SCUOLA INTERNAZIONALE SUPERIORE
DI STUDI AVANZATI

Mathematics Area

Master in High Performance Computing

High Performance Programming Paradigms

Applied to Computational Fluid Dynamic Simulations

Advisor:

Dott. Luca Heltai

Candidate:

Mauro Bardelloni

2014-2015

Contents

Introduction

Most of the time spent to solve a numerical problem is taken by a linear system. There are cases in which the solution is straightforward: lower triangular, upper triangular or diagonal matrices. However, in every day life, solutions of linear systems are far for being trivial: usually partial differential equation (PDE) problems lead to complicated linear systems and it could be even worse when you have to deal with a system of PDEs.

This is the situation where HPC is not enough and the starting point of my thesis. Literature and software libraries provide a huge amount of resources and it is very unlikely that you could do better than thousand of scientists and billions of lines of code improved for years. You might improve slightly your code, and not more but if you want to significantly improve your code, you will have to take into account something else: what you need is a new *High Performance Point of View*.

1. Preconditioner

Experience proves that almost every matrix leads to a difficult linear system: there is no nice linear system that you can assure it is easy or fast to solve. For example, consider the following matrix [38]:

$$[A]_{i,j} = \begin{cases} a_i, & \text{if } i > j \\ a_0, & \text{if } i = j \\ a_j, & \text{if } i < j \end{cases} \tag{1}$$

where a_i are arbitrary real numbers and $i, j \in \{0, \dots, n-1\}$. Given $b \in \mathbb{R}^n$, we are interested in finding a solution $x \in \mathbb{R}^n$ to the equation:

$$Ax = b. \tag{2}$$

The fastest known stable factorization direct methods for solving such a linear system requires $\mathcal{O}(n^2)$ floating point computations. Consider that the well-known Gauss-Elimination algorithm requires $\mathcal{O}(n^3)$.

A real improvement to the convergence of (2) is given by Strang in [29]. He noticed that the matrix (1) could be approximated with a matrix $\tilde{A}$ that leads to an equivalent system of (2). The resulting system is solvable in $\mathcal{O}(n \log n)$ operations. The idea of Stang is to multiply Ax and b for the same non degenerate square matrix: this is the idea behind the notion of *preconditioner*:

(1.1) **DEFINITION:** A *preconditioner* P is a non-degenerate square matrix such that

$$P^{-1}Ax = P^{-1}b$$

is a "simpler" linear system than $Ax = b$.

Indeed, the problem becomes the right choice of a preconditioner: we need a preconditioner that is cheap to compute and that at the same time is a good approximation of the inverse matrix. The literature is full of recipe about the choice of the best preconditioner (see [38] for a review) and in many cases it is nothing but a mix of blocks obtained from the original system matrix and cooked in the right doses.

2. New landscape, new spyglass

Notice that in Definition 1.1 we are actually interested in the *action* of P^{-1} on a vector: when we use an iterative solver, we do not need to know the value of $P^{-1}A$ but $P^{-1}(Ax)$. In term of math, this means that we should talk about *linear operators* instead of *matrices.* This is a real change of point of view and it is something that it is not fully considered in the classical programming languages: we would like to implement the *action* of *multiplication*, *transpose-multiplication*, *sum and multiplication*, and *sum and transpose-multiplication* without *physically* assembling any object.
Consider the case where we have a matrix A and a matrix B and our problem requires $A * B$. Probably, one would assemble $C = A * B$ and then would use this new object. A finer analysis shows that we could multiply for B and then for A every time without assembling C: ideally, a `LinearOperator` class should do that without having to explicitly write any additional code and without loss of performances. The tools we choose to solve this problem are the so called *lambda expressions*: these objects are designed in order to have highest level of ductility and performance (see [36] and [30] for an extensive explanation).
In this thesis we show how to implement a linear algebra class based on *lambda expressions* objects (Chapter I) and we apply this class to PDE problem. Furthermore, we provide benchmarks that show no loss of performances and a working implementation of a PDE solver that shows the power of this syntax (Chapter III).

3. Do not reinvent the wheel

Our intent is to use existing libraries as much as possible. First of all, we need reliable open source libraries: for instance `deal.II`. This is a strengthened library for finite element with several developer that use it for scientific research. Moreover, speaking about parallelism, `deal.II` has its own interface for *Trilinos* and *Petsc* libraries.

`deal.II` is a very versatile and powerful tool but it has a drawback: in the case you are interested in solving several problems and they are similar to each other you have to rewrite a lot of lines of code changing few words. This consideration led us to write `deal2lkit` ([27] and Chapter II).

`deal2lkit` is a wrapper of classes of `deal.II` provided with a powerful feature of parsing parameter file: user can change problem parameters without rebuilding the project. This means flexibility and time saving.

Our goal is a multiphysics solver. We use the tools developed in the class `LinearOperator` and the structure of `deal2lkit` to write a high level library named π-`DoMUS` (Chapter III).

Using π-`DoMUS` (Parallel Deal.II MUltiphysics Solver) we are able to write a solver for every common PDE and to recover all the previous listed features (performance and fexibility).

The intent of this thesis is to show an *HPP* (High Performance Programmin) interface that is able to improve performance in the case *HPC* is not enough.

4. Real life problems

Is *HPP* really necessary? The Millennium Prize Problems are seven problems in mathematics that were stated by the Clay Mathematics Institute in 2000. As of November 2015, six of the problems remain unsolved. A correct solution to any of the problems results in a US \$1,000,000 prize (sometimes called a Millennium Prize) being awarded by the institute. One of these problems is the existence and smoothness of the $3D$ Navier–Stokes equations. The equations can be stated as follows:

$$\begin{cases} \rho\frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v}\cdot\nabla)\mathbf{v} = -\nabla p + \nu\Delta\mathbf{v} + \mathbf{f}(\boldsymbol{x},t) \\ \operatorname{div} v = 0. \end{cases}$$

where v is the velocity field, p the pressure, and $\mathbf{f}$ represents the external forces.

Even if no existence of solutions has been already proved, Navier-Stokes equation is fundamental for its applications: shipbuilding, airplane, and also cars. Therefore, the numerical simulation of its behavior is necessary. We can represent its discretization as following:

$$\begin{pmatrix} F & B^t \\ B & 0 \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \tag{3}$$

where F stands for $\rho\frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v}\cdot\nabla)\mathbf{v} - \nu\Delta\mathbf{v}$, B^t for $-\nabla p$, and B for $\operatorname{div} v$.

Practice shows that the convergence of (3) depends strongly on ν, ρ, grid size, time discretization, initial condition, and boundary condition. Moreover, the term $(\mathbf{v}\cdot\nabla)\mathbf{v}$ enormously complicates the study of the solution.

A preconditioner for such a problem is really far from being trivial. A lot of papers are devoted to this goal and research is still in progress. A common choice is to use

$$\begin{pmatrix} A^{-1} & 0 \\ S^{-1}BA^{-1} & -S^{-1} \end{pmatrix}$$

where $S := BA^{-1}B^t$, as preconditioner for (3). In many applications the crucial choice for the preconditioner is transferred to S. There are a wide range of possibilities, the of them are listed below (see [13] for more details):

- SIMPLE
$$S^{-1} \approx -(B\hat{A}^{-1}B^t)^{-1}$$
  $\hat{A}$ an approximation of A (usually $\hat{A} = \operatorname{diag}(A)$)
- BFBt [14]
$$S^{-1} \approx -(BM^{-1}B^t)^{-1}(BM^{-1}AM^{-1}B^t)(BM^{-1}B^t)^{-1}$$
  where M is I, $\operatorname{diag} A$, or X (Mass matrix on the velocity space)
- Olshanskii's preconditioner [26]
$$S^{-1} \approx -Q^{-1}BL^{-1}AL^{-1}B^tQ^{-1}$$
  where L is the Laplacian matrix and Q the mass matrix.

In this thesis we are going to show how the research of a good preconditioner like these could be tackled implementing a code (Chapter III) able to rapidly tests a wide combination of ingredients (Chapter III). This result is obtained thanks to the concept of `LinearOperator` class that ensure performance and flexibility (Chapter I).

LinearOperator Class I

I.1. Introduction

Expression templates [11, 37] are a well known optimization technique to avoid the creation of large, temporary objects in arithmetic expressions. The idea 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 extend, an example is the C++ linear-algebra library `Eigen` [15].

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 "template overhad" 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 very generic, high-level interface is required (see Section I.2). In particular, we don't 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 to encapsulate a wide variety of concrete matrix and vector implementations.

The expression syntax is developed within the framework of the finite-element library deal.II [7] and has been added to the library starting from version 8.3 [6]. 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, which could be readily adjusted with very minor changes to any concrete choice of naming and call signature.

The Chapter is structured as follows. In section I.2 we define the vector, matrix and solver interfaces that are used. Then, in the following two sections, 3 and 4, the `LinearOperator` and `PackagedOperation` class are defined. Implementations aspects for vector space operations are discussed and a generic strategy for encapsulating concrete matrix objects into the `LinearOperator` framework is given. Section I.5 presents two detailed examples, an application of the `LinearOperator` to prescribe Dirichlet boundary conditions with an operator (and without manipulating the underlying matrix object); and an implementation of a preconditioner for the Stokes problem. Short performance comparisons are given that show that the overhead introduced by `std::function` objects is negligible. We draw some conclusions in Section I.6.

I.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` concept. 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`:

```cpp
class Matrix {
  template<typename Vector>
  void vmult(Vector &dst, const Vector &src);

  template<typename Vector>
  void vmult_add(Vector &dst, const Vector &src);

  template<typename Vector>
  void Tvmult(Vector &dst, const Vector &src);

  template<typename Vector>
  void Tvmult_add(Vector &dst, const Vector &src);
};
```

Here, `Tvmult` applies the corresponding transpose matrix vector multiplication and the variants with `vmult_add` and `Tvmult_add` add 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. 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:

```cpp
class Vector {
  typedef ... number_type;

  Vector &operator=(const Vector &);
  Vector &operator=(number_type);

  Vector &operator+=(const Vector &);
  Vector &operator-=(const Vector &);
  Vector &operator*=(number_type);
  Vector &operator/=(number_type);
};
```

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

```
v = 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 behavior.

Another design decision that becomes apparent in 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:

```
template<typename Vector>
class Solver {
  template<typename Matrix, typename Preconditioner>
  solve(const Matrix          &A,
        Vector                &x,
        const Vector          &b,
        const Preconditioner &prec);
};
```

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 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.

I.3. A LINEAR OPERATOR CLASS

The solver interface introduced in the previous section is very 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 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.

I.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 transfered immediately: a linear operator has a notion of applying itself (`vmult`), or its transposed operation (`Tvmult`). 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` and `Tvmult` variants and consequently support multiple range and domain vector types.

The question arises (at least from an implementational standpoint) which strategy to 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:

```
template <typename Range, typename Domain>
class LinearOperator
{
public:
  std::function<void(Range  &v, const Domain &u)> vmult;
  std::function<void(Range  &v, const Domain &u)> vmult_add;
  std::function<void(Domain &v, const Range  &u)> Tvmult;
  std::function<void(Domain &v, const Range  &u)> Tvmult_add;

  std::function<void(Range  &v, bool fast)> reinit_range_vector;
  std::function<void(Domain &v, bool fast)> reinit_domain_vector;

  // ...
};
```

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`. The boolean `fast` is an implementational detail that controls whether the vector in question is also zeroed out while it is resized (`fast == false`), or not.

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 I.3.3:

```
template <typename Range, typename Domain> class LinearOperator
{
public:
  // ...

  LinearOperator();
  LinearOperator(const LinearOperator<Range, Domain> &) = default;
  template<typename Op> LinearOperator(const Op &op)
  {
    *this = linear_operator<Range, Domain, Op>(op);
  }

  LinearOperator<Range, Domain> &
  operator=(const LinearOperator<Range, Domain> &) = default;

  template <typename Op>
  LinearOperator<Range, Domain> &operator=(const Op &op)
  {
    *this = linear_operator<Range, Domain, Op>(op);
    return *this;
  }
};
```

I.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:

```
template <typename Range, typename Intermediate, typename Domain>
LinearOperator<Range, Domain>
```

```
operator*(const LinearOperator<Range, Intermediate> &first_op,
          const LinearOperator<Intermediate, Domain> &second_op)
{
  LinearOperator<Range, Domain> return_op;

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

  return_op.vmult = [first_op, second_op](Range &v, const Domain &u)
  {
    GrowingVectorMemory<Intermediate> vector_memory;

    Intermediate *i = vector_memory.alloc();
    second_op.reinit_range_vector(*i, /*bool fast =*/ true);
    second_op.vmult(*i, u);
    first_op.vmult(v, *i);
    vector_memory.free(i);
  };

  // etc.

  return return_op;
}
```

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 straightforward 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);
  };

  // etc...

  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.

I.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);
  }
```

Remark. Wrapping in a lambda function as opposed to a direct assignment (`op.vmult = matrix.vmult`) ensures that the linear operator wrapper is compatible with a wide variety of templated or non-templated function signatures.

