`StenSeal` – A General Software Library for High-Performance Finite-Difference Computations

Jonatan Werpers Karl Ljungkvist Ylva Rydin

May 2, 2017

Abstract

1 Introduction

Finite difference methods are commonly used for spatial discretization of partial differential equations (PDEs). They provide a simple and potentially highly efficient means of approximating solutions of both time dependent and time independent problems. If designed and implemented correctly they can also be very robust and make efficient use of computer resources. Moreover, they can also provide high order schemes without putting severe constraints on time steps when integrating in time.

Traditionally, finite-difference methods have been designed, analyzed and implemented on a problem by problem basis. This has resulted in specific codes for specific equations and specific boundary conditions. While this shortens implementation time for the individual developer in the short term there has been and will be a lot of duplicated effort designing and optimizing software interfaces. A potential explanation for this is that for a long time there were no tools for easily tackling new problems and equations. As a result there seems to be a lack of a general high-performance software library for large-scale finite-difference computations.

During the last 20 years, research has shown that the *summation by parts* framework (SBP) for finite-difference methods combined with the weak imposition of boundary conditions using the *simultaneous approximation term* technique (SAT) can provide such tools. The framework provides mathematical tools that allow deriving accurate and stable finite-difference schemes for many combinations of PDE and boundary conditions. A few examples can be found in [11, 4].

While general frameworks such as Matlab or SciPy have successfully been used in many cases to prove the applicability of the schemes, they lack the ability to scale to large problems. Once large-scale simulations are to be performed, where performance is vital, implementations in low-level languages

such as C++ have typically been done on a specialized case-to-case basis. Due to this, new numerical schemes and methods often get stuck at the prototyping level and never make it into the large-scale simulations. Similarly, access to the methods becomes restricted for the rest of the community, hurting the advance of the research field at large.

In contrast to this, in the finite-element community, there are numerous general low-level open-source libraries where programmers can write high-performance large-scale simulations without too much effort. One such FEM frameworks is `deal.II`, which is general enough to be used in a wide range of applications, while at the same time offering excellent performance up to hundreds of thousands of cores [3]. Another important example is the the FEniCS project, which includes a domain-specific language reducing the application-specific code to just a few lines [2].

To our knowledge, no similar effort for finite-difference computations exists. Based on this, the motivation for this project is to explore the needs and constraints that are necessary for such a framework, consider the design choices for the software interface, and to provide a pilot implementation capable of simplifying application programming while maintaining performance and flexibility.

The rest of this paper is structured as follows, in Section 2 we provide a brief introduction to finite-difference methods and the SBP-SAT framework. In Section 3 we discuss strategies to arrive at a high performance implementation of a finite-difference scheme and in Section 4 we discuss the requirements and structure of a a general software interface for finite-difference methods allowing both flexibility and optimization. Finally in Section 5 we provide some numerical results from our implementation.

2 Finite Differences

To obtain an approximate solution to an initial-boundary-value problem with any discretization method, the method must be consistent and stable. If these two properties are fulfilled the approximate solution will converge to the true solution [6]. With methods based on finite-difference methods it is trivial to derive arbitrarily high order of accuracy in the approximations of derivatives. However, it may be a complicated matter to achieve and prove stability for general finite-difference methods. The SBP framework ensure that the discretization of the derivatives preserve certain properties of the continuous derivatives. This allows a stability analysis based on standard methods for establishing well-posedness of the continuous equations. With this you are given a clear path from a new PDE to a stable finite-difference scheme. We shall now go into more detail and explain the basics of the framework.

To develop a stable discretization of a PDE that converges to the true

solution is a difficult task requiring mathematical analysis. Before making an attempt to numerically solve a PDE we have to ensure that the PDE is formulated in a way allowing stable discretization. In other words, we have to show that our PDE is *well-posed*. A common method used to perform this analysis is the energy method [5]. This method uses the integration by parts formula which can be expressed in terms of the standard L^2 inner product as

$$(u, v_x) = uv|_0^1 - (u_x, v), \quad u, v \in L^2[0,\ 1].$$

To introduce finite-difference methods, the advection equation

$$\begin{array}{lll} u_t + u_x = 0, & 0 \leq x \leq 1, & t \geq 0, \\ u = g_l, & x = 0, & t \geq 0, \\ u = f_1, & 0 \leq x \leq 1, & t = 0. \end{array} \tag{1}$$

