LinearOperator — a generic, high-level expression syntax for linear algebra

Matthias Maier*, Mauro Bardelloni†, Luca Heltai†

We introduce an *expression syntax* for the evaluation of matrix-matrix, matrix-vector and vector-vector operations. The implementation is similar to the well-known general concept of *expression templates* as used, for example, in the C++ linear-algebra libraries `Eigen` and `Blaze`. The novelty of the approach that is discussed here lies in the use of new C++11 features like *lambda expressions* and `std::function` objects that avoids the majority of the implementational complexity that usually comes with a pure template solution.

A concrete implementation of the expression syntax has been developed within the framework of the finite-element library `deal.II`, but it is fairly generic: the `LinearOperator` implementation only requires a minimal vector and matrix interface, that all of `deal.II`'s concrete vector and matrix types adhere to. This makes the interface fully transparent with respect to the concrete implementation, in particular to the storage strategy (full matrix, sparse structure), and memory strategy (local, shared, distributed).

The paper concludes with a number of performance comparisons and examples that demonstrate that the framework results in efficient, short and concise code. The performance comparisons show that the overhead introduced by `std::function` objects is negligible for moderately sized matrices, even when compared to native expression-template implementations.

1 Introduction

Expression templates [4, 10] are a well known optimization technique to avoid the creation of large, temporary objects in arithmetic expressions. This is especially important for matrix-matrix, matrix-vector and vector-vector operations that frequently occur in computational linear algebra. With matrix and vector objects that easily go into the gigabytes of memory requirements, temporaries have to be avoided as much as possible. Nevertheless, an intuitive syntax for working with linear algebra objects is desirable.

*School of Mathematics, University of Minnesota, 127 Vincent Hall, 206 Church Street SE, Minneapolis, MN 55455, USA, (msmaier@umn.edu).

†SISSA - International School for Advanced Studies, Via Bonomea 265, 34136 Trieste, Italy, (mauro.bardelloni@sissa.it, luca.heltai@sissa.it).

The idea behind expression templates is to overload `operator+`, `-`, `*`, etc., to build up an arithmetic syntax tree with the help of the C++ template mechanism instead of performing the arithmetic operation immediately by returning an intermediate object. The arithmetic operations are performed later when the expression is complete and an evaluation of the expression is actually requested. A number of numerical libraries make use of expression templates to a certain extent. Examples are the C++ linear-algebra libraries `Eigen` [5], or `Blaze` [6].

Although expression templates offer a good incarnation of the "generic programming" paradigm [8], they are a difficult technique to master, that is not easily adapted to existing numerical libraries, or *collections* of libraries, and it has non-negligible implementational complexity.

We present an alternative approach of building up an expression syntax for matrix-matrix, matrix-vector, and vector-vector operations. It uses the C++11 [1] features *lambda expressions* and *lambda captures*, as well as `std::function` objects, instead of a templates-only approach. This avoids the majority of the implementational complexity that usually comes with a pure template solution. Only two class signatures are required: A class `LinearOperator` to encapsulate a linear operation with two template parameters denoting its domain and range, and a class `PackagedOperation` to store a (partially applied) expression with a template parameter for its range space in which the result can be stored. Our expression syntax is suitable to encapsulate a wide variety of concrete matrix, vector, and linear solver classes because only a generic, high-level interface is required (see Section 2). In particular, we do not make any assumptions on the underlying memory model, or type of execution (sequential, or with thread/process parallelization). No random access to data, or other low-level access is required. This naturally rules out some low-level optimization techniques that require such access (or detailed information about the expression that is formed up), but on the other hand it allows encapsulation of a wide variety of concrete matrix and vector implementations.

The expression syntax is developed within the framework of the finite-element library `deal.II` and has been added to the library starting from version 8.3 [2]. However, we stress the point that the implementation that is presented in this work is otherwise *generic*. The only `deal.II` specific portion is the concrete form of the vector and matrix interfaces we assume to be present, and that `LinearOperator` and `PackagedOperation` mimic. These interfaces can be readily adjusted with minor changes to any concrete choice of naming and call signature. We provide also two minimal examples of other interfaces and concrete types, by adapting the `LinearOperator` class to work with the `Eigen` and `Blaze` libraries. All examples and benchmark codes are available (under the GNU Lesser General Public License version 2.1) on a public GitHub repository [7].

The overhead of dynamic `std::function` objects and dynamic temporary storage pools used in `LinearOperator` compared to optimal hand-written code, or (smart) expression templates, does not depend on the matrix size. The overhead is generally negligible for matrix sizes beyond 1000×1000.

The paper is structured as follows. In Section 2 we define the vector, matrix, and solver

interfaces. In Sections 3 and 4, we define the LinearOperator and PackagedOperation classes. We discuss implementation aspects for vector space operations and present a generic strategy for encapsulating concrete matrix objects into the LinearOperator framework. Section 5 presents a number of performance comparisons between LinearOperator and low-level implementations based on the deal.II, Eigen, and Blaze libraries. Section 6 presents a detailed real-life example, where LinearOperator and PackagedOperation are used to implement a preconditioner for the Stokes problem. A short performance comparison to a hand-written preconditioner is presented. We draw some conclusions in Section 7.

2 Vector, matrix and solver interfaces

In this section we introduce the vector, matrix and solver interfaces we will use to describe and implement the LinearOperator template class. We use the deal.II finite-element library for our concrete implementation. It provides a large variety of matrix and vector types (serial and MPI distributed variants, as well as wrappers to external libraries) and offers a standardized, high-level interface for all vector and matrix types.

A matrix object describes a linear operation. As such we require at least the following minimal interface for applying its action on a source vector src and storing the result in a destination vector dst:

```
1 class Matrix
2 {
3   template<typename Vector>
4   void vmult(Vector &dst, const Vector &src);
5
6   template<typename Vector>
7   void vmult_add(Vector &dst, const Vector &src);
8 };
```

Here, the variant vmult_add adds the result of the matrix vector multiplication to dst instead of replacing its former contents with the result. Depending on the concrete matrix type (such as full matrices, sparse matrices, MPI-distributed variants, or block matrices) many more member functions for accessing and manipulating a matrix are available, and the concrete signature of the vmult function, etc., may vary. It is only important to be able to call vmult, etc., with a compatible vector type. The power of this approach lies in the fact that using this interface is (almost) completely opaque with respect to the concrete implementation, or operations being performed.

Similarly, the guaranteed minimal interface for vectors—beside the possibility to use them in a call to vmult—is:

```
1 class Vector
2 {
3   typedef double number_type;
4
5   Vector &operator=(const Vector &);
```

```
6    Vector &operator=(number_type);
7
8    Vector &operator+=(const Vector &);
9    Vector &operator-=(const Vector &);
10   Vector &operator*=(number_type);
11   Vector &operator/=(number_type);
12 };
```

The roles of the operators `=`, `+=`, `-=`, `*=` and `/=` are straight-forward with the exception of the special assignment operator `=` that takes a scalar number. This is syntactic sugar to allow the mathematically common expression

```
1 v = 0.0; // v is of type Vector
```

to zero out a vector. One could have also implemented this with a `zero()` member function, or similar. We will only assume that assigning a 0 to zero out is a well defined operation, all other assignments of a scalar values are allowed to be undefined behaviour.

Another design decision that becomes apparent in the above interface is that no function requires intermediate storage. With matrix and vector objects that easily go into the gigabytes of memory requirements on modern platform, it is very important to prevent the user of the library from any accidental space leak that, e. g., a temporary resulting from an `operator+` would require. `deal.II` ensures this by forbidding all such implicit intermediates by simply not implementing those interfaces.

The iterative solver interfaces in `deal.II` for solving a linear equation $Ax = b$ with a given method and a preconditioner `prec` are fully templated and thus fairly generic:

```
1 template<typename Vector>
2 class Solver
3 {
4   template<typename Matrix, typename Preconditioner>
5   solve(const Matrix          &A,
6         Vector                &x,
7         const Vector          &b,
8         const Preconditioner &prec);
9 };
```

It is assumed that `Matrix`, `Preconditioner`, and `Vector` adhere to the interfaces presented above. (In case of a preconditioner, usually only `vmult` has to be implemented).

Remark. In the following we will assume that the above matrix and vector interfaces are the smallest level of granularity we have access to. This naturally rules out some optimizations and approaches that can be used for non-distributed linear algebra, but allows us to readily apply the developed framework to all scenarios of different matrix and vector implementations imaginable. Besides from the `deal.II` implementation, we provide two more backends in Section 5, where we substitute the `deal.II` internal classes with `Eigen` [5] and `Blaze` [6] implementations of dense and sparse matrices and vectors, by writing simple wrappers and plugins that adhere to the above interfaces.

3 A linear operator class

The solver interface introduced in the previous section is generic in the sense that any matrix or preconditioner object can be used provided that it implements (parts of) the above matrix interface. As an example, consider two matrices B and C. If a preconditioner $B + k\,C$ (with some scalar k) should be used, then there is no necessity to construct an actual matrix, say D, that physically stores $B + k\,C$. It completely suffices to provide an object whose `vmult` function *performs* the operation $(B + k\,C)v$ when applied to a given vector v. However, there is a slight problem with the above interface in the sense that it is unnecessarily verbose—compared to the fact that the mathematical expression $B + k\,C$ already encodes all necessary information. A possible implementation of the hypothetical preconditioner is

```
template<typename Matrix>
class MyPreconditioner
{
  MyPreconditioner(const Matrix &B, const Matrix &C, double k);

  template<typename Vector>
  void vmult(Vector &dst, const Vector &src) {
    C.vmult(dst, src);
    dst *= k;
    B.vmult_add(dst, src);
  }
};
```

One of the main motivations of the approach presented in the next subsection is the idea to transform the mathematical expression $B + k\,C$ into objects adhering to the above matrix interface and freeing the user from writing unnecessary boiler-plate code.

3.1 LinearOperator

To obtain an expression syntax for the above matrix and vector interfaces, we need a class concept that stores a computational expression. The concept of a linear operator is a good starting point for this because the current matrix interface can be transferred immediately: a linear operator has a notion of applying itself (`vmult`). Further, a linear operator has a well defined domain (of definition) and range. This is in contrast to the above matrix interface that usually only has templated `vmult` variants and consequently support multiple range and domain vector types.

The question that arises naturally (at least from an implementational standpoint) is: what strategy should we follow? It turns out that *knowing* the corresponding range and domain of a linear operator—and how to construct vectors belonging to the respective spaces—is not only very useful but sometimes required, e. g., the concatenation of two matrix objects without corresponding range and domain is ill-defined. We thus define with the help of C++11 `std::function` objects:

```
1  template <typename Range, typename Domain>
2  class LinearOperator
3  {
4  public:
5    std::function<void(Range  &v, const Domain &u)> vmult;
6    std::function<void(Range  &v, const Domain &u)> vmult_add;
7
8    std::function<void(Range  &v)> reinit_range_vector;
9    std::function<void(Domain &v)> reinit_domain_vector;
10
11   ...
12 };
```

Here, `vmult` and its variants shall carry the usual meaning. `reinit_range_vector` and `reinit_domain_vector` are function objects that shall reinitialize a vector `v` such that it is suitable as a source or destination vector in an application of `vmult`.