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 provide 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, bool fast)
    {
      v.reinit(matrix.m(), fast);
    }

    // ...
  };
} /* namespace internal */

  // in linear_operator:

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

This allows to specialize 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,
                              bool fast)
    {
      // 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 a *constexpr* `true` if the addresses of u and v are the same, otherwise it return `false`.

I.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 *null operator*:

```
operator+(const LinearOperator<Range, Domain> &first_op,
          const LinearOperator<Range, Domain> &second_op)
{
  if (first_op.is_null_operator)
    return second_op;
 if (second_op.is_null_operator)
    return first_op;

  // Do the general case here
  // ...

}
```

The complete implementation of the `null_operator` simply provides `vmult` and `Tvmult` methods that zero out the destination vector, while `vmult_add` and `Tvmult_add` leave the `Range` vector untouched:

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

  return_op.is_null_operator = true;

  return_op.vmult = [](Range &v, const Domain &u)
  {
    v = 0.;
  };

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

  return_op.Tvmult = [](Range &v, const Domain &u)
  {
    v = 0.;
  };

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

  return return_op;
}
```

I.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:

```
template <typename Range, typename Domain>
class BlockLinearOperator : public LinearOperator<Range, Domain>
{
public:
  // ...

  typedef LinearOperator<typename Range::BlockType, typename Domain::BlockType>
      BlockType;

  std::function<unsigned int()> n_block_rows;
  std::function<unsigned int()> n_block_cols;
  std::function<BlockType(unsigned int, unsigned int)> block;
};
```

We provide helper functions which fills the above functions starting from standard deal.II block matrices, as well as from arrays of linear operators:

```
// An m by n block sparse matrix
BlockSpaseMatrix<double> A(m,n);
...
auto B = block_operator(A);

// Now we can access each sub-block as an individual
// linear operator:

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

// A block operator encapsulating two sub-blocks of A
auto subA = block_operator({{B00}, {B10}});
```

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.

Also this operator makes heavy use of `std::function` objects and lambda functions. Such a flexibiliy 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 1000×1000). Section I.5.2 analyses in details the run-time penalty associated with such objects, and shows its full potential in writing block based preconditioners for complex partial differential equations.

I.4. A PackagedOperation

In this section a further generalization of the linear operator concept shall be discussed 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 will define 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, bool fast)> 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. An implicit conversion operator can be defined that automatically converts the packaged operation to its result such that 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, /*bool fast=*/ true);
    apply(result_vector);
    return result_vector;
  }
};
```

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 straightforward:

```
template <typename Range, typename Domain>
PackagedOperation<Range>
operator*(const LinearOperator<Range, Domain> &op,
          const Domain &u)
{
  PackagedOperation<Range> return_comp;

  return_comp.reinit_vector = op.reinit_range_vector;

  return_comp.apply = [op, &u](Range &v)
  {
    op.vmult(v, u);
  };

  // ...

  return return_comp;
}
```

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

```
template <typename Range>
PackagedOperation<Range> operator-(const Range &offset,
                                   const PackagedOperation<Range> &comp)
{
  PackagedOperation<Range> return_comp;

  return_comp.reinit_vector = comp.reinit_vector;

  return_comp.apply = [&offset, comp](Range &v)
  {
    comp.apply(v);
    v *= -1.;
    v += offset;
  };

  // ...

  return return_comp;
}
```

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. With this implementation, 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;
```

I.5. Examples

This section contains two examples that use the new `LinearOperator` concept. First, a strategy to wrap constraints around a matrix object is presented that does not need access to the elements of the underlying matrix object, permitting the use of abstract linear operators with constraints. The second example is a Schur complement preconditioner for the Stokes problem. Finally, performance comparisons between optimized, hand-written implementations, and an implementation with `LinearOperator` are made.

I.5.1. FORMULATING LINEAR CONSTRAINTS WITH A `LINEAROPERATOR`

In finite element codes, it is often necessary to modify the resulting linear system in such a way that certain constraints are satisfied, e. g., when imposing Dirichlet boundary conditions or when solving the problem in a geometry with hanging nodes.

The classical way to proceed when there are degrees of freedom which are constrained (see, e. g., [28]) is to eliminate them during the assembly process, either by removing them altogether from the resulting linear system, or by replacing their rows and columns by appropriate replacements.

The alternative we present here consists in imposing the constraints as a linear operator and not by manipulating the original matrix.

Remark. This has the positive side effect that the matrix object (that should be wrapped) does not need to be a classical matrix. It can also be a linear operator object with no direct access to the underlying storage.

This is especially important in those situations in which the matrix is never stored, and it is not even possible to construct a single specific entry of the matrix itself. A notable example is when the matrix vector product may be replaced by a sequence of approximate operations which reproduce to a certain accuracy the original matrix-vector product, but do not give access to the elements of the matrix itself. This is the case in Boundary Element Methods, where there are several techniques to reduce the count of operations for a full matrix vector product from n^2 to $n\log(n)$ or even lower counts, generally known as Fast Multiple Methods, which exploit such approximations without ever forming a full matrix.

In such cases, the procedure we devise here is the only available option, since the only interface we have with the matrix is the matrix-vector operation `vmult` (or its transpose `Tvmult`).

Algebraic formulation of linear constraints

Consider a generic linear system of the form

$$Ax = b, \tag{I.1}$$

with a system matrix $A \in \mathcal{M}at(n,n)$ and a right hand side $b \in \mathbb{R}^n$.

In many numerical methods it is often convenient to assemble the matrix A without taking into account additional constraints on the degrees of freedom, such as, for example, Dirichlet boundary conditions or continuity conditions across faces with hanging nodes in finite element methods. Such constraints can be later removed from the system matrix by modifying the rows and columns of A and b.

Let us assume that of the n degrees of freedom, m are constrained, with linear constraints of the following type:

$$\sum_j \tilde{\alpha}_j^k x_j + \tilde{\kappa}_j = 0, \quad k = 1, \dots, m, \tag{I.2}$$

with coefficients $\tilde{\alpha}_j^k, \tilde{\kappa}_j \in \mathbb{R}$. Without loss of generality, one can renumber and renormalise the constraints in such a way that all constrained degrees of freedom appear first, and that for the k-th linear constraint we have that $\tilde{\alpha}_k^k \neq 0$. If the set of constrained degrees of freedom is well-formed, then one can eliminate them from the linear system [28], and they can be rewritten in the following form:

$$x_k = \sum_{j>m} \alpha_j^k x_j + \kappa_j, \quad k = 1, \dots, m, \tag{I.3}$$

where, in the simple case where the constrained degrees of freedom do not depend on each other, $\alpha_j^k = -\tilde{\alpha}_j^k / \tilde{\alpha}_k^k$ and $\kappa_j = \tilde{\kappa}_j / \tilde{\alpha}_k^k$. The more complex case of *circulant* contraints, in which j may also be smaller than m in equation (I.3), can be simplified through trivial manipulation to the same form, when the constraints are well formed.

We denote the vector of *constrained degrees of freedom* by $v_c = \{x_k\}_{k=1}^m$. Those degrees of freedom can be fully expressed in terms of the remaining degrees of freedom, as in equation (I.3). Conversely, $v_n = \{x_k\}_{k=m+1}^n$ shall denote the vector of *unconstrained degrees of freedom* and do not appear on the left side.

Let α denote the $m \times (n-m)$ - matrix $\{\alpha_j^k\}_{jk}$ appearing in equation (I.3). If a vector v satisfies the linear constraints (I.3), then we can rewrite them in two equivalent matrix forms:

$$\begin{pmatrix} v_c \\ v_n \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & \alpha \\ 0 & I \end{pmatrix}}_{=:C} \cdot \begin{pmatrix} v_c \\ v_n \end{pmatrix} + \underbrace{\begin{pmatrix} \kappa \\ 0 \end{pmatrix}}_{:=\kappa'}. \tag{I.4}$$

$$\underbrace{\begin{pmatrix} I & -\alpha \\ 0 & 0 \end{pmatrix}}_{=:B=I-C} \cdot \begin{pmatrix} v_c \\ v_n \end{pmatrix} = \begin{pmatrix} \kappa \\ 0 \end{pmatrix}. \tag{I.5}$$

We observe that the *affine operator* $P : v \to Cv + \kappa'$ is idempotent, i.e., $P^2 = P$, since P always leaves untouched the unconstrained degrees of freedom v_n and sets the constrained ones to $\alpha v_n + \kappa$. If a vector x is such that it satisfies the constraints (I.3), then $P(x) = x$. Moreover, the matrices B and C are such that $BC = C^t B^t = 0$.

Exploiting these properties, we can use the matrix $B = I - C$ to reformulate the solution of system (I.1) subject to the constraints (I.3) using a Lagrange multiplier λ as follows:

$$\begin{cases} Ax + B^t \lambda = b, \\ Bx = \kappa'. \end{cases} \tag{I.6}$$

Since by construction the solution x of system (I.6) satisfies the constraints $Bx = \kappa'$, we can replace x with $P(x)$ and multiply by C^t to obtain a simplified system on the variable x only

$$C^t AC\, x = C^t\,(b - A\,\kappa'). \tag{I.7}$$

Unfortunately system (I.7) is underdetermined, since $C^t AC$ has rank $(m-k)$ (its first k rows are all zero). This is expected, since we removed the Lagrange multiplier λ from (I.6) when we multiplied by C^t, and uniqueness of a solution was guaranteed by the presence of the Lagrange multiplier.

Observing more closely the structure of $C^t AC$, we can recover the solution of system (I.6) by summing a matrix with complementary rank, for example

$$S := C^t AC + I_c, \qquad C^t AC := \begin{pmatrix} 0 & 0 \\ 0 & (C^t AC)_{nn} \end{pmatrix}, \qquad I_c := \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix}. \tag{I.8}$$

Then there exist a unique solution x to the system

$$S\tilde{x} = C^t\,(b - A\,\kappa') \tag{I.9}$$

$$x = C\tilde{x} + \kappa', \tag{I.10}$$

and it coincides with the solution x of system (I.6).

The deal.II library provides functions which assemble directly the matrix S and the right hand side in (I.9), through its `ConstraintMatrix` class, and then *distribute* the constraints to the final solution through equation (I.10).

An implementation of a similar procedure (which only uses knowledge about the action of the matrices C and A) would require two linear operator objects plus two packaged operation objects, one for the right hand side of equation (I.9) and one for the right hand side of equation (I.10).

Implementation with `LinearOperator` and `PackagedOperation`

For this example, we will compare with the specific implementation of the `ConstraintMatrix` which can be found in `deal.II`, although our strategy will work also for other implementations.

First we construct a `LineaOperator` that encodes the action of the matrix C in equation (I.4):

```
template<typename Matrix>
LinearOperator<Range, Domain>
constraints(const ConstraintMatrix &cm, const Matrix &m);
```

Referring to `deal.II` implementation, such a matrix exists in `deal.II` in a form which is not that of a standard matrix, since its implementation is more efficient using a specialised class (called `ConstraintMatrix`) that essentially stores the coefficients of (I.4) in efficient data structures (see [28]). With it, it is straightforward to implement a function that returns C as a `LinearOperator` for a given `ConstraintMatrix`. We report only the `vmult` function implementation, as the other members follow a very similar structure:

```
  // ... constraints(cm, A) ...
  // vmult operation for a constraint matrix, available
  // as a ConstraintMatrix of deal.II (here called cm)

  return_op.vmult = [&cm](Range &v, const Domain &u)
  for (auto i : v.locally_owned_elements())
    if (cm.is_constrained(i))
    {
      const auto entries = cm.get_constraint_entries (i);
      for (types::global_dof_index j=0; j < entries->size(); ++j)
      {
        auto pos = (*entries)[j].first;
        v(i) =  u(pos) * (*entries)[j].second;
      }
    }
    else
      v(i) = u(i);
 }
```

In a very straightforward manner it is then possible to implement the I_c operator as (only vmult is shown):

```
  // ... identity_of_constraints(cm) ...
  // vmult operation for the Ic part of the constrained operator
  return_op.vmult = [&cm](Range &v, const Domain &u) {
    v = 0;
    for (auto i : v.locally_owned_elements())
      if (cm.is_constrained(i))
        v(i) = u(i);
  }
```

The solution to system (I.9) can now be expressed as

```
// Given a matrix A and the constraint matrix cm, build all operators:
const auto op_C  = constraints<Range, Domain, Matrix>(cm, A);
const auto op_Ic = identity_of_constraints<Range, Domain>(cm);
const auto op_Ct = transpose_operator<Domain, Range>(op_C);
const auto op_A  = linear_operator<Range, Domain>(A);

const auto op_S = op_Ct * op_A * op_C + Ic;

// We assume both solver and preconditioner are available
const auto op_S_inv = inverse_operator(op_S, solver, preconditioner);

// The packaged operation on the right hand side is automatically
// applied on x_tilde
Vector<double> x_tilde = op_S_inv * op_Ct*(system_rhs-op_A*kappa_prime);

// Distribute constraints to the solution
x_tilde = op_C*(x_tilde + kappa_prime);