will be used as an example. This equation is *well-posed* if the initial and boundary conditions are chosen correctly [10]. Let the domain $x \in [0, 1]$ be discretized using N+1 equidistant grid points,

$$x_i = ih, \quad i = 0, 1, \ldots N, \quad h = \frac{1}{N},$$

and let the vector $u = [u_0, u_1, \ldots, u_N]^T$ represent the discrete solution vector in each time step. The spatial derivative is approximated by the finite-difference operator D_1 as

$$u_x = -D_1 u + \mathcal{O}(h^p),$$

where p is the order of accuracy of the difference operator. To ensure that the numerical solution is converging to the true solution a stability analysis similar to the continuous analysis is performed. This analysis is simple if the finite-difference operator has a property that mimics integration by parts called the summation-by-parts (SBP) property.

Consider a positive definite diagonal matrix H with the weights of a quadrature rule on the diagonal which introduces the inner product $(u, v)_H = u^T H v$. To access the boundary elements, we introduce the vectors

$$e_l = [1, 0, \ldots, 0]^T \quad \text{and} \quad e_r = [0, \ldots, 0, 1]^T.$$

A finite-difference operator D_1 approximating $\partial/\partial x$ has the summation by parts property if

$$(u, D_1 v)_H = u^T e_r e_r^T v - u^T e_l e_l^T v - (D_1 u, v)_H.$$

By employing an operator with the SBP-property we can make a stable semi-discrete approximation of (1) as

$$u_t = D_1 u + \tau(e_l^T u - g_l),$$

where τ is a penalty parameter chosen to enforce the boundary conditions [10].

In the present study, we focus on a more difficult problem, namely the 1D wave equation with a variable coefficient,

$$\begin{array}{lll} u_{tt} = (cu_x)_x, & 0 \leq x \leq 1, & t \geq 0, \\ u = g_l, & x = 0, & t \geq 0, \\ u = g_r, & x = 1, & t \geq 0, \\ u = f_1, u_t = f_2, & 0 \leq x \leq 1, & t = 0. \end{array} \tag{2}$$

as a benchmark problem. While simple, this test problem gives rise to many of the difficulties that occur when solving more complex PDEs, such as treatment of variable coefficients. When solving 2 and 3-dimensional problems with finite-difference methods all domains have to be blocked mapped to square domains. This mapping gives rise to variable coefficients in the PDE. Hence, it is crucial for a finite-difference library to handle problems with variable coefficients efficiently.

There are several techniques to solve the wave equation using finite-difference methods. In this work we have focused on two different approaches that are both highly accurate but give rise to schemes that require different treatment in a matrix free setting. The first approach is the narrow second derivative operators introduced by Mattsson in [8]. With these operators the following stable finite-difference approximation

$$u_{tt} = D_2^{(\bar{c})} u + \tau_l (e_l^T u - g_l) + \tau_r (e_r^T u - g_r), \quad c > 0,$$

is obtained, where τ_l and τ_r are penalty parameters used to enforce the boundary conditions. The operator $D_2^{(\bar{c})}$ has the sparsity pattern presented in Figure 1. This operator can be divided into two parts, namely the boundary stencil which is marked by red dots in the figure, and the inner stencil marked by the blue dots. Both the boundary stencils and the inner stencil are weighted by the coefficient $c(x)$. However, in a setting where $c(x)$ is constant the inner stencil is the same for each point and the right boundary stencil is equal to the left boundary stencil mirrored over the off-diagonal.

The second approach to approximate the wave equation (2) is the upwind operators introduced by Mattson in [9] which yield the following discretization

$$u_{tt} = D_+ \bar{c} D_- u + \tau_l (e_l^T u - g_l) + \tau_r (e_r^T u - g_r), \quad c > 0. \tag{3}$$

The upwind operators have the sparsity patterns presented in Figure 2. Note that here, in contrast to the compact operators, the left and right boundary blocks are not the same. The differential stencils are the same for both upwind operators, but D_- is applied from the left while D_+ is applied from the right. Therefore inner stencils in D_+ marked with blue dots is the same as the inner stencil of D_- but flipped and shifted to the right with a sign