Beside the usual default copy constructor and assignment operator we also implement a default constructor that will populate all `std::function` objects with a default implementation throwing an error upon invocation. Further, templated variants of the copy constructor and assignment operator are provided that use the `linear_operator` wrapper that will be discussed in Section 3.3:

```
1  template <typename Range, typename Domain> class LinearOperator
2  {
3  public:
4    ...
5
6    LinearOperator();
7    LinearOperator(const LinearOperator<Range, Domain> &) = default;
8    template<typename Op> LinearOperator(const Op &op)
9    {
10     *this = linear_operator<Range, Domain, Op>(op);
11   }
12
13   LinearOperator<Range, Domain> &
14   operator=(const LinearOperator<Range, Domain> &) = default;
15
16   template <typename Op>
17   LinearOperator<Range, Domain> &operator=(const Op &op)
18   {
19     *this = linear_operator<Range, Domain, Op>(op);
20     return *this;
21   }
22 };
```

3.2 Vector space operations

With the help of the abstract `vmult` and `vmult_add` functions it is now possible to implement vector space operations on linear operators. The *key idea* is to capture the individual subexpressions (in form of their corresponding `vmult` and `vmult_add` `std::function` objects) of the operands by a *lambda-capture*. As an example, consider the concatenation of two compatible linear operators:

```
1  template <typename Range, typename Intermediate, typename Domain>
2  LinearOperator<Range, Domain>
3  operator*(const LinearOperator<Range, Intermediate> &first_op,
4            const LinearOperator<Intermediate, Domain> &second_op)
5  {
6    LinearOperator<Range, Domain> return_op;
7
8    return_op.reinit_domain_vector = second_op.reinit_domain_vector;
9    return_op.reinit_range_vector  = first_op.reinit_range_vector;
10
11   return_op.vmult = [first_op, second_op](Range &v, const Domain &u)
12   {
13     GrowingVectorMemory<Intermediate> vector_memory;
14
15     Intermediate *i = vector_memory.alloc();
16     second_op.reinit_range_vector(*i);
17     second_op.vmult(*i, u);
18     first_op.vmult(v, *i);
19     vector_memory.free(i);
20   };
21
22   ...
23
24   return return_op;
25 }
```

For temporary storage of the intermediate result a memory pool provided by `deal.II` is used that avoids unnecessary allocation and deallocation operations.

Remark. At this abstract level of concatenation of two opaque `vmult` function objects, temporary storage of intermediate results cannot be avoided. One might argue that for a plain matrix-matrix-vector product $y = A\,B\,x$ of two matrices A and B and a vector x the resulting operation could be fused into a single set of stacked loops,

$$y_i = \sum_{j,k} A_{ij} B_{jk} x_k,$$

that avoids intermediate storage. However, the goal of the discussion is to develop a mechanism that provides syntactic sugar for *completely abstract* linear algebra operations—and on this level of abstraction fusing of loops might not be possible (for certain data

structures), or not desirable, e. g., for distributed data structures fusing might involve prohibitively expensive communication between computing nodes.

The conceptually simpler multiplication with a scalar number, as well as addition and subtraction can be implemented in a straight-forward manner. As an example consider the addition of two linear operators:

```
template <typename Range, typename Domain>
LinearOperator<Range, Domain>
operator+(const LinearOperator<Range, Domain> &first_op,
          const LinearOperator<Range, Domain> &second_op)
{
  LinearOperator<Range, Domain> return_op;

  return_op.reinit_range_vector = first_op.reinit_range_vector;
  return_op.reinit_domain_vector = first_op.reinit_domain_vector;

  return_op.vmult = [first_op, second_op](Range &v, const Domain &u)
  {
    first_op.vmult(v, u);
    second_op.vmult_add(v, u);
  };

  return_op.vmult_add = [first_op, second_op](Range &v, const Domain &u)
  {
    first_op.vmult_add(v, u);
    second_op.vmult_add(v, u);
  };

  ...

  return return_op;
}
```

Remark. In a similar fashion it is possible to define in-place variants of all operations, `+=`, `-=`, `*=` (for concatenation as well as scalar multiplication) that replace the left-hand object.

3.3 Constructing a LinearOperator

A crucial, so far missing, ingredient is a strategy of how to construct a linear operator out of a given data structure such as a matrix. For this, we define a function

```
template <typename Range, typename Domain, typename Matrix>
LinearOperator<Range, Domain> linear_operator(const Matrix &matrix)
{
  LinearOperator<Range, Domain> return_op;

  // populate return_op...

```

```
    return return_op;
}
```

that takes a reference to a matrix object and converts it to a `LinearOperator`. The matrix object must remain a valid object throughout the whole lifetime of the `LinearOperator` object. With the help of a lambda expression the corresponding `vmult` (`vmult_add`, etc.) function of the matrix object can be encapsulated in a straightforward manner:

```
  op.vmult = [&matrix](Range &v, const Domain &u)
  {
    matrix.vmult(v,u);
  }
```

The last missing ingredient for the `linear_operator` wrapper is a mechanism for deriving `reinit_range_vector` and `reinit_domain_vector`. Due to the fact that a wide variety of data structures shall be supported, a general interface cannot be easily defined. An alternative strategy is to use *template specialization* of a helper class to distinguish between the vector types in question. The selection of the most specialized variants happens in the *second phase lookup*. Thus, it is possible to have a fairly generic implementation in the header file defining `LinearOperator` and providing specializations for certain types in completely different header files (that only need to be imported in a compilation unit actually using the types in question):

```
namespace internal
{
  template<typename Vector>
  struct ReinitHelper
  {
    template <typename Matrix>
    static
    void reinit_range_vector (const Matrix &matrix, Vector &v)
    {
      v.reinit(matrix.m());
    }

    ...
  };
}
```

The helper class is then use in the definition of `LinearOperator`:

```
return_op.reinit_range_vector = [&matrix_exemplar](Range &v)
{
  internal::ReinitHelper<Range>::reinit_range_vector(matrix, v);
};
```

This allows specialization for vector types that need a different setup. The split of the `Vector` and `Matrix` template parameter to belong to the struct and to the member function, respectively, allows to keep the `Matrix` template while specializing (or partially specializing) the `Vector` parameter:

```
namespace internal
{
  template <typename> struct ReinitHelper;

  template<>
  struct ReinitHelper<SpecialVector>
  {
    template <typename Matrix>
    static
    void reinit_range_vector (const Matrix &matrix,
                              SpecialVector &v)
    {
      // special setup...
    }

    ...
  };
```

Remark. Encapsulation of matrix objects into a `linear_operator` wrapper can also be used to provide safeguard against common user errors: For most `vmult` variants the source and destination vectors must be different storage locations. Encapsulating the call to `vmult` allows to easily provide fall-back code for this condition:

```
op.vmult = [&matrix](Range &v, const Domain &u)
{
  if (PointerComparison::equal(&v, &u))
    {
      // vmult with intermediate storage
    }
  else
    {
      matrix.vmult(v,u);
    }
};
```

Here, `PointerComparison::equal` returns `true` if the addresses of u and v are the same, otherwise it returns `false`.

Remark. As a byproduct of our design, the construction of a `LinearOperator` allows one to implement in a straight forward manner and with minimal boiler plate code a common interface, equipped with an embedded intuitive mathematical syntax, to *arbitrary external libraries*, provided that the outcome of the operation is *captured* by the `vmult` of the `LinearOperator` itself. As a powerful example, consider the following interface where a standard CBLAS `cblas_dgemv` routine is used to compute matrix-vector products between existing `deal.II` objects instead of using built-in library operations:

```
FullMatrix<double> A(n,n);
```

```
2  LinearOperator<Vector<double>, Vector<double> > op;
3
4  op.vmult = [&A, n] (Vector<double> &dst, const Vector<double> &src)
5  {
6    cblas_dgemv (CblasRowMajor, CblasNoTrans,
7                 n, n, 1.0, &A(0,0), n,
8                 &src(0), 1, 0.0, &dst(0), 1);
9  }
```

Such interfaces are very easy to write, in many occasions they remove the need to construct full wrappers for external libraries, they are fully compatible with `deal.II` solver routines, and they are agnostic of the underlying data structure, providing a very powerful way to exploit (or explore) external libraries in a non-intrusive way.

3.4 Eliding null operations

Consider a matrix $A \in \mathcal{M}at(n,n)$ and a vector $x \in \mathbb{R}^n$. In the worst case scenario of a full matrix, evaluating $A\,x$ requires $\sim n^2$ operations. However, if A is the *null matrix*, we would like to avoid all operations and simply set the result to zero in `vmult`. Similarly, a significant speed-up can be achieved for more complex operations, such as for example the evaluation of $(A + B)\,x$, where $A \in \mathcal{M}at(n,n)$, $B \in \mathcal{M}at(n,n)$, and $x \in \mathbb{R}^n$. In the most general case of full matrices, this operation would require $\sim 2n^2$ operations. If either B or A are a null matrix, at least half of the operations can be avoided.

In order to implement this strategy of eliding unnecessary operations we augment the `LinearOperator` class with a member object of type `bool`, `is_null_operator`, that describes whether the object represents a null matrix. Whenever this variable is *true*, the resulting object of an arithmetic operation can be simplified.

As an example, consider the `+` operator optimized using `is_null_operator`:

```
1  operator+(const LinearOperator<Range, Domain> &first_op,
2            const LinearOperator<Range, Domain> &second_op)
3  {
4    if (first_op.is_null_operator)
5      return second_op;
6    if (second_op.is_null_operator)
7      return first_op;
8
9    // Do the general case here
10   ...
11 }
```

The complete implementation of the `null_operator` simply provides a `vmult` method that zero out the destination vector, while `vmult_add` leaves the `Range` vector untouched:

```
1  LinearOperator<Range, Domain>
2  null_operator(const LinearOperator<Range, Domain> &op)
3  {
4    auto return_op = op;
```

```
5
6     return_op.is_null_operator = true;
7
8     return_op.vmult = [](Range &v, const Domain &u)
9     {
10      v = 0.;
11    };
12
13    return_op.vmult_add = [](Range &v, const Domain &u)
14    {};
15
16    return return_op;
17  }
```

3.5 LinearOperator for block structures

While it is readily possible to use the `LinearOperator` class to encapsulate block structures (block matrices acting on block vectors), it is often desirable to retain access to the underlying block structure. For this reasons we implement a derived class `BlockLinearOperator` that inherits the public interface from `LinearOperator` with the addition of three more function objects that provide information about the block structure:

```
1  template <typename Range, typename Domain>
2  class BlockLinearOperator : public LinearOperator<Range, Domain>
3  {
4  public:
5    ...
6
7    typedef LinearOperator<typename Range::BlockType, typename
         Domain::BlockType> BlockType;
8
9    std::function<unsigned int()> n_block_rows;
10   std::function<unsigned int()> n_block_cols;
11   std::function<BlockType(unsigned int, unsigned int)> block;
12 };
```

We provide helper functions which fill the above functions starting from standard `deal.II` block matrices:

```
1  BlockSpaseMatrix<double> A(m, n);
2  ...
3  auto B = block_operator(A);
```

Now we can access each sub-block as an individual `LinearOperator`:

```
1  auto B00 = B.block(0,0);
2  auto B10 = B.block(1,0);
```

Using such a structure, it is possible to use the `BlockLinearOperator` as a whole, as well as by accessing its composing blocks, by means of the member function `block`, like in

the snippet above.

This operator makes heavy use of `std::function` objects and lambda functions. Such a flexibility comes with a run-time penalty, which makes such an object efficient only when the encapsulated linear operators have a large individual size, (i.e., matrix blocks greater than roughly 1000×1000). Sections 5 and 6 analyze in detail the run-time penalty associated with such objects, and show its full potential in writing block based preconditioners for complex partial differential equations.

4 A PackagedOperation

In this section we discuss a further generalization of the linear operator concept that applies the same concept of *expression construction* to matrix-vector products, e. g., the evaluation of a residual

```
Vector<double> residual = b - A * x;
```

with a `LinearOperator A`, and vectors `b` and `x`. The key point is that the above syntax should not require any intermediate storage. We define the above binary operations in such a way that they yield an object of type `PackagedOperation`:

```
template <typename Range>
class PackagedOperation
{
public:
  ...

  std::function<void(Range &v)> apply;
  std::function<void(Range &v)> apply_add;

  std::function<void(Range &v)> reinit_vector;
};
```

which—similarly to `LinearOperator`—stores the knowledge of how to `apply` (or `apply_add`) a computation to a vector and how to initialize a vector such that it is suitable to hold the result. We define an implicit conversion operator that automatically converts the packaged operation to its result such that the above assignment to a *vector* type `residual` is possible:

```
template <typename Range>
class PackagedOperation
{
public:
  ...

  operator Range() const
  {
    Range result_vector;
    reinit_vector(result_vector);
```

```
11      apply(result_vector);
12      return result_vector;
13    }
14  };
```

With the *move assignment* semantics introduced in C++11 [1] the creation of a result vector and subsequent assignment does not imply any additional runtime cost. The multiplication of a linear operator with a vector is straight forward:

```
1   template <typename Range, typename Domain>
2   PackagedOperation<Range>
3   operator*(const LinearOperator<Range, Domain> &op,
4             const Domain &u)
5   {
6     PackagedOperation<Range> return_comp;
7
8     return_comp.reinit_vector = op.reinit_range_vector;
9
10    return_comp.apply = [op, &u](Range &v)
11    {
12      op.vmult(v, u);
13    };
14
15    ...
16
17    return return_comp;
18  }
```

Similarly, subtraction of a `PackagedOperation` from a vector:

```
1   template <typename Range>
2   PackagedOperation<Range> operator-(const Range &offset,
3                                      const PackagedOperation<Range> &comp)
4   {
5     PackagedOperation<Range> return_comp;
6
7     return_comp.reinit_vector = comp.reinit_vector;
8
9     return_comp.apply = [&offset, comp](Range &v)
10    {
11      comp.apply(v);
12      v *= -1.;
13      v += offset;
14    };
15
16    ...
17
18    return return_comp;
19  }
```

Again, all lambda objects that are created store references to vectors. This implies that (similarly to matrices that are wrapped into a `LinearOperator` object) all vectors must

remain valid objects throughout the lifetime of the PackagedOperation in which they are used. As demonstrated in the following Section, in terms of performance, the one-liner

```
Vector<double> residual = b - linear_operator(A) * x;
```

is equivalent to

```
Vector<double> residual;
residual.reinit(A.n());
A.vmult(residual, x);
residual *= -1.;
residual += b;
```

5 Performance benchmarks

This section presents a number of detailed performance comparisons between the implementation with LinearOperator and a direct low-level implementation in deal.II [2], as well as a comparison between LinearOperator variants that use Eigen [5] and Blaze [6] as backend for concrete matrix and vector types, and their respective expression templates version. The benchmark code used in this section is available on a public GitHub Repository [7].

The dynamic std::function objects and dynamic temporary storage pools (for intermediate results) used in our implementation entails a small runtime penalty compared to hand-written and optimized low-level implementations. We compare the runtime performances of LinearOperator and PackagedOperation variants with equivalent direct low-level implementations in deal.II, Eigen and Blaze. The comparison is done for the following four test cases:

Case 1: a matrix-vector multiplication: Mv;

Case 2: a power expression: $M^3 v$;

Case 3: a combination of matrix operations: $(M + 3\,Id)\,M\,v$;

Case 4: a combination of matrix and vector operations: $M(x + y + z)$.

The first two test cases are trivial and are meant to isolate the overhead introduced by LinearOperator, as well as to point out how fusing nested loops exploiting the knowledge of the low-level structure may not always be the best option, even if the implementation allows it. Test cases three and four involve, respectively, a non-trivial expression constructed through the linear combination of LinearOperator objects and a combination of LinearOperator and PackagedOperation objects, and are meant to expose some common pitfalls related to the blind usage of high-level syntax.

All tests are performed on both dense $n \times n$-matrices A,

$$A_{i,j} := 1 + \frac{1}{(i+1)(j+1)}$$

and sparse $n \times n$ matrices S that are the system matrix of a Laplace problem (discretized with linear finite elements on the unit square). Each operation is repeated $10,000$ times for different matrix sizes, in order to reduce random fluctuations on the final result, and the total computational time is reported both in graphical form (Figures 1 – 14, pages 19 – 27) and in tabular form (Tables 1 – 8, pages 33 – 34) using a laptop with a 2,8 GHz Intel Core i7 processor, and 16 GB of RAM (1600 MHz DDR3), running in serial on a single thread.

Independently of the particular low-level implementation, the `LinearOperator` and `PackagedOperation` variants of the four test cases have the following structure:

Case 1 (`LO`):

```
1 const auto op = linear_operator(matrix);
2 for (unsigned int i = 0; i < iter; ++i) {
3   x = op * x;
4   x /= x.l2_norm();
5 }
```

Case 2 (`LO`):

```
1 const auto op = linear_operator(matrix);
2 for (unsigned int i = 0; i < iter; ++i) {
3   x = op * op * op * x;
4   x /= x.l2_norm();
5 }
```

Case 3 (`LO`):

```
1 const auto op = linear_operator(matrix);
2 const auto reinit = op.reinit_range_vector;
3 for (unsigned int i = 0; i < iter; ++i) {
4   x = (3.0 * identity_operator(reinit) + op) * op * x;
5   x /= x.l2_norm();
6 }
```

Case 4 (`LO`):

```
1 const auto op = linear_operator(matrix);
2 for (unsigned int i = 0; i < iter; ++i) {
3   x = op * (x + y + z);
4   x /= x.l2_norm();
5 }
```

Remark. The performance penalty of `LinearOperator` and `PackagedOperation` can be isolated in two main parts:

- a call to an opaque `std::function` object: while in general this does not introduce too much overhead *per-se*, it may prevent good compilers from inlining, for example, `vmult` statements;

– runtime creation or destruction of temporary objects: even though LinearOperator and PackagedOperation are small objects, their construction and destruction at run time may still impact the overall performance of an algorithm if placed, for example, inside inner tight loops.

One way to avoid the second type of overhead is to construct LinearOperator and PackagedOperation outside of loops. This is the standard use case, for example when constructing a preconditioner: Typically, all LinearOperator and PackagedOperation objects are constructed prior to calling a solver, and internally the solver only sees overhead of the first type, i.e., those related to calling an opaque std::function.

A way to isolate this standard behaviour in our benchmarks is to replace the creation and destruction of LinearOperator and PackagedOperation inside the tight loops above with a single application of a PackagedOperation object, that is constructed outside of the loop:

Case 1 (LO fast):

```
const auto step = linear_operator(matrix) * x;
for (unsigned int i = 0; i < iter; ++i) {
  step.apply(x);
  x /= x.l2_norm();
}
```

Case 2 (LO fast):

```
const auto op = linear_operator(matrix);
const auto step = op * op * op * x;
for (unsigned int i = 0; i < iter; ++i) {
  step.apply(x);
  x /= x.l2_norm();
}
```

Case 3 (LO fast):

```
const auto op = linear_operator(matrix);
const auto reinit = op.reinit_range_vector;
const auto step = (3.0 * identity_operator(reinit) + op) * op * x;
for (unsigned int i = 0; i < iter; ++i) {
  step.apply(x);
  x /= x.l2_norm();
}
```

Case 4 (LO fast):

```
const auto step = linear_operator(matrix) * (x + y + z);
for (unsigned int i = 0; i < iter; ++i) {
  step.apply(x);
  x /= x.l2_norm();
}
```

In Figures 1 – 14, we report the first version of the benchmarks (involving creation and destruction of temporary `PackagedOperation` objects in each loop) by “`LO`”, and denote the second variant with “`LO fast`”.

5.1 Comparison with direct low-level implementation in `deal.II`

In this section, we use `deal.II` for the backend providing full and sparse matrix types (`FullMatrix` and `SparseMatrix`) for the `LinearOperator` objects. The comparison is done against the following low-level implementation code:

Case 1 (`deal.II native`):

```
Vector<double> tmp(n);
for (unsigned int i = 0; i < iter; ++i) {
  matrix.vmult(tmp, x);
  x = tmp;
  x /= x.l2_norm();
}
```

Notice that in the low-level implementation of Case 1, a temporary vector is necessary, since `deal.II` does not guard internally that the destination and source vector are different. The result of the comparison for Case 1 is presented in Figure 1 on page 19.

Case 2 (`deal.II native`):

```
Vector<double> tmp(n);
for (unsigned int i = 0; i < iter; ++i) {
  matrix.vmult(tmp, x);
  matrix.vmult(x, tmp);
  matrix.vmult(tmp, x);
  x = tmp;
  x /= x.l2_norm();
}
```

In this case we expect a slightly better result with the low-level implementation, since we are exploiting knowledge about the operation to save one temporary vector allocation. The internal memory management of the `LinearOperator` object has a slight overhead, visible in Figure 2 (page 20) for very small matrix sizes. It is negligible for larger sized matrices.

Case 3 (`deal.II native`):

```
Vector<double> tmp(n);
for (unsigned int i = 0; i < iter; ++i) {
  matrix.vmult(tmp, x);
  matrix.vmult(x, tmp);
  x.add(3., tmp);
  x /= x.l2_norm();
}
```

The same considerations we made for Case 2 are also valid for Case 3: A slight overhead for the `LinearOperator` variant with respect to the direct implementation is visible in Figure 3 on page 20.

Case 4a (`deal.II native`):

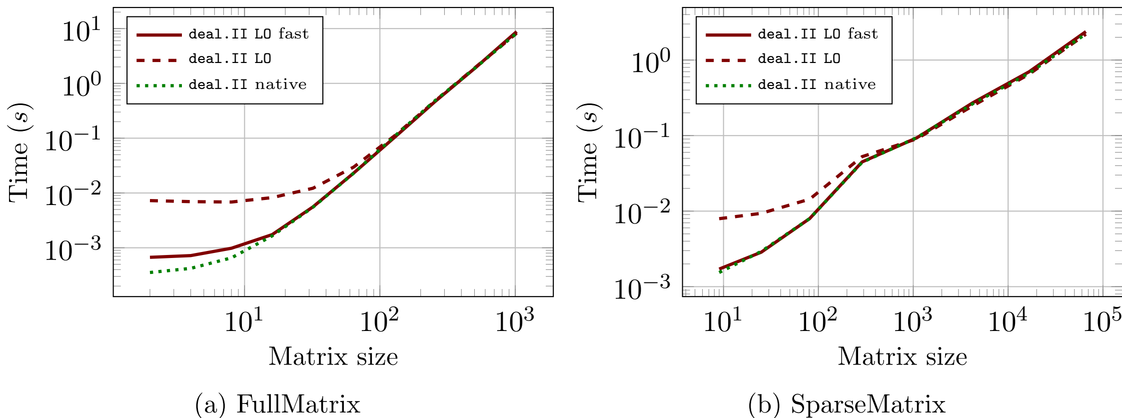
```
1 Vector<double> tmp(n);
2 for (unsigned int i = 0; i < reps; ++i) {
3   matrix.vmult(tmp, x);
4   matrix.vmult_add(tmp, y);
5   matrix.vmult_add(tmp, z);
6   x = tmp;
7   x /= x.l2_norm();
8 }
```

We notice from Figure 4 on page 20 that the implementation in Case 4a does not compare well with the `LinearOperator` and `PackagedOperation` variant, since the order of the operations is suboptimal in this implementation. A better result is obtained in Case 4b below where summation between vectors is performed prior to matrix-vector multiplication.

Case 4b (`deal.II native`):

```
1 Vector<double> tmp(n);
2 for (unsigned int i = 0; i < reps; ++i) {
3   tmp = x;
4   tmp += y;
5   tmp += z;
6   matrix.vmult(x, tmp);
7   x /= x.l2_norm();
8 }
```

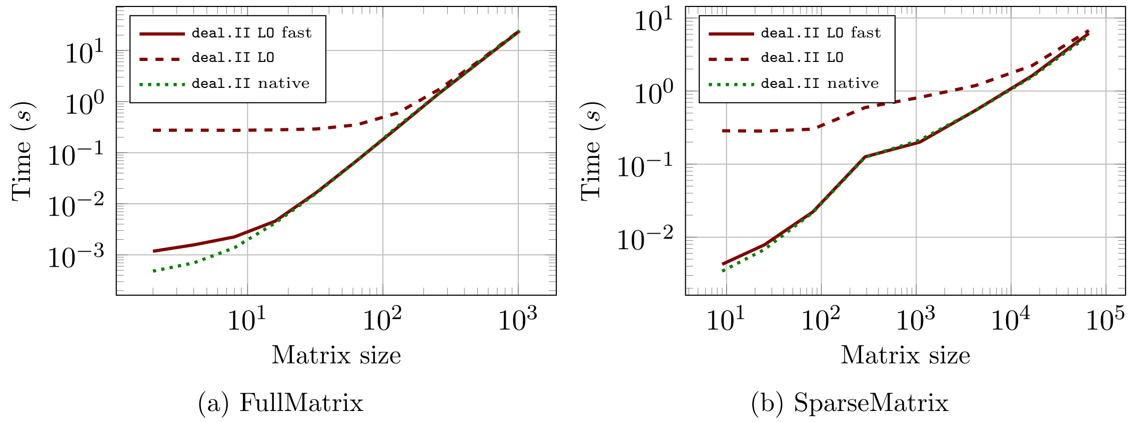
The results from the optimal version of the low-level implementation of **Case 4b**, collected in Figure 5, confirm that the overhead of using `LinearOperator` and `PackagedOperation` is only noticeable for very small matrix sizes of $n < 300$, and is thus negligible in practice.



(a) FullMatrix

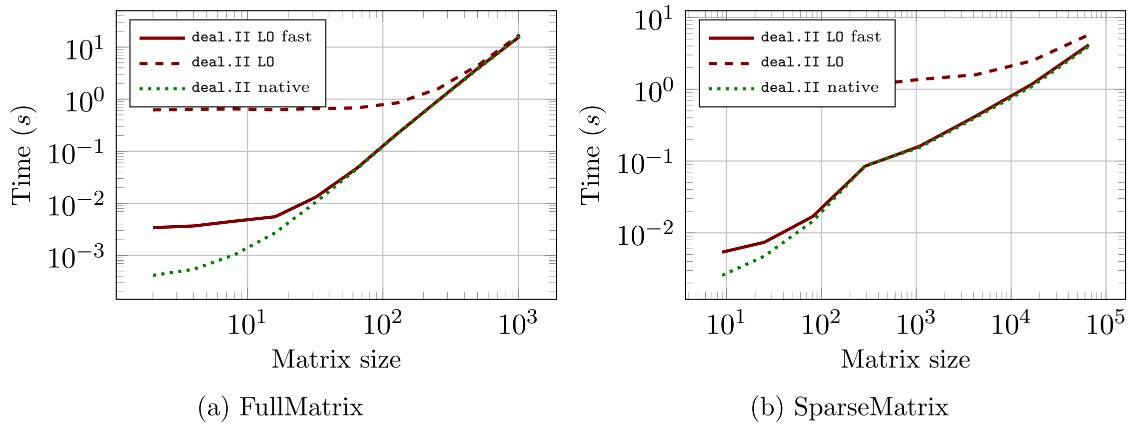
(b) SparseMatrix

Figure 1: Case 1, Mv, `LinearOperator` VS `deal.II native`



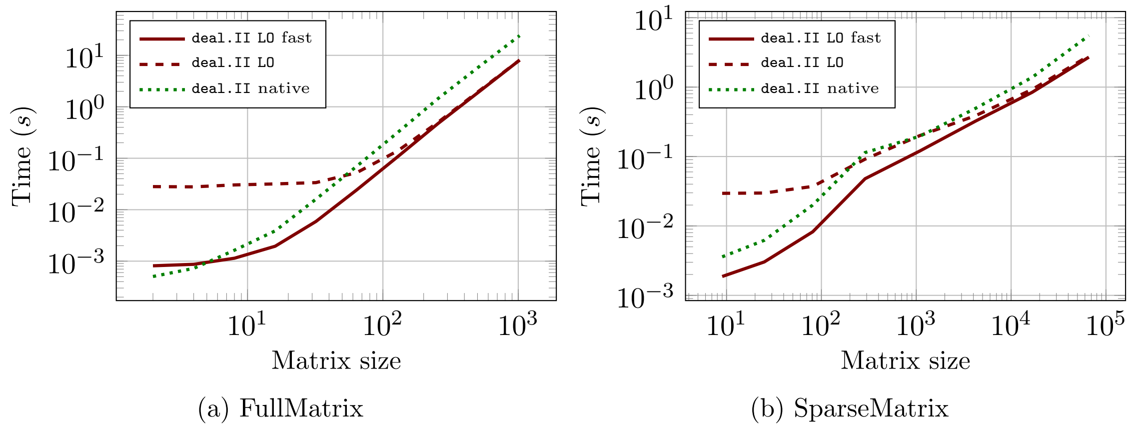
(a) FullMatrix (b) SparseMatrix

Figure 2: Case 2, $M^3 v$, `LinearOperator` VS `deal.II native`



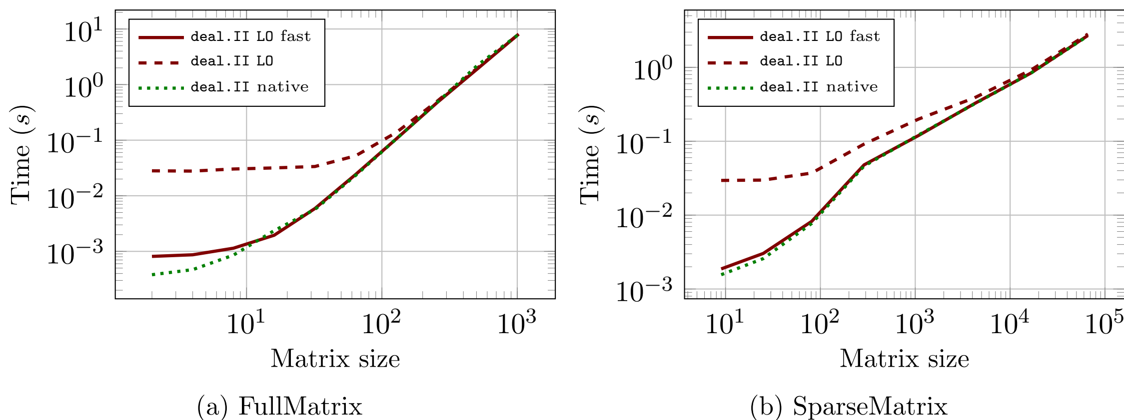
(a) FullMatrix (b) SparseMatrix

Figure 3: Case 3, $(M + 3Id)Mv$, `LinearOperator` VS `deal.II native`



(a) FullMatrix (b) SparseMatrix

Figure 4: Case 4a, $M(x + y + z)$, `LinearOperator` VS `deal.II native`



(a) FullMatrix

(b) SparseMatrix

Figure 5: Case 4b, $M(x+y+z)$, `LinearOperator` VS `deal.II native`

5.2 Comparison with direct low-level implementation in `Eigen`

`Eigen` [5] is a `C++` template library for linear algebra that offers a performant, high-level interface based on expression templates. Expression templates allow to intelligently remove temporaries and enable lazy evaluation, when it is appropriate.

In this regard, the expression-syntax mechanism introduced by `LinearOperator` and `PackagedOperation` is superfluous when used with a backend that already supports such a high-level syntax. Nevertheless, we provide such an interface for `Eigen` here in order to estimate the overhead of `LinearOperator` and `PackagedOperation` compared to a native expression-template mechanism. As a positive side effect, this demonstrates how versatile our approach is: Interfacing with `Eigen` via `LinearOperator` is fully compatible with `deal.II`—no additional wrapper is required. This allows, for example, to use the full suite of linear solvers and preconditioners implemented in `deal.II` with data objects from `Eigen`. In this section we use `Eigen` as a backend for dense and sparse matrices (`Eigen::Matrix` and `Eigen::SparseMatrix`).

Only a minimal amount of *glue-code* is required, to make `Eigen` compatible to the `LinearOperator` interface. This is accomplished by exploiting the `plugin` mechanism of the `Eigen` library: Prior to including any header files a plugin header is specified with the help of a macro:

```
1 #define EIGEN_MATRIX_PLUGIN "eigen_plugin.h"
2 #include <Eigen/Dense>
3 #include <Eigen/Sparse>
```

The header file `eigen_plugin.h` can now be used to implement the missing `Vector` interfaces expected by `LinearOperator` and `PackagedOperation`; for example,

```
1 inline
2 Eigen::Matrix<_Scalar, _Rows, _Cols, _Options, _MaxRows, _MaxCols>&
3 operator=(const _Scalar &s) {
```

```
4    this->fill(s);
5    return (*this);
6  }
7  ...
```

After the minimal interface has been provided, a linear operator can be constructed by simply populating its `vmult` operator through a lambda capture:

```
1  typedef Eigen::Matrix<double, Eigen::Dynamic, 1> EVector;
2  LinearOperator<EVector, EVector> lo;
3  lo.vmult = [&Ematrix] (EVector &d, const EVector &s) {
4    d = Ematrix*s;
5  };
```

This allows the `LinearOperator` variants to be written using the same syntax used previously. The low-level implementation using `Eigen` is straightforward, thanks to their template expression syntax.

Case 1 (`Eigen native`):

```
1  for (unsigned int i = 0; i < iter; ++i) {
2    Ex = Ematrix * Ex;
3    Ex /= Ex.norm();
4  }
```

In the low-level implementation of Case 1, no temporary vector is necessary, since `Eigen` automatically detects when a temporary object is needed and allocates the memory on its own. The result of the comparison for Case 1 is presented in Figure 6 on page 23.

Case 2a (`Eigen native`):

```
1  for (unsigned int i = 0; i < iter; ++i) {
2    Ex = Ematrix * Ematrix * Ematrix * Ex;
3    Ex /= Ex.norm();
4  }
```

In Case 2a, the `Eigen` expression-template mechanism detects that two matrix-matrix multiplications are requested, and fuses the two loops internally, effectively providing a matrix vector multiplication between M^3 and the given vector. Although formally correct, such loop fusion is unnecessarily expensive, even for small matrix sizes, as Figure 7 shows. The `LinearOperator` variant, offers a clear advantage in this case: Due to the fact that we do not have direct access to the underlying matrix object, only the matrix-vector product can be used. A fair comparison is achieved by forcing `Eigen` to isolate the matrix-vector product. This is done in the following test case.

Case 2b (`Eigen native`):

```
1  for (unsigned int i = 0; i < iter; ++i) {
2    Ex = Ematrix * (Ematrix * (Ematrix * Ex));
3    Ex /= Ex.norm();
4  }
```

In this case we expect a better result with the low-level implementation, since it does not have the overhead of LinearOperator at run time. The slight overhead is visible in Figure 8 (page 24) for small matrix sizes, which is clearly negligible for larger matrices.

Case 3 (Eigen native):

```
for (unsigned int i = 0; i < iter; ++i) {
  Ex = 3 * Ematrix * Ex + Ematrix * (Ematrix * Ex);
  Ex /= Ex.norm();
}
```

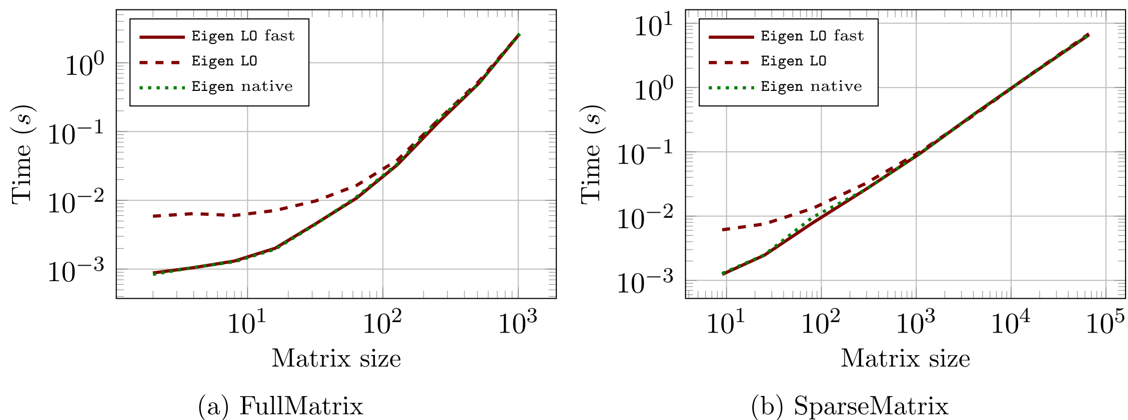
Eigen does not support an abstract *identity* operator, which is in contrast provided by the LinearOperator implementation. As such, Case 3, which involves the operation $(M+3Id)Mv$, cannot be constructed automatically by the template mechanism in Eigen, and has to be expanded manually, resulting in an overall cost which is higher than the LinearOperator counterpart, as Figure 9 on page 24 shows.

Case 4 (Eigen native):

```
for (unsigned int i = 0; i < iter; ++i) {
  Ex = M * (Ex + Ey + Ez);
  Ex /= Ex.norm();
}
```

The results of the last example are shown in Figure 10 on page 25.

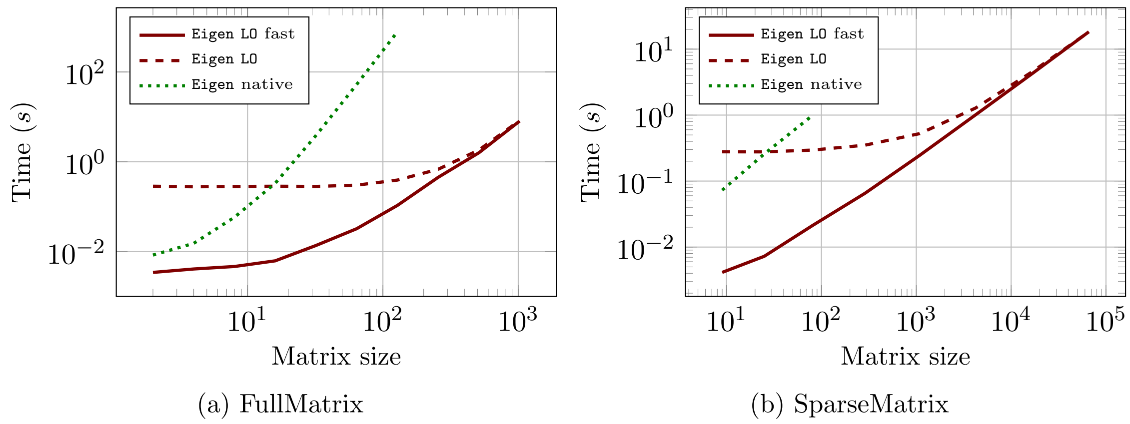
In conclusion, the performance tests in this section confirm our statement that the overhead introduced by LinearOperator and PackagedOperation objects is already negligible for medium sized matrices with $n > 1000$, even when comparing with a native expression-template implementation. Moreover, Cases 2 and 3 are an example where a clever usage of an abstract identity operator and lazy evaluation of matrix-vector products with dynamic intermediate storage have beneficial effects compared to a low-level expression template that blindly avoid temporary by fusing matrix-matrix products.



(a) FullMatrix

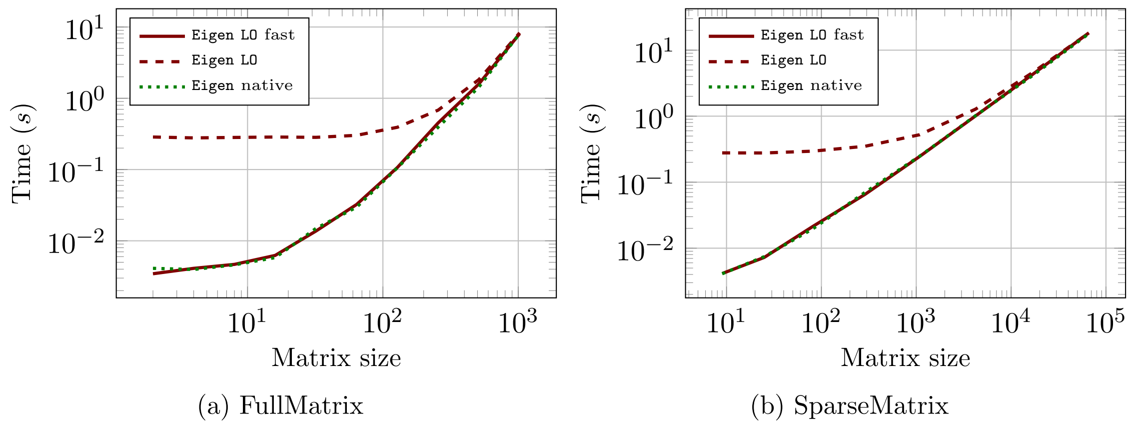
(b) SparseMatrix

Figure 6: Case 1, Mv, LinearOperator VS Eigen



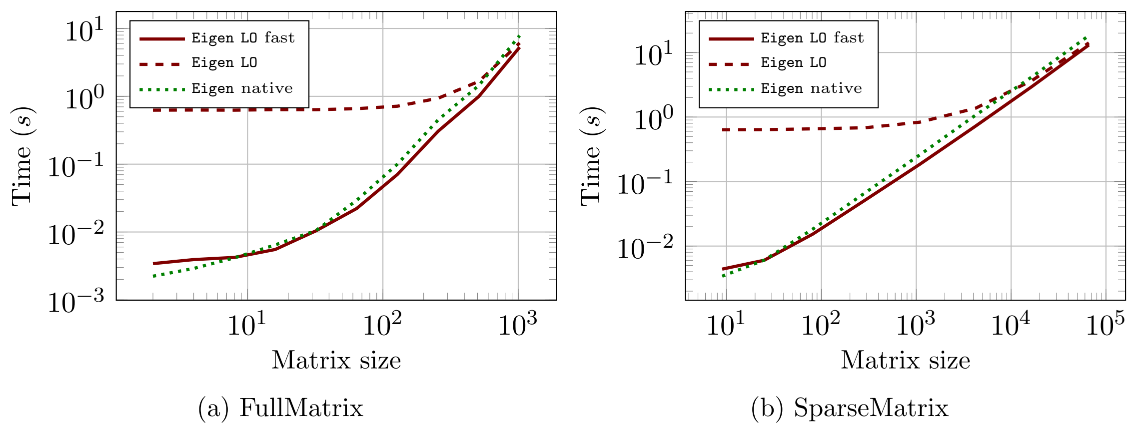
(a) FullMatrix (b) SparseMatrix

Figure 7: Case 2a, M^3v, `LinearOperator` VS `Eigen`



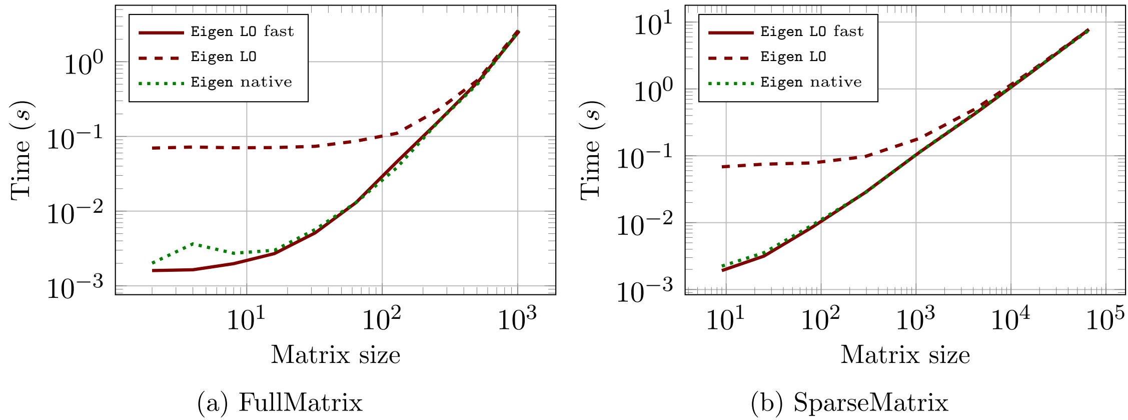
(a) FullMatrix (b) SparseMatrix

Figure 8: Case 2b, $M(M(Mv))$, `LinearOperator` VS `Eigen`



(a) FullMatrix (b) SparseMatrix

Figure 9: Case 3, $(M + 3Id)Mv$, `LinearOperator` VS `Eigen`



(a) FullMatrix (b) SparseMatrix

Figure 10: Case 4, $M(x+y+z)$, `LinearOperator` VS `Eigen`

5.3 Comparison with direct low-level implementation in `Blaze`

`Blaze` [6] is a `C++` math library for dense and sparse arithmetic that uses newly introduced *smart* expression templates. In contrast to plain expression templates, the *smart* counterparts also make efforts in avoiding unnecessary matrix-matrix multiplications. This addresses, for example, the overhead we encountered in Case 2a with `Eigen` in Section 5.2.

Similarly to the comparison with `Eigen`, we use the full and sparse matrices provided by `Blaze` (`blaze::DynamicMatrix` and `blaze::CompressedMatrix`) as low-level backends for the `LinearOperator` objects. Unfortunately, `Blaze` does not support a plugin mechanism as provided by `Eigen`. Thus, we need a small wrapper around the `blaze::DynamicVector` type in order to use `LinearOperator` and `PackagedOperation` objects with `Blaze`.

Case 1 (`Blaze native`):

```
1  for (unsigned int i = 0; i < iter; ++i) {
2    Bx = Bmatrix*Bx;
3    Bx /= std::sqrt(blaze::trans(Bx)*Bx);
4  }
```

Similarly to what happens in `Eigen`, the low-level implementation of Case 1 using `Blaze` does not require explicit temporary vectors which are created by the smart expression template mechanism. The result of the comparison for Case 1 is presented in Figure 11 on page 26.

Case 2 (`Blaze native`):

```
1  for (unsigned int i = 0; i < iter; ++i) {
2    Bx = Bmatrix*Bmatrix*Bmatrix*Bx;
3    Bx /= std::sqrt(blaze::trans(Bx)*Bx);
4  }
```

In Case 2 the `Blaze` smart expression template mechanism is capable of detecting that the two matrix-matrix multiplications are in fact not necessary, and reduces the operation

to three matrix-vector products, as in the `LinearOperator` variant. Figure 12 on page 27 shows the comparison between the two, with little overhead in the `LinearOperator` variant for large matrix sizes.

Case 3 (`Blaze native`):

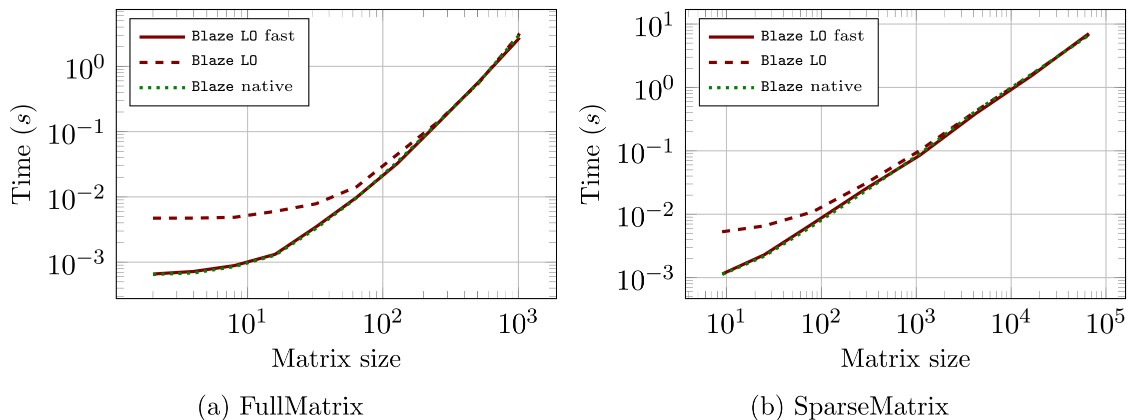
```
1    for (unsigned int i = 0; i < iter; ++i) {
2      Bx = 3*Bmatrix*Bx+Bmatrix*Bmatrix*Bx;
3      Bx /= std::sqrt(blaze::trans(Bx)*Bx);
4   }
```

Similarly to `Eigen`, no abstract *identity* operator is provided by `Blaze`. Consequently, the overall cost for Case 3 is higher than the `LinearOperator` counterpart, as Figure 13 (page 13).

Case 4 (`Blaze native`):

```
1   for (unsigned int i = 0; i < iter; ++i)
2   {
3      Bx = Bmatrix*(Bx+By+Bz);
4      Bx /= std::sqrt(blaze::trans(Bx)*Bx);
5   }
```

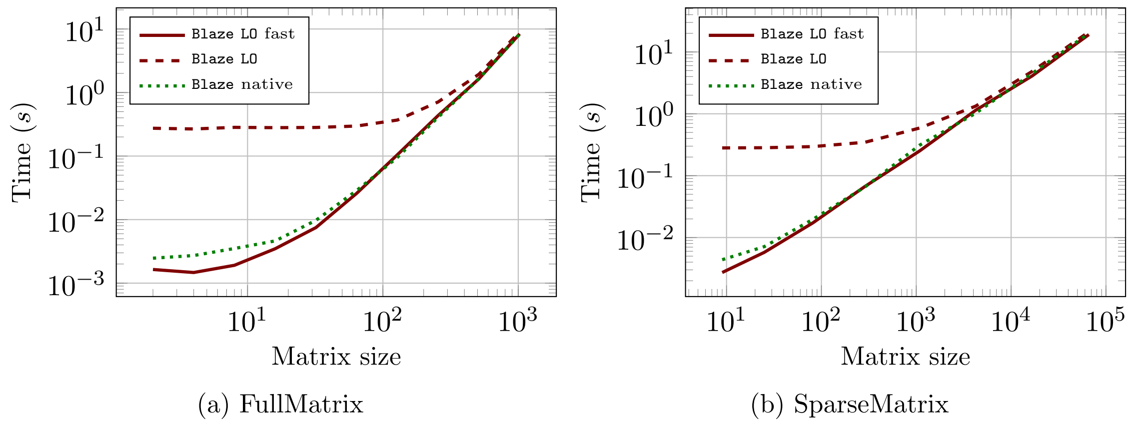
The last example of this section, shown in Figure 14, confirms that the overhead introduced by `LinearOperator` and `PackagedOperation` objects is negligible for large matrix sizes, even when comparing with native smart expression template implementations.



(a) FullMatrix

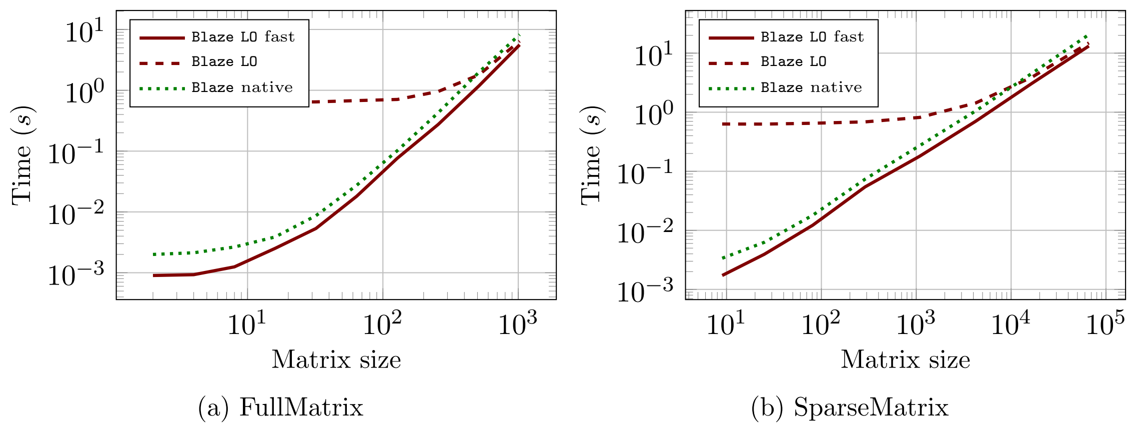
(b) SparseMatrix

Figure 11: Case 1, Mv, `LinearOperator` VS `Blaze`



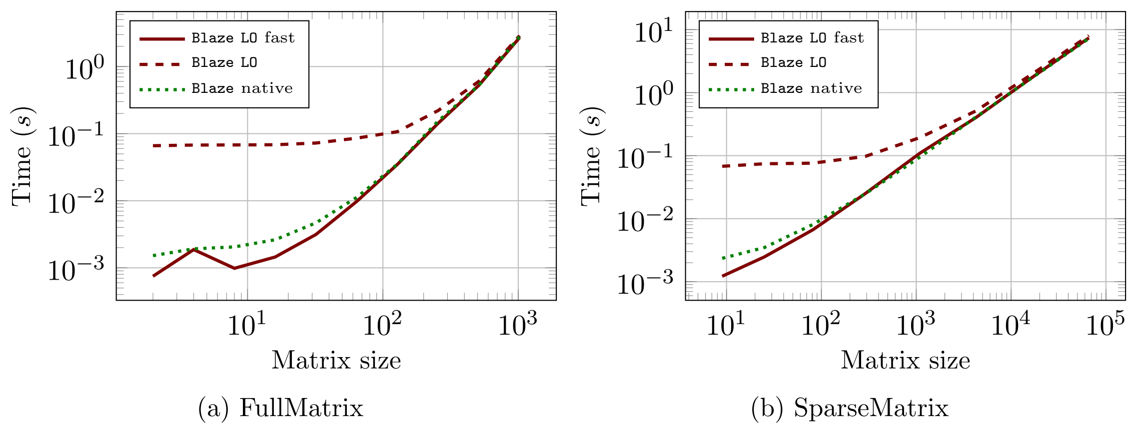
(a) FullMatrix (b) SparseMatrix

Figure 12: Case 2, M^3v, `LinearOperator` VS `Blaze`



(a) FullMatrix (b) SparseMatrix

Figure 13: Case 3, $(M + 3Id)Mv$, `LinearOperator` VS `Blaze`



(a) FullMatrix (b) SparseMatrix

Figure 14: Case 4, $M(x + y + z)$, `LinearOperator` VS `Blaze`

5.4 Global comparison and summary

For completeness, we report the timing results of all comparisons made in this section (with their best option, where available) in tabular form in an Appendix (starting on page 33). Tables 1 – 8 (pages 33 – 34) list the timings for all four cases and for all low-level backends (`deal.II`, `Eigen` and `Blaze`).

Some interesting features emerge from the corresponding tables: both `Eigen` and `Blaze` provide low-level implementations of dense matrix-vector products that are generally more efficient than their `deal.II` counterparts. On the other hand, the implementation of compressed-row sparse matrices in `deal.II` seems to surpass the `Eigen` and `Blaze` counterparts.

The `LinearOperator` variants, involving a single creation of a `PackagedOperation` object outside the loop, have negligible overhead, and in some cases behave better than their corresponding low-level implementation (thanks, for example in case 3, to the smart treatments of temporary identity operators).

The overhead introduced by the creation and destruction of a `PackagedOperation` in each inner loop is constant in time (on a 2,8 GHz Intel Core i7 processor, 20,000 temporary creations and destructions take around 0.28 seconds). This implies that for small matrix sizes, such overhead is clearly noticeable, and its importance decreases when the matrix size reaches a few hundred elements for the dense case, and a few thousand elements for the sparse case.

In Tables 1 – 8 we report the `LinearOperator` version (`LO`) with intermediates (in the loop) with the wording `PO+backend` and use `LO+backend` for the fast variants (`LO fast`).

6 A practical example—A preconditioner for the Stokes Problem

A good way to show the power of an expression syntax for matrix-vector operations is a real life example. Writing a suitable preconditioner for complex problems is far from being trivial, and a non user-friendly approach often leads to mistakes and bugs which are quite difficult to catch. In this section a Schur-complement preconditioner for the Stokes problem is presented that serves as an example how the new `LinearOperator` concept tremendously simplifies the readability of numerical code, while maintaining the same performance as hand-crafted, low-level variants.

6.1 Statement of the problem

Consider the stationary Stokes problem

$$-\Delta u + \nabla p = f, \qquad \nabla \cdot u = 0. \tag{1}$$

The corresponding system matrix of this problem has the saddle-point structure

$$\mathbb{M} = \begin{pmatrix} A & B^t \\ B & 0 \end{pmatrix}.$$

It is well known that a good preconditioner for this system is given by (see, e. g., [3])

$$\mathbb{P} = \begin{pmatrix} A & B^t \\ 0 & -S \end{pmatrix}^{-1}, \tag{2}$$

where $S = BA^{-1}B^t$ is the corresponding *Schur complement*.

We are going to focus on the action of the inverse of (2) on a generic vector. In the following we assume that $A \in \mathcal{M}at(n,n)$ and $B \in \mathcal{M}at(m,n)$ are results of a suitable discretization (of stable test function spaces) such that the linear system is well-posed.

6.2 A low-level implementation

A straight forward implementation of the preconditioner $\mathbb{P}$ is to compute the action of $\mathbb{P}^{-1}$ on a given vector $(u,p)^t$:

$$\begin{pmatrix} v \\ q \end{pmatrix} = \begin{pmatrix} A & B^t \\ 0 & -S \end{pmatrix}^{-1} \cdot \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} A^{-1} & A^{-1}B^tS^{-1} \\ 0 & -S^{-1} \end{pmatrix} \cdot \begin{pmatrix} u \\ p \end{pmatrix}. \tag{3}$$

This leads to the following low-level pseudocode implementation of the preconditioner:

```
1 v = inv(A) * u;
2 u_tmp = inv(S) * p;
3 u_tmp = B^t * u_tmp;
4 u_tmp = inv(A) * u_tmp;
5 v += u_tmp;
6 q = -inv(S) * p;
```

This approach is unnecessarily expensive because it consists of two additional (and otherwise identical) solve operations (with `inv(A)` and with `inv(S)`) and an intermediate vector. This can be optimized by some minor code refactoring:

```
1 q = inv(S) * p;
2 v = u;
3 v += Bt * q;
4 v = inv(A) * v;
5 q *= -1;
```

We stress the point that, although the derivation of this pseudocode is straight forward, it is nonetheless non-trivial. We demonstrate in the next subsection that the same pseudocode can be derived on an abstract level with the help of the `LinearOperator` template class.

6.3 A high-level implementation with the LinearOperator template class

Let $L = (l_{ij})_{ij}$ be a *regular block lower-triangular matrix* consisting of linear operators l_{ij} and invertible diagonal blocks l_{ii}. For a given right hand side $b = (b_j)_j$, let the task be to find a block vector $x = (x_i)_i$ such that:

$$Lx = b. \tag{4}$$

This equation can be solved by block-wise *forward substitution*:

$$\begin{cases} x_0 = l_{0,0}^{-1} \cdot b_0, \\ x_i = l_{i,i}^{-1} \cdot \Big(b_n - \sum_{j=0}^{i-1} l_{i,j} \cdot x_j\Big). \end{cases} \tag{5}$$

Similarly, a system of equations with an upper block triangular matrix U can be solved by block-wise *backward substitution*:

$$\begin{cases} x_n = u_{n,n}^{-1} \cdot b_n, \\ x_i = u_{i,i}^{-1} \cdot \Big(b_n - \sum_{j=i+1}^{n} u_{i,j} \cdot x_j\Big). \end{cases} \tag{6}$$

Both algorithms can be implemented in a straightforward manner. We created two functions with the following signature:

```
1 template <size_t n, typename Range, typename Domain>
2 BlockLinearOperator<n, n, Domain, Range>
3 block_back_substitution(
4   const BlockLinearOperator<n, n, Range, Domain> &block_operator,
5   const BlockLinearOperator<n, n, Domain, Range> &diagonal_inverse);
```

This allows us to write the inverse of the preconditioner in a “natural”, high-level way without loss of (algorithmic) performance. Notice that in (6) we only use the inverses of diagonal blocks, and, more importantly, they are used only once. The following listing illustrates the implementation of the `vmult` operation of the block back-substitution operator, which assumes that all input objects are of type `BlockLinearOperator`. The other functions have a very similar implementation.

```
1  return_op.vmult = [block_operator, diagonal_inverse]
2                    (Range &v, const Range &u)
3  {
4    const unsigned int m = block_operator.n_block_rows();
5    if (m == 0)
6      return;
7
8    v.block(m-1) = diagonal_inverse.block(m-1, m-1) * u.block(m-1);
9
10   for (int i = m - 2; i >= 0; --i)
```

```
11      {
12        auto &dst = v.block(i);
13        dst = u.block(i);
14        dst *= -1.;
15        for (int j = i + 1; j < m; ++j)
16          dst += block_operator.block(i, j) * v.block(j);
17        dst *= -1.;
18        dst = diagonal_inverse.block(i, i) * dst;
19      }
20  };
```

With these prerequisites at hand, the solution process for a Stokes system, including the construction of a block triangular preconditioner can be implemented in very few lines of code, showing the full power of the expression syntax. Assuming that preconditioners `precA` and `precS`, as well as solvers `Asolver` and `Ssolver` for A and S, respectively, are available and with a solver `Gsolver` for the global system:

```
1   auto Bt = transpose(B);
2   auto Ainv = inverse_operator(A , Asolver, Aprec);
3   auto S = B * Ainv * Bt;
4   auto Sinv = inverse_operator(S, Ssolver, Sprec);
5
6   auto system_matrix = block_operator({{A, Bt}, {B, 0}});
7   auto diagonal_inverse = block_diagonal_operator({Ainv, -1. * Sinv});
8   P_inv = block_back_substitution(system_matrix, diagonal_inverse);
9
10  auto system_inverse =  inverse_operator(system_matrix, Gsolver, P_inv);
11
12  solution = system_inverse * rhs;
```

6.4 Benchmarks

In this subsection we present a small test to assess the performance of the `LinearOperator` implementation of the system preconditioners when compared to a standard implementation on a parallel cluster, using the MPI parallelization strategy. The computations were performed on a cluster with 11 nodes. Each node is equipped with two CPUs, with 10 cores (E5) Intel Xeon E5-2680 v2 each. The standard implementation is taken from the example program "step-32" of the `deal.II` library [9]. It implements the optimized algorithm presented in Subsection 6.2. The `LinearOperator` implementation uses the *back-substitution algorithm* presented in equation (6). The benchmark consists of performing 100 matrix vector multiplications with the two preconditioners.

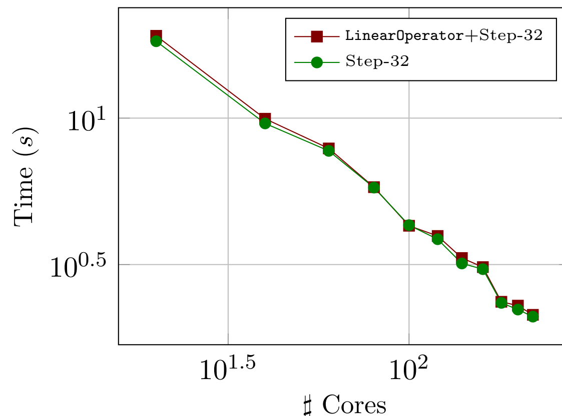


Figure 15: Computational cost of a Stokes preconditioner multiplication, `deal.II` native implementation VS `LinearOperator` implementation in `step-32` [9], while varying the number of cores.

7 Conclusion

We introduced an expression syntax for the evaluation of vector space operations. It uses the new C++11 features of `std::function` objects and lambda expressions to avoid the usual template complexity associated with an implementation via pure expression templates. The introduced framework is generic, it requires only a minimal interface that vector and matrix classes must adhere to. We gave a number of performance comparisons and examples that demonstrate that the framework results in efficient, short and concise code. The numerical examples demonstrate that the runtime overhead introduced by the `std::function` objects and lambda functions is negligible, even when compared to native expression templates implementations like `Eigen` or `Blaze`.

Acknowledgements

This work was partially supported by the project OpenViewSHIP, "Sviluppo di un ecosistema computazionale per la progettazione idrodinamica del sistema elica-carena", supported by Regione FVG - PAR FSC 2007-2013, Fondo per lo Sviluppo e la Coesione and by the project "TRIM - Tecnologia e Ricerca Industriale per la Mobilità Marina", CTN01-00176-163601, supported by MIUR, the Italian Ministry of Instruction, University and Research.

The authors wish to thank the anonymous referees for their precious and constructive comments.

Appendix

Timing results in tabular form for all comparisons made in Section 5.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
2	$3.54 \cdot 10^{-4}$	$6.7 \cdot 10^{-4}$	$7.26 \cdot 10^{-3}$	$8.38 \cdot 10^{-4}$	$8.79 \cdot 10^{-4}$	$5.88 \cdot 10^{-3}$	$6.47 \cdot 10^{-4}$	$6.58 \cdot 10^{-4}$	$4.72 \cdot 10^{-3}$
4	$4.21 \cdot 10^{-4}$	$7.2 \cdot 10^{-4}$	$6.95 \cdot 10^{-3}$	$1.06 \cdot 10^{-3}$	$1.05 \cdot 10^{-3}$	$6.41 \cdot 10^{-3}$	$6.84 \cdot 10^{-4}$	$7.2 \cdot 10^{-4}$	$4.73 \cdot 10^{-3}$
8	$6.55 \cdot 10^{-4}$	$9.81 \cdot 10^{-4}$	$6.84 \cdot 10^{-3}$	$1.28 \cdot 10^{-3}$	$1.31 \cdot 10^{-3}$	$6.02 \cdot 10^{-3}$	$8.55 \cdot 10^{-4}$	$8.86 \cdot 10^{-4}$	$4.87 \cdot 10^{-3}$
16	$1.65 \cdot 10^{-3}$	$1.74 \cdot 10^{-3}$	$8.23 \cdot 10^{-3}$	$1.95 \cdot 10^{-3}$	$2.01 \cdot 10^{-3}$	$7.12 \cdot 10^{-3}$	$1.28 \cdot 10^{-3}$	$1.32 \cdot 10^{-3}$	$6 \cdot 10^{-3}$
32	$5.5 \cdot 10^{-3}$	$5.57 \cdot 10^{-3}$	$1.22 \cdot 10^{-2}$	$4.53 \cdot 10^{-3}$	$4.59 \cdot 10^{-3}$	$9.82 \cdot 10^{-3}$	$3.28 \cdot 10^{-3}$	$3.46 \cdot 10^{-3}$	$7.84 \cdot 10^{-3}$
64	$2.3 \cdot 10^{-2}$	$2.31 \cdot 10^{-2}$	$2.97 \cdot 10^{-2}$	$1.12 \cdot 10^{-2}$	$1.08 \cdot 10^{-2}$	$1.65 \cdot 10^{-2}$	$9.69 \cdot 10^{-3}$	$9.85 \cdot 10^{-3}$	$1.43 \cdot 10^{-2}$
128	0.11	0.1	0.11	$3.38 \cdot 10^{-2}$	$3.29 \cdot 10^{-2}$	$3.85 \cdot 10^{-2}$	$3.44 \cdot 10^{-2}$	$3.26 \cdot 10^{-2}$	$4.49 \cdot 10^{-2}$
256	0.49	0.47	0.48	0.15	0.14	0.15	0.14	0.14	0.14
512	1.96	1.96	2	0.5	0.5	0.54	0.57	0.58	0.56
1,024	8.25	8.81	8.24	2.64	2.64	2.63	3.09	2.75	3.18

Table 1: Case 1, Mv, Comparison of full-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
9	$1.54 \cdot 10^{-3}$	$1.71 \cdot 10^{-3}$	$7.94 \cdot 10^{-3}$	$1.28 \cdot 10^{-3}$	$1.24 \cdot 10^{-3}$	$6.12 \cdot 10^{-3}$	$1.12 \cdot 10^{-3}$	$1.14 \cdot 10^{-3}$	$5.29 \cdot 10^{-3}$
25	$2.93 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$	$9.35 \cdot 10^{-3}$	$2.49 \cdot 10^{-3}$	$2.46 \cdot 10^{-3}$	$7.54 \cdot 10^{-3}$	$2.18 \cdot 10^{-3}$	$2.29 \cdot 10^{-3}$	$6.54 \cdot 10^{-3}$
81	$7.97 \cdot 10^{-3}$	$7.98 \cdot 10^{-3}$	$1.44 \cdot 10^{-2}$	$9.59 \cdot 10^{-3}$	$7.85 \cdot 10^{-3}$	$1.32 \cdot 10^{-2}$	$6.57 \cdot 10^{-3}$	$7.1 \cdot 10^{-3}$	$1.09 \cdot 10^{-2}$
289	$4.43 \cdot 10^{-2}$	$4.48 \cdot 10^{-2}$	$5.26 \cdot 10^{-2}$	$2.56 \cdot 10^{-2}$	$2.57 \cdot 10^{-2}$	$3.19 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$	$2.48 \cdot 10^{-2}$	$3.02 \cdot 10^{-2}$
1,089	$9.4 \cdot 10^{-2}$	$9.33 \cdot 10^{-2}$	$9.06 \cdot 10^{-2}$	$9.37 \cdot 10^{-2}$	$9.36 \cdot 10^{-2}$	$9.94 \cdot 10^{-2}$	$8.97 \cdot 10^{-2}$	$8.49 \cdot 10^{-2}$	0.1
4,225	0.26	0.27	0.25	0.4	0.4	0.39	0.42	0.39	0.42
16,641	0.67	0.7	0.65	1.65	1.64	1.66	1.64	1.55	1.61
66,049	2.22	2.38	2.2	6.64	6.58	6.86	6.67	7.07	6.85

Table 2: Case 1, Mv, Comparison of sparse-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
2	$4.81 \cdot 10^{-4}$	$1.18 \cdot 10^{-3}$	0.28	$4.11 \cdot 10^{-3}$	$3.45 \cdot 10^{-3}$	0.29	$2.47 \cdot 10^{-3}$	$1.64 \cdot 10^{-3}$	0.27
4	$6.97 \cdot 10^{-4}$	$1.57 \cdot 10^{-3}$	0.28	$3.96 \cdot 10^{-3}$	$4.11 \cdot 10^{-3}$	0.28	$2.72 \cdot 10^{-3}$	$1.47 \cdot 10^{-3}$	0.27
8	$1.38 \cdot 10^{-3}$	$2.25 \cdot 10^{-3}$	0.27	$4.6 \cdot 10^{-3}$	$4.67 \cdot 10^{-3}$	0.28	$3.51 \cdot 10^{-3}$	$1.91 \cdot 10^{-3}$	0.28
16	$4.21 \cdot 10^{-3}$	$4.57 \cdot 10^{-3}$	0.28	$5.81 \cdot 10^{-3}$	$6.24 \cdot 10^{-3}$	0.29	$4.62 \cdot 10^{-3}$	$3.48 \cdot 10^{-3}$	0.28
32	$1.59 \cdot 10^{-2}$	$1.64 \cdot 10^{-2}$	0.29	$1.49 \cdot 10^{-2}$	$1.37 \cdot 10^{-2}$	0.28	$9.74 \cdot 10^{-3}$	$7.49 \cdot 10^{-3}$	0.28
64	$6.89 \cdot 10^{-2}$	$6.96 \cdot 10^{-2}$	0.35	$2.97 \cdot 10^{-2}$	$3.24 \cdot 10^{-2}$	0.3	$2.87 \cdot 10^{-2}$	$2.59 \cdot 10^{-2}$	0.3
128	0.33	0.31	0.6	0.11	0.11	0.39	$9.53 \cdot 10^{-2}$	0.11	0.37
256	1.4	1.4	1.69	0.39	0.45	0.68	0.41	0.44	0.71
512	5.95	5.84	6.2	1.44	1.59	1.83	1.65	1.64	1.94
1,024	24.4	24.2	24.6	8.13	8.06	8.28	8.28	8.37	9

Table 3: Case 2, M^3v, Comparison of full-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
9	$3.43 \cdot 10^{-3}$	$4.26 \cdot 10^{-3}$	0.29	$4.1 \cdot 10^{-3}$	$4.13 \cdot 10^{-3}$	0.28	$4.38 \cdot 10^{-3}$	$2.72 \cdot 10^{-3}$	0.28
25	$6.77 \cdot 10^{-3}$	$7.84 \cdot 10^{-3}$	0.28	$7.45 \cdot 10^{-3}$	$7.25 \cdot 10^{-3}$	0.28	$7.13 \cdot 10^{-3}$	$5.76 \cdot 10^{-3}$	0.28
81	$2.17 \cdot 10^{-2}$	$2.21 \cdot 10^{-2}$	0.3	$1.96 \cdot 10^{-2}$	$2.13 \cdot 10^{-2}$	0.3	$1.96 \cdot 10^{-2}$	$1.72 \cdot 10^{-2}$	0.3
289	0.12	0.13	0.6	$7.02 \cdot 10^{-2}$	$6.55 \cdot 10^{-2}$	0.35	$6.59 \cdot 10^{-2}$	$6.72 \cdot 10^{-2}$	0.35
1,089	0.21	0.2	0.82	0.25	0.25	0.52	0.31	0.25	0.59
4,225	0.54	0.55	1.19	1.02	1.01	1.28	1.02	1.14	1.33
16,641	1.57	1.62	2.23	4.07	4.24	4.52	4.48	4.09	4.84
66,049	5.78	6.17	6.74	17.8	18.3	18.2	20	19.1	21.1

Table 4: Case 2, M^3v, Comparison of sparse-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
2	$4.14 \cdot 10^{-4}$	$3.42 \cdot 10^{-3}$	0.62	$2.23 \cdot 10^{-3}$	$3.42 \cdot 10^{-3}$	0.63	$2 \cdot 10^{-3}$	$9.01 \cdot 10^{-4}$	0.63
4	$5.42 \cdot 10^{-4}$	$3.68 \cdot 10^{-3}$	0.64	$2.9 \cdot 10^{-3}$	$3.92 \cdot 10^{-3}$	0.63	$2.13 \cdot 10^{-3}$	$9.29 \cdot 10^{-4}$	0.63
8	$1.03 \cdot 10^{-3}$	$4.55 \cdot 10^{-3}$	0.65	$4.16 \cdot 10^{-3}$	$4.22 \cdot 10^{-3}$	0.63	$2.65 \cdot 10^{-3}$	$1.25 \cdot 10^{-3}$	0.74
16	$2.72 \cdot 10^{-3}$	$5.53 \cdot 10^{-3}$	0.63	$6.45 \cdot 10^{-3}$	$5.52 \cdot 10^{-3}$	0.64	$3.88 \cdot 10^{-3}$	$2.52 \cdot 10^{-3}$	0.64
32	$1.06 \cdot 10^{-2}$	$1.32 \cdot 10^{-2}$	0.66	$1.05 \cdot 10^{-2}$	$1.04 \cdot 10^{-2}$	0.64	$8.67 \cdot 10^{-3}$	$5.34 \cdot 10^{-3}$	0.65
64	$4.56 \cdot 10^{-2}$	$4.7 \cdot 10^{-2}$	0.68	$2.93 \cdot 10^{-2}$	$2.22 \cdot 10^{-2}$	0.66	$2.74 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	0.68
128	0.22	0.22	0.85	0.1	$7.06 \cdot 10^{-2}$	0.72	0.1	$7.66 \cdot 10^{-2}$	0.71
256	0.94	0.95	1.59	0.45	0.31	0.94	0.43	0.27	0.96
512	3.93	4.08	4.61	1.45	1	1.66	1.98	1.18	1.79
1,024	16.2	16	17.3	7.86	5.3	6.12	8.24	5.6	6.43

Table 5: Case 3, $(M + 3Id)Mv$, Comparison of full-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
9	$2.56 \cdot 10^{-3}$	$5.39 \cdot 10^{-3}$	0.65	$3.41 \cdot 10^{-3}$	$4.39 \cdot 10^{-3}$	0.63	$3.37 \cdot 10^{-3}$	$1.72 \cdot 10^{-3}$	0.63
25	$4.73 \cdot 10^{-3}$	$7.39 \cdot 10^{-3}$	0.64	$6.06 \cdot 10^{-3}$	$6.03 \cdot 10^{-3}$	0.64	$6.27 \cdot 10^{-3}$	$3.93 \cdot 10^{-3}$	0.63
81	$1.45 \cdot 10^{-2}$	$1.69 \cdot 10^{-2}$	0.65	$1.83 \cdot 10^{-2}$	$1.53 \cdot 10^{-2}$	0.66	$1.8 \cdot 10^{-2}$	$1.22 \cdot 10^{-2}$	0.65
289	$8.61 \cdot 10^{-2}$	$8.48 \cdot 10^{-2}$	1.13	$6.69 \cdot 10^{-2}$	$5.15 \cdot 10^{-2}$	0.68	$7.36 \cdot 10^{-2}$	$5.44 \cdot 10^{-2}$	0.69
1,089	0.16	0.16	1.37	0.26	0.19	0.83	0.27	0.18	0.82
4,225	0.4	0.42	1.59	1.07	0.72	1.37	1.05	0.7	1.43
16,641	1.1	1.18	2.49	4.21	2.98	3.69	4.57	3.09	3.93
66,049	3.96	4.16	5.74	18.3	13.1	14.1	20.6	13.2	14.9

Table 6: Case 3, $(M + 3Id)Mv$, Comparison of sparse-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
2	$3.8 \cdot 10^{-4}$	$8.12 \cdot 10^{-4}$	$2.81 \cdot 10^{-2}$	$2.01 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	$6.98 \cdot 10^{-2}$	$1.52 \cdot 10^{-3}$	$7.49 \cdot 10^{-4}$	$6.61 \cdot 10^{-2}$
4	$4.69 \cdot 10^{-4}$	$8.68 \cdot 10^{-4}$	$2.78 \cdot 10^{-2}$	$3.64 \cdot 10^{-3}$	$1.64 \cdot 10^{-3}$	$7.21 \cdot 10^{-2}$	$1.92 \cdot 10^{-3}$	$1.87 \cdot 10^{-3}$	$6.76 \cdot 10^{-2}$
8	$8.56 \cdot 10^{-4}$	$1.14 \cdot 10^{-3}$	$3.03 \cdot 10^{-2}$	$2.73 \cdot 10^{-3}$	$1.98 \cdot 10^{-3}$	$7.05 \cdot 10^{-2}$	$2.05 \cdot 10^{-3}$	$9.84 \cdot 10^{-4}$	$6.79 \cdot 10^{-2}$
16	$2.36 \cdot 10^{-3}$	$1.95 \cdot 10^{-3}$	$3.17 \cdot 10^{-2}$	$2.99 \cdot 10^{-3}$	$2.7 \cdot 10^{-3}$	$7.09 \cdot 10^{-2}$	$2.62 \cdot 10^{-3}$	$1.45 \cdot 10^{-3}$	$6.83 \cdot 10^{-2}$
32	$5.68 \cdot 10^{-3}$	$5.9 \cdot 10^{-3}$	$3.36 \cdot 10^{-2}$	$5.68 \cdot 10^{-3}$	$5.12 \cdot 10^{-3}$	$7.37 \cdot 10^{-2}$	$4.68 \cdot 10^{-3}$	$3.14 \cdot 10^{-3}$	$7.25 \cdot 10^{-2}$
64	$2.33 \cdot 10^{-2}$	$2.42 \cdot 10^{-2}$	$5.24 \cdot 10^{-2}$	$1.3 \cdot 10^{-2}$	$1.3 \cdot 10^{-2}$	$8.65 \cdot 10^{-2}$	$1.12 \cdot 10^{-2}$	$9.75 \cdot 10^{-3}$	$8.57 \cdot 10^{-2}$
128	0.11	0.11	0.14	$3.81 \cdot 10^{-2}$	$4.56 \cdot 10^{-2}$	0.11	$3.51 \cdot 10^{-2}$	$3.48 \cdot 10^{-2}$	0.11
256	0.48	0.48	0.5	0.15	0.15	0.22	0.15	0.14	0.22
512	2.26	1.95	2.02	0.52	0.54	0.58	0.53	0.53	0.6
1,024	8	8.09	8.09	2.62	2.56	2.71	2.79	2.71	2.89

Table 7: Case 4, $M(x + y + z)$, Comparison of full-matrix benchmarks.

Size	deal.ii	LO+deal.ii	PO+deal.ii	Eigen	LO+Eigen	PO+Eigen	Blaze	LO+Blaze	PO+Blaze
9	$1.56 \cdot 10^{-3}$	$1.87 \cdot 10^{-3}$	$2.95 \cdot 10^{-2}$	$2.24 \cdot 10^{-3}$	$1.92 \cdot 10^{-3}$	$6.84 \cdot 10^{-2}$	$2.35 \cdot 10^{-3}$	$1.22 \cdot 10^{-3}$	$6.78 \cdot 10^{-2}$
25	$2.58 \cdot 10^{-3}$	$3.03 \cdot 10^{-3}$	$2.98 \cdot 10^{-2}$	$3.55 \cdot 10^{-3}$	$3.16 \cdot 10^{-3}$	$7.49 \cdot 10^{-2}$	$3.48 \cdot 10^{-3}$	$2.47 \cdot 10^{-3}$	$7.43 \cdot 10^{-2}$
81	$7.72 \cdot 10^{-3}$	$8.23 \cdot 10^{-3}$	$3.71 \cdot 10^{-2}$	$9.22 \cdot 10^{-3}$	$8.58 \cdot 10^{-3}$	$7.82 \cdot 10^{-2}$	$8.06 \cdot 10^{-3}$	$6.64 \cdot 10^{-3}$	$7.6 \cdot 10^{-2}$
289	$4.62 \cdot 10^{-2}$	$4.79 \cdot 10^{-2}$	$9.16 \cdot 10^{-2}$	$2.77 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	$9.71 \cdot 10^{-2}$	$2.47 \cdot 10^{-2}$	$2.49 \cdot 10^{-2}$	$9.71 \cdot 10^{-2}$
1,089	0.12	0.12	0.2	0.11	0.11	0.18	$9.47 \cdot 10^{-2}$	0.11	0.19
4,225	0.32	0.32	0.39	0.44	0.43	0.5	0.41	0.4	0.51
16,641	0.84	0.84	0.92	1.83	1.8	1.89	1.72	1.74	1.97
66,049	2.67	2.7	2.82	7.5	7.72	7.77	7.23	7.39	8.02

Table 8: Case 4, $M(x + y + z)$, Comparison of sparse-matrix benchmarks.

References

[1] Standard for Programming Language C++, 2011. ISO/IEC 14882:2011.

[2] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, and B. Turcksin. The deal.ii library, version 8.3. *Archive of Numerical Software*, 4(100):1–11, 2016.

[3] M. Benzi and A. J. Wathen. *Some Preconditioning Techniques for Saddle Point Problems*, volume 13 of *Mathematics in Industry*, pages 195–211. Springer Berlin Heidelberg, 2008.

[4] K. Budge. C++ optimization and excluding middle-level code. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 107–121, 1994.

[5] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[6] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[7] M. Maier, M. Bardelloni, and L. Heltai. LinearOperator Benchmarks, Version 1.0.0, Mar. 2016. http://dx.doi.org/10.5281/zenodo.47202.

[8] D. R. Musser and A. A. Stepanov. Generic programming. volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 1989.

[9] The deal.II Authors. step-32. https://www.dealii.org/developer/doxygen/deal.II/step_32.html.

[10] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):280–305, 1995.