT = TIN, typename TERR = long double>
TIN error_of_table (
            const LookupTable< TIN, TOUT > * impl)   [inline]
```

Use Brent's method to find the maximum of $|f(x) - L(x)|/(a_{\text{tol}} + r_{\text{tol}}|f(x)|)$ over each subinterval of L.

Note

Must be careful about the last interval b/c tableMaxArg >= maxArg (and we don't care about error outside of table bounds)

Todo This parallelizes reasonably well, but does it really use the best pragma possible?

brent's method occasionally spends much more time on single intervals (stragglers) so there's some potential for

The documentation for this struct was generated from the following file:

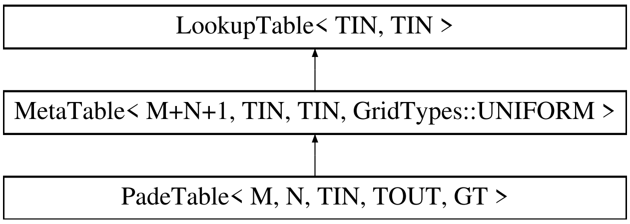
- LookupTableGenerator.hpp

8.26 PadeTable< M, N, TIN, TOUT, GT > Class Template Reference

LUT using [M/N] pade approximants.

#include <PadeTable.hpp>

Inheritance diagram for PadeTable< M, N, TIN, TOUT, GT >:



Public Member Functions

- **PadeTable** (const MetaTable< M+N+1, TIN, TOUT, GT > &L)
- **PadeTable** (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())
- TOUT operator() (TIN x) const final

Public Member Functions inherited from MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TIN, GT > & operator= (MetaTable< N, TIN, TIN, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TIN **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TIN, GT > & operator+= (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator-=** (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TIN, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TIN diff (unsigned int s, TIN x) const

  Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TIN, GT > &L) noexcept

  Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >

- std::string **m_name**

  name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

  bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

  fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

  > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

  order of accuracy of implementation
- std::size_t **m_dataSize**

  size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

  = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

  length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

  < holds polynomials coefficients

8.26.1 Detailed Description

template<unsigned int M, unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes↩
::UNIFORM>
class func::PadeTable< M, N, TIN, TOUT, GT >

LUT using [M/N] pade approximants.

Polynomial coefficients are calculated using Armadillo.

```
// PadeTable requires the user's mathematical function is templated
template <typename T>
T foo(T x){ return x; }

int main(){
  double min = 0.0, max = 10.0, step = 0.0001;
  // build every possible PadeTable
  UniformPadeTable<4,3,double> L1({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<3,3,double> L2({FUNC_SET_F(foo,double)}, {min, max, step});
```