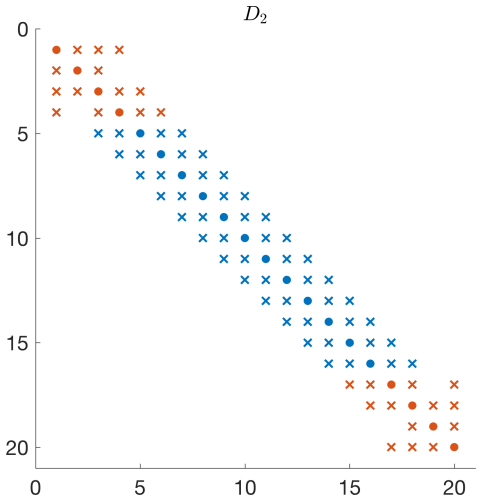

Figure 1: Sparsity pattern of the fourth order narrow operator $D_2^{(\bar{c})}$ with $N = 20$.

change. Further the boundary blocks in D_+ are the same as boundary blocks in D_- but with opposite signs, flipped and mirrored over the off-diagonal.

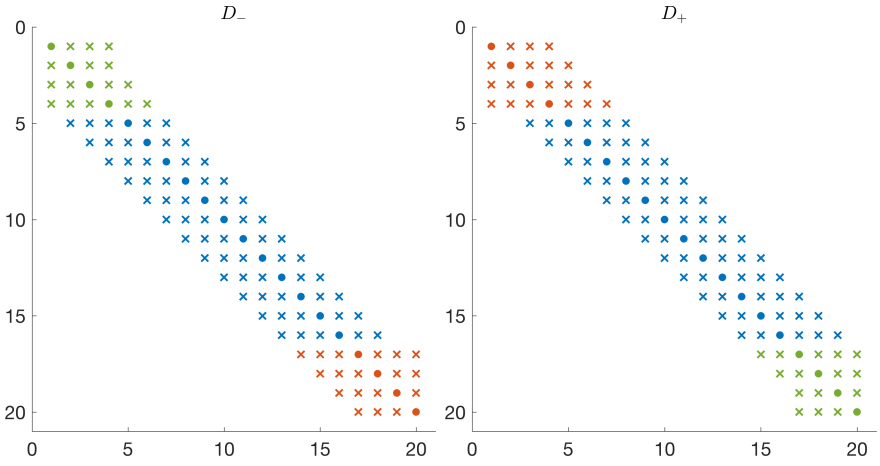

Figure 2: Sparsity patterns of fifth order Upwind operators with $N = 20$.

3 Performance

When optimizing a program code for efficiency, both the properties of the algorithm and the hardware have to be taken into account. Firstly, algorithmic optimizations must be considered, such as reducing the operation count, or using more efficient data structures and memory access patterns. Secondly, the algorithm must be expressed such that the compiler can pro-

duce machine-code instructions which fully utilize the available hardware resources, including memory bandwidth, cache memory, and vector units.

Arguably the most important characteristic of an algorithm is its *computational intensity*, which is defined as the number of floating point operations (flops) per data accessed. Depending on the corresponding balance of the hardware, the algorithm will either be limited by the available bandwidth or computational power.

Due to fundamental power-related limitations in the hardware, the ratio of flops per data of modern processors has increased steadily during recent years, and is now for both multi-core CPUs and GPUs as high as 40-80 flops per double-precision access (see e.g. Section 3.4 in [7]).

For an m-point stencil evaluated for all N points, the number of operations will be $(2m-1)N$ while the number of variables that need to be accessed is between $2N$ and $(m+2)N$ doubles, depending on if a constant or more complicated variable coefficient is used. This yields a computational intensity between about 2 and m, with n never reaching more than about 25 in the most extreme cases. In practice, ideal memory bandwidth utilization is almost impossible to achieve, so the effective intensity will likely be even lower. It is thus clear that a finite-difference computation will always be bandwidth-bound.

Naturally, the most direct way to better performance for a bandwidth-bound algorithm is to reduce the amount of memory accessed. Since the computational units of the processor are mostly idle, it is usually worth trading memory accesses for additional computations, for instance by recomputing values rather than reading them from memory. Of course, computations are not completely for free, and the right balance between computations and bandwidth is exactly flop-per-byte ratio of the computer system under consideration.

In the following, we explore two ways to reduce the memory footprint of the algorithm. Then a couple of techniques for generation of more efficient machine code are discussed.

