SIAM J. SCI. COMPUT.
Vol. 41, No. 3, pp. C194–C218

DIRECT FUNCTION EVALUATION VERSUS LOOKUP TABLES: WHEN TO USE WHICH?*

KEVIN R. GREEN†, TANNER A. BOHN‡, AND RAYMOND J. SPITERI§

Abstract. The speed of mathematical function evaluations can significantly contribute to the overall performance of numerical simulations. Two common approaches to evaluate a mathematical function are by direct evaluation or by means of lookup tables of precomputed values. Direct evaluation is the standard approach, and lookup tables are common in graphics applications and field-programmable gate array computing. We address the usage of direct function evaluation versus lookup tables for evaluating general univariate functions on general-purpose CPUs for large-scale numerical simulation. We introduce a small C++ library called `FunC` (for **Fun**ction **C**omparator) that can be run on a given system to quickly and conveniently evaluate the performance of direct evaluation relative to various lookup table implementations for arbitrary user-specified functions. A user can then make an informed decision as to which method can be expected to be most efficient for their application. Using functions relevant to the Chaste software library for the simulation of cardiac models, `FunC` successfully predicts that lookup table implementations outperform direct evaluation. After implementing higher-order lookup tables in Chaste to test these predictions, we demonstrate the effects of table size on two simulations, one small and one large. For the small simulation, we see an increase in the simulation speed of up to 25% for tables using cubic interpolation over those using linear interpolation and up to 144% for cubic tables over direct evaluation. For the more realistic large simulation, we see an increase in the simulation speed by up to 86% for lookup tables over direct evaluation.

Key words. lookup table, direct function evaluation, heart simulation, cardiac potentials

AMS subject classifications. 65A05, 65Y20, 68U20

DOI. 10.1137/18M1201421

1. Introduction.

Many sophisticated scientific computing codes require evaluation of complicated mathematical functions f. These functions may be evaluated billions of times during code execution, and hence a considerable portion of the overall computational effort can be devoted to them.

There are two main approaches for evaluating such functions. The first and more natural approach to evaluating mathematical functions is via *direct evaluation*. What constitutes *direct evaluation* in the present context is the implementation of an evaluation strategy that relies on pre-existing routines that evaluate such functions as accurately as possible in a given precision, similar to how intrinsic functions are used. Examples of such functions include `exp`, `sin`, and `cos` from `math.h`, functions under `boost.math` in C++, and combinations or compositions thereof. The second approach is via the implementation of *lookup tables* (LUTs), whereby a set of precomputed function values are stored in memory for quick retrieval using a hash followed by some form of local approximation (e.g., interpolation or Taylor series) in order to achieve a desired accuracy.

Multivariate LUTs are commonly used in graphics programming for operations

*Submitted to the journal's Software and High-Performance Computing section July 19, 2018; accepted for publication (in revised form) March 21, 2019; published electronically June 4, 2019.
http://www.siam.org/journals/sisc/41-3/M120142.html

†Department of Mathematics and Statistics, University of Saskatchewan, Saskatoon, SK, S7N 5E6, Canada (kevin.green@usask.ca).

‡Department of Applied Mathematics, Western University, London, ON, N6A 5B7, Canada (tannerbohn@gmail.com).

§Department of Computer Science, University of Saskatchewan, Saskatoon, SK, S7N 5C9, Canada (spiteri@cs.usask.ca).

such as color-space transformations or texture mapping [20]; indeed, the current colloquial usage of LUT generally refers to this area of application. These multivariate functions can be included in the framework of this study, but such transformations are typically not as sensitive to error as, e.g., numerical solutions of differential equations, and thus a LUT implementation based on a coarsely sampled space with linear interpolation often suffices in the realm of graphics programming.

The concept of a LUT is quite general in computing because its use can be applied to any quantity that involves storage of precomputed values to be later reused rather than recomputed. A prominent example of this in scientific libraries is FFTW (Fastest Fourier Transform in the West), which uses LUTs in two different contexts: for memorization of its computation protocols and for precomputation of FFT twiddle factors [11]. Another example is RRTM (Rapid Radiative Transfer Model), which uses LUTs for multivariate absorption coefficients [18, 6]. More recent advancements investigate the use of hardware LUTs for computing sigmoidal activation functions in hardware neural networks, e.g., as in Dias, Sales, and Osorio [10].

A specific software library that makes precise use of our vision of LUTs is the Cancer, Heart, and Soft Tissue Environment (Chaste) [28], which is a C++ library that uses biophysical models of excitable cells to simulate electrical activity in network or continuum tissue domains. The ordinary differential equations (ODEs) used for the cell models have a wide range of complexity, and some of them require repeated evaluation of complicated right-hand side functions. An example of a particular right-hand side function that arises from the ten Tusscher–Panfilov model [25] of human epicardial cells (TTP2006Epi) is

$$f(x) = \frac{0.67(1 + e^{0.14285714285714285(x+35)})^{-1} + 0.33}{562e^{-0.004166666666666666(x+27)^2} + 31(1 + e^{0.1(25-x)})^{-1} + 80(1 + e^{0.1(x+30)})^{-1}}, \tag{1}$$

which combines various evaluations of the exponential function. This function must be evaluated at least once for every cell, every time step of a simulation, and potentially with many different values of its input argument, x. Furthermore, this function is only one of 40 similar recurring patterns identified in the cell model that have many different shift and scaling factors in the exponentials.

Because scientific codes such as those mentioned above often involve compositions, additions, and products of elementary functions, significant efforts towards optimizing the evaluation of elementary functions have been made; see, e.g., Yamamoto, Kitamura, and Yamane [31]. A special-purpose compiler that is capable of generating LUTs that use Taylor-series interpolants based on accuracy and memory constraints is provided by Zhang et al. in [32] and can produce both C++ software implementations as well as field-programmable gate array hardware modules.

The MESA tool by Wilcox, Strout, and Bieman [29, 30] takes things a step further by automating the detection of candidate areas where LUTs may be beneficial through C/C++ source-code analysis as well as providing a more sophisticated algorithm for determining an optimized LUT implementation. The LUTs for this tool, however, are limited to constant and linear interpolation—higher-order interpolation methods are deemed too expensive for practical purposes. For the application domains investigated in [30], the authors conclude that "the LUT data must reside primarily in midlevel cache to be effective."

We argue that the use of higher-order tables has many more upsides than down. In a way similar to how one justifies the usage of higher-order discretization methods for differential equations, namely through the tradeoff of error versus computational

cost and an increase in storage efficiency, we note that higher-order LUTs benefit from these features as well. The only downside to using higher-order LUTs comes from a slight increase in flops required for their evaluation. The results of subsection 4.5 demonstrate that this increase in flops is effectively negligible for large scientific computing problems.

LUTs are typically used in scenarios where function evaluations are repeated a large number of times and the results need not be on the order of roundoff error. Treating the construction and storage of the tables as a sunk cost, it is clear that the use of LUTs pays off computationally when it costs less to retrieve and postprocess a small number of data points than it does to perform all the instructions needed for a direct evaluation. Determining the tipping point of this tradeoff, however, is generally a daunting task because there are many factors that come into play, e.g., hardware and compiler optimizations that are hard to account for in a purely theoretical approach. Locating this tipping point precisely is not crucial, but being comfortably on the appropriate side of it can make a significant difference to the performance of a code.

This paper addresses the problem of generating software LUT implementations for arbitrary mathematical functions in the practical sense of determining whether it is beneficial to use a software LUT implementation over a given direct evaluation implementation as well as potential circumstances that can affect *which* software LUT implementation to use. The generality we allow in f destroys any hope of generating LUT implementations based on a priori considerations such as theory for bounding the truncation and rounding errors of a function evaluation that is typically used in the literature for specific elementary and special functions.

To this end, we present a simple C++ **Fun**ction **C**omparator library (`FunC`) [12] that can be used to conveniently evaluate the performance of different implementations of a scalar function evaluation and compare their performance. The library is designed such that specifying the details of an implementation is straightforward so that new C++ implementations can be easily added. We do not consider here the inclusion of implementations from other languages, but this can be done as well by defining a new implementation type that derives from the `EvaluationImplementation` base class (described below) and provides the allocation in its constructor and appropriate function evaluation in the overloaded brackets.

The results of this study focus on comparing the performance of direct evaluation and uniform interpolation LUTs, i.e., LUTs based on uniformly spaced data. We demonstrate significant performance improvements of LUTs over direct evaluation on two commonly used biophysical cell models; the improvements are large enough to recommend the inclusion of higher-order LUTs in Chaste. For large-scale simulations, it is shown that even LUT implementations that are incapable of remaining in cache nonetheless allow for substantial performance improvements over direct evaluation.

The remainder of this paper is organized as follows. In section 2, we describe `FunC` and its features. In section 3, we describe how LUTs are currently used in the Chaste library. In section 4, we give results of (i) the `FunC` framework applied to the elementary exponential function, (ii) the application of `FunC` to functions from cardiac cell models, and (iii) the use of LUTs in Chaste simulation tests. Finally in section 5, we summarize our main results and discuss potential improvements to `FunC` to handle more challenging scenarios. All timing results in this paper are generated on the system specified in Table 1 while the system was in use for other computational problems, i.e., under a use-case that can generally be expected of shared high-performance computing resources.

TABLE 1
System used for timing results.