```
  UniformPadeTable<5,2,double> L3({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<4,2,double> L4({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<3,2,double> L5({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<2,2,double> L6({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<6,1,double> L7({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<5,1,double> L8({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<4,1,double> L9({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<3,1,double> L10({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<2,1,double> L11({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformPadeTable<1,1,double> L12({FUNC_SET_F(foo,double)}, {min, max, step});
  auto val = L(0.87354);
}
```

Note

This class only works if TOUT and TIN can both be cast to double. Armadillo Mat<T>'s is_supported_↩ elem_type<T> will only let us do arithmetic with float or double (not even long double) and arma::field is useless.

Evaluate by using parentheses, just like a function

Each member function is marked const

Available template values are all M,N such that 0 < N <= M and M+N<=7

Template values where M < N are not supported

Requires both Armadillo and Boost version 1.71.0 or newer to generate

8.26.2 Member Function Documentation

operator()()

template<unsigned int M, unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::↩ UNIFORM>
TOUT operator() (
TIN x) const [inline], [final], [virtual]

Find the subinterval $[x_k, x_{k+1})$ that x belongs to, fetch the coefficients of the polynomial $p_k(x)$ from m_table[k], and use Horner's method to compute $p_k(x)$.

Todo PadeTable & LinearRawInterpTable must override this operator. Maybe operator() will be faster if each implementation provides their own operator() and diff(). If the vtable isn't optimized out then perchance removing the use of virtual will remove that overhead.

surely this could use openmp simd...

Reimplemented from MetaTable< M+N+1, TIN, TIN, GridTypes::UNIFORM >.

The documentation for this class was generated from the following file:

- PadeTable.hpp

8.27 polynomial_helper< TOUT, N, B > Struct Template Reference

A typedef for std::array<TOUT,N> along with some functions that interpret the array as polynomial coefficients.

8.27.1 Detailed Description

template<typename TOUT, unsigned int N, bool B>
struct func::polynomial_helper< TOUT, N, B >

A typedef for std::array<TOUT,N> along with some functions that interpret the array as polynomial coefficients.

Note

By convention, we write polynomials coefficients with increasing powers of x:

$$p(x) = \text{m\_table}[x0].\text{coefs}[0] + \text{m\_table}[x0].\text{coefs}[1]x + ... + \text{m\_table}[x0].\text{coefs}[N-1]x^{N-1}.$$

Polynomial arrays are sometimes aligned (currently only aligned for float or double)

Template Parameters

B	Boolean: determines whether an array of polynomials over TOUT are aligned with alignas(sizeof(TOUT)∗alignments[N])

Note

Polynomials can store other things, such as

- 3D LUTs may have coefs for x & y dimensions of each subrectangle,
- Coefficients of f's derivatives

The documentation for this struct was generated from the following file:

- Polynomial.hpp

8.28 polynomial_helper< TOUT, N, false > Struct Template Reference

Arrays of this type of polynomial are not aligned.

#include <Polynomial.hpp>

Public Member Functions

- constexpr unsigned int **size** () const noexcept

Public Attributes

- TOUT **coefs** [N]

8.28.1 Detailed Description

template<typename TOUT, unsigned int N>
struct func::polynomial_helper< TOUT, N, false >

Arrays of this type of polynomial are not aligned.

The documentation for this struct was generated from the following file:

- Polynomial.hpp

8.29 polynomial_helper< TOUT, N, true > Struct Template Reference

Arrays of this type of polynomial are aligned.

#include <Polynomial.hpp>

Public Member Functions

- constexpr unsigned int **size** () const noexcept

Public Attributes

- TOUT **coefs** [N]

8.29.1 Detailed Description

template<typename TOUT, unsigned int N>
struct func::polynomial_helper< TOUT, N, true >

Arrays of this type of polynomial are aligned.

The documentation for this struct was generated from the following file:

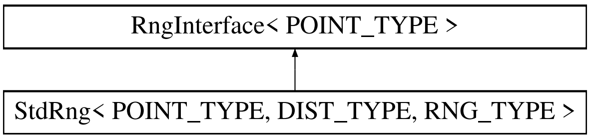
- Polynomial.hpp

8.30 RngInterface< POINT_TYPE > Class Template Reference

Abstract interface for classes that can generate random numbers.

#include <RngInterface.hpp>

Inheritance diagram for RngInterface< POINT_TYPE >:



Public Member Functions

- virtual void init (unsigned int seed)=0
- virtual unsigned int seed ()=0
- virtual POINT_TYPE get_point ()=0

8.30.1 Detailed Description

template<typename POINT_TYPE>
class func::RngInterface< POINT_TYPE >

Abstract interface for classes that can generate random numbers.

8.30.2 Member Function Documentation

get_point()

```
template<typename POINT_TYPE>
virtual POINT_TYPE get_point ()   [pure virtual]
```

get a random point from the random distribution

Implemented in StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >.

init()

```
template<typename POINT_TYPE>
virtual void init (
                 unsigned int seed)   [pure virtual]
```

build a random generator for sampling from a distribution

Implemented in StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >.

seed()