3.1 A Matrix-free Approach

The first and single most important observation we can make is that using a matrix-based approach, where stencil weights are stored explicitly row for row, is going to be very inefficient. Not only is the matrix extremely sparse in the traditional sense, i.e. having mostly zeros, but due to the structured mesh, each row will be of very similar structure, and in many cases have the exact same values. Because of this, we instead implement the operators as opaque matrix-like objects, that for all purposes behave as matrices, but internally makes use of this special structure that all finite-difference operators share.

This has the greatest impact for constant coefficients, appearing e.g. for

Cartesian meshes, where every single row will have identical structure and weights. In that case, it is sufficient to store these once, reducing the storage space for the operator from mN doubles to m doubles for a stencil of width m and a vector of size N, if the boundary blocks are neglected.

For coefficients defined by a function, such as appears for general non-Cartesian meshes, the weights in the resulting operator are in general distinct, but by recomputing them on the fly instead of precomputing and storing them explicitly, we can reduce the data footprint of the operator dramatically. How much data we need to access depends on the specific form of the function. For instance, for the case of a mesh defined by transfinite interpolation, this will be equal to the positions of the boundary points, which is a great improvement due to the lower dimensionality of the boundary.

Finally, if the coefficient is completely general, e.g. for a mesh defined by a vector of node-point positions, it could still be worth not precomputing the rows. In d dimensions, the node-position vector will amount to d values per node, which is less than the $1 + (m - 1)d$ weights per node of the combined resulting stencil.

3.2 Loop Blocking

The second important algorithmic consideration for finite-difference computations is to loop over solution vectors in manner which minimizes unnecessary memory accesses and efficiently utilizes the caches.

For instance, for the Upwind class of operators, the second derivative is calculated by applying two consecutive first-derivative operators. If this is performed in the naive with two full loops and a full-sized temporary vector, two additional accesses to main memory are introduced, both of size proportional to the number of points N. Clearly, there is a strict locality in which each individual intermediate result is only needed for very few final values. To exploit this, we suggest a loop-blocking approach in which only B intermediate results are stored simultaneously. Here, B should be small enough to fit in the cache, but large enough to not cause unnecessarily many recomputations of overlapping values.

Similarly, computing the Laplacian in higher dimensions will comprise evaluating several stencils on the same input vector and combining the resulting contributions. By fusing these loops, values that are loaded into the cache can be reused to avoid additional accesses to main memory.

Another issue in higher dimensions than one is that there will be a lot of strided memory accesses when evaluating stencils along other dimensions than the first. To avoid this resulting in intensive cache thrashing, the fastest running loop direction can be blocked to match the cache line size. Of course, when different stencil loops are fused, the situation becomes more complicated, but using multi-dimensional blocking there are still considerable

improvements to be made over the naive approach.

3.3 Generating Efficient Machine Code

Modern CPUs are able to achieve a very high single-core performance by utilizing long instruction pipelines, wide vector units, deep cache hierarchies, hardware memory- and instruction prefetching, and advanced branch prediction. In order to be efficiently utilized, the processor needs to be served with instructions with minimal dependencies, predictable memory access patterns, and vectorization. Fortunately, modern compilers can achieve most of these things without manual programming, as long as the relevant information is expressed in the implementation.

The single most important information that can be provided to the compiler, is that the stencils are of fixed compile-time constant width. This lets the compiler unroll these loops, giving a large number of predictable operations for prefetching, pipelining, and vectorization. Similarly, if the offsets or even the stencil weights can be made compile-time known, these can be inlined by the compiler resulting in predictable memory accesses, and in some cases compiled into the instructions themselves. Furthermore, compile-time known offsets combined with unrolling of the main outer loop over vector components, can be exploited to infer reuse of data between loop iterations. By using the techniques in Section 4, we can make sure that offsets and weights are compile-time known in the cases where it is possible.

4 Software Interface

The challenge when designing an interface for high-performance finite-difference computations to find the right level of abstraction, to both offer the user great flexibility with respect to the type of operators and applications that can be implemented, and at the same time guarantee that all the optimization in Section 3 can be made.

In addition, whenever possible, the interface should not force the user to care about unnecessary details which can be inferred automatically by the library. For instance, since the generalization from one to higher dimensions is independent of the operator, we try to hide dimensionality as much as possible in the interface. Similarly, the treatment of complicated mapped geometries, resulting in different types of variable coefficients, is also something that can be hidden such that the user code is identical except for the definition of the geometry.

In C++, the traditional way to achieve compile-time evaluation of expressions is through template meta-programming, but this often comes at the price of greatly reduced readability of the source code. Since the C++11 revision, and even more so since the C++14 revision, it is now possible to use the `constexpr` keyword to declare functions and variables that can be

evaluated at compile time, which is a great simplification over template metaprogramming. In `StenSeal`, we exploit this together with operator overloading to create a simple expression system allowing the user to define stencils using symbolic expressions. For example, the code in Listing 1 defines a centered three-point second-derivative stencil. The compiler will be able to evaluate this code at compile time, and produce an object with a member function for applying the stencil where all possible quantities are compile-time constants, including loop bound, index offsets, and even weights.

Listing 1: Creation centered three-point stencil

```
const stenseal::Symbol u;
constexpr Stencil<3> stencil = 1.0*u[-1] + (-2.0)*u[0] + u[1];
```

Additionally, using `constexpr` we are able to apply a stencil at compile time, which is useful for the compact operators where the actual stencil can be generated at compile-time for geometries where the variable coefficient due to the mapping is compile-time known, such as for Cartesian grids.

4.1 Abstractions

As mentioned above, the `StenSeal` library separates abstract finite difference stencils from the geometry, size, and dimension of the computational grid to which they are applied.

The domain is represented using one of two different template classes; `CartesianGeometry` for domains which can be directly discretized by a Cartesian mesh, and `GeneralGeometry`, for domains that are discretized through a mapping from a Cartesian mesh to an arbitrary set of node point locations. Information about the mapping is obtained through the template class `Metric`, which is parametrized with the geometry type. For instance, there is a function `inverse_jacobian` in `Metric`, which returns a the inverse Jacobian of the mapping, in the form of a coefficient object with a function `get` returning the value at a given grid point index. The exact type of the coefficient depends on the type of the geometry. For `CartesianGeometry`, the mapping is the identity mapping, in which case the coefficient is of the class `OneCoefficient`, which does not store any data and simply returns the value one for any index. For `GeneralGeometry`, we instead use the class `VectorCoefficient`, which internally stores a vector with the values at each grid point. In addition to the `get` function, for getting the value at single grid point index, the coefficient classes also have functions for getting a whole set of consecutive values in the form of a `std::array`, both from the interior and on the left and right boundaries.

With this interface, the library can easily be extended with new classes for other types of geometries, without modification to other classes in the

library. For instance, we plan to add a class for the case when the geometry is defined by a transfinite interpolation of points on the boundary.

The stencil operators themselves are based on abstractions on three levels. On the lowest level, we have abstractions for stencil operations, such as the class template `Stencil`, which represents a single one-dimensional stencil. Listing 1 shows how a `Stencil` can be conveniently constructed using symbolic expressions. Furthermore, the class template `StencilTensor` is a higher-dimensional block of `Stencil`s used to represent boundary stencil blocks with several rows. `StencilTensor` is also used when stencil weights are themselves formed using a stencil, e.g. based on a metric coefficient. `Stencil` and `StencilTensor` have functions for applying the stencil both at a given position in a discrete solution vector, and to a fixed `std::array` with length matching the width of the stencil. Moreover, there are similar functions for a left-right flipped application of the stencil, which is useful for when applying a left boundary block on the right side. To the simplify the notation, we provide the convenience `typedef`s for the most common dimensionality of `StencilTensor`, namely `StencilTensor2D` and `StencilTensor3D` for 2D and 3D, respectively. These stencils would also provide the basis for an implementation of a SAT boundary condition treatment.

On the second level, we have abstractions for one dimensional SBP operators, which are comprised of a single `Stencil` in the interior and left and right boundary block `StencilTensor`s. There are three template classes representing different variants of SBP operator types. `SymmetricSBP` is the most restrictive case, where the same stencil is applied on the right and left boundaries. Note that here, "symmetric" refers to the structure of the difference stencils, which are symmetric under left-right flip and negation, and not the symmetry of the matrix representation. An example of a difference operator of the `SymmetricSBP` type is the compact D_1 operator discussed in Section 2.

For SBP operators where there is no such symmetry, `AsymmetricSBP` is used instead, which expects distinct `StencilTensor`s for the left and right boundary blocks, in addition to the `Stencil` used in the interior. The upwind D_+ and D_- operators discussed in Section 2 are examples of `AsymmetricSBP` operators.

Finally, for the compact setting where second derivative $D_2^{(\bar{c})}$ operator has interior and boundary stencils which are themselves formed based on stencils applied to a variable coefficient $\bar{c}$, the class template `VariableSymmetricSBP` is used. Just like for the `SymmetricSBP`, "symmetric" again refers to the fact that the coefficient stencils are identical for left and right boundaries, meaning that the resulting SBP operator is symmetric too for domain-symmetric coefficients such as a constant coefficient.

In addition to the SBP differential operators, there is also a class template `Quadrature` representing a quadrature integration formula with the same

block structure as the SBP operators, but with only one weight per row. Just like `SymmetricSBP`, the blocks for the left and right boundaries are equal.

All of the SBP classes and `Quadrature` have an `apply` function which applies them to a whole discrete function vector writing the output to another corresponding vector. However, for `VariableSymmetricSBP`, the `apply` function also expects the coefficient $\hat{c}$ which is of one of the coefficient types mentioned above.

Finally, on the highest level we have class templates for concrete differential operators for actual meshes with known dimension, geometry, and size. The class template `UpwindLaplace` for the Laplace operator of the upwind kind expects template parameters for the dimensionality, the type of the underlying first-derivative SBP operator, and the type of the geometry. The class template `CompactLaplace` for the Laplacian of the compact type expects template parameters for the dimensionality, the type of the underlying compact second- and first-derivative SBP operators, and the type of the geometry.

Both the upwind and compact Laplace classes have a function `apply` which applies them to a full sized grid function with size corresponding to the underlying geometry. These classes also include a function `matrix` which initializes a `dealii::SparseMatrix` with the corresponding matrix representation, which is useful for our performance comparisons.

Listing 2: Creation and application of a finite-difference operator

```
const int dim = 1;
std::array<unsigned int,dim> n_nodes{ 100 };
typedef stenseal::CartesianGeometry<dim> GeometryType;
Geometry geometry(n_nodes);  // unit interval by default

const stenseal::Symbol u;

constexpr stenseal::Stencil<3>
  interior((0.5)*u[-2] + (-2.0)*u[-1] + (1.5)*u[0]);
constexpr stenseal::StencilTensor2D<2,2>
  left_block((-1.0)*u[0] + 1.0*u[1],
             (-1.0)*u[-1] + 1.0*u[0]);
constexpr stenseal::StencilTensor2D<2,4>
  right_block(0.4*u[-2] + (-1.6)*u[-1] + 1.0*u[0] + 0.2*u[1],
              0.0*u[-3] + 2.0*u[-2] + (-5.0)*u[-1] +3.0*u[0]);

typedef stenseal::AsymmetricSBP<3,2,2,4,2> OperatorType;
constexpr OperatorType Dm (interior, left_block, right_block);

stenseal::UpwindLaplace<dim,OperatorType,GeometryType> op(Dm,geometry);

dealii::Vector<double> u(geometry.get_n_nodes_total()),
dealii::Vector<double> d2u(geometry.get_n_nodes_total());

op.apply(d2u,u); // apply Laplace on discrete function
```

Listing 2 shows an example of how all the parts come together to form a full operator and apply it to differentiate a discrete function. The code for `StenSeal` is available on GitHub [1].

5 Numerical experiments

To test the performance of the difference operator implementations while minimizing the influence of other factors we apply the 1D Laplace operator to a random vector 1000 times. This is done for both upwind and narrow SBP operators of order 2, 4 and 6, for different vector sizes N, in one setting with a constant coefficient and one with a variable coefficient given in a vector. For reference, our matrix-free implementation is compared to sparse matrix implementation utilizing `dealii::SparseMatrix`, see Figure 3.

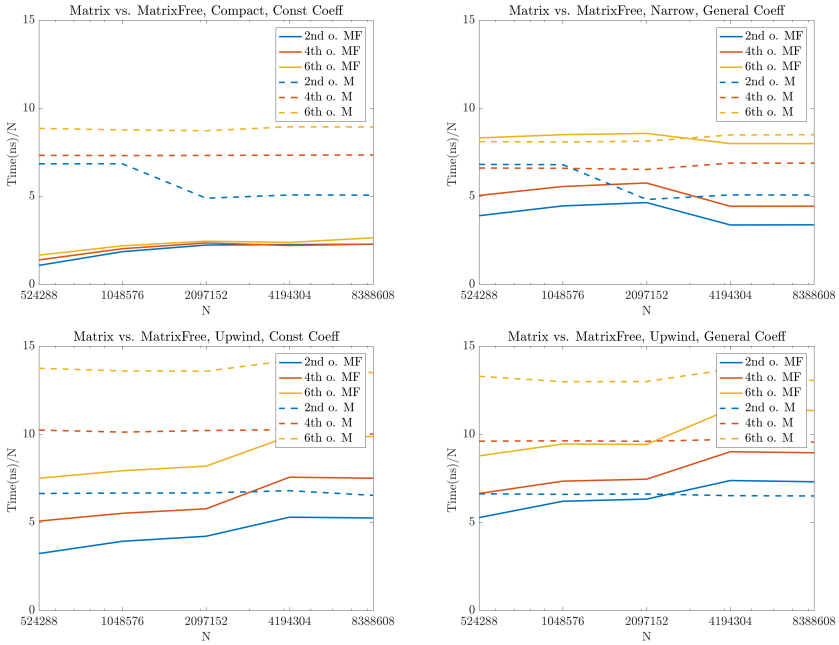

Figure 3: Results from performance test comparing the matrix-free (MF) and the matrix (M) implementation.

As expected, the speed up for the matrix-free approach is significant in the constant coefficient settings. The largest gain is seen with the narrow operators where all matrix-free implementations are about twice as fast as the 2-nd order matrix implementation. The speed up is not as large in the upwind case since two operators D_+ and D_- are applied one at a time compared to the narrow case where only one operator is applied. To reduce this performance loss for the upwind operators the loop blocking approach suggested in Section 3 should be used. In the variable coefficient case an

extra vector has to be accessed from the main memory which reduces the performance of the matrix-free implementation. Nevertheless, a small gain compared to the matrix implementation is seen in a majority of the tests.

6 Conclusion

We have designed a software interface for general higher order finite-difference methods. The interface is flexible enough to accommodate a large variety of applications while still providing abstractions strict enough to allow the library to be efficiently implemented.

The benchmarks in Section 5 show that the library achieves high performance when compared to the main competing method for general finite difference methods. Due to the memory characteristics of the computations this difference will grow as we move from 1D to 2D and 3D.

The library includes abstractions both on a fairly low level, and on a relatively high level. The low-level abstractions allow detailed customization by the programmer and also constitute important building blocks for higher-level abstractions, which are more suitable for manipulation by the application programmer.

One area to focus on in future work is making the library practical for solving PDEs. This includes refining and extending the current abstractions to handle boundary conditions using the SAT technique, as well as implementing application of operators in 2D and 3D. Another area to focus on is the performance of the implementation. While the interface is designed to allow performance optimizations some important concepts like for example loop-blocking have not been exploited.

References

[1] `StenSeal` Code Reepository. `http://github.com/kalj/stenseal`.

[2] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[3] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 8.5. *submitted*, 2017.

[4] M. H. Carpenter, D. Gottlieb, and S. S. Abarbanel. Time-Stable Boundary Conditions for Finite-Difference Schemes Solving Hyperbolic Systems: Methodology and Application to High-Order Compact Schemes. *Journal of Computational Physics*, 111(2):220–236, 1994.

[5] B. Gustafsson, H.-O. Kreiss, and J. Oliger. *Time dependent problems and difference methods*. John Wiley & Sons, Inc., 1995.

[6] P. D. Lax and R. D. Richtmyer. Survey of the stability of linear finite difference equations. *Communications on Pure and Applied Mathematics*, 9(2):267–293, 1956.

[7] K. Ljungkvist. Techniques for Finite Element Methods on Modern Processors, Jan 2015. IT licentiate theses / Uppsala University, Department of Information Technology.

[8] K. Mattsson. Summation by parts operators for finite difference approximations of second-derivatives with variable coefficients. *Journal of Scientific Computing*, 51:650–682, 2012.

[9] K. Mattsson. Diagonal-norm upwind sbp operators. *Journal of Computational Physics*, 355:283–310, 2017.

[10] M. Svärd and J. Nordström. Review of summation-by-parts-operators schemes for initial-boundary-value problems. *Journal of Computational Physics*, 268(0):17 – 38, 2014.

[11] K. Virta and K. Mattsson. Acoustic Wave Propagation in Complicated Geometries and Heterogeneous Media. *Journal of Scientific Computing*, 61(1):90–118, 2014.