CPU	Intel Xeon CPU E5-2620 0 @ 2.00GHz
RAM	32GB DDR4 @ 1333MHz
OS	Linux 3.13.0-135-generic #184-Ubuntu SMP Wed. Oct. 18 11:55:51 UTC 2017
compiler	`gcc` (Ubuntu/Linaro 6.3.0-18ubuntu2 14.04) 6.3.0 20170519

2. Description of `FunC`. Because modern computing environments can be quite complicated, it is often difficult to predict a priori whether an algorithm or its implementation will outperform another with the same purpose. Thus, in order to decide whether it is preferable to use direct evaluation or a LUT for evaluating a mathematical function within a given computing environment, we posit that an effective method is to directly gather experimental evidence by performing and timing function evaluations. This approach has the capacity to take into account most of the factors that may affect direct evaluation and LUT times; timings can be run using the same compiler, hardware, and even data distribution as the eventual program for which the decision is being made.

We now provide a brief description of a simple C++ library, `FunC`, and the generation and implementation of uniform interpolation LUTs used in this paper. Example usage of the library and further implementation details are available on GitHub [12]. `FunC` currently does not provide any direct analysis for the parallel evaluation of any of the described implementations at any level. We note, however, that all of the described implementations can run unimpeded in a parallel environment and could be evaluated within the framework of `FunC` with some additional effort.

2.1. `FunC` main components. The main idea of `FunC` is that any given mathematical function evaluation has an implementation that can be viewed as a derived class of an abstract base class that we call `EvaluationImplementation`. The choice of components for this base class are follows:

1. a function object (`EvaluationFunctor`) that contains the function to be evaluated and potentially its derivatives—absolutely necessary for determining the function being evaluated; generalizes evaluation to implementations that can make use of additional information about the function;
2. the argument domain on which the function is to be evaluated—necessary for LUT implementations that use uniformly spaced data (as in this study); useful for all finite domains;
3. the order of accuracy of the implementation—not strictly necessary, but often known; useful when generating LUTs to meet a specific tolerance as described in subsection 2.3.
4. the size of data necessary for evaluation of the implementation—necessary for a given implementation; useful for logging/comparison purposes; and
5. the name of the implementation—not necessary in principle but useful for categorizing and logging purposes.

The `EvaluationImplementation` base class has a pure virtual bracket operator (it is expected to be callable like a standard function); this must be provided by derived classes to be usable. It also provides a virtual destructor and a virtual method called `print_details`, which may be overwritten by derived classes.

The comparison class (`ImplementationComparator`) is designed to estimate the times required for implementation evaluations with consistent overhead while maintaining a level of abstraction that makes it easy to incorporate into a larger system

of code. `ImplementationComparator` takes the following entities upon construction:

1. a vector of pointers to existing implementations to evaluate,
2. a number of evaluations to test, and
3. an optional seed for a random number generator.

With an `ImplementationComparator` in place, multiple timings of the specified evaluations for each implementation can be taken. At present, no detailed statistical analysis of the results is built in beyond sorting according to fastest, mean, or slowest observed run times.

2.2. Abstraction of direct evaluation. The derived class (`DirectEvaluation`) of `EvaluationImplementation` simply evaluates the function that is supplied in its constructor. An immediate issue precipitated by this choice of implementation is the possible difference between this abstraction and direct evaluation in the form of a preprocessor macro. The resolution of this issue is not entirely straightforward. One may expect that a macro substitution before compilation would allow more optimization to be performed on the evaluation. To test this hypothesis, we perform function evaluation benchmarking on both the abstracted and macro variants for two different cases: (i) the exponential $\exp(x)$ and (ii) the function (1). The timing results are shown in Figure 1.

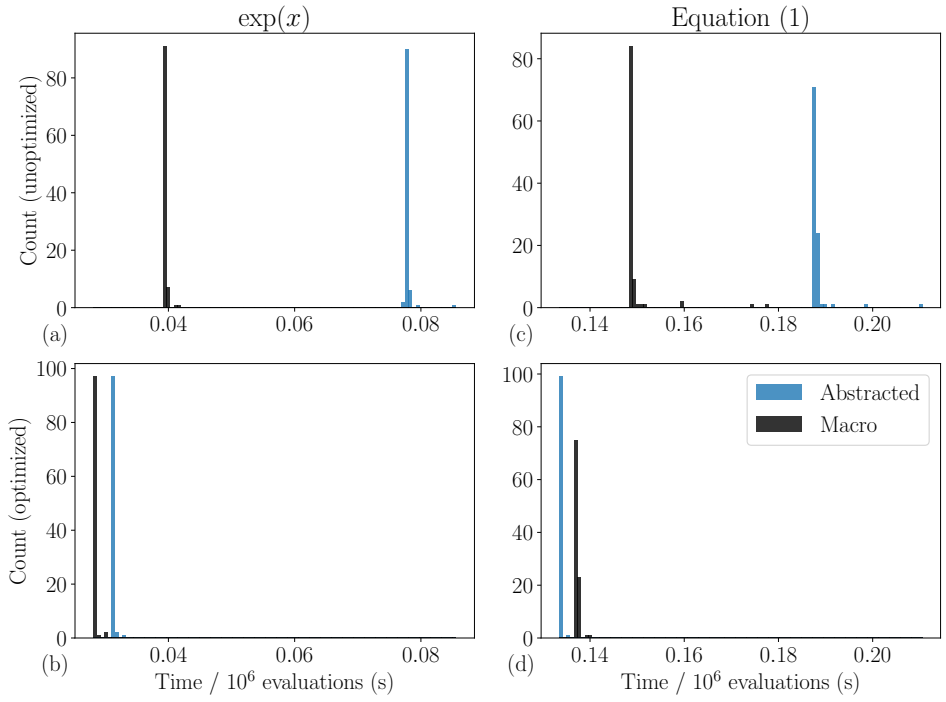

FIG. 1. *Execution times for abstracted and macro direct evaluations of* $\exp(x)$ *and* (1). *Histograms are generated from* 100 *runs of each case on the system described in Table* 1. *Optimized results are obtained with the* `-O2` *flag.*

From Figure 1(a)–(b), we observe that the overhead of calling the abstracted variant is clear in both the unoptimized and optimized variants of $\exp(x)$. This

overhead is also clear from Figure 1(c) in the unoptimized compilation of (1), showing approximately the same magnitude of difference between the abstracted and macro variants as $\exp(x)$. This difference, however, represents a much smaller percentage of the evaluation time for the more complicated function (1).

In contrast, the comparison of optimized results for (1) tells a different story. Figure 1(d) shows that the abstracted evaluation outperforms the macro substitution in this case, and it also outperforms the unoptimized direct evaluation; cf. Figure 1(c).

Without looking at the generated machine code, the explanation can only be hypothesized; e.g., the optimizer for the macro evaluation may at times be trying to do too much for the evaluation of (1), leading to inefficient use of the instruction cache. Such outcomes are highly compiler and machine dependent. Nonetheless, it seems safe to expect that the differences in evaluation times incurred by using abstracted direct evaluations instead of macro substitution are much smaller than the differences observed in comparing direct evaluations to LUT implementations for sufficiently complicated mathematical functions. In fact, comparison of Figure 1 with the results of subsection 4.2 shows that the differences between the optimized abstracted and macro evaluations is less than 10% of the total time for the worst-case LUT performance. Henceforth, we assume it is acceptable to use abstracted function evaluations in our performance comparisons.

2.3. Uniform interpolation LUT. Because we consider only uniform interpolation lookup tables for this study, our usage of LUT is taken to mean "uniform interpolation LUT." Our view of LUTs is through the abstract `UniformLookupTable` intermediate class. This class derives from `EvaluationImplementation` but remains abstract because it does not yet define the bracket operator. It does provide members that are useful for *any* LUT implementation making use of uniformly spaced data. A general destructor that cleans up the data array used for implementation is implemented at this level. The benefit of having this intermediate class is that components useful for generating LUTs can still be quite general, and a registry of implemented `UniformLookupTable` types available to use in the library can be maintained.

Specific LUT types then derive from `UniformLookupTable` and implement their own data allocation (through their constructor) and their own evaluation (through their bracket operator). The `UniformCubicPrecomputedInterpolationTable` implementation is shown in Figure 2 to serve as an example. When new classes that derive from `UniformLookupTable` are coded, they can be registered with a singleton registry within the library. The implementation of this registry is such that additional `UniformLookupTable` types may be implemented outside of the library yet still placed within the registry.

`UniformLookupTableGenerator` is an additional class that is responsible for the generation of any LUT implementation that exists in the singleton registry. LUTs can be generated by their desired step size, by a desired evaluation tolerance, or by a desired implementation size. Generation by step size is straightforward; it is, in fact, the default method used for constructing LUTs with `FunC`.

For LUT generation according to a tolerance, the maximum evaluation error must be determined for a LUT with a given step size and then the step size adjusted accordingly. This is where the generality of f comes in to play—the error cannot be practically bound based on theory applied to general f. Instead, the maximum error

is computed numerically for a table with a given step size. The error measure used is

$$E = \max_x \frac{\left|f(x) - \tilde{f}(x)\right|}{\frac{1}{2}(\left|f(x)\right| + \left|\tilde{f}(x)\right|)}, \tag{2}$$

where f represents the direct evaluation and $\tilde{f}$ an alternative implementation. Strictly speaking, E is a measure of the *relative difference* to the average of the magnitudes of f and $\tilde{f}$. However, direct evaluations are often accurate down to machine precision. In such cases, E is indeed a relative error, and we adopt a notation that reflects this. This definition of E helps to alleviate problems associated with general relative error formulae, as discussed in [26].