// The above is equivalent to cm.distribute(x_tilde) in deal.II
```

The two consecutive packaged operation could have been condensed into one by using the one-liner

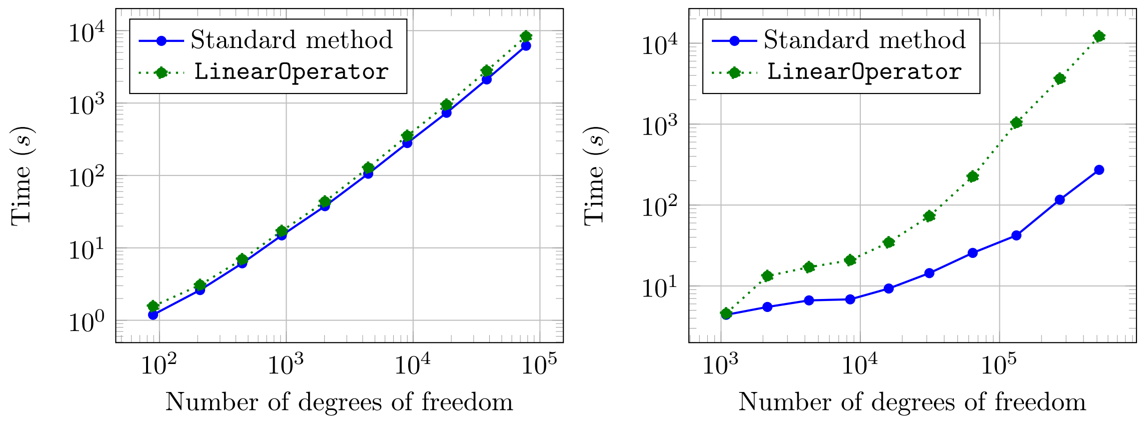
```
Vector<double> solution =
      op_C*(op_S_inv * op_Ct*(system_rhs-op_A*kappa_prime) ) + kappa_prime;
```

whose expression is identical to the mathematical formulation expressed in system (I.9) and (I.10) together.

Benchmark

We present some benchmarks obtained with the solution of a Laplace problem using Dirichlet boundary conditions and a non-uniform refinement strategy. The comparison has been made with two example test programs that can be downloaded from the deal.II library web page, one for the serial case [33] and one for the parallel case [32].

We replaced the original implementation of constraint matrix with one based on the `LinearOperator` class. The solutions are identical in the two cases, and Figure I.1 shows the different timing of the two methods in the serial and parallel case, indicating a negligible overhead for the serial case (left), and a considerable overhead for the parallel case (right).



(a) 1 CPU (b) 16 CPUs

Figure I.1: Execution time over matrix size for a Laplace problem where the Dirichlet boundary condition and the hanging node constraints are enforced by direct application to the matrix (Standard Method), as well as applied indirectly by encapsulation in a linear operator (`LinearOperator`).

In the parallel case, the overhead of the `LinearOperator` approach is due to the overhead in the communication. In the standard approach, elimination of the constraints is performed once, and no communication about constrained degrees of freedom is required after elimination as occurred. Without access to the underlying matrix structure, at every application of the constrained linear operation, a certain amount of communication is unavoidable, making this approach only useful for those cases where the matrix is not assembled or not available. In those cases the overhead is unavoidable, and the `LinearOperator` approach is the only available option.

I.5.2. A PRECONDITIONER FOR THE STOKES PROBLEM

Perhaps a better way to show the power of 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 lead to mistakes and bugs which are quite difficult to catch. In this section we provide a use case of `LinearOperator` which simplifies tremendously the readability of numerical codes, while maintaining the same performances of hand-crafted, low level, codes.

Statement of the problem

In the following we use the stationary Stokes Problem as a case study:

$$\begin{aligned} -\Delta u + \nabla p &= f, \\ \nabla \cdot u &= 0. \end{aligned} \tag{I.11}$$

The corresponding system matrix of this problem has the following structure:

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

It is well known that a good preconditioner for this system is given by (see, e. g., [9])

$$\mathbb{P} = \begin{pmatrix} A & B^t \\ 0 & -S \end{pmatrix}^{-1}, \tag{I.12}$$

where $S = BA^{-1}B^t$ is the corresponding *Schur complement* (see [9]).

We are going to focus on the action of the inverse of (I.12) 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 ansatz spaces) such that the linear system is well-posed.

A low-level implementation

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

$$\begin{pmatrix} v \\ q \end{pmatrix} = \begin{pmatrix} A & B^t \\ 0 & -S \end{pmatrix}^{-1} \cdot \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} A^{-1} & A^{-1}B^tS^{-1} \\ 0 & -S^{-1} \end{pmatrix} \cdot \begin{pmatrix} u \\ p \end{pmatrix}. \tag{I.13}$$

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

```
v = A^{-1} * u;
u_tmp = S^{-1} * p;
u_tmp = B^t * u_tmp;
u_tmp = A^{-1} * u_tmp;
v += u_tmp
q = -S^{-1} * p;
```

This approach is unnecessarily expensive because it consists of two additional (and otherwise identical) solve operations (with `A^{-1}` and with `S^{-1}`) and an intermediate vector. This can be optimized by some minor code refactoring:

```
q = S^{-1} * p;
v = u;
v += Bt * q;
v = A^{-1} * v;
q *= -1;
```

We stress the point that, although the derivation of this pseudocode is straightforward, 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` concept.

A high-level implementation with the `LinearOperator` concept

Let $L = (l_{ij})_{ij}$ be a *regular block lower-triangular matrix* consisting of linerar 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:

$$Lx = b. \tag{I.14}$$

This equation can be solved by blockwise *forward substituion*:

$$\begin{cases} x_0 = l_{0,0}^{-1} \cdot b_0, \\ x_i = l_{i,i}^{-1} \cdot \Big(b_n - \sum_{j=0}^{i-1} l_{i,j} \cdot x_j \Big). \end{cases} \tag{I.15}$$

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

$$\begin{cases} x_n = u_{n,n}^{-1} \cdot b_n, \\ x_i = u_{i,i}^{-1} \cdot \Big(b_n - \sum_{j=i+1}^{n} u_{i,j} \cdot x_j \Big). \end{cases} \tag{I.16}$$

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

```cpp
template <size_t n, typename Range, typename Domain>
BlockLinearOperator<n, n, Domain, Range>
block_back_substitution(
  const BlockLinearOperator<n, n, Range, Domain> &block_operator,
  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 (I.16) 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 follow a very similar implementation.

```cpp
  // ...
  return_op.vmult = [block_operator, diagonal_inverse]
                    (Range &v, const Range &u)
  {
    const unsigned int m = block_operator.n_block_rows();
    if (m == 0)
      return;

    v.block(m-1) = diagonal_inverse.block(m-1, m-1) * u.block(m-1);

    for (int i = m - 2; i >= 0; --i)
      {
        auto &dst = v.block(i);
        dst = u.block(i);
        dst *= -1.;
        for (int j = i + 1; j < m; ++j)
          dst += block_operator.block(i, j) * v.block(j);
        dst *= -1.;
        dst = diagonal_inverse.block(i, i) * dst;
      }
  };
```

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:

```
// Assemble A and B
// together with precA and precS, two preconditioners
// for A and S respectively, and Asolver, Ssolver, and Gsolver
// respectively suitable solvers for A, S and the global system
...
// Transpose linear operator of B
auto Bt = transpose(B);
// Inverse of A
auto Ainv = inverse_operator(A , Asolver, Aprec);
// Schur complement, or alternatively: S = B*Aprec*Bt
auto S = B*Ainv*Bt;
auto Sinv = inverse_operator(S, Ssolver, Sprec);

auto system_matrix = block_operator({{A, Bt}, {B, 0}});
auto diagonal_inverse = block_diagonal_operator({Ainv, -1*Sinv});
P_inv = block_back_substitution(system_matrix, diagonal_inverse );

// Global solver
auto system_inverse =  inverse_operator(system_matrix, Gsolver, P_inv);

// Solve the problem!
auto solution = system_inverse * rhs;
```

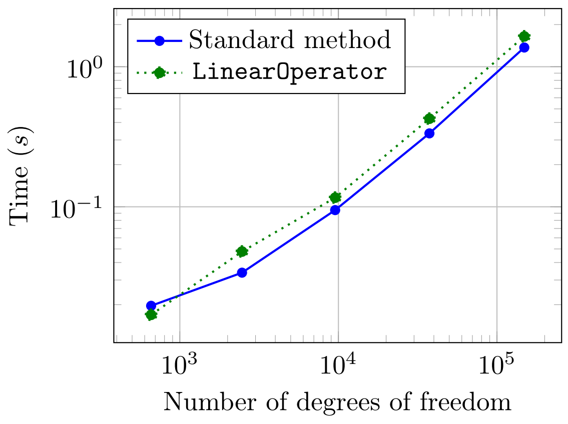
I.5.3. SOME BENCHMARKS

In this subsection we present two small tests to assess the preformance of the `LinearOperator` implementation of the system preconditioners when compared to a standard implementation (see, for example, the example `step-32` of the deal.II library [31]). The computations were performed on a 2.6GHz dual-core Intel Core i5 processor (i5) and on an Intel Xeon E5-2680 v2 with 10 cores (E5).

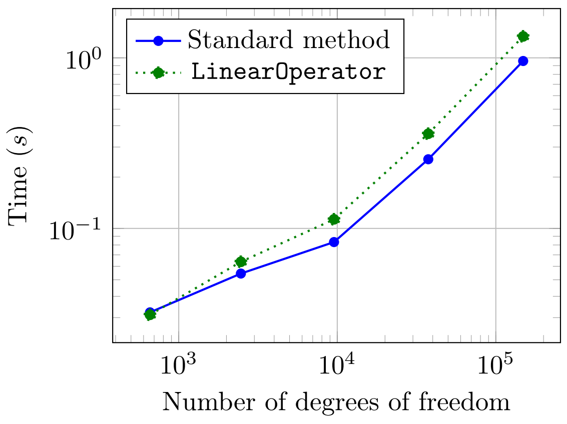
The standard implementation is taken from the example program “step-32” of the *deal.ii* library [31]. It implements the optimized algorithm presented in Subsection I.5.2.

The `LinearOperator` implementation uses the *back substitution algorithm* presented in equation (I.16).

The benchmark we use consists in performing 500 matrix vector multiplication with the two preconditioners. The average execution time over 5 different runs is plotted in figures I.2 and I.3 for i5 and E5 processors respectively.

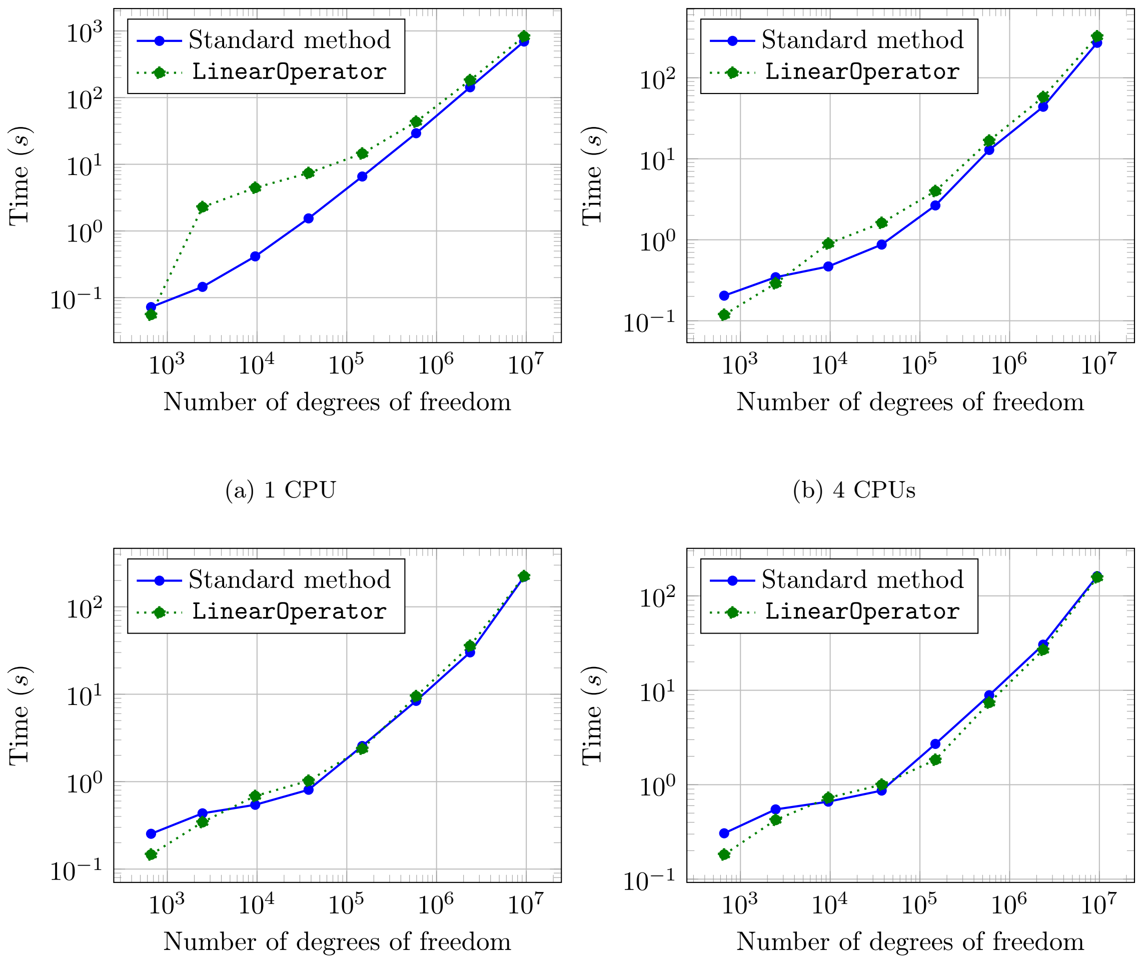


(a) 1 CPU



(b) 2 CPUs

Figure I.2: Execution time of the different algorithms running in serial (left) and in parallel (right) on a dual-core Intel Core i5 processor as a function of the matrix size. The resulting execution time is a mean value over five runs.



