Computers and Mathematics with Applications 72 (2016) 1–24

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

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

Matthias Maier [a], Mauro Bardelloni [b], Luca Heltai [b,*]

[a] School of Mathematics, University of Minnesota, 127 Vincent Hall, 206 Church Street SE, Minneapolis, MN 55455, USA
[b] SISSA - International School for Advanced Studies, Via Bonomea 265, 34136 Trieste, Italy

ARTICLE INFO

Article history:
Received 21 October 2015
Received in revised form 4 April 2016
Accepted 16 April 2016
Available online 17 May 2016

Keywords:
Expression templates
Linear algebra
High performance computing

ABSTRACT

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 avoid 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 [1,2] are well known optimization techniques 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.

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` [3], or `Blaze` [4].

* Corresponding author.
E-mail addresses: msmaier@umn.edu (M. Maier), mauro.bardelloni@sissa.it (M. Bardelloni), luca.heltai@sissa.it (L. Heltai).

http://dx.doi.org/10.1016/j.camwa.2016.04.024
0898-1221/

Although expression templates offer a good incarnation of the "generic programming" paradigm [5], they are difficult techniques to master, that are not easily adapted to existing numerical libraries, or *collections* of libraries, and they have 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 [6] 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 [7,8]. 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 [9].

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 – besides the possibility to use them in a call to `vmult` – is:

```
1  class Vector
2  {
3    typedef double number_type;
4
```