Assuming that the LUT interpolation error is approximated well by its asymptotic expansion, we can combine the order of the interpolation with a Newton-like iteration in log-log space to hone in on a table step size that gives the desired error. A brief description of such a procedure goes as follows. For a given implementation with step size h, we can evaluate (2), meaning we can effectively view $E = E(h)$. For a LUT with interpolation order p, we can write

$$E(h) = Ch^p, \tag{3}$$

where C is an arbitrary constant. Taking the natural logarithm of this equation results in

$$\log E(h) = \log C + p \log h,$$

having derivative

$$\frac{\mathrm{d} \log E(h)}{\mathrm{d} \log h} = p. \tag{4}$$

To strictly satisfy a specified error tolerance τ, the goal is to find a step size h such that

$$E(h) \lesssim \tau, \tag{5}$$

leading to

$$\log E(h) - \log \tau \lesssim 0. \tag{6}$$

Thus, finding an approximate solution to (6) as an equality can be done by a Newton-like iteration of the form

$$\log h_i = \log h_{i-1} - \frac{1}{p} \left(\log E(h_{i-1}) - \log \tau\right), \quad i = 1, 2, \ldots. \tag{7}$$

In practice, we have found that the convergence of such an iteration is generally insensitive to whether the initial step size is larger or smaller than the converged solution. Because of roundoff errors, however, this approach is not guaranteed to yield a solution that strictly satisfies (5).

To guarantee satisfaction of (5), we use a bracketing method in $\log E(\log h)$ space. The specific bracketing method we use is the `bracket_and_solve_root` function from the `boost` C++ libraries [4]. This function expands a bracket around an initial guess

until it is guaranteed to contain a root, and then performs TOMS algorithm 748 [1] to obtain the root to a desired precision. The benefit of this algorithm is that because it delivers a bracket around the root to some tolerance, we can choose the lower end of the bracket to guarantee (5). In `FunC`, we allow the combination of both the log-Newton iteration (7) followed by the bracketing algorithm; however, in many cases using just the bracketing algorithm to find a step size h that satisfies (5) should suffice.

Numerical algorithms for optimization are generally capable of finding the location of the optimum with an accuracy of $\sqrt{\epsilon_{\text{mach}}}$. Therefore, determining the location of the maximum evaluation error should be performed in a precision that is roughly double that of the desired working precision. In C++, this can also be accomplished easily with `boost`. For example, `boost` provides the `brent_find_minima` function, which is an implementation of Brent's method for minimization [5] that can be run in quadruple precision. For a LUT, this can be run on every interval in the implementation, after which the global maximum error can be determined.

The precision of an approximation to a function f with a LUT is limited by a combination of the rounding errors present in direct evaluation of f itself and the rounding errors of the interpolation method used. Because approximations of f of $\mathcal{O}(\epsilon_{\text{mach}})$ are assumed to not be required, LUT implementations suffice.

The type of table used is called a *precomputed interpolation* table. Specifically, a `UniformXXXPrecomputedInterpolationTable` of step size h, where `XXX` is one of {`Linear`, `Quadratic`, `Cubic`, ...}, precomputes and stores interpolating polynomial coefficients on each interval (x_i, x_{i+1}) using the points $x_i, x_i + h/n, x_i + 2h/n, \ldots, x_{i+1}$, where $h = x_{i+1} - x_i$ is the step size of the table and $n > 0$ is the degree of the interpolating polynomial. Evaluation of such a table for a given x then only requires determination of the interval in which x lies (an index mapping or trivial hash function) and an evaluation of the polynomial at $x - x_i$ via Horner's method [13]. For example, see Figure 2 for the entirety of the implementation of a `UniformCubicPrecomputedInterpolationTable`. For the case of $n = 0$ (piecewise-constant interpolation), `FunC` uses a `UniformConstantInterpolationTable`. Accordingly, only the hash is required to determine the closest point in the table to the desired evaluation point; the "precomputed" coefficients are really just the function evaluations at the uniformly spaced points. Although advantages of simplicity and computational efficiency are readily apparent, this approach is prone to requiring significantly larger table sizes than higher-order LUTs.

We have implemented other table types in `FunC` based on local Taylor approximations and Hermite interpolation but omit detailed results for the sake of brevity. Instead, we mention that their implementation can both be handled in a similar way to the precomputed interpolation tables, namely, by computing and storing polynomial coefficients followed by Horner evaluation. For all of the functions we have tried, the Taylor tables consistently underperform relative to the precomputed LUTs of the same order, and the cubic Hermite precomputed LUT performs similarly to the cubic precomputed LUTs.

2.4. Best- and worst-case performance scenarios. The performance of implementations that require precomputed values depends on where the needed precomputed values are present in the memory hierarchy at the time of the evaluation. For example, a LUT that is more often in cache is likely to outperform one that is not, and so LUT size may play a large role in performance. This situation is further confounded by other active processes in a given system, something that cannot generally be accounted for.

```
#define IMPL_NAME UniformCubicPrecomputedInterpolationTable
REGISTER_ULUT_IMPL(IMPL_NAME);
UniformCubicPrecomputedInterpolationTable::UniformCubicPrecomputedInterpolationTable(
     EvaluationFunctor<double,double> *func, UniformLookupTableParameters par)
  : UniformLookupTable(func, par)
{
  /* Base class variables */
  m_name = STR(IMPL_NAME);
  m_order = 4;
  m_numTableEntries = 4*(m_numIntervals+1);
  m_dataSize = (unsigned) sizeof(double)*m_numTableEntries;
  /* Allocate and set table */
  m_table.reset(new double[m_numTableEntries]);
  for (int ii=0;ii<m_numIntervals;++ii) {
    const double x = m_minArg + ii*m_stepSize;
    m_grid[ii]     = x;
    /* compute & store polynomial coefficients */
    const double y0 = (*mp_func)(x);
    const double y1 = (*mp_func)(x+m_stepSize/3);
    const double y2 = (*mp_func)(x+2*m_stepSize/3);
    const double y3 = (*mp_func)(x+m_stepSize);
    m_table[4*ii]   = y0;
    m_table[4*ii+1] = -11*y0/2+9*y1-9*y2/2+y3;
    m_table[4*ii+2] = 9*y0-45*y1/2+18*y2-9*y3/2;
    m_table[4*ii+3] = -9*y0/2+27*y1/2-27*y2/2+9*y3/2;
  }
}
double UniformCubicPrecomputedInterpolationTable::operator()(double x)
{
  double   dx = m_stepSize_inv*(x-m_minArg); // scaled & shifted x coord
  unsigned x0 = (unsigned) dx;               // scaled location of previous table entry
  dx         -= x0;                          // scaled distance from previous grid point
  x0         *= 4;                           // table index value
  // interpolation
  return m_table[x0]+dx*(m_table[x0+1]+dx*(m_table[x0+2]+dx*m_table[x0+3]));
}
```

FIG. 2. *Entirety of the implementation file for the uniform cubic precomputed interpolation table. Shown are registration, construction, and evaluation of a* `UniformCubicPrecomputedInterpolationTable` *using member variables from the parent* `UniformLookupTable` *class.*

Nonetheless, we can provide a rule of thumb in the direction of the worst case: *If LUT data have a low likelihood of staying in cache, then the implementation with the fewest flops generally performs the best.* In the worst case, we can assume that LUT implementations must always be fetched from RAM. When every evaluation requires a fetch from RAM, all implementations are approximately equivalent in terms of getting data to the right location, differing only in which operations they perform on the data.

It is this worst-case scenario that is most relevant for scientific computing. Developing an approach to approximating this performance on a given system is essential for meaningful results that translate to real numerical simulation, and this is the main component that makes `FunC` valuable as a tool external to any simulation code.

At the other extreme, a similar rule of thumb applies to the best-case scenario: *If LUT data have a high likelihood of staying in cache, then the implementation with the fewest flops generally performs the best.* This is because if the data are always available in cache, LUT implementations are again on equal footing for fetch timings, and all that comes down to their evaluation is how fast the operations can be performed. Fewer operations directly results in less time for evaluation.

Summarizing the above, regardless of whether all LUTs are always in cache (or always out of cache), the implementation that requires the fewest flops would be expected to have the best performance because the memory latencies in loading the table data would be expected to be similar. The nuance comes when dealing with tables of differing sizes; such a case begins to confound the memory latencies of different cache levels and potentially even main memory.

This is too complex a problem to completely characterize. An implementation's ability to stay in cache is inversely proportional to its size, but it also depends on everything else on which a processor is working. We can account for the size, but not the "everything else." In addition, considering even the different levels of cache in modern CPUs adds yet another factor.

All of the above considered, we approximate the worst case by generating implementations that are far larger than a practical implementation would generally require and evaluate them at a large number of random sample points. Measuring the time T_{worst} for a group of evaluations, we then choose the best time for each desired implementation—the best performance under the worst-case scenario. Similarly for the best case, we generate implementations that are far smaller than a practical implementation would require and measure that time T_{best} for a small number of evaluations, again taking the fastest time. The time here gives us an approximation of the time required for an implementation to be evaluated when its data are already in cache—the best performance under the best-case scenario.

One can estimate the total evaluation time T_{total} by forming a convex combination of the best-case and the worst-case times of a given implementation, parameterized linearly by the probability that the implementation would be in cache, P_{cache}:

$$T_{\text{total}} = P_{\text{cache}} T_{\text{best}} + (1 - P_{\text{cache}})\, T_{\text{worst}}. \tag{8}$$

Equation (8) allows the total time to be predicted given the value of P_{cache}. Due to the many factors already discussed, however, it is difficult to measure P_{cache} in practice.

However, (8) does allow a comparison of implementations by equating T_{total} for two separate implementations i and j to determine a break-even point,

$$P^i_{\text{cache}} T^i_{\text{best}} + \left(1 - P^i_{\text{cache}}\right) T^i_{\text{worst}} = P^j_{\text{cache}} T^j_{\text{best}} + \left(1 - P^j_{\text{cache}}\right) T^j_{\text{worst}}.$$

Rearranging, we can write

$$P^j_{\text{cache}} = \alpha_{ij} P^i_{\text{cache}} + \beta_{ij}, \tag{9}$$

with

$$\alpha_{ij} = \frac{T^i_{\text{worst}} - T^i_{\text{best}}}{T^j_{\text{worst}} - T^j_{\text{best}}}, \quad \beta_{ij} = \frac{T^j_{\text{worst}} - T^i_{\text{worst}}}{T^j_{\text{worst}} - T^j_{\text{best}}}.$$

The parameter β_{ij} can be interpreted simply by setting $P^i_{\text{cache}} = 0$ in (9). If implementation j *cannot* stay in cache at least this fraction of the time throughout a numerical simulation, then it will *not* outperform implementation i.