(a) 1 CPU

(b) 4 CPUs

(c) 8 CPUs

(d) 16 CPUs

Figure I.3: Execution time of the different algorithms on a cluster of 10-core Intel Xeon processors as a function of the matrix size. The resulting execution time is a mean value over five runs for different number of processors.

I.6. 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 overhead” associated with an implementation via pure expression templates. The introduced framework is very generic, it requires only a minimal interface that vector and matrix classes must adhere to. We gave a number of examples that demonstrate that the framework results in short and concise code. The numerical examples demonstrate that the runtime overhead introduced by the `std::function` objects and lambda functions is negligible.

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.

deal2lkit II

II.1. Introduction

The solution of partial differential equations by means of a finite element method always requires at least the following steps:

- generation of a geometrical grid to represent the domain of the simulation;
- definition of the discrete functional space for the solution;
- application of proper boundary conditions;
- actual solution of the algebraic problem;
- post-processing of the result (data output and error analysis).

Such a structure usually implies that different problems share a considerable amount of code. A natural response to such common background lies in the use of open source libraries as building blocks for advanced numerical solvers. The general purpose finite element library `deal.II` [4, 7] is one of the most successful libraries of this kind, and allows considerable simplification when writing complex finite element codes. `deal.II` supports massively parallel simulations [3], hp adativity [8], geometric multigrid [19], discontinuous finite elements [20], and matrix free simulations [23] among many other capabilities.

The `deal.II` library has been written with generality in mind, and allows the solution of several classes of finite element problems. Its flexibility can be attributed to the granularity and modularity of the code base, in which only the building blocks of finite element codes are programmed, and the semantic for the solution of an actual problem is left to users of the library. This approach has the advantage that `deal.II` can be used to solve virtually any problem that can be written into a partial differential equation, but leaves to the user the burden to stich together the various building blocks. A typical approach is to start from one of the many example programs that the library comes with (more than 50), and modify it to suite the needs of the user. While the approach copy-modify-run may be well suited for a single person working on a single project, it falls short when one wishes to reuse the same code base to solve possibly very different problems. The biggest difficulty comes from the fact that most of the tasks above have slightly different specialisations depending on the problem at hand. These specialisation are usually difficult to generalize, since they depend, for example, on the number of variables of a problem, the types of boundary conditions one would like to impose, or the type of norm one would like to use when computing errors during the post-processing phase of a program.

In this Chapter, we present a brief overview of `deal2lkit`: a library of modules built on top of `deal.II` that drastically reduces the amount of repeated lines of code between different projects, by introducing an extensive use of parameter files into every step of a general finite element code.

`deal2lkit` features also interfaces for other scientific libraries in order to tackle problems of increasing difficulties. We have constructed convenience wrappers around the SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (`SUNDIALS` [17]), that proved to be a winning strategy in many applications which require the solution of non-linear time dependent problems [25]. `deal2lkit` offers also an interface to the Open Asset Import Library (`Assimp`), which is used to extend the compatibility of the `deal.II` library towards grid generation software and 3D CAD manipulation tools.

`deal2lkit` is distributed under the free GNU Lesser General Public License (LGPL) and is available from the `deal2lkit` homepage at https://github.com/mathLab/deal2lkit. The library is tested by means of the continuous integration service hosted by Travis CI, where more than ninety tests are run both in debug and release mode before any change to the library is merged in the main distribution.

`deal2lkit` has been developed with full support for parallel environments, by exploiting hybrid multithread and multiprocessors paradigm [3].

The Chapter is organized as follows: in Section II.2 we present some of the modules of `deal2lkit`; Section II.3 shows some example applications which are useful to grasp the capabilities of `deal2lkit` and in Section II.4 we draw some conclusions and address future development.

II.2. Modules

II.2.1. PARAMETERACCEPTOR

In general, a *parameter file* is used to steer the execution of a program at run time, without the need to recompile the executable, with clear advantages in terms of *human-time*. Morevoer, without modifying the source code, the possibility to introduce bugs is prevented.

In the `deal.II` library, reading and writing parameter files is done through the `ParameterHandler` class, that provides a standard interface to an input file that can be used to feed run-time parameters to a program, such as time step sizes, geometries, right hand sides, etc.

`deal.II` supports the standard *xml* or *JSON* formats, or a custom text format which resemble `bash` files with support for sections, as in the following example:

Generated code

```
subsection Nonlinear solver
  set Nonlinear method = Gradient
  # this is a comment
  subsection Linear solver
    set Solver                        = CG
    set Maximum number of iterations = 30
  end
```

```
end
```

Typically, the following four steps are required to let a program use a parameter file:

- make sure that the program knows what entries will be in the file;
- create a parameter file with default values if one does not exist;
- parse all entries of the file (possibly raising exceptions if the entries were not previously declared, or if the parsed entries contain illegal values);
- assign the parsed entries to local variables of the program.

The `ParameterHandler` class of the `deal.II` library provides facilities to perform the above four steps, through the following methods:

- `ParameterHandler::print_parameters`
- `ParameterHandler::read_input`
- `ParameterHandler::declare_entry`
- `ParameterHandler::get_*`

In large programs, where the number of parameters easily exceeds hundreds of entries, managing the above four actions for different classes is far from trivial. The `deal.II` documentation advocates the creation of a class that would store all parameters of the problem, with two methods:

- `declare_parameters(prm)`
- `parse_parameters(prm)` or `get_parameters(prm)`

that should be called by the program before writing or reading a parameter file, and right after having read the parameter file into an object `prm` of type `ParameterHandler`.

Such an approach has the advantage that bookkeeping is simple, if compared to a scattered approach where each class keeps track of its own parameters, but it suffers one big draw back: it is not reusable for problems of different type and it has still the defect that one has to separate declaration and recovery of each parameter, as in the following short example:

```
 void NonLinEq::declare_parameters (ParameterHandler &prm) {
  prm.enter_subsection ("Nonlinear solver");
  {
    prm.declare_entry ("Nonlinear method",
                       "Newton-Raphson",
                       ParameterHandler::RegularExpressions::Anything());
    eq.declare_parameters (prm);
  }
  prm.leave_subsection ();
}
```

The complementary part of this code is contained in the `parse_parameters` method, which actually fills the values of the local variables.

```
void NonLinEq::parse_parameters (ParameterHandler &prm) {
  prm.enter_subsection ("Nonlinear solver");
  std::string method = prm.get ("Nonlinear method");
  eq.get_parameters (prm);
  prm.leave_subsection ();
}
```

According to the proposed design in the `deal.II` documentation, such separation is necessary (with a consequent proliferation of several places where one has to keep track of what variables have been declared and what variables have been assigned locally) since the declaration, reading and writing of a parameter file, and the assignment to local variables have to be done *exactly* in this sequence.

`deal2lkit` implements a *global subscription mechanism* and a *local subscription mechanism* through the base class `ParameterAcceptor`, which maintains compatibility with all classes written following the `deal.II` suggested construction, and provides an additional method which removes the necessity to split the declaration and parsing of parameters.

The *global subscription mechanism* is such that whenever a class that was derived by `ParameterAcceptor` is constructed, a static registry in the base class is updated with a pointer to the derived class. Such registry is traversed upon invocation of the single function `ParameterAcceptor::initialize("file.prm")` which in turn calls the method `declare_parameters` for each of the registered classes, reads the file `"file.prm"`, (creating it first with default values if it does not exist) and subsequently calls the method `parse_parameters`, again for each of the registered classes.

The base class `ParameterAcceptor` conforms to the standard advocated in the `deal.II` documentation, and it has a pure virtual method `declare_parameters` and a virtual method `parse_parameters` which can be overloaded as shown above. However, the base class also has a default implementation of `parse_parameters` which exploits a *local subscription mechanism* by storing in a local registry a pointer to all variables that were declared through the `add_parameter` method of `ParameterAcceptor`. Such method has the same syntax of the `ParameterHandler::add_entry` method, with the addition of two arguments: a `ParameterHandler` object on which `ParameterHandler::add_entry` will be called, and a reference to the variable that should hold the entry when a `ParameterHandler::get_*` method is called. Such variable is stored in a registry (local to the class instantiation) which is traversed by the default implementation of `ParameterAcceptor::parse_parameters`. Specialized implementations are provided for the most commonly used variable types.

When writing from scratch a class derived from `ParameterAcceptor`, if the user implements the `declare_parameters` method using *only* the above `add_parameter` method, then it is guaranteed that, upon calling the default implementation of `ParameterAcceptor::parse_parameters`, all variables that were stored in the registry are automatically populated with the values from the parameter file, without having to do so manually.

If a particular action needs to be taken after a class has parsed its parameters, the user can overload the virtual method `ParameterAcceptor::parse_parameters_call_back`, which by default does nothing.

A typical usage of this chain of classes is the following:

```
// This is your own class, derived from ParameterAcceptor
```

```
class MyClass : public ParameterAcceptor {
    virtual declare_parameters(ParameterHandler &prm) {
      add_parameters(prm,
                     &member_par,
                     "Par name",
                     "50", // Default value
                     ParameterHandler::RegularExpressions::Integer(0,100),
                     "long description about Par name");
    }
 private:
    // A problem parameter
    const unsigned int member_par;
}

int main() {
    // Make sure you build your class BEFORE calling
    // ParameterAcceptor::initialize()
    MyClass example;

    // With this call, all derived classes will have their
    // parameters initialized, including the class example above
    ParameterAcceptor::initialize("file.prm",
        "file_without_long_descriptions.prm");
    return 0;
}
```

Running the code above in an empty directory, will result in the creation of two files: `file.prm`, containing

Generated code

```
# long description about Par name
set Par name = 50
```

and `file_without_long_descriptions.prm` containing:

Generated code

```
set Par name = 50
```

The `ParameterAcceptor::RegularExpressions` are used to set the expected type of variable (integer in the above example) and, if available, a range of the allowed values. In the remainder of the Chapter, for the sake of brevity, the "long descriptions" will be omitted.

The second file name is optional. If one changes the value in the parameter file `"file.prm"` and runs the program again, this change will be automatically reflected on the `example.member_par` variable.

All modules in `deal2lkit` are derived from `ParameterAcceptor`. Declaration of the entries is required once, by specifying at the same time the variables that will hold the parameter. Once all objects have been constructed, the static method `ParameterAcceptor::initialize` will fill the local parameters of each derived class automatically, greatly simplifying the bookkeeping of the parameters, and allowing for immediate reuse of all classes in different programs, without having to rewrite a global parameter class, or without having to manually keep track of what classes have had their parameters declared or parsed.

As a convention, in `deal2lkit` all modules derived from `ParameterAcceptor` implement a default constructor which takes one or more optional arguments. The first optional argument is always the name of the section in the parameter file that the derived class should use to fill its local variables. By default, the utility function `deal2lkit::type` is used to fill the section name with a human readable version of the class name itself. For example, according to the above example, there will be a section named `MyClass`. Next options are the default values that will be written in the parameter file. There is no need to specify these options, as the user can always change the content of the file to make sure that the right parameters are used at run time, but this possibility allows one to design a program that does something sensible on the first run, without having to change any parameter file.

II.2.2. PARSED GRID GENERATOR

The interface for generating a grid through a parameter file is managed by the `ParsedGridGenerator<dim,spacedim>` class, which inherits from `ParameterAcceptor`. We can specify either the type of the grid that must be generated (e.g., rectangle, sphere, ball, etc) or read it from a file. In the first case, we exploit the `dealii::GridGenerator` functions of the `deal.II` library, effectively providing a parameterized wrapper around `GridGenerator` (hence the name). Otherwise, we simply specify the file to be read, which must be in any of the format recognised by the `deal.II` library.

The constructor of `ParsedGridGenerator<dim,spacedim>` takes optional strings which allow the user to decide what are the *default* values that will be written on the parameter file:

```
ParsedGridGenerator (const std::string section_name="",
                     const std::string grid_type="rectangle",
                     const std::string input_grid_file="",
                     const std::string opt_point_1="",
                     const std::string opt_point_2="",
                     const std::string opt_colorize="false",
                     const std::string opt_double_1="1.0",
                     const std::string opt_double_2="0.5",
                     const std::string opt_int="1",
                     const std::string opt_vec_of_int="",
                     const std::string mesh_smoothing="none",
                     const std::string output_grid_file="");
```

The first optional argument specifies the section name within the parameter file. If the section name is empty, by default it is set to `ParsedGridGenerator<x,x>` where `x,x` will be replaced with the actual `dim` and `spacedim` numbers with which the user instantiated the class.