```
template<typename POINT_TYPE>
virtual unsigned int seed ()   [pure virtual]
```

return the current seed

Implemented in StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >.

The documentation for this class was generated from the following file:

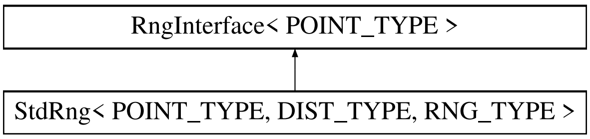
- RngInterface.hpp

8.31 StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE > Class Template Reference

An implementation of RngInterface, intended to be used with the generators/ distributions defined in std::random. Only LookupTableComparator uses this class, and it's for sampling random arguments.

`#include <StdRng.hpp>`

Inheritance diagram for StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >:



Public Member Functions

- StdRng (std::unique_ptr< DIST_TYPE > dist)
- template<typename ... DIST_TYPE_ARGS>
  StdRng (DIST_TYPE_ARGS ... args)
- void init (unsigned int seed)
- unsigned int seed ()
- POINT_TYPE get_point ()

8.31.1 Detailed Description

template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE = std::mt19937>
class func::StdRng< POINT_TYPE, DIST_TYPE, RNG_TYPE >

An implementation of RngInterface, intended to be used with the generators/ distributions defined in std::random. Only LookupTableComparator uses this class, and it's for sampling random arguments.

Note

Will take ownership if given a probability distribution

Will build its own probability distribution corresponding to DIST_TYPE if given the correct constructor args.

Builds its own number generator when init(seed) is called

Usage example:

```
// uniform_real_distribution<double> generated from std::mt19937
// within the range 0.0 to 1.0
StdRng<double> rng(0.0,1.0); // build a std::uniform_real_distribution<double>
rng.init(2020);              // build a std::mt19937
rng.get_point();             // return a random number

// or if you want to get fancy and use a faster generator, here's
// a normal distribution with mean 0.0 and standard deviation 1.0,
// using points generated from minstd_rand0
StdRng<float,std::normal_distribution<float>,minstd_rand0> rng2(0.0,1.0));
```

8.31.2 Constructor & Destructor Documentation

StdRng() [1/2]

```
template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE
= std::mt19937>
StdRng (
            std::unique_ptr< DIST_TYPE > dist) [inline]
```

Take ownership of the pointer to probability distribution passed in

StdRng() [2/2]

```
template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE
= std::mt19937>
template<typename ... DIST_TYPE_ARGS>
StdRng (
            DIST_TYPE_ARGS ... args) [inline]
```

Create a new StdRng based on the template type and its parameters

8.31.3 Member Function Documentation

get_point()

```
template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE
= std::mt19937>
POINT_TYPE get_point () [inline], [virtual]
```

Get a random point from the given distribution

Implements RngInterface< POINT_TYPE >.

init()

```
template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE
= std::mt19937>
void init (
            unsigned int seed) [inline], [virtual]
```

Set the seed and generator of the distribution

Implements RngInterface< POINT_TYPE >.

seed()

```
template<typename POINT_TYPE, class DIST_TYPE = std::uniform_real_distribution<double>, class RNG_TYPE
= std::mt19937>
unsigned int seed () [inline], [virtual]
```

Return the seed

Implements RngInterface< POINT_TYPE >.

The documentation for this class was generated from the following file:

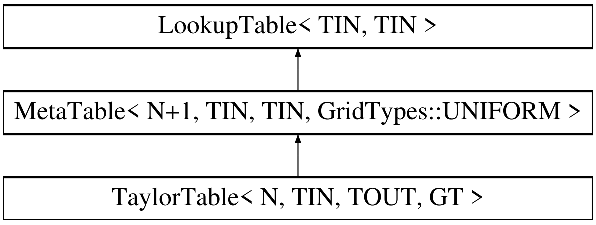
- StdRng.hpp

8.32 TaylorTable< N, TIN, TOUT, GT > Class Template Reference

LUT using degree 1 to 7 truncated Taylor series.

#include <TaylorTable.hpp>

Inheritance diagram for TaylorTable< N, TIN, TOUT, GT >:



Public Member Functions

- **TaylorTable** (const MetaTable< N+1, TIN, TOUT, GT > &L)
- **TaylorTable** (const FunctionContainer< TIN, TOUT > &func_container, const LookupTableParameters< TIN > &par, const nlohmann::json &jsonStats=nlohmann::json())

Public Member Functions inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- MetaTable ()=default
- MetaTable< N, TIN, TIN, GT > & operator= (MetaTable< N, TIN, TIN, GT > L)
- std::string name () const final
- TIN min_arg () const final
- TIN max_arg () const final
- TIN **tablemax_arg** () const
- unsigned int order () const final
- std::size_t size () const final
- unsigned int num_subintervals () const final
- TIN step_size () const final
- std::pair< TIN, TIN > bounds_of_subinterval (unsigned int intervalNumber) const final
- void print_json (std::ostream &out) const final
- unsigned int **num_table_entries** () const
- unsigned int **ncoefs_per_entry** () const
- TIN **table_entry** (unsigned int i, unsigned int j) const
- std::array< TIN, 4 > **transfer_function_coefs** () const
- MetaTable< N, TIN, TIN, GT > & operator+= (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator-=** (const MetaTable< N, TIN, TIN, GT > &other)
- MetaTable< N, TIN, TIN, GT > & **operator*=** (const TIN &a)
- MetaTable< N, TIN, TIN, GT > & **operator/=** (const TIN &a)
- std::pair< unsigned int, TIN > hash (TIN x) const
- TIN operator() (TIN x) const override
- TIN diff (unsigned int s, TIN x) const

  Return the sth derivative of L at x: ie return $p_k^{(s)}(x)$.

Public Member Functions inherited from LookupTable< TIN, TIN >

- LookupTable ()=default

Additional Inherited Members

Public Types inherited from LookupTable< TIN, TIN >

- using **input_type**
- using **output_type**

Protected Member Functions inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- void swap (MetaTable< N, TIN, TIN, GT > &L) noexcept

  Copy-swap pattern, necessary for operator=

Protected Attributes inherited from MetaTable< N+1, TIN, TIN, GridTypes::UNIFORM >

- std::string **m_name**

  name of implementation type
- TIN **m_minArg**
- TIN **m_maxArg**

  bounds of evaluation
- TIN **m_stepSize**
- TIN **m_stepSize_inv**

  fixed grid spacing (and its inverse)
- TIN **m_tableMaxArg**

  > m_maxArg if (m_maxArg-m_minArg)/m_stepSize is non-integer
- unsigned int **m_order**

  order of accuracy of implementation
- std::size_t **m_dataSize**

  size of relevant data for impl evaluation
- unsigned int **m_numIntervals**

  = (m_tableMaxArg - m_minArg)/m_stepSize;
- unsigned int **m_numTableEntries**

  length of m_table (usually = m_numIntervals + 1)
- __attribute__((aligned)) std TransferFunction< TIN > m_transferFunction

  < holds polynomials coefficients

8.32.1 Detailed Description

template<unsigned int N, typename TIN, typename TOUT = TIN, GridTypes GT = GridTypes::UNIFORM>
class func::TaylorTable< N, TIN, TOUT, GT >

LUT using degree 1 to 7 truncated Taylor series.

```
// TaylorTable requires the user's mathematical function is templated
template <typename T>
T foo(T x){ return x; }

int main(){
  double min = 0.0, max = 10.0, step = 0.0001;
  UniformTaylorTable<1,double> L1({FUNC_SET_F(foo,double)}, {min, max, step}); // uniform
  UniformTaylorTable<2,double> L2({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformTaylorTable<3,double> L3({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformTaylorTable<4,double> L4({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformTaylorTable<5,double> L5({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformTaylorTable<6,double> L6({FUNC_SET_F(foo,double)}, {min, max, step});
  UniformTaylorTable<7,double> L6({FUNC_SET_F(foo,double)}, {min, max, step});
  NonUniformTaylorTable<1,double> L6({FUNC_SET_F(foo,double)}, {min, max, step}); // nonuniform
  NonUniformTaylorTable<7,double> L6({FUNC_SET_F(foo,double)}, {min, max, step});
  auto val = L(0.87354);
}
```

Note

Each member function is marked const

evaluate by using parentheses, just like a function

The documentation for this class was generated from the following file:

- TaylorTable.hpp

8.33 Timer Class Reference

Starts a timer when created. Stops when stop() is called and returns the duration in seconds with duration().

#include <Timer.hpp>

Public Member Functions

- Timer ()
- void stop ()
- double duration ()

8.33.1 Detailed Description

Starts a timer when created. Stops when stop() is called and returns the duration in seconds with duration().

Note

The timer's duration is static after calling stop().

This measures the elapsed time in seconds.

8.33.2 Constructor & Destructor Documentation

Timer()

Timer () [inline]

Start the timer

8.33.3 Member Function Documentation

duration()

double duration () [inline]

Return how long the timer ran for

stop()

void stop () [inline]

Stop the timer

The documentation for this class was generated from the following file:

- Timer.hpp

8.34 TransferFunction< TIN > Class Template Reference

Transforms a uniformly spaced partition of $[a, b]$ into a nonuniform partition of $[a, b]$.

#include <TransferFunction.hpp>

Public Member Functions

- **TransferFunction** (const std::array< TIN, 4 > &inv_coefs)
- template<typename TOUT>
  TransferFunction (const FunctionContainer< TIN, TOUT > &fc, TIN minArg, TIN tableMaxArg, TIN step↩ Size)
- TIN inverse (TIN x) const
- TIN inverse_diff (TIN x) const
- TIN operator() (TIN x) const
- std::array< TIN, 4 > **get_coefs** () const
- TIN **min_arg** () const
- TIN **max_arg** () const

8.34.1 Detailed Description

template<typename TIN>
class func::TransferFunction< TIN >

Transforms a uniformly spaced partition of $[a, b]$ into a nonuniform partition of $[a, b]$.

For efficiency, we require a Transfer function is simply an increasing cubic polynomial such that $p(a) = 0, p(b) = b/$stepSize. To help compete against uniform lookup tables, part of the operator() must be baked into those coefficients (explaining why stepSize appears in the formula above).

When given a FunctionContainer with a defined first derivative then this will construct a valid TransferFunction. Let f be a function defined over $[a, b]$. Then, construct $S : [a, b] \to [a, b]$ as

$$S(x) = a + \frac{b-a}{c}\int_a^x \frac{1}{\sqrt{1 + f'(t)^2}}dt,$$

where $c = \int_a^b \frac{1}{\sqrt{1+f'(t)^2}}dt$. This way, $S(a) = a, S(b) = b$. Computing S^{-1} has to be fast so we approximate is as a monotone Hermite cubic polynomial and then rebuild S as $(S^{-1})^{-1}$ using Boost's newton_raphson_iterate.

Note

Can be constructed with a std::array<TIN,4> of polynomial coefficients.

An example: the identity transfer function which is the linear polynomial with coefs {-m_minArg/m_step↩ Size,1/m_stepSize,0,0}.

We need Boost version 1.71.0 or newer to generate a candidate transfer function. Boost is not required if TransferFunction is given coefficients.

Resulting nonuniform LUT performs better if f' is largest near the endpoints [a,b]. If f' is largest near the middle of an interval (eg e^{-x^2} when $a < -3$ and $b > 3$) then the grid remains uniform. This is an issue with the way we approximate S!

Todo Currently, transfer functions are basically useless if f' is not extreme near the endpoints of its domain. (b/c we use cubic hermite interpolation at the endpoints).

We want to accurately approximate S anywhere $f'(x)$ is extreme! The best possible polynomial approximating S might be promising...

Note

A cubic polynomial $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ is monotone over $\mathbb{R}$ if and only if $a_2^2 < 3a_1a_3$. Maybe we can compute a polynomial minimizing $\max_t |f(t) - p(t)|$ such that $p(a) = a, p(b) = b$, and $a_2^2 < 3a_1a_3$. That would be a much better general purpose solution. The search space is convex!

8.34.2 Constructor & Destructor Documentation

TransferFunction()

```
template<typename TIN>
template<typename TOUT>
TransferFunction (
        const FunctionContainer< TIN, TOUT > & fc,
        TIN minArg,
        TIN tableMaxArg,
        TIN stepSize) [inline]
```

Build the coefficients in g_inv

8.34.3 Member Function Documentation

inverse()

```
template<typename TIN>
TIN inverse (
        TIN x) const [inline]
```

Evaluate the inverse using Horner's method

inverse_diff()

```
template<typename TIN>
TIN inverse_diff (
        TIN x) const [inline]
```

Evaluate the derivative of the inverse using Horner's method

operator()()

```
template<typename TIN>
TIN operator() (
        TIN x) const [inline]
```

Use newton-raphson_iteration on p. Recall that each coef in g was divided by h and we subtracted by m_minArg

The documentation for this class was generated from the following file:

- TransferFunction.hpp

9 File Documentation

9.1 File List

Here is a list of all documented files with brief descriptions:

9.2 FunctionContainer.hpp File Reference

Define a FunctionContainer with the (necessary) convenience function get_nth_func(), the "variadic" macro #FUNC_SET_F.

```
#include <stdexcept>
#include <functional>
#include "config.hpp"
#include <boost/math/differentiation/autodiff.hpp>
```

Classes

- struct nth_differentiable< TIN, TOUT, N >
  These structs provide an indexed typedef for Boost's autodiff_fvar type.
- struct nth_differentiable< TIN, TOUT, 0 >
  Base case for nth_differentiable *when N=0.*
- class FunctionContainer< TIN, TOUT >
  Wrapper for std::function<TOUT(TIN)> *and some optional* std::function*s of Boost's automatic differentiation type.*

Macros

- #define FUNC_SET_F_ONE_TYPE(F, TYPE)
- #define FUNC_SET_F_TWO_TYPE(F, TIN, TOUT)
- #define FUNC_GET_MACRO_FUNCTION_CONTAINER(_1, _2, _3, NAME, ...)
- #define FUNC_SET_F(...)
  Macro to list out FunctionContainer constructor arguments. Call as FUNC_SET_F(foo,template-type...) *where.*

Typedefs

- template<typename T, unsigned int N>
  using **adVar** = autodiff_fvar<T,N>

9.2.1 Detailed Description

Define a FunctionContainer with the (necessary) convenience function get_nth_func(), the "variadic" macro #FUNC_SET_F.

Note

Boost's autodiff.cpp is only included if CMake found a new enough version of Boost

9.2.2 Macro Definition Documentation

FUNC_GET_MACRO_FUNCTION_CONTAINER

```
#define FUNC_GET_MACRO_FUNCTION_CONTAINER(
        _1,
        _2,
        _3,
        NAME,
        ...)
```

Value:

```
NAME
```

FUNC_SET_F

```
#define FUNC_SET_F(
        ...)
```

Value:

```
FUNC_GET_MACRO_FUNCTION_CONTAINER(__VA_ARGS__, FUNC_SET_F_TWO_TYPE,
    FUNC_SET_F_ONE_TYPE, )(__VA_ARGS__)
```

Macro to list out FunctionContainer constructor arguments. Call as FUNC_SET_F(foo,template-type...) where.

- foo is your function name, and
- template-type is as list of the 1 or 2 types it maps between.

FUNC_SET_F_ONE_TYPE

```
#define FUNC_SET_F_ONE_TYPE(
        F,
        TYPE)
```

Value:

```
F<TYPE>, F<func::adVar<TYPE,1>>,                    \
F<func::adVar<TYPE,2>>, F<func::adVar<TYPE,3>>, \
F<func::adVar<TYPE,4>>, F<func::adVar<TYPE,5>>, \
F<func::adVar<TYPE,6>>, F<func::adVar<TYPE,7>>
```

FUNC_SET_F_TWO_TYPE

```
#define FUNC_SET_F_TWO_TYPE(
        F,
        TIN,
        TOUT)
```

Value:

```
F<TIN,TOUT>, F<func::adVar<TIN,1>,func::adVar<TOUT,1>>,                                    \
F<func::adVar<TIN,2>,func::adVar<TOUT,2>>, F<func::adVar<TIN,3>,func::adVar<TOUT,3>>, \
F<func::adVar<TIN,4>,func::adVar<TOUT,4>>, F<func::adVar<TIN,5>,func::adVar<TOUT,5>>, \
F<func::adVar<TIN,6>,func::adVar<TOUT,6>>, F<func::adVar<TIN,7>,func::adVar<TOUT,7>>
```

9.3 Polynomial.hpp File Reference

Define a polynomial and provide several helper functions.

#include <string>
#include <ostream>

Classes

- struct polynomial_helper< TOUT, N, true >

  Arrays of this type of polynomial are aligned.
- struct polynomial_helper< TOUT, N, false >

  Arrays of this type of polynomial are not aligned.

Typedefs

- template<typename TOUT, unsigned int N>
  using **polynomial** = polynomial_helper<TOUT,N,std::is_floating_point<TOUT>::value>

Functions

- constexpr unsigned int **factorial** (unsigned int n)
- constexpr unsigned int **permutation** (unsigned int n, unsigned int k)
- template<unsigned int N, typename TOUT, typename TIN = TOUT>
  TOUT polynomial_diff (polynomial< TOUT, N > p, TIN x, unsigned s)

  Compute $p^{(s)}(x)$, the sth derivative of p at x.
- template<unsigned int N, typename TOUT, typename TIN = TOUT>
  polynomial< TOUT, N > taylor_shift (polynomial< TOUT, N > p, TIN a, TIN b, TIN c, TIN d)

  Given a polynomial $p : [a, b] \to \mathbb{R}$, compute the coefficients of $q : [c, d] \to \mathbb{R}$ such that $q(x) = p([(b-a)x + (ad - bc)]/(d - c))$ by expanding p in a Taylor series.
- template<unsigned int N, typename TOUT, typename TIN = TOUT>
  TOUT eval (polynomial< TOUT, N > p, TIN x)

  Compute $p(x)$.
- template<unsigned int N, typename TOUT>
  std::string polynomial_print (const polynomial< TOUT, N > &p)
- template<unsigned int N, typename TOUT>
  std::ostream & operator<< (std::ostream &out, const polynomial< TOUT, N > &p)
- template<unsigned int N, typename TOUT>
  std::string to_string (const polynomial< TOUT, N > &p)

9.3.1 Detailed Description

Define a polynomial and provide several helper functions.

9.3.2 Function Documentation

eval()

template<unsigned int N, typename TOUT, typename TIN = TOUT>
TOUT eval (
    polynomial< TOUT, N > p,
    TIN x) [inline]

Compute $p(x)$.

Note

p cannot be empty

operator<<()

template<unsigned int N, typename TOUT>
std::ostream & operator<< (
    std::ostream & out,
    const polynomial< TOUT, N > & p)

Print basic info about a polynomial

polynomial_diff()

template<unsigned int N, typename TOUT, typename TIN = TOUT>
TOUT polynomial_diff (
    polynomial< TOUT, N > p,
    TIN x,
    unsigned s) [inline]

Compute $p^{(s)}(x)$, the sth derivative of p at x.

Note

p cannot be empty

polynomial_print()

template<unsigned int N, typename TOUT>
std::string polynomial_print (
    const polynomial< TOUT, N > & p)

Convenient debugging method for printing a polynomial. TODO this could wrap operator<<

taylor_shift()

```
template<unsigned int N, typename TOUT, typename TIN = TOUT>
polynomial< TOUT, N > taylor_shift (
        polynomial< TOUT, N > p,
        TIN a,
        TIN b,
        TIN c,
        TIN d) [inline]
```

Given a polynomial $p : [a, b] \to \mathbb{R}$, compute the coefficients of $q : [c, d] \to \mathbb{R}$ such that $q(x) = p([(b-a)x + (ad - bc)]/(d - c))$ by expanding p in a Taylor series.

This is used all over FunC (for example, special case for rightmost interval, Nonuniform LUTs, and Taylor/Pade tables). Optimizations are very welcome!

to_string()

```
template<unsigned int N, typename TOUT>
std::string to_string (
        const polynomial< TOUT, N > & p) [inline]
```

wraps operator<<

9.4 cxx17utils.hpp File Reference

Beta features for building multivariate LUTs via currying. Each feature requires C++17.

```
#include "LookupTable.hpp"
#include <boost/math/tools/minima.hpp>
#include <boost/math/tools/roots.hpp>
#include <boost/math/special_functions/next.hpp>
#include <boost/math/special_functions/sign.hpp>
#include <boost/multiprecision/cpp_bin_float.hpp>
```

Classes

- struct curriedLUT< N, TIN, TOUT, classname >
- struct curriedLUT< 0, TIN, TOUT, classname >

Functions

- template<unsigned int N, typename TERR, class LUT, typename F>
  TERR **metric** (LUT L, F f)

  Compute the error in a multivariate LUT.
- template<unsigned int N, typename TIN, typename TOUT, template< typename... > class classname, class F, typename... TIN2>
  constexpr curriedLUT< N-1, TIN, TOUT, classname >::type ndimLUT (F f, const std::vector< LookupTableParameters< TIN > > &params, TIN2... other)

  BETA FEATURE: Define a LUT with N input variables.

9.4.1 Detailed Description

Beta features for building multivariate LUTs via currying. Each feature requires C++17.

9.4.2 Function Documentation

ndimLUT()

```
template<unsigned int N, typename TIN, typename TOUT, template< typename... > class classname, class F, typename...
TIN2>
curriedLUT< N-1, TIN, TOUT, classname >::type ndimLUT (
        F f,
        const std::vector< LookupTableParameters< TIN > > & params,
        TIN2... other) [constexpr]
```

BETA FEATURE: Define a LUT with N input variables.

Call as func::ndimLUT<ndim,type1,type2,luttype>(f, params) See examples/2D_lut.cpp for example usage

Todo This is not compatible with function derivatives (needed for nonuniform partition & Taylor tables). Is that impossible to support anyways?

Template argument classname should be variadic! Is this possible?

10 Todo List

Class ArgumentRecord< TIN >

Implement functions to_json & from_json

The histogram will never have that many buckets so we could likely get away with simply making every one of this class's member variables threadprivate.

Perhaps we should use boost histogram instead but that would add an additional dependency

Member ArgumentRecord< TIN >::ArgumentRecord (nlohmann::json jsonStats)

Maybe optionally provide an ostream?

Class CompositeLookupTable< TIN, TOUT >

Implement to/from_json. We can use the unique_ptr<LookupTable> version of from_json in LookupTableFactory to build each member LUT easily

Class FailureProofTable< LUT_TYPE >

This class will support to_json but not from_json because it needs a FunctionContainer. Add another constructor to build this class from a FunctionContainer and a filename

Member func::ndimLUT (F f, const std::vector< LookupTableParameters< TIN > > &params, TIN2... other)

This is not compatible with function derivatives (needed for nonuniform partition & Taylor tables). Is that impossible to support anyways?

Template argument classname should be variadic! Is this possible?

Member LinearRawInterpTable< TIN, TOUT, GT >::operator() (TIN x) const override

is there a way to make this work with nonuniform grids in a way that works with our model?

Class LookupTableComparator< TIN, TOUT >

LookupTableComparator's constructor should accept any callable type

Class LookupTableGenerator< TIN, TOUT, TERR >

Newton's iterations are currently unused because sometimes it'll try building a LUT so large it'll kill mortal computers. There must be a way to use it, but I'm not sure how.

Member LookupTableGenerator< TIN, TOUT, TERR >::OptimalStepSizeFunctor::error_of_table (const LookupTable< TIN, TOUT > ∗impl)

This parallelizes reasonably well, but does it really use the best pragma possible?

brent's method occasionally spends much more time on single intervals (stragglers) so there's some potential for

Member MetaTable< N, TIN, TOUT, GT >::diff (unsigned int s, TIN x) const

Make this function virtual and override in PadeTable and LinearRawInterpTables

Member MetaTable< N, TIN, TOUT, GT >::operator() (TIN x) const override

PadeTable & LinearRawInterpTable must override this operator. Maybe operator() will be faster if each implementation provides their own operator() and diff(). If the vtable isn't optimized out then perchance removing the use of virtual will remove that overhead.

surely this could use openmp simd...

Class TransferFunction< TIN >

Currently, transfer functions are basically useless if f' is not extreme near the endpoints of its domain. (b/c we use cubic hermite interpolation at the endpoints).