Equation (9) allows us to determine the regions in $(P^i_{\text{cache}}, P^j_{\text{cache}})$ space where one implementation is guaranteed to outperform another. We are not aware of the extent to which such a model is useful in a general practical sense because we are mainly interested in scenarios where the worst-case performance likely holds.

2.4.1. Usage of `FunC`. The usage of `FunC` can be split into three stages to be managed by the user:

1. implementation generation,
2. implementation comparison, and
3. data postprocessing.

Implementation generation requires a different approach for different implementation types in general. Because we are only dealing with direct function evaluation and LUTs, we note that direct evaluation does not require additional generation steps, but all LUTs can be generated at the same abstraction level using the `UniformLookupTableGenerator` class. `UniformLookupTableGenerator` has the following methods:

- `generate_by_tol`,
- `generate_by_step`, and
- `generate_by_impl_size`

that allow generation according to a desired tolerance, a step size, or an implementation's data size. Each is useful in its own way. The generation according to a tolerance generates an implementation according to the algorithm described in subsection 2.3. This is generally the slowest method of generation because it involves iteratively generating implementations at different step sizes and evaluating their errors. However, once this has been performed for a given function at a given tolerance, the step size of the resulting implementation can be stored and only requires the use of the `generate_by_step` function for future use. The generation by implementation size is useful for generating the implementations needed for our best- and worst-case approximations as described in subsection 2.4, but it may also be useful in the presence of a strict memory requirement.

Implementation comparison is handled with the `ImplementationComparator` class. This requires a vector of pointers to existing implementations to do its work. Once initialized with the implementations, it can be run simply by calling its `run_timings` method that times a user-specified number of uniformly sampled randomized evaluations on the appropriate domains and repeats this for a user-specified number of trials.

Finally, `ImplementationComparator` contains methods that can be used for data postprocessing as well. First, the timing statistics (minimum, maximum, and mean runtimes) may be computed using the `compute_timing_statistics` method. Next, the compared implementations may be sorted (in place) according to one of the criteria using the `sort_timings` method. Finally, the information from the comparison may be output in some way.

`FunC` provides routines for summarized output in various forms that can be sorted by minimum, maximum, or mean runtimes. Detailed output of timings can be formulated in terms of a json file. The json output contains all of the `raw` runtimes of all of the registered implementations for all of the trials as well as the summarized `min`, `max`, and `mean` data of each for quick reference. The json output also contains information about the number of evaluations in each trial, `nEvals`, and the total number of trials performed, `nTrials`.

A simple main program file that goes through these steps, showing the basic components needed to use the `FunC` library and some sample output, is found in Appendix A in the supplementary material.

3. Chaste LUT implementations and profiling. The Chaste library already uses linear LUTs for cell model function evaluations when built as an optimized vari-

ant [7]. We note that this is not a precomputed interpolation table that stores the polynomial coefficients before computation. Rather, the polynomial coefficients are computed with every evaluation. Such an implementation may be acceptable for the linear case because (i) it uses the same amount of memory as a constant LUT with the same step size (which is half that of a precomputed linear LUT) and (ii) the coefficient calculations require a single flop, in contrast, e.g., with the complexity of the coefficients in the cubic interpolation shown in Figure 2.

Within Chaste, the PyCml tool [21] is responsible for generating the C++ code for evaluation of cell model ODE functions. PyCml is a Python program that reads a CellML file [9] and generates either the relevant direct evaluation code or identifies expensive, recurring function evaluations and automatically generates linear LUT implementations. The LUT parameters can be supplied via an additional XML file along with the CellML file. For example, there are 15 single-variable functions that are tagged for table lookup in the Luo–Rudy cell model (LR1991) [16], and 40 such functions for the TTP2006Epi model, all of which can be found in Appendix B in the supplementary material.

Because there are often many functions that are required to be evaluated for the same variable, the LUT implementation for all functions of the same variable are grouped such that the table entries at specific variable values are as close as possible in memory. We have extended the functionality of PyCml to include the generation of precomputed cubic LUTs that maintain this closeness in memory.

3.1. Two-dimensional bidomain profiling and benchmarking. Chaste includes its own profiling capabilities. The first results presented are based on Chaste release 3.4 and its `TestPerformance01.hpp` test file. This test file computes a small simulation of a two-dimensional bidomain model [27] using the LR1991 cell model.

The bidomain model is widely accepted as a useful model for describing the electrical activity in myocardial tissue. It is a multiscale reaction-diffusion equation that couples systems of nonlinear ODEs describing the chemical reactions and the local flow of ions at the cell membrane level with a system of partial differential equations (PDEs) that describe the diffusion of the electrical activation through the cardiac muscle. To account for both of these effects, the tissue is divided into two co-located domains, the intracellular and the extracellular. These domains are separated by a nonconductive medium, the cell membrane, such that intracellular and extracellular currents can only influence each other via ion exchange through the membrane. Because the bidomain model is a continuum-based model, individual myocardial cells are generally not modelled. Rather, their net effect is captured in an averaged sense.

In a spatial domain $\Omega \subset \mathbb{R}^d$ of dimension d and time interval $[t_0, t_f]$, the bidomain model can be written as

$$\frac{\partial \mathbf{s}}{\partial t} = \mathbf{f}(t, \mathbf{s}, v), \tag{10a}$$

$$\chi C_m \frac{\partial v}{\partial t} + \chi I_{\text{ion}}(t, \mathbf{s}, v) = \nabla \cdot (\sigma_i \nabla v) + \nabla \cdot (\sigma_i \nabla u_e), \tag{10b}$$

$$0 = \nabla \cdot (\sigma_i \nabla v) + \nabla \cdot ((\sigma_i + \sigma_e) \nabla u_e), \tag{10c}$$

subject to boundary conditions on $\partial\Omega \times [t_0, t_f]$ given by

$$\hat{\mathbf{n}} \cdot (\sigma_i \nabla v + \sigma_i \nabla u_e) = 0, \tag{10d}$$

$$\hat{\mathbf{n}} \cdot (\sigma_e \nabla u_e) = 0, \tag{10e}$$

where $\mathbf{s} = \mathbf{s}(\mathbf{x}, t)$ is a vector describing the cellular state at location $\mathbf{x} \in \Omega$ and time $t \in [t_0, t_f]$, $v = v(\mathbf{x}, t)$ is the transmembrane potential, and $u_e = u_e(\mathbf{x}, t)$ is the extracellular potential. The ionic currents, represented by $I_{\text{ion}}(v, \mathbf{s})$, and $\mathbf{f}(t, \mathbf{s}, v)$ are nonlinear terms related to the cell model, and σ_i and σ_e are the intracellular and extracellular conductivity tensors, respectively. Finally, χ is the area of the cell membrane per unit volume, C_m is the capacitance of the cell membrane per unit area, and $\hat{\mathbf{n}}$ is the outward unit normal to the domain of consideration, $\partial\Omega$.

The simulation is performed on a 0.2 cm $\times$ 0.2 cm square domain, split into 128 equally sized triangular elements. Timings of the different components of the simulation are recorded while it runs. The total time of bidomain model simulations depends mostly on two components, (i) advancing the cell model ODEs, which are obtained at each node from the reaction PDEs upon spatial discretization, for each time step, and (ii) solving the linear system that arises from discretization of the potential-conserving diffusion PDEs for each time step. Further details on the numerical solution of the bidomain model can be found in, e.g., [24] and references therein.

For the benchmarks performed in this study, first-order Godunov operator splitting is used [23]. The cell model ODEs and the diffusion PDEs are advanced (separately) via the backward Euler method. The ODE system is nonlinear and thus requires a Newton iteration to update its state. The accuracy of LUT approximations can have an effect on the convergence of this iteration, potentially undermining convergence if the function evaluations are too inaccurate.

We have modified the `TestPerformance01.hpp` test file to also use the TTP2006Epi cell model, which is more computationally intensive than the LR1991 model and, because it is more stiff [17], can be expected to require more right-hand side evaluations. We document the effect of using LUTs for function evaluations with both of these cell models using the gcc compiler with `-O2` optimization—Chaste's `GccOpt` variant.

3.2. Niederer monodomain profiling and benchmarking. The benchmark problem proposed by Niederer et al. in [19] is a cardiac simulation problem that was designed to allow consistent benchmarking across various continuum cardiac solvers. The Niederer benchmark uses the monodomain model [24], obtained from the bidomain model (10) under the assumption that anisotropies in the internal and external conductivity tensors are proportional, i.e., $\sigma_i = \lambda\sigma_e$, where λ is a constant scalar:

$$\frac{\partial \mathbf{s}}{\partial t} = \mathbf{f}(t, \mathbf{s}, v), \tag{11a}$$

$$\chi C_m \frac{\partial v}{\partial t} + \chi I_{\text{ion}}(t, \mathbf{s}, v) = \frac{\lambda}{1+\lambda} \nabla \cdot (\sigma_i \nabla v), \tag{11b}$$

subject to the boundary conditions

$$\hat{\mathbf{n}} \cdot (\sigma_i \nabla v) = 0. \tag{11c}$$

The Niederer benchmark uses TTP2006Epi to specify the cell model functions $\mathbf{f}(t, \mathbf{s}, v)$ and $I_{\text{ion}}(t, \mathbf{s}, v)$, where $\mathbf{s}$ has 18 variables [25].

We solve the Niederer benchmark on a three-dimensional domain of size 0.3 cm $\times$ 0.7 cm $\times$ 2.0 cm that is split into cubes of size $(0.02 \text{ cm})^3$ that are in turn split equally into two tetrahedral elements. The result is 105,000 equally sized elements over the spatial domain of the simulation. Although this is by no means an inordinately large simulation by current standards, it is indeed large enough such that a single state vector is on the order of the size of tens of MB. Combining this with the required matrix-vector multiplications needed for solving the finite-element discretization of

the problem, it is clear that this cannot all fit within the cache of a contemporary CPU.

The time interval for the solution is 40 ms. The time integration is performed using a semi-implicit first-order method that applies backward Euler to the tissue equations and forward Euler to the cell model with a time step of 5×10^{-3} ms. Again, the simulation is performed with `-O2` optimization in Chaste's `GccOpt` build.

4. Results. We determine the effects of using LUT implementations by considering five different scenarios:

1. implementation size for evaluation at tolerance,
2. best- and worst-case behaviors of implementations,
3. complexity of function being evaluated,
4. benchmarking in situ implementations for small problems, and
5. benchmarking in situ implementations for large problems.

We begin by considering the relative sizes of LUT implementations for evaluation to specific tolerances followed by performance evaluation of the different-order LUT implementations on a given system. To determine the performance of direct evaluations of various functions and how this compares to the LUT performance, we run two separate sets of functions with varying complexity through `FunC`. To determine the impact of smaller LUT implementations in Chaste, we run the two two-dimensional cardiac bidomain simulations described. Finally, to determine the impact of direct evaluation versus LUT implementations in Chaste on a problem with a more realistic size, we run the Niederer benchmark.

4.1. LUT accuracy and size. Although higher-order LUT implementations require more information for a given function evaluation, their overall memory footprint is generally smaller (for a given accuracy) as order increases. As demonstration of this, Figure 3 shows the accuracy of various LUT implementations as a function of both table step size and table implementation size for the function in (1) evaluated on the interval $x \in (-250, 550)$. Both visualizations show the expected scaling for each implementation.

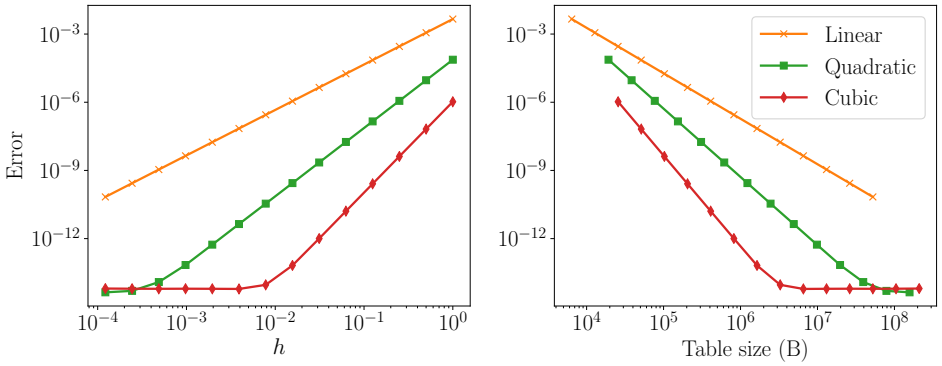

FIG. 3. *Error versus step size and error versus table size for* (1), $x \in (-250, 550)$. *For a desired error, one can find the appropriate step size and the subsequent table size for a given implementation.*

As a specific illustration, we consider the case where an error tolerance of 10^{-9} is required for evaluations of (1). Figure 3 shows that the quadratic table takes about

ten times more storage than the cubic table, and that the linear table takes more than twenty times more storage than the quadratic. Thus, the implementation size of the cubic LUT is at least a factor of 200 less than the implementation size of the linear LUT. Going from a linear to a cubic LUT would reduce the storage requirement from the order of tens of megabytes to the order of tens (to hundreds) of kilobytes. Such a difference in size is likely to dramatically influence the caching behavior of the LUT and hence its performance.

Looking to an even lower tolerance, e.g., 10^{-13}, and extrapolating the data for the linear table, we can determine that a linear LUT would require on the order of 100GB of storage, whereas a cubic LUT only requires on the order of 1 MB. Such a storage requirement for the linear LUTs makes them impractical by contemporary standards at these lower tolerances.

Now we profile the behavior of implementation evaluations for tables of varying tolerance. Some results for $\exp(x)$ and (1) are displayed in Figures 4 and 5, respectively. Each type of implementation was performed independently, i.e., given free

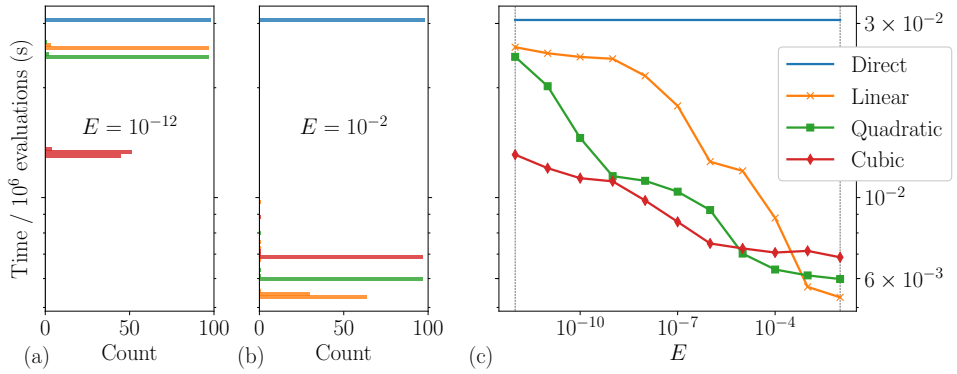

FIG. 4. *Timings for evaluation of* $\exp(x)$ *for varying evaluation error tolerance.* (a) *Histogram of execution times for* 100 *runs of each implementation at tolerance* $E = 10^{-12}$. (b) *Histogram of execution times for* 100 *runs of each implementation at tolerance* $E = 10^{-2}$. (c) *Work-precision diagram for the different implementations (minimum execution time of* 100 *runs). This plot does not reflect the error of the direct evaluation.*

rein over cache usage and access. At small tolerances, where all tables are small, we see replication of the best-case scenario. All tables fit into cache, and so the linear table outperforms the quadratic table, which in turn outperforms the cubic table. As the error is decreased, the larger linear LUTs suffer performance hits most prominently. The decreased performance of the LUT implementations can be attributed to their data no longer fitting entirely in cache, and the repeated evaluation for a random argument requires more and more fetching of data from RAM. With a tolerance of 10^{-12}, the linear table almost performs at its worst case even when no other program data are being manipulated.

4.2. Best and worst cases. The best- and worst-case scenarios for a given LUT implementation evaluation are approximated by using atypically small and atypically large variants of the implementations, respectively. Figure 6 shows the full distribution of runtimes for both scenarios of the linear, quadratic, and cubic LUTs. We note, in particular, that both are shown to be ranked linear, quadratic, and cubic from best

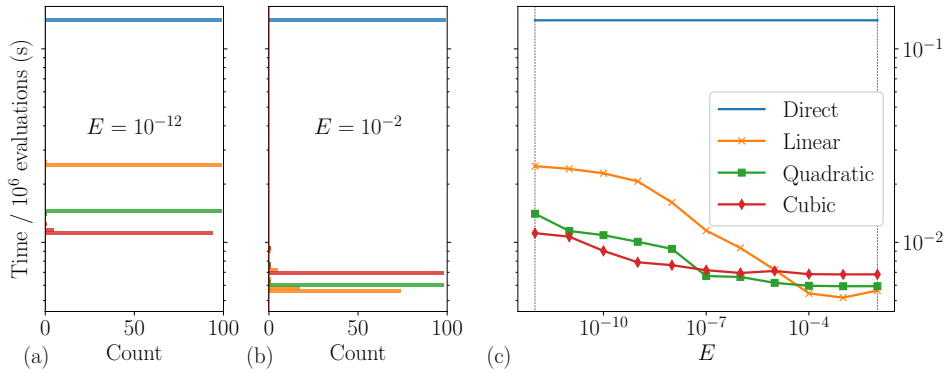

FIG. 5. *Timings for evaluation of* (1) *for varying evaluation error tolerance.* (a) *Histogram of execution times for* 100 *runs of evaluation at* $E = 10^{-12}$. (b) *Histogram of execution times for* 100 *runs of evaluation at* $E = 10^{-2}$. (c) *Work-precision diagram for the different LUTs (minimum execution time of* 100 *runs). This plot does not reflect the actual error of the direct evaluation.*

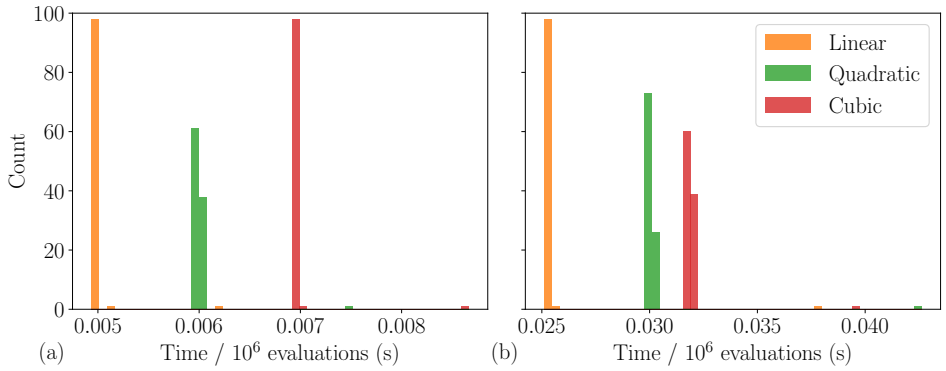

FIG. 6. (a) *Distribution of* 100 *attempts at determining the best-case scenario (table entries always in cache).* (b) *Distribution of* 100 *attempts at determining the worst-case scenario (every evaluation requires a fetch from RAM).*

to worst, in accordance with our postulated rules of thumb. Also of note is the factor of approximately five between the best- and worst-case times.

Because we have already established the importance of data size in the evaluation time of an implementation, we also consider the best-case cubic LUT time (∼0.007 s) in comparison to the worst-case linear LUT time (∼0.025 s), noting that a factor of about 3.5 exists between these. This means under a hypothetical situation where a cubic implementation is guaranteed to remain in cache and a linear implementation guaranteed to not, the performance of the cubic implementation would be 250% faster than that of the linear.

Using the system specified in Table 1, values for the parameters of the simple cache probability model (9) are presented in Table 2, with a graphical representation partitioning the best-performing LUT implementations in a head-to-head format

in Figure 7.

TABLE 2
Approximated parameters for (9) *for the system specified in Table* 1.

i	j	α_{ij}	β_{ij}
1 – Linear	2 – Quadratic	0.845	0.198
1 – Linear	3 – Cubic	0.812	0.267
2 – Quadratic	3 – Cubic	0.962	0.076

The black lines in the subfigures of Figure 7 represent the break-even points where the implementations should perform equally. The regions are shaded according to which implementation is expected to outperform (orange = linear, green = quadratic, red = cubic). For example, in Figure 7(a), holding $P^1_{\text{cache}} = 0$ (every evaluation requires a fetch from RAM) and varying P^2_{cache}, we see that the break-even point lies at $P^2_{\text{cache}} = \beta_{12} = 0.198$, meaning that if the quadratic LUT data can be in cache at least 19.8% of the time, it will outperform a linear table that is never in cache. Holding $P^2_{\text{cache}} = 1$ (its table entries are always in cache when needed) and varying P^1_{cache}, we find that the break-even point is $P^1_{\text{cache}} = (1 - \beta_{12})/\alpha_{12} = 0.949$, meaning that the linear table that is in cache 94.9% of the time will outperform an always-in-cache quadratic table.

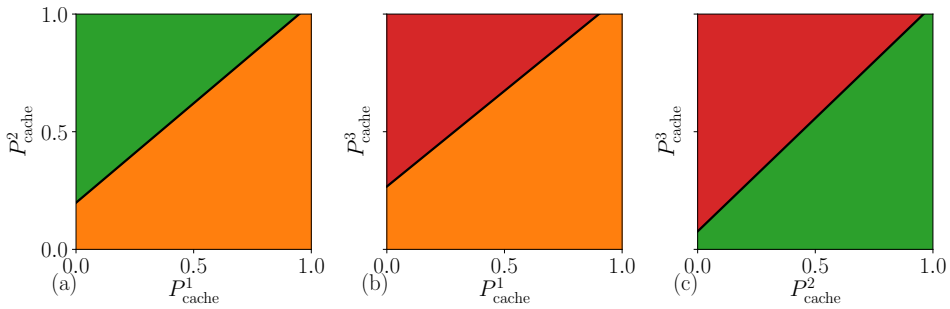

FIG. 7. *Outperformance partitions for j versus i LUT implementations for three combinations of comparisons. Subfigures are colored according to which implementation is expected to outperform in each of the regions (orange = linear, green = quadratic, red = cubic). (Figure in color online.)*

4.3. Function complexity. We now consider how direct evaluation compares to LUTs in terms of evaluation times for functions of varying complexity.

We extracted the default parameters for the linear LUTs from the Chaste source and computed the error (2). Figure 8 shows the errors for the default (linear) LUT implementation in Chaste along with quadratic and cubic LUTs with suitably chosen parameters (as described shortly). The table step size for the Chaste linear LUT is 0.001 mV and is a fixed size for every table used in every cell model in Chaste. This step size seems to be chosen in accordance with the study of Cooper, Spiteri, and Mirams [8], from which we quote, "we have noted that a fine lookup table resolution is often needed to maintain a desired accuracy," referring to the overall accuracy of

a simulation, rather than the accuracy of any given function evaluation. The linear LUTs are defined on the interval $x \in (-250, 550)$ mV.

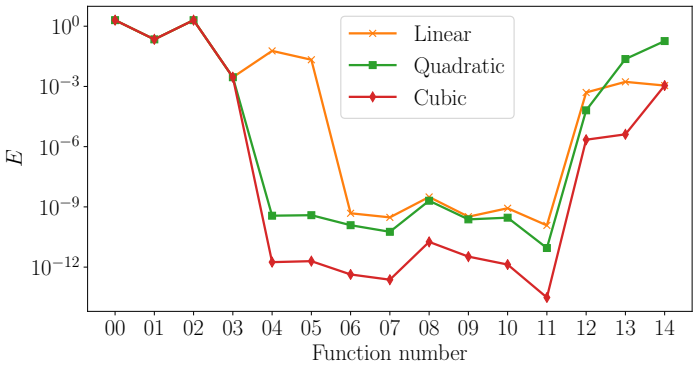

FIG. 8. *Errors for each function identified in the* LR1991 *cell model. The LUT implementations all use the interval* $(-250, 550)$*, and the table step sizes are* 0.001 *mV,* 0.04 *mV, and* 0.1 *mV for the linear (Chaste default), quadratic, and cubic LUTs, respectively.*

The step sizes for the quadratic and cubic tables were chosen such that the error for every function is similar to the error of the default linear LUT. We recall that Chaste's LUT evaluation is performed on the set as a whole, with each function being represented by a table with the same parameters. Through an ad hoc procedure, we find that a quadratic table with step size 0.04 mV and a cubic table with step size 0.1 mV produce function errors similar to those of Chaste's default choice.

We note, in particular, the error of the piecewise functions (00, 01, 02, 03, 14). When using LUTs, the overall accuracy for piecewise functions is limited by the maximum jump discontinuity within the function's domain.

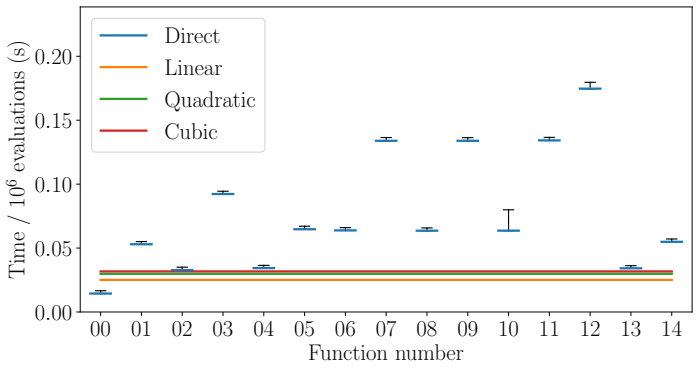

FIG. 9. *Comparison of timings for direct evaluations of the functions in the* LR1991 *set. Noted are the worst-case timings established for the linear, quadratic, and cubic LUT implementations. Direct evaluation times are presented with their median (of* 100 *runs) as the thicker blue lines, with black bars extending to minimum and maximum measured times. (Figure in color online.)*

Timings of direct evaluations for the functions in the LR1991 set and their relation to the worst-case behaviors for the LUT implementations are found in Figure 9. The

visualizations use a thick blue line at the median time of 100 experiments, with black bars extending to the absolute best and worst observed times. The linear, quadratic, and cubic LUT worst-case times are also displayed for reference. The important feature here is that all but one of the functions in the set take longer to evaluate directly than the worst-case for all of the LUT implementations. As can likely be inferred by the lengths of the various code snippets used to define the functions in Appendix B in the supplementary material, the most expensive function of the set is number 12; its direct evaluation takes about six times longer to evaluate than any of the LUT implementations.

We perform the same procedure with all of the functions in the TTP2006Epi set as well, with accuracy results shown in Figure 10. We note here that functions 06–09 are only piecewise continuous. Functions 27 and 29 have singularities in their domain, making the error computation a gross underestimate. Function 34 crosses 0, and so according to (2), results in an error of at best 2. Potential remedies to such scenarios are discussed in section 5.

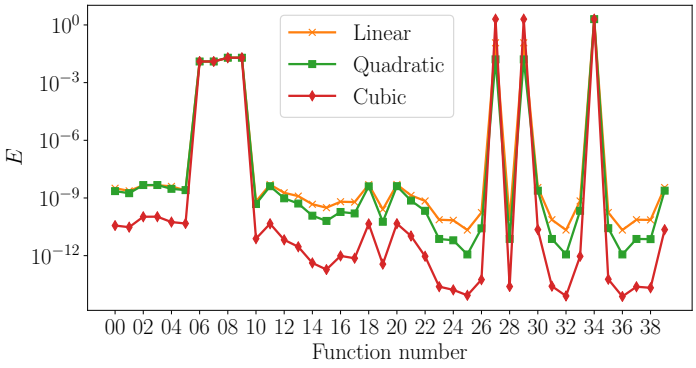

FIG. 10. *Errors for each function identified in the* TTP2006Epi *cell model. The LUT implementations all use the interval* $(-250, 550)$ *mV, and the table step sizes are* 0.001 *mV,* 0.04 *mV, and* 0.1 *mV for the linear (Chaste default), quadratic, and cubic LUT implementations, respectively.*

Timings of direct evaluations for all of the functions in the TTP2006Epi set and their relation to the worst-case behaviors for the LUT implementations are found in Figure 11. In this experiment, *every* function in the set takes longer to evaluate directly than the worst-case behavior of the linear LUT implementation. The latter functions in the set perform comparably to the worst case of any of the LUT implementations, and the most expensive function to evaluate, number 10, takes about five times longer with direct evaluation than the worst-case of any of the LUT implementations.

4.4. Two-dimensional bidomain performance testing. To get an idea of the influence of implementation type for function evaluations in cardiac simulation, we look at three components of the solving process: the ODE solve time, the linear system solve time (KSP, the name of the PETSc [2] linear solver class used in Chaste), and the total CPU execution time. One hundred runs are performed with each of direct evaluation, linear LUT, and cubic LUT implementations for the LR1991 model, with the distributions of each of these components shown in Figure 12.

As expected, the place where LUT implementations make the most difference is

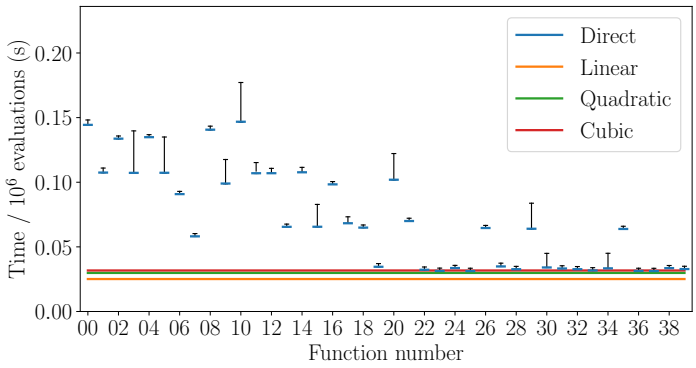

FIG. 11. *Comparison of timings for direct evaluations for the functions in the* TTP2006Epi *set. Noted are the worst-case timings established for the linear, quadratic, and cubic LUT implementations. Direct evaluation times are presented with their median (of* 100 *runs) as the thicker blue lines, with black bars extending to minimum and maximum measured times. (Figure in color online.)*

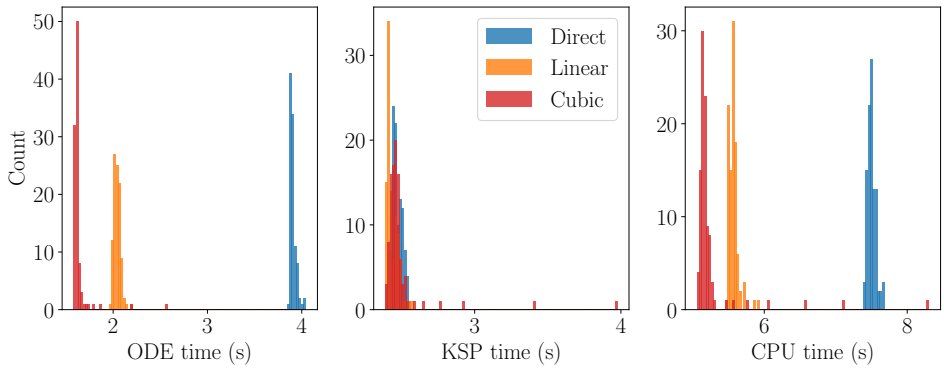

FIG. 12. *Two-dimensional bidomain performance testing of the Luo–Rudy cell model. Visualized are the runtime distributions of different components of the solving procedure, ODE solve time, linear system solve time, and total CPU time, for* 100 *different simulations. Each subfigure shows three cases of implementation for the functions found in Figure* SM4 *in the supplementary material: direct evaluation, linear LUT, and cubic LUT with precomputed coefficients.*

in the ODE solving phase. The ODE solver performs many right-hand side function evaluations. On the other hand, there are no LUT function evaluations for the KSP solver; the only influence of the LUT implementation is in the initialization of the right-hand side of the Krylov iteration, potentially affecting convergence. As expected, there does not seem to be any significant difference in the KSP runtimes. The absolute differences in ODE runtimes are similar to the absolute differences in CPU times because this component dominates the overall computational cost. Comparing the times of the ODE phase for the LR1991 model, we present the speedups in Table 3. For the LR1991 model, moving from direct evaluation to a linear LUT implementation gives significant speedup, which is further increased by moving to a

cubic LUT implementation.

TABLE 3

Speedups of ODE component in Chaste solving, computed from the fastest runtimes observed with each implementation. Speedup is the ratio of runtimes.

LR1991		**TTP2006Epi**	
from Direct to Linear:	1.95	from Direct to Linear:	1.09
from Linear to Cubic:	1.25	from Linear to Cubic:	1.12
from Direct to Cubic:	2.44	from Direct to Cubic:	1.23

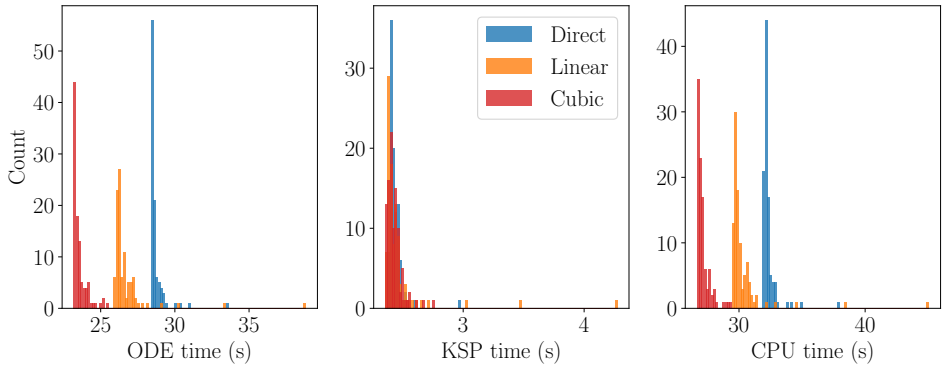

FIG. 13. *Small domain two-dimensional performance testing of the ten Tusscher–Panfilov cell model. Visualized are the distributions of different components of the solving procedure, ODE solve time, linear system solve time, and total CPU time, for* 100 *simulations. Each subfigure shows three cases of implementation for the automatically identified expensive functions: direct evaluation, linear LUT, and cubic LUT with precomputed coefficients.*

The same process is applied to the TTP2006Epi model, with distributions of runtimes in Figure 13 and speedups in Table 3. For the different implementations within the TTP2006Epi model, moving from direct evaluation to linear LUT results in a modest speedup (9%), and going from linear to cubic LUTs produces a slightly more significant speedup (23%).

4.5. Niederer benchmark. Runtime results of the Niederer benchmark are found in Table 4. The relationships between ODE, KSP, and CPU times are qualitatively the same as for the previous problem. The RHS time has now been added because it forms a significant portion of the runtime in this benchmark, whereas it did not for the previous one. RHS is a measurement of time taken to set up the right-hand side of the tissue equation (11b) throughout the simulation. From Table 5, we observe that linear and cubic LUT implementations perform similarly for this problem, having less than a 1% difference in total CPU time and about a 30% improvement over the direct evaluation. Looking at just the solution of the cell model ODEs (11a), where function evaluation speeds have the largest effect, we see a performance improvement of over 86%.

5. Summary and future directions. We designed a C++ library called `FunC` that can be run in a computing environment with user-specified functions to help rank the relative execution speeds of various implementations. `FunC` was designed to help

TABLE 4
Runtime results for the Niederer benchmark problem in seconds. A single run of each was performed. The linear LUT uses a step size of 0.001 *mV; the cubic LUT uses* 0.1 *mV.*

Implementation	ODE time	KSP time	RHS time	CPU time
Direct evaluation	2830.3	372.0	2690.5	5908.8
Linear LUT	1515.4	369.5	2658.0	4558.8
Cubic LUT	1517.7	368.8	2655.8	4562.0

TABLE 5
Speedups of ODE and CPU times for the Niederer benchmark problem.

ODE speedup		**CPU speedup**	
from Direct to Linear:	1.867	from Direct to Linear:	1.296
from Linear to Cubic:	0.998	from Linear to Cubic:	0.999
from Direct to Cubic:	1.864	from Direct to Cubic:	1.295

organize various implementation styles with implementation details being as transparent as possible. The capabilities of `FunC` were demonstrated through (i) showing the large difference in implementation sizes from linear to cubic LUTs, (ii) showing the performance of implementations when *only* function evaluations are required, (iii) approximating the best- and worst-case scenarios for an implementation's performance on a given system, (iv) comparing the direct evaluation speeds of sets of functions with worst-case LUT implementations, and (v) showing the performance of nontrivial numerical simulations when different implementations are chosen for mathematical function evaluation. Regarding this final point, we note that the accuracy to which a LUT is evaluated can substantially affect the accuracy and stability of a numerical solution. Such issues are difficult to characterize explicitly and are highly application-dependent; consequently, we have avoided discussion of this entirely.

We demonstrated that under a scenario where only mathematical function evaluation of a uniform random variable is performed at a chosen accuracy, the size and order of LUT implementations affect their performance. Linear LUT implementations can quickly become too big to fit in cache and hence inefficient, even when function evaluation is the only thing performed in a program. Cubic LUT implementations on the other hand generally require orders of magnitude less memory for their data, and so we only see this inefficiency as the evaluation tolerance is lowered, and it is not as pronounced as with linear LUTs.

Considering the small-scale two-dimensional bidomain simulation using the LR1991 cell model, more significant improvement is seen in changing implementation from direct evaluation to linear LUTs than from linear to cubic LUTs, when evaluation tolerance is held approximately constant. For the larger and more complicated TTP2006Epi cell model, we found that the benefit of going from linear to cubic LUT implementations had a more pronounced effect than changing from direct evaluation to linear LUT implementations, but overall there was much less difference between direct evaluation and the use of cubic LUT implementations than for the LR1991 model, again with evaluation tolerance held approximately constant.

With the larger simulations of the three-dimensional monodomain Niederer benchmark problem, we see that both the linear and cubic LUT implementations could produce a significant increase in the simulation speed. Using the cubic LUT resulted in a negligible ($\lesssim 0.2\%$) loss of performance compared to the linear LUT, but it was still substantially faster ($\approx 30\%$) than the simulation with direct evaluation.

Some limitations of this study, and thus areas for future improvement, are as

follows.

1. Dealing with functions that have rapidly varying first derivatives can benefit from using nonuniform meshes, which are not considered here. Some work on elementary functions is reported in, e.g., [22] and [14], and is often centered around circuit design and field-programmable gate arrays. As demonstrated in this work, software LUT implementations can show significant improvement over direct evaluation as well, and accordingly we expect such nonuniform LUT methods implemented at the software level to be beneficial in some scenarios.
2. Functions with jump discontinuities can be addressed by using some form of composite implementation that is specified for each region of the domain where the function is continuous. A composite implementation could be defined in terms of the LUTs used in this paper, nonuniform LUTs, or even combinations of these and other styles of implementations.
3. This study only considers LUT implementations with polynomial interpolation in the monomial basis with Horner evaluation. Other bases can be used, and in fact representation in the Bernstein basis [15] with evaluation by de Casteljau's algorithm is a good choice if numerical stability and accurate evaluation are both issues [3]. Because it requires more flops for evaluation at a given order, we expect this approach to perform more slowly than the monomial evaluations in the worst case.

The most computationally intensive part of `FunC` is the determination of the numerical error in (2) and thus the creation of implementations that satisfy a tolerance. However, once this is performed for a given implementation type of a specific function at a specific tolerance on a specific domain, it need not be performed again. Another view is that `FunC` can be used simply as a *check* on the implementations being used, giving some confidence that the function evaluations are accurate enough. We foresee `FunC` as being useful in situations similar to what we have done in Chaste, where a database of implementation parameters can be maintained for a given set of functions evaluated to a specified set of tolerances. `FunC` can then simply be run in its timing mode, comparing direct evaluations with worst-case implementation times on a given system before deciding whether or not a direct evaluation should be used.

The effectiveness of using LUTs is generally limited by the number and complexity of the required mathematical function evaluations. In addition, the memory requirements for multivariate functions make LUT evaluation impractical because each independent variable must be discretized. With these constraints in mind, our final thoughts on the use of software LUT implementations in large-scale simulation are as follows:

1. Generalized software LUT implementations can improve the performance of a code if the complexity of the function evaluations is high enough;
2. higher-order LUT implementations can outperform linear LUTs when the total size of a program is small enough relative to available cache; and
3. the use of higher-order LUT implementations allows the incorporation of higher demands on function evaluations for a given implementation size, e.g., lower evaluation tolerances or larger domains of evaluation, without sacrificing much in terms of performance.

It is this final point which is of most importance in practice, allowing for more robust software in general.

REFERENCES

[1] G. E. Alefeld, F. A. Potra, and Y. Shi, *Algorithm* 748*: Enclosing zeros of continuous functions*, ACM Trans. Math. Softw., 21 (1995), pp. 327–344, https://doi.org/10.1145/210089.210111.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page*, http://www.mcs.anl.gov/petsc, 2017.

[3] W. Boehm and A. Müller, *On de Casteljau's algorithm*, Comput. Aided Geom. Design, 16 (1999), pp. 587–605, https://doi.org/10.1016/S0167-8396(99)00023-0.

[4] Boost, *Boost C++ Libraries*, http://www.boost.org/, 2017 (last accessed 2018-02-26).

[5] R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[6] S. A. Buehler, P. Eriksson, and O. Lemke, *Absorption lookup tables in the radiative transfer model ARTS*, J. Quant. Spectrosc. Radiat. Transf., 112 (2011), pp. 1559–1567, https://doi.org/10.1016/j.jqsrt.2011.03.008.

[7] J. Cooper, S. Mckeever, and A. Garny, *On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations*, in Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '06, ACM, New York, 2006, pp. 12–20.

[8] J. Cooper, R. J. Spiteri, and G. R. Mirams, *Cellular cardiac electrophysiology modeling with Chaste and CellML*, Front. Physiol., 5 (2015), 511, https://doi.org/10.3389/fphys.2014.00511.

[9] A. A. Cuellar, C. M. Lloyd, P. F. Nielsen, D. P. Bullivant, D. P. Nickerson, and P. J. Hunter, *An overview of CellML* 1.1*, a biological model description language*, SIMULATION, 79 (2003), pp. 740–747, https://doi.org/10.1177/0037549703040939.

[10] M. A. Dias, D. O. Sales, and F. S. Osorio, *Automatic generation of LUTs for hardware neural networks*, Neurocomputing, 180 (2016), pp. 108–120, https://doi.org/10.1016/j.neucom.2015.07.111.

[11] M. Frigo and S. G. Johnson, *The design and implementation of FFTW*3, Proc. IEEE, 93 (2005), pp. 216–231.

[12] K. R. Green, T. Bohn, and R. J. Spiteri, `FunC` *source code*, https://github.com/uofs-simlab/func, 2018.

[13] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002, https://doi.org/10.1137/1.9780898718027.

[14] S. F. Hsiao, C. S. Wen, and P. H. Wu, *Compression of lookup table for piecewise polynomial function evaluation*, in 2014 17th Euromicro Conference on Digital System Design, IEEE, 2014, pp. 279–284, https://doi.org/10.1109/DSD.2014.53.

[15] G. G. Lorentz, *Bernstein Polynomials*, Mathematical Expositions, no. 8, University of Toronto Press, Toronto, 1953.

[16] C. Luo and Y. Rudy, *A model of ventricular cardiac action potential*, Circ. Res., 68 (1991), pp. 1501–1526.

[17] M. E. Marsh, S. T. Ziaratgahi, and R. J. Spiteri, *The secrets to the success of the Rush–Larsen method and its generalizations*, IEEE Trans. Biomed. Eng., 59 (2012), pp. 2506–2515.

[18] E. J. Mlawer, S. J. Taubman, P. D. Brown, M. J. Iacono, and S. A. Clough, *Radiative transfer for inhomogeneous atmospheres: RRTM, a validated correlated-k model for the longwave*, J. Geophys. Res-Atmos., 102 (1997), pp. 16663–16682, https://doi.org/10.1029/97JD00237.

[19] S. A. Niederer, E. Kerfoot, A. P. Benson, M. O. Bernabeu, O. Bernus, C. Bradley, E. M. Cherry, R. Clayton, F. H. Fenton, A. Garny, E. Heidenreich, S. Land, M. Maleckar, P. Pathmanathan, G. Plank, J. F. Rodríguez, I. Roy, F. B. Sachse, G. Seemann, O. Skavhaug, and N. P. Smith, *Verification of cardiac tissue electrophysiology simulators using an N-version benchmark*, Philos. Trans. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci., 369 (2011), pp. 4331–4351, https://doi.org/10.1098/rsta.2011.0139.

[20] M. Pharr and R. Fernando, *GPU Gems* 2*: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA, 2005.

[21] PyCml, *CellML Tools in Python*, https://chaste.cs.ox.ac.uk/cellml/, 2011.

[22] T. Sasao, S. Nagayama, and J. T. Butler, *Numerical function generators using LUT cascades*, IEEE Trans. Comput., 56 (2007), pp. 826–838, https://doi.org/10.1109/TC.2007.1033.

[23] R. J. Spiteri and S. Torabi Ziaratgahi, *Operator splitting for the bidomain model revisited*, J. Comput. Appl. Math., 296 (2016), pp. 550–563.

[24] J. Sundnes, G. T. Lines, and X. Cai, *Computing the Electrical Activity in the Heart*, Springer-Verlag, Berlin, 2006.

[25] K. H. W. J. ten Tusscher and A. V. Panfilov, *Alternans and spiral breakup in a human ventricular tissue model*, Am. J. Physiol. Heart Circ. Physiol., 291 (2006), pp. H1088–H1100, https://doi.org/10.1152/ajpheart.00109.2006.

[26] L. Tornqvist, P. Vartia, and Y. O. Vartia, *How should relative changes be measured?*, Amer. Statist., 39 (1985), pp. 43–46, https://www.jstor.org/stable/2683905.

[27] L. Tung, *A Bi-domain Model for Describing Ischemic Myocardial D-C Potentials*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1978.

[28] University of Oxford, Department of Computer Science, *Chaste — Cancer, Heart and Soft Tissue Environment*, 2014, http://www.cs.ox.ac.uk/chaste/.

[29] C. Wilcox, M. M. Strout, and J. M. Bieman, *Mesa: Automatic generation of lookup table optimizations*, in Proceedings of the 4th International Workshop on Multicore Software Engineering, IWMSE '11, 2011, ACM, New York, pp. 1–8, https://doi.org/10.1145/1984693.1984694.

[30] C. Wilcox, M. M. Strout, and J. M. Bieman, *Tool support for software lookup table optimization*, Scientific Programming, 19 (2011), pp. 213–229.

[31] A. Yamamoto, Y. Kitamura, and Y. Yamane, *Computational efficiencies of approximated exponential functions for transport calculations of the characteristics method*, Ann. Nucl. Energy, 31 (2004), pp. 1027–1037, https://doi.org/10.1016/j.anucene.2004.01.003.

[32] Y. Zhang, L. Deng, P. Yedlapalli, S. P. Muralidhara, H. Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and X. Sun, *A special-purpose compiler for look-up table and code generation for function evaluation*, in Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, Leuven, Belgium, 2010, pp. 1130–1135, http://dl.acm.org/citation.cfm?id=1870926.1871200.