`ParsedGridGenerator` can be used both in serial and parallel settings, and a typical usage of this class is

```
// 2D square - serial mesh
// by default it constructs the rectangle whos opposite corner points
// are p1=(10.0,10.0) and p2=(20.0,20.0)
ParsedGridGenerator<2,2> tria_builder_2d("2D mesh",
                                         "rectangle",
                                         "",
                                                  "10.0,10.0",
                                                  "20.0,20.0",
                                                  "true");

// 3D parallelepiped - parallel distributed mesh
// by default it constructs the parallelepiped whos opposite corner
// points are p1=(7.0,8.0,9.0) and p2=(15.0.16.0,16.7)
ParsedGridGenerator<3,3> tria_builder_3d("3D mesh",
                                                   "rectangle",
                                                   "",
                                                   "7.0,8.0,9.0",
                                                   "15.0,16.0,16.7");

// call ParameterAcceptor
ParameterAcceptor::initialize("file.prm", "no_descriptions.prm");

// Construct a serial mesh following the indications in "file.prm"
// in the section "2D mesh"
Triangulation<2,2> *tria_serial   = tria_builder_2d.serial();

// Construct a parallel::distributed::Triangulation following the
// indications in "file.prm'', in the section "3D mesh"
parallel::distributed::Triangulation<3,3> *tria_mpi  =
      tria_builder_3d.distributed(MPI_COMM_WORLD);
```

Once the function `ParameterAcceptor::initialize("no_descriptions.prm")` is called, the `no_descriptions.prm` is filled with the following entries:

Generated code

```
subsection 3D mesh
  set Colorize                    = false
  set Grid to generate            = rectangle
  set Input grid file name        =
  set Mesh smoothing alogrithm    = none
  set Optional Point<spacedim> 1 = 7.0,8.0,9.0
  set Optional Point<spacedim> 2 = 15.0,16.0,16.7
  set Optional double 1           = 1.0
  set Optional double 2           = 0.5
  set Optional int 1              = 1
  set Optional vector of dim int = 1,1,1
  set Output grid file name       =
end
subsection 2D mesh
  set Colorize                    = true
  set Grid to generate            = rectangle
  set Input grid file name        =
  set Mesh smoothing alogrithm    = none
  set Optional Point<spacedim> 1 = 10.0,10.0
  set Optional Point<spacedim> 2 = 20.0,20.0
  set Optional double 1           = 1.0
  set Optional double 2           = 0.5
  set Optional int 1              = 1
  set Optional vector of dim int = 1,1
  set Output grid file name       =
end
```

If the user would then change the parameter file to generate a sphere, or read a file, no change in the code would be necessary, as at run time the new parameters would be used.

The option `Output grid file name`, if set to non-empty, will allow the user to call the method `ParsedGridGenerator<dim,spacedim>::write(tria)` to output the given triangulation to a file. The format of the output file is decided by the extension of the output file name.

II.2.3. PARSED FINITE ELEMENT

The existence and stability of solutions to partial differential equations is strictly dependent on the selected solution space, which defines the finite element to use. The class `ParsedFiniteElement<dim,spacedim>`, derived from `ParameterAcceptor`, allows the definition of the finite element type from a parameter file. The constructor takes several optional arguments, which are reported below, that are used to fill the default values of the parameter file:

```
ParsedFiniteElement (const std::string &name="",
                     const std::string &default_fe="FE_Q(1)",
                     const std::string &default_component_names="u",
                     const unsigned int n_components=0,
                     const std::string &default_coupling="",
                     const std::string &default_preconditioner_coupling="");
```

The first entry is the name if the section for `ParameterHandler`. The second one is the `FiniteElement`, which follows the same form returned by the `dealii::FiniteElement::get_name()` function. For example, for the Stokes problem in two or three dimension, the following string could be used "`FESystem[FE_Q(2)^dim-FE_Q(1)]`", generating a Taylor-Hood mixed finite element space of order two for the velocity and one for the pressure. Further optional entries are the component names, the allowed number of components (0 means an arbitrary number) and the system and preconditioner couplings, which may be used to define the blocks of a `dealii::BlockVector` or a `dealii::BlockMatrix`.

If one specifies a given number of components at construction time, then the program will throw an exception if the user changes the parameter file in a way that creates a finite element space with the wrong number of components.

A typical usage of this class is reported in the snippet below:

```
// finite element for 2D Stokes problem
ParsedFiniteElement<2,2> fe_builder("Parsed Finite Element for Stokes",
                                    "FESystem[FE_Q(2)^2-FE_Q(1)]",
                                    "u,u,p");

ParameterAcceptor::initialize("file.prm", "no_descriptions.prm");

// pointer to a newly created finite element following the content
// of "file.prm"
FiniteElement<2,2> *fe = fe_builder();
```

After that the above code is run, the `no_descriptions.prm` file is filled as follows:

Generated code

```
subsection Parsed Finite Element for Stokes
  set Block coupling                 =
  set Blocking of the finite element = u,u,p
  set Finite element space           = FESystem[FE_Q(2)^2-FE_Q(1)]
  set Preconditioner block coupling  =
end
```

An extended use of the methods in `ParsedFiniteElement` can be found, for example, in the πDoMUS (Parallel Deal.II MUltiphysics Solver) application (`https://github.com/mathLab/pi-DoMUS`) [12].

II.2.4. PARSED FUNCTIONS AND BOUNDARY CONDITION

Forcing terms and boundary conditions are often expressed by means of functions, and one would like to have the possibility to change them without recompiling the user code. The `deal2lkit` library offers three classes (derived from `ParameterAcceptor`) to define such functions at run time as parsed objects: `ParsedFunction<dim,ncomponents>`, `ParsedMappedFunctions<dim,ncomponents>`, and `ParsedDiricheltBCs<dim,spacedim,ncomponents>`.

`ParsedFunction<dim,ncomponents>` is a thin wrapper for the `deal.II` class `dealii::Functions::ParsedFunction`, which simply derives the `deal.II` version from `ParameterAcceptor`, in order to put in place the subscription strategy of `deal2lkit` (see Section II.2.1). The constructor takes a name for the section and an expression string, which is used to set the expression of the underlying `dealii::Functions::ParsedFunction` as soon as the parameters are parsed. For example, if we declare a `ParsedFunction` as follows:

```
ParsedFunction<2,1> forcing_term("Forcing term",
                                 "sin(2*pi*(x-t))");
```

then, within the parameter file, the following section can be found:

Generated code

```
subsection Forcing term
  set Function constants  =
  set Function expression = sin(2*pi*(x-t))
  set Variable names      = x,y,t
end
```

and the `forcing_term` can be used in place of a `Function<2>` object. The second value for the template parameter specifies the number of components for the function, which can be greater or equal than one.

As for each class derived from `ParameterAcceptor`, the strings given in the constructor are the default values set in the parameter file, but they can be changed within the parameter file itself.

`ParsedMappedFunctions<dim,ncomponents>` is a more complex collection of functions acting on given ids (boundary_id, material_id, etc.) and on specified components. Neumann boundary conditions and forcing terms can be easily handled with this class. The constructor takes:

- the name for the section of the Parameter Handler to use;
- the name of the variable to which the component belongs. For example, for Stokes equations, if we name the velocity `u` and the pressure `p`, then `component_names=u,u,p`, in 2D, and `component_names=u,u,u,p` for a 3D framework;

- a list of ids and components where the boundary conditions must be applied, which is a string with the following pattern `boundary_id = component; other_component % other_id = comp; other_comp`. The components can be either set with numbers (e.g., first component is 0, second is 1) or with the name of the variable defined in `component_names`;
- a list of ids and expressions defined over the ids (if this string is left empty, a `ZeroFunction` is imposed on the above specified ids and components).
- A list of constants that can be used in the above expressions.

A typical use of this class is the following:

```
// create mapped_functions object - 2D Stokes problem

ParsedMappedFunctions<2,3>
mapped_functions("Mapped functions",
"u,u,p",
"0=u % 6=ALL",
"0=x;y;0 % 6=y*k;0;k",
"k=1");

...

unsigned int id = cell->material_id();

std::vector<double> fs(n_q_points);

mapped_functions.get_mapped_function(id)->value_list(
fe_values.get_quadrature_points(),
fs);
```

After the code is run for the first time, the following section will appear in the parameter file:

Generated code

```
subsection Mapped functions
  set IDs and component masks = 0=u % 6=ALL
  set IDs and expressions     = 0=x;y;0 % 6=y*k;0;k
  set Known component names   = u,u,p
  set Used constants          = k=1
end
```

`ParsedDiricheltBCs<dim,spacedim,ncomponents>` is a specialization of the above, which allows to set Dirichlet boundary conditions using `deal.II` utilities. This class is derived from `ParsedMappedFunctions<spacedim,ncomponents>` and provides wrappers for the functions `dealii::VectorTools::interpolate_boundary_values`, `dealii::VectorTools::project_boundary_values` and `dealii::VectorTools::compute_nonzero_normal_flux`. The constructor takes:

- the name for the section of the Parameter Handler to use;
- the name of the variable to which the component belongs, as for the class `ParsedMappedFunctions`;

- a list of ids and components where the boundary conditions must be applied, which is a string with the following pattern `boundary_id = component; other_component % other_id = comp; other_comp`. The normal component can be specfied postponing `".N"` to the name of the components (e.g., `u.N` is the normal part of `u`);
- a list of ids and expressions defined over the ids (if this string is left empty, homogeneous boundary conditions are imposed on the above specified ids and components);
- A list of constants that can be used in the above expressions.

A typical use of this class is the following:

```
// create dirichlet object - 2D Stokes problem

ParsedDirichletBCs<2,2,3>
dirichlet("Dirichlet BCs",
"u,u,p",
"0=u % 3=u.N",
"0=x;y;0 % 3=x;2;0");
...
// the following functions apply the boundary conditions for the
// boundary id 0
dirichlet.interpolate_boundary_values(dof_handler,constraints);

// in order to apply the boundary conditions also to the
// boundary id 3, where the normal component is set,
// the following function must be called
dirichlet.compute_nonzero_normal_flux(dof_handler,constraints);
```

Once the above code is run, the following section is included in the parameter file:

Generated code

```
subsection Dirichlet BCs
  set IDs and component masks = 0=u % 3=u.N
  set IDs and expressions     = 0=x;y;0 % 3=x;2;0
  set Known component names   = u,u,p
  set Used constants          =
end
```

II.2.5. PARSED SOLVER

In `deal.II`, to solve a linear system one needs to specify in every program the solver type, the preconditioner, and the solver parameters (such as how many iterations to perform before giving up in an iterative solver, or to what tolerance we should iterate to).

The typical implementation in `deal.II` looks like the following snippet of code:

```
// Maximum 1000 iterations, to 1e-8 absolute tolerance
 SolverControl solver_control (1000, 1e-8);

 SolverCG<VEC> solver(solver_control,
                      SolverCG<VEC>::AdditionalData());
```

```
// Assuming preconditioner was constructed somewhere above...
solver.solve ( system_matrix,
                dst, src,
                preconditioner );
```

Following the philosophy of the other deal2lkit modules, the ParsedSolver<VECTOR> class allows to change the solver type (Conjugate Gradient, in the above snippet of code) and the entries of SolverControl from a parameter file

```
ParsedSolver<VEC> inverse("Solver", "cg",
                          unsigned int max_iterations= 1000,
                          double reduction= 1e-8,
                          linear_operator<VEC> matrix = Identity<VEC>(),
                          linear_operator<VEC> preconditioner =
                              Identity<VEC>());
```

The constructor takes as optional entries the section name, the solver type, the maximum number of iterations, and the reduction required to reach convergence. If the operators in which this solver should act (i.e, the system matrix, op, and the preconditioner, prec) are known in advance, they can be supplied in the constructor. They default to the identity operator, and they can be assigned later by setting op and prec. A typical usage of this class is as follows:

```
ParsedSolver<VEC> matrix_inv("Solver");
ParameterAcceptor::initialize("...");
matrix_inv.op   = linear_operator<VEC>(matrix);
matrix_inv.prec = linear_operator<VEC>(preconditioner);

solution = matrix_inv*rhs;
```

Once the above code is run, within the parameter file the following section is populated:

Generated code

```
subsection Solver
  set Log frequency = 1
  set Log history   = false
  set Log result    = true
  set Max steps     = 1000
  set Reduction     = 1e-08
  set Solver name   = cg
  set Tolerance     = 1.e-10
end
```

ParsedSolver is derived from the LinearOperator class of deal.II [24], which allows the use of very effective syntax expressions, making the solution of the system computable with the very simple expression:

```
solution = matrix_inv * rhs;
```

The supported solver types are those provided by the deal.II library, namely, CG, BICSTAB, GMRES, FGMRES, MINRES, QMRS and Richardson. One of the main advantage of such a solver it's the possibility to combine it using the expression syntax of LinearOperators in the construction, for example, of block preconditioners for complex problems.

II.2.6. POST-PROCESSING AND ERROR ANALYSIS

deal2lkit provides two classes to handle post processing through a parameter file, namely, ParsedDataOut<dim,spacedim> and ErrorHandler<ntables>, which are both derived from ParameterAcceptor.

The class ParsedDataOut<dim,spacedim> provides a wrapper for dealii::DataOut, to write solution vectors in any of the formats supported by deal.II, automatically splitting the output of several files at once when the code is run in parallel. The constructor takes the name of the parameter section, the format of the output (e.g., vtu), an optional incremental suffix for creating a progressive directories/subdirectories for every run, the base name of the output file (the default is set to solution) and an optional MPI_COMMUNICATOR object. A typical usage of this class is the following:

```
// constructor
ParsedDataOut<dim,spacedim> data_out("Section name", "vtu");
...
std::stringstream suffix;
suffix << "." << number_of_this_cycle;
data_out.prepare_data_output(*dof_handler, suffix.str());
data_out.add_data_vector (vector_storing_solution_of_stokes, "u,u,p");
data_out.add_data_vector (another_vector, "random");
data_out.write_data_and_clear();
```

The function prepare_data_output initialises the internal data structures and prepares the output files, whose names are constructed as a combination of the base name, an optional user supplied suffix, eventually a processor number and the output suffix. The function add_data_vector is a wrapper for the method dealii::DataOut::add_data_vector and it automatically distinguish between vector and scalar fields according to the names used in the add_data_vector method. Finally, write_data_and_clear saves the output file and releases the pointers to the internal data structures. It is worth mentioning that the lines of codes that must be written are the same for both a serial or parallel application, with the exception of the constructor which needs an MPI_COMMUNICATOR in the parallel case.

If an exact (or a reference) solution is known, the class ErrorHandler<n_tables> gives the possibility to calculate the error of the numerical solution in various norms (i.e., L^2, H^1, L^∞, W_1^∞) where both the norms and the exact solution can be parsed from the parameter file. It is templated over the number of tables that the user wishes to generate. The constructor of this class takes the name of the section of the parameter, the name of the components (i.e., for Stokes in 2D it could be “u,u,p”), and the norms to be used. An example use of this class is the following:

```
// constructor. we are solving Stokes equations in 2D
ErrorHandler<1> eh("Error Tables",
                   "u, u, p",
                   "L2, Linfty, H1; AddUp; L2");
...
for (unsigned int cycle=0; cycle<max; ++cycle)
{
  if(cycle != 0)
  refine_mesh();

11      setup_and_solve_all();
12
13      eh.error_from_exact(dof_handler,
14      vector_storing_solution,
15      exact_solution);
16    }
17    eh.output_table(std::cout);
```

Line 4 of the above code shows that by default this code will compute the error for the velocity in L^2, L^∞, H^1 norm, whereas the error for the pressure would be computed only in L^2 norm. The `AddUp` entry is used to add different components of the same vector-valued variable. After running the code for the first time, the user can change any of the above options at run time by modifying the input file parameter.

The example below shows a complete program using `ParsedGridGenerator`, `ParsedFunction` and `ErrorHandler` classes to estimate the error of interpolation for bi-linear finite element spaces:in particular we want to measure the rate of convergence of the interpolation of the cosine function.

```
ParsedGridGenerator<2> gg;
ErrorHandler<> eh;
ParsedFunction<2,1> pf("Function", "cos(2*pi*x)*cos(2*pi*y)");

ParameterAcceptor::initialize("error.prm","used.prm");

auto tria = gg.serial();
FE_Q<2> fe(1);
DoFHandler<2> dh(*tria);

for (unsigned int i=0; i<5; ++i)
  {
    tria->refine_global(1);
    dh.distribute_dofs(fe);
    Vector<double> sol(dh.n_dofs());
    VectorTools::interpolate(dh, pf, sol);
    eh.error_from_exact(dh, sol, pf);
  }
eh.output_table(deallog.get_file_stream());
```

In the `error.prm` file it is possible to specify several output parameters. For example: error file format, table name, and the norms we want to evaluate. In this case we selected a `tex` file format and as norms *Linfty, L2, and H1*, and we asked to output the table also on the terminal:

Generated code

```
subsection deal2lkit::ErrorHandler<1>
  set Compute error             = true
  set Error file format         = tex
  set Output error tables       = true
  set Solution names            = u
  set Solution names for latex  = u
  set Table names               = error
  set Write error files         = false
  subsection Table 0
    set Add convergence rates           = true
    set Extra terms                     = cells,dofs
    set Latex table caption             = error
    set List of error norms to compute = Linfty, L2, H1
    set Rate key                       =
  end
end
```

The output of the program will look like the following:

Generated code

```
cells dofs    u_Linfty         u_L2           u_H1
4     9    1.183e-01 -    5.156e-02 -    2.615e-01 -
16    25   3.291e-02 1.85 1.333e-02 1.95 1.272e-01 1.04
64    81   8.449e-03 1.96 3.360e-03 1.99 6.313e-02 1.01
256   289  2.126e-03 1.99 8.418e-04 2.00 3.150e-02 1.00
1024  1089 5.325e-04 2.00 2.106e-04 2.00 1.574e-02 1.00
```

Since we specified the `tex` format, the program will generate the file `error0.tex`, which reads:

Generated code

```
\documentclass[10pt]{report}
\usepackage{float}


\begin{document}
\begin{table}[H]
\begin{center}
\begin{tabular}{|r|r|c|c|c|c|c|c|} \hline
\# cells & \# dofs &
\multicolumn{2}{|c|}{$\| u - u_h \|_\infty $} &
\multicolumn{2}{|c|}{$\| u - u_h \|_0 $} &
\multicolumn{2}{|c|}{$\| u - u_h \|_1 $}\\ \hline
4 & 9 & 1.183e-01 & - & 5.156e-02 & - & 2.615e-01 & -\\ \hline
16 & 25 & 3.291e-02 & 1.85 & 1.333e-02 & 1.95 & 1.272e-01 & 1.04\\ \hline
64 & 81 & 8.449e-03 & 1.96 & 3.360e-03 & 1.99 & 6.313e-02 & 1.01\\ \hline
256 & 289 & 2.126e-03 & 1.99 & 8.418e-04 & 2.00 & 3.150e-02 & 1.00\\ \hline
1024 & 1089 & 5.325e-04 & 2.00 & 2.106e-04 & 2.00 & 1.574e-02 & 1.00\\ \hline
\end{tabular}
\end{center}
\end{table}
\end{document}
```

If `error0.tex` is included in a tex file, it will result in the following table:

# cells	# dofs	$\|u-u_h\|_\infty$		$\|u-u_h\|_0$		$\|u-u_h\|_1$	
4	9	1.183e-01	-	5.156e-02	-	2.615e-01	-
16	25	3.291e-02	1.85	1.333e-02	1.95	1.272e-01	1.04
64	81	8.449e-03	1.96	3.360e-03	1.99	6.313e-02	1.01
256	289	2.126e-03	1.99	8.418e-04	2.00	3.150e-02	1.00
1024	1089	5.325e-04	2.00	2.106e-04	2.00	1.574e-02	1.00

II.2.7. SUNDIALS INTERFACE

`deal2lkit` features an interface for the SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (`SUNDIALS`) [17], which is implemented in the class `SundialsInterface`.

The class `IDAInterface` is a wrapper to the Implicit Differential-Algebraic solver (`IDA`) [18], provided within the `SUNDIALS` library, which is a general purpose solver for systems of Differential-Algebraic Equations (DAEs).

Citing from the `SUNDIALS` documentation:

Consider a system of Differential-Algebraic Equations written in the general form

$$\begin{cases} F(t, y, \dot{y}) = 0\,, \\ y(t_0) = y_0\,, \\ \dot{y}(t_0) = \dot{y}_0\,. \end{cases} \tag{II.1}$$

where $y, \dot{y}$ are vectors in $\mathbb{R}^n$, t is often the time (but can also be a parametric quantity), and $F : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. Such problem is solved using Newton iteration augmented with a line search global strategy [18]. The integration method used in `IDA` is the variable-order, variable-coefficient BDF (Backward Differentiation Formula), in fixed-leading-coefficient form [10]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i}\, y_{n-i} = h_n\, \dot{y}_n\,, \tag{II.2}$$

where y_n and $\dot{y}_n$ are the computed approximations of $y(t_n)$ and $\dot{y}(t_n)$, respectively, and the step size is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ are uniquely determined by the order q, and the history of the step sizes. The application of the BDF method (II.2) to the DAE system (II.1) results in a nonlinear algebraic system to be solved at each time step:

$$G(y_n) \equiv F\left(t_n, y_n, \frac{1}{h_n}\sum_{i=0}^{q} \alpha_{n,i}\, y_{n-i}\right) = 0\,. \tag{II.3}$$

The Newton method leads to a linear system of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)})\,, \tag{II.4}$$

where $y_{n(m)}$ is the m-th approximation to y_n, J is the approximation of the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}\,, \tag{II.5}$$

and $\alpha = \alpha_{n,0}/h_n$. It is worthing metioning that the scalar α changes whenever the step size or method order changes.

As far as the solution of the linear system is concerned, the `deal2lkit` class `SundialsInterface` exploits the linear algebra classes included in the `deal.II` library (e.g., solvers, preconditioners, `LinearOperator`, etc.).

A user that would want to use the `SUNDIALS` interface, should derive its problem class from the `SundialsInterface` class, and implement all pure virtual methods.

The following snippet of code shows in details the methods needing overloads in order to actual solve a DAE system of the form (II.1):

```
class MyClass : public SundialsInterface<VEC>
{
public:

```

```
MyClass ();

void run ();

/*********************************************************
 * Public interface from SundialsInterface
 *********************************************************/
virtual shared_ptr<VEC>
create_new_vector() const;

/** Returns the number of degrees of freedom. Pure virtual function. */
virtual unsigned int n_dofs() const;

/** This function is called at the end of each iteration step for
 * the ode solver. Once again, the conversion between pointers and
 * other forms of vectors need to be done inside the inheriting
 * class. */
virtual void output_step(const double t,
                         const VEC &solution,
                         const VEC &solution_dot,
                         const unsigned int step_number,
                         const double h);

/** This function will check the behaviour of the solution. If it
 * is converged or if it is becoming unstable the time integrator
 * will be stopped. If the convergence is not achived the
 * calculation will be continued. If necessary, it can also reset
 * the time stepper. */
virtual bool solver_should_restart(const double t,
                                   const unsigned int step_number,
                                   const double h,
                                   VEC &solution,
                                   VEC &solution_dot);

/** For dae problems, we need a
 residual function. */
virtual int residual(const double t,
                     const VEC &src_yy,
                     const VEC &src_yp,
                     VEC &dst);

/** Setup Jacobian system (and preconditioner). */
virtual int setup_jacobian(const double t,
                           const VEC &src_yy,
                           const VEC &src_yp,
                           const VEC &residual,
                           const double alpha);


/** Solve the linear system. */
virtual int solve_jacobian_system(const double t,
                                  const VEC &y,
                                  const VEC &y_dot,
                                  const VEC &residual,
                                                    const double alpha,
                                                    const VEC &src,
                                                    VEC &dst) const;

   /** And an identification of the
    differential components. This
    has to be 1 if the
    corresponding variable is a
    differential component, zero
    otherwise.  */
   virtual VEC &differential_components() const;

...
   private:
     // This class provides the interface to the IDA solver
     IDAInterface<VEC> ida;
};

...

void MyClass::run()
{
   ...
   // once the grid has been constructed, the FE has been setup
   // and all necessary variables have been initialized
   // IDA can be called and the job is done :)
   ida.start_ode(solution, solution_dot, max_time_iterations);
}
```

The functions declared after the comment `Public interface from SundialsInterface` are problem-dependent and therefore must be implemented by the end user.

II.3. Examples

This Section deals with several examples of actual solution of partial differential equations with `deal2lkit`. Such examples are part of the `deal2lkit` library itself and the source codes can be found within the folder `examples`.

II.3.1. SIMPLE POISSON PROBLEM

We start the examples section with a trivial problem, that shows very effectively the potential for *high performance programming* using `deal2lkit`. This trivial example program *does not* follow best practices in scientific programming, and has everything in the `main` function for the sake of brevity.

We solve the poisson problem

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \tag{II.6}$$

with Dirichlet boundary conditions, or homogenous Neumann boundary conditions, and where κ, f, and Ω can all be constructed from a parameter file.

We report here the full listing excluding header files and comments:

```cpp
// header files excluded
using namespace deal2lkit;
using namespace dealii;

// Two simple parameters for this problem
class PoissonParameters : public ParameterAcceptor
{
public:
  virtual void declare_parameters(ParameterHandler &prm)
  {
    add_parameter(prm, &n_cycles, "Number of cycles", "4");
    add_parameter(prm, &initial_refinement, "Initial refinement", "3");
  }
  unsigned int n_cycles;
  unsigned int initial_refinement;
};

// Main function
int main(int argc, char **argv)
{
  Utilities::MPI::MPI_InitFinalize mpi_init(argc, argv);

  // Change this if you want a one or three dimensional simulation
  const unsigned int dim = 2;
  // embedded in spacedim Euclidean space
  const unsigned int spacedim = 2;

  // All the generators
  PoissonParameters par;
  ParsedGridGenerator<dim,spacedim> pgg;
  ParsedFiniteElement<dim,spacedim> pfe;

  // Parametric functions
  ParsedDirichletBCs<dim,dim,1> bcs;
  ParsedFunction<spacedim> kappa("Kappa", "1.0");
  ParsedFunction<spacedim> force("Forcing term");
  ParsedFunction<spacedim> exact("Exact solution");

  // Linear algebra classes
  SparsityPattern sparsity;
  SparseMatrix<double> matrix;
  SparseILU<double> prec;
  ConstraintMatrix constraints;

  Vector<double> solution;
  Vector<double> rhs;

  // Parsed inverse
  ParsedSolver<Vector<double> > inverse("Solver", "cg", 1000, 1e-8,
    linear_operator<Vector<double> >(matrix),
    linear_operator<Vector<double> >(matrix, prec));

  shared_ptr<Triangulation<dim,spacedim> > tria;
  shared_ptr<FiniteElement<dim,spacedim> > fe;

  ErrorHandler<1> eh;
  ParsedDataOut<dim,spacedim> pdo;

  // And now the parameter file is initialized, written if
  // non-existant, and then read
  ParameterAcceptor::initialize("poisson.prm", "used_parameters.prm");

  tria = SP(pgg.serial());
  pgg.write(*tria);

  fe = SP(pfe());
  DoFHandler<dim,spacedim> dh(*tria);

  tria->refine_global(par.initial_refinement);
  QGauss<dim> quad(2*fe->degree+1);

  // Execute par.n_cycles refinements
  for (unsigned int i=0; i<par.n_cycles; ++i)
    {
      dh.distribute_dofs(*fe);
      std::cout << "Cycle " << i
                << ", cells: " << tria->n_active_cells()
                << ", dofs: " << dh.n_dofs() << std::endl;

      DynamicSparsityPattern dsp(dh.n_dofs());
      DoFTools::make_sparsity_pattern (dh, dsp);
      sparsity.copy_from(dsp);

      matrix.reinit (sparsity);
      solution.reinit (dh.n_dofs());
      rhs.reinit (dh.n_dofs());

      constraints.clear();
      bcs.interpolate_boundary_values(dh, constraints);
      constraints.close();

      // Solve a poisson problem
      MatrixCreator::create_laplace_matrix(dh, quad, matrix, force,
                                           rhs, &kappa, constraints);
      prec.initialize(matrix);

      solution = inverse*rhs;
      constraints.distribute(solution);

      pdo.prepare_data_output(dh, std::to_string(i));
      pdo.add_data_vector(solution, "solution");
      pdo.write_data_and_clear();

      eh.error_from_exact(dh, solution, exact);

      tria->refine_global(1);
    }

  eh.output_table();
  return 0;
}
```

After we run the code for the first time, a file similar to the following (short listing) parameter file is created in `used_parameters.prm` (we show here a changed version of this file, where we added a manufactured solution to test convergence of Q1 finite elements on a square):

Generated code

```
subsection Dirichlet BCs
  set IDs and component masks = 0=ALL
  set IDs and expressions     =
  set Known component names   = u
  set Used constants          =
end
subsection Exact solution
  set Function constants  =
  set Function expression = sin(2*pi*x)*sin(2*pi*y)
  set Variable names      = x,y,t
end
subsection Forcing term
  set Function constants  =
  set Function expression = 8*pi*pi*sin(2*pi*x)*sin(2*pi*y)
  set Variable names      = x,y,t
end
subsection Kappa
  set Function constants  =
  set Function expression = 1.0
  set Variable names      = x,y,t
end
subsection PoissonParameters
  set Initial refinement = 2
  set Number of cycles   = 5
end
subsection Solver
  set Log frequency = 1
  set Log history   = false
  set Log result    = true
  set Max steps     = 1000
  set Reduction     = 1e-08
  set Solver name   = cg
  set Tolerance     = 1.e-10
end
subsection deal2lkit::ErrorHandler<1>
  set Compute error            = true
  set Error file format        = tex
  set Output error tables      = true
  set Solution names           = u
  set Solution names for latex = u
  set Table names              = error
  set Write error files        = false
  subsection Table 0
```

```
    set Add convergence rates          = true
    set Extra terms                    = cells,dofs
    set Latex table caption            = error
    set List of error norms to compute = Linfty, L2, H1
    set Rate key                       =
  end
end
subsection deal2lkit::ParsedDataOut<2, 2>
  set Incremental run prefix =
  set Output partitioning    = false
  set Problem base name      = solution
  set Solution names         = u
  subsection Solution output format
    set Output format = vtu
    set Subdivisions  = 1
  end
end
subsection deal2lkit::ParsedFiniteElement<2, 2>
  set Block coupling                 =
  set Blocking of the finite element = u
  set Finite element space           = FE_Q(1)
  set Preconditioner block coupling  =
end
subsection deal2lkit::ParsedGridGenerator<2, 2>
  set Colorize                    = false
  set Grid to generate            = rectangle
  set Input grid file name        =
  set Mesh smoothing alogrithm    = none
  set Optional Point<spacedim> 1 = 0,0
  set Optional Point<spacedim> 2 = 1,1
  set Optional double 1           = 1.0
  set Optional double 2           = 0.5
  set Optional int 1              = 1
  set Optional vector of dim int = 1,1
  set Output grid file name =
end
```

The final output of the code is something similar to (only last cycle is shown):

Generated code

```
Cycle 4, cells: 4096, dofs: 4225
DEAL:cg::Starting value 0.615860
DEAL:cg::Convergence step 41 value 4.15887e-09
DEAL:PrepareOutput::Will write on file: ./solution4.vtu
DEAL:AddingData::Added data: solution
DEAL:WritingData::Wrote output file.
DEAL::Reset output.
cells dofs    u_Linfty         u_L2           u_H1
   16   25 2.088e-01    - 1.218e-01    - 1.996e+00    -
   64   81 7.554e-02 1.47 3.039e-02 2.00 1.003e+00 0.99
  256  289 2.063e-02 1.87 7.601e-03 2.00 5.031e-01 1.00
 1024 1089 5.273e-03 1.97 1.901e-03 2.00 2.518e-01 1.00
 4096 4225 1.325e-03 1.99 4.752e-04 2.00 1.259e-01 1.00
```

The full program consists of less than one hundred lines of actual code, and allows one to arbitrarily change almost everything of the program itself (with the exception of the problem to solve) without recompiling the code.

II.3.2. HEAT EQUATION SOLVED WITH THE SUNDIALS INTERFACE

In order to demonstrate the potential of the `deal2lkit` library, we solved the heat equation

$$\frac{\partial u(x,t)}{\partial t} - \nabla \cdot (D \nabla u(x,t)) = f(x,t) \quad \text{for} \quad x \in \Omega = (0,1)\times(0,1) \quad \forall t \in (0,1), \tag{II.7}$$

where $D = 0.5$ and

$$f(x,t) = 2\pi y\,(y-1)\cos\left(-2\pi(t-x)\right) - 2D\left[2\pi^2 y\,(y-1)\sin\left(-2\pi(t-x)\right) - sin\left(-2\pi(t-x)\right)\right]. \tag{II.8}$$

At the boundaries, we impose the exact solution of the problem

$$u|_{\partial\Omega} = y\,(1-y)\sin\left(2\pi(x-t)\right) \tag{II.9}$$

and $\dfrac{\partial u(t=0)}{\partial t} = 0$ has been used as initial condition.

The aforementioned problem has been solved relying on the `SundialsInterface` class, which features a variable-order, variable-coefficient BDF (Backward Differentiation Formula) integration method. During the transient, the mesh is adaptively refined in order to satisfy a given accuracy, which is set by the user. We use a Kelly error estimator [21] to compute an *a posteriori* estimate for the error on each cell, and when the L^∞ norm of the error estimator is greater than a threshold, the mesh is accordingly refined. Snapshots of the computed solution are shown below. The statistics of the calls to the different functions when the code is run on two processors with `mpirun -np 2` are given here:

Generated code

```
+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |       3.1s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Jacobian matrix        |        95 |   0.0761s  |       2.5% |
| Compute error estimator         |       122 |    0.298s  |       9.6% |
| Setup Jacobian                  |        95 |   0.0777s  |       2.5% |
| Solve system                    |       281 |    0.309s  |        10% |
| Post-processing                 |       101 |     1.61s  |        52% |
| Residual                        |       281 |    0.726s  |        23% |
| Setup dof systems               |        23 |   0.0323s  |         1% |
+---------------------------------+-----------+------------+------------+
```

As it can be seen, the jacobian matrix has been assembled only 95 times, even though the system has been solved 281 times. The mesh has been refined 22 times, because the setup dof systems has been called 23 times (one call is for the first time that the mesh is created). More than 50% of the time is spent for storing the solution, which, in this case, has been saved every 0.01 seconds.

During the transient, the mesh has been h-adaptively refined and coarsened relying on the Kelly error estimator and they are reported in Fig. II.1. The corresponding computed solutions are depicted in Fig. II.2.

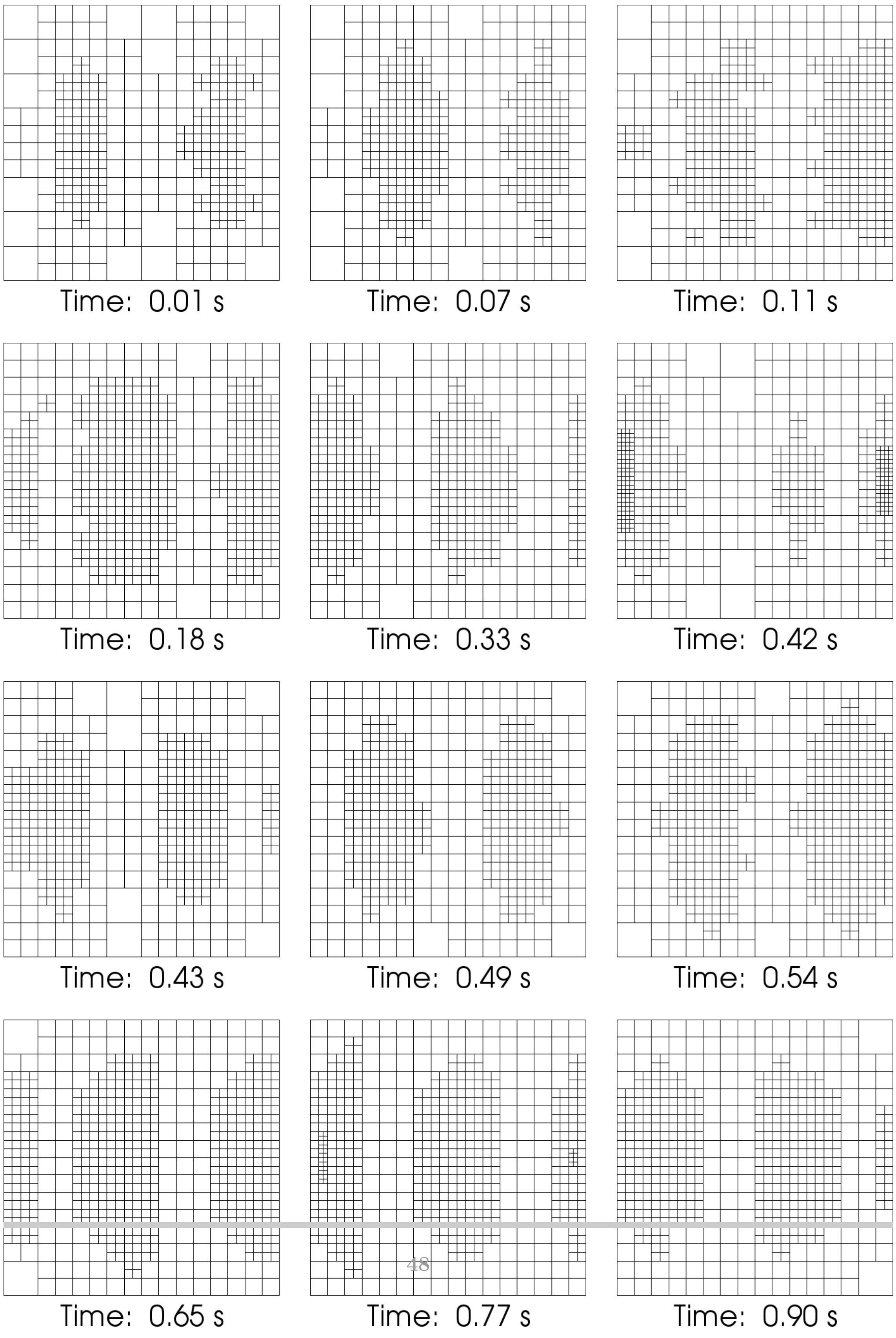


Figure II.1: Computed meshes during the transient.

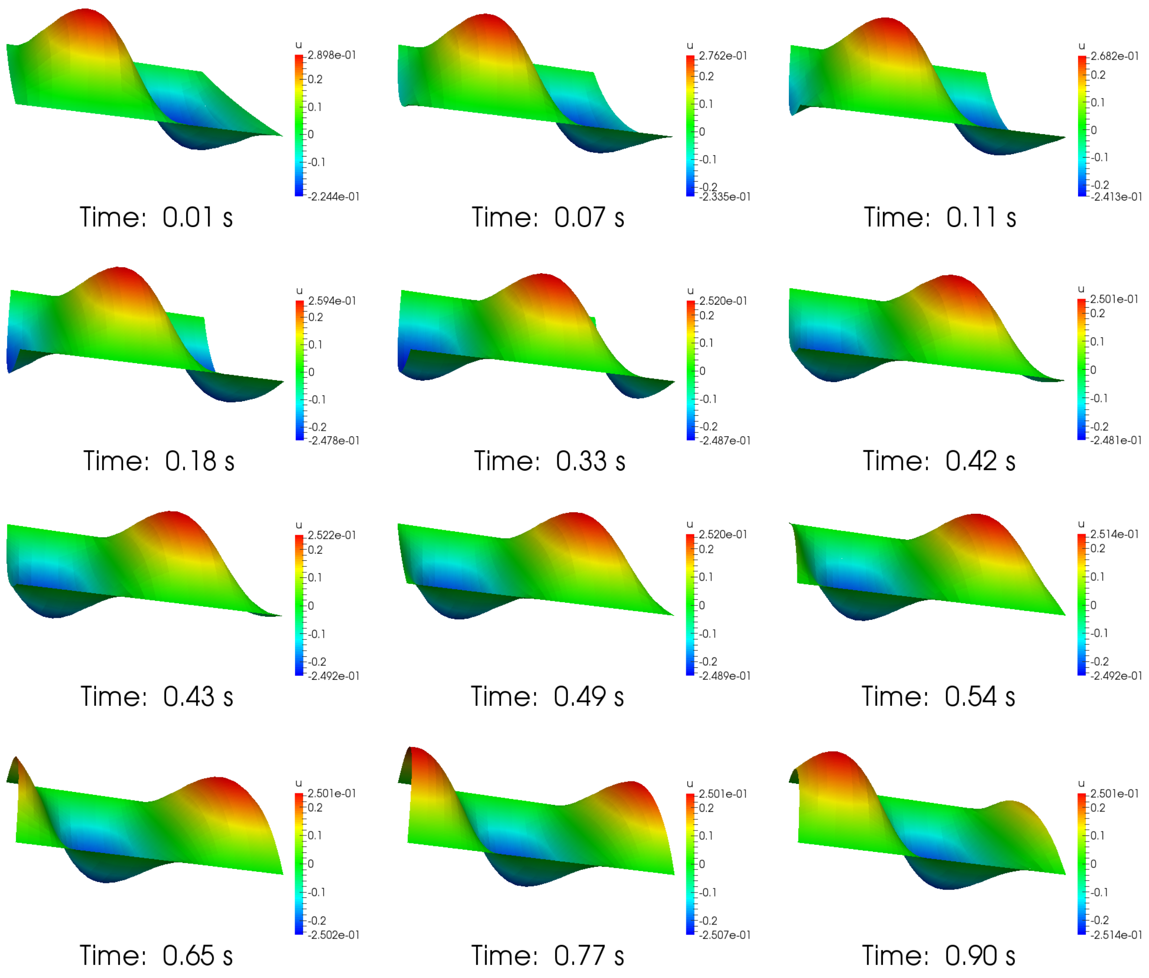


Figure II.2: Computed solution during the transient.

II.3.3. FLOW PAST A CYLINDER: STOKES EQUATIONS

As a last example, we provide an example saddle-point problem: the flow past a cylinder for low Reynolds number in a 2D settings. We used the `IdaInterface` to solve the dynamic Stokes equations:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} - \nu\Delta\mathbf{u} + \nabla p = 0 \quad \text{for} \quad x \in \Omega = (-1,3)\times(-1,1) \quad \forall t \in (0,20), \\ \operatorname{div}\mathbf{u} = 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in} \quad \Omega, \\ \mathbf{u} = f(x,t) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{on} \quad \partial\Omega, \\ \mathbf{u}(t=0) = (0,0) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in} \quad \Omega, \end{cases} \tag{II.10}$$

where $\mathbf{u}$ is the fluid velocity, ν is the kinematic viscosity, p is the pressure, and $f(t)$ represents the applied Dirichlet boundary conditions.On the left boundary (`boundary_id=1`) we impose a parabolic velocity profile, which is multiplied by a *ramp* function within the time interval $(0, 0.1)$, and we applied the no-slip condition on the cylinder surface (`boundary_id=5`).

Generated code

```
subsection Dirichlet BCs
  set IDs and component masks = 1=u % 5=u
  set IDs and expressions     = 1=(t<.1 ? (-1.0)*t*10.0*(y-1)*(y+1)/6.0 : \
                                (-1.0)*(y-1)*(y+1)/6.0 ); 0; 0 % 5=0; 0; 0
  set Known component names   = u,u,p
  set Used constants          =
end
```

We discretized the problem using the second order Taylor Hood finite element.

Generated code

```
subsection Finite Element
  set Block coupling                 = 1,1; 1,0
  set Blocking of the finite element = u,u,p
  set Finite element space           = FESystem[FE_Q(2)^dim-FE_Q(1)]
  set Preconditioner block coupling  = 1,0; 0,1
end
```

The mesh was imported by the file named `grid-2.2.ucd`

Generated code

```
subsection Domain
  set Grid to generate           = file
  set Input grid file name       = ../source/grid-2.2.ucd
  ...
end
```

During the transient, the mesh was locally refined using a Kelly error estimator [21] for the velocity components. Figure II.3 shows the initial mesh and the locally coarsened and refined mesh.

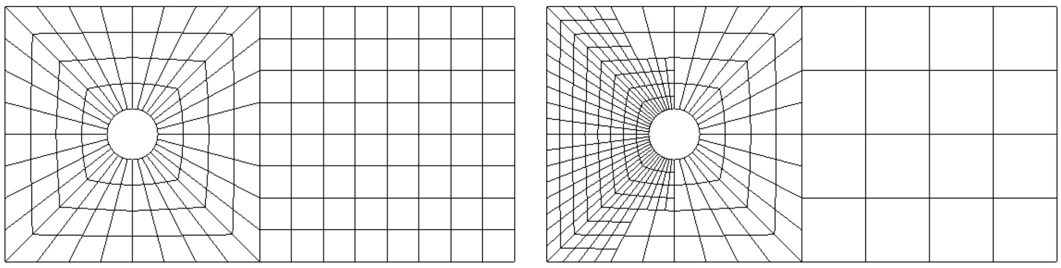


Figure II.3: Initial (left) and refined (right) mesh for the Stokes example.

We provide two snapshots of the solution, at time=0.2 and 20 seconds, in Fig. II.4.

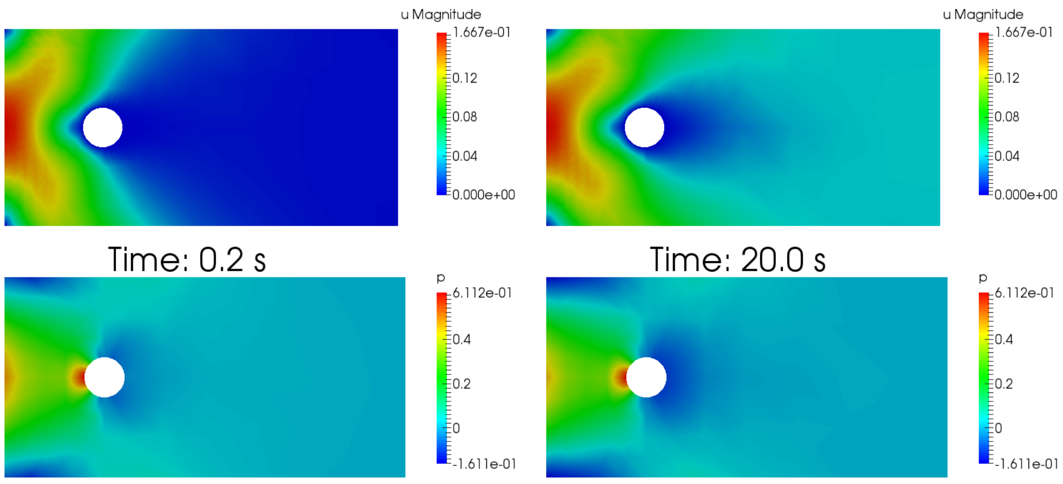


Figure II.4: Velocity magnitude (top) and pressure (bottom) at time=0.2 and 20.0 seconds.

Finally, we report the statistics of the summary of the calls to the various functions.

Generated code

```
+---------------------------------------------+------------+------------+
| Total wall clock time elapsed since start   |      4.71s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Jacobian matrix        |        28 |     0.425s |         9% |
| Compute error estimator         |       203 |     0.559s |        12% |
| Setup Jacobian                  |        28 |     0.427s |       9.1% |
| Solve system                    |        77 |      1.13s |        24% |
| Post-processing                 |       201 |      2.25s |        48% |
| Residual                        |        77 |     0.193s |       4.1% |
| Setup dof systems               |         4 |    0.0648s |       1.4% |
+---------------------------------+-----------+------------+------------+
```

It is worth mentioning that the system matrix has been assembled only 28 times, while the system has been solved 77 times. Within the parameter file, we set to output the solution every 0.1 seconds therefore two hundred times plus one at time=0. Since the system has been solved with a lower frequency, the solution that is stored is obtained through interpolation.

II.4. Conclusions

In this Chapter we presented version 1.0.0 of the `deal2lkit` library, which is a collection of modules for `deal.II` designed to provide a *high performance programming* experience to both beginner and advanced users of the `deal.II` library. One of the key feature of `deal2lkit` is the possibility to access most of the repetitive tasks related to writing complex finite element codes using parameter files. We gave a general overview of some of the main classes. The full documentation is generated automatically using Doxygen, and it is available online at `http://mathlab.github.io/deal2lkit`. Example programs were used to show the potential of `deal2lkit`, which allows fast prototyping of time dependent, linear and non-linear, scalar and vector problems, which are fully parallel, and support adaptive mesh refinement.

`deal2lkit` is in continuous development and new functionalities are constantly implemented to enrich those tools that are useful to develop prototype finite element codes efficiently, in a well tested environment. Currently the main application using `deal2lkit` is π-DoMUS, the Parallel Deal.II MUlti-physics Solver [34]. π-DoMUS exploits all of `deal2lkit` latest functionalities, including symbolic calculation of Jacobians and residuals, relying on the Sacado package [2] of Trilinos [16].

π-DoMUS III

III.1. Introduction

Nowadays, many remarkable simulations and industrial applications require a lot of hours for a single result: the running time is often measured in days, weeks, or even months. In this framework, you are interested in optimizing your code as much as possible and *HPC* is essential to reach this task: non parallel codes or source files without an header of math libraries are unthinkable on a cluster.

There are many optimized libraries for almost every purpose (e.g. `deal.II`, Trilinos, PETSc, BLAS, Plasma, P4est, tbb, etc ..) and with this in mind we do not intent to reinvent the wheel. We think that our contribution to these libraries in term of *HPC* could only "slightly" improve our codes. Nevertheless, advangarde research requires a result as soon as possible. Our solution to this request is the only available for a scientist: turn the code upside down as quickly as possible and test as much as possible solutions with some small benchmarks. This paradigm is summed up with the acronym *HPP* (High Performance Programmin): flexible code to focus your attention on the algorithm. In practice, you do not want to spend hours to optimize your code improving *for loops* or other low level ingredients if existing libraries have done better codes than you could write: it is very probable that an existing library could reach a lower order of time with respect to your own code. Ideally, you should try to optimize the algorithms (the math behind your code) as much as possible, before trying to optimize the code. This can improve your code of orders of magnitude of time.

In literature, you can find libraries devoted to specific problems: fluid dynamics, elasticity, mechanics, etc. One of the most successful libraries of this kind is ASPECT (Advanced Solver for Problems in Earth's ConvecTion, [5]). The purpose of this library is to handle problems concerning simulation of convection in the Earth mantle (see [22, 5]). ASPECT is what we mean for *HPP*: it is based on `deal.II` for the finite element content and `deal.II` uses Trilinos, PETSc, tbb, UMFPACK, and Boost for example for the *HPC* part.

Our intent is to handle multiphysic PDEs, in particular non-linear ones: the kind of PDEs we have in mind, admit a weak formulation (residual formulation) and sometime an energetic formulation. In the next section we will introduce the π-`DoMUS` (Parallel Deal.II MUltiphysics Solver, [34]) project. In terms of finite element library we make the same choice of ASPECT and base our library on `deal.II`. Since we are looking for a flexible tool we integrate `deal2lkit` to parse parameter. Eventually, we take advantages of the library IDA of `SUNDIALS` (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) for the advancement in time of our equations.

The main characteristic of π-`DoMUS` is the capability to construct automatically the system matrices (or system preconditioners) starting from a residual (or energetic) formulation which does not require the user to compute manually any Jacobian (or Hessian) matrix. Such goal is obtained by using the advanced package Sacado, included in the Trilinos library. Such package has the capability of computing symbolic derivatives of quantities in an automated and efficient way at compile time, resulting in assembly routines which perform similarly to those assembled manually, at a fraction of *uman time*.

III.2. Beyond the code

π-`DoMUS` (Parallel Deal.II MUltiphysics Solver, [34]) is a solver for PDEs in *residual form* or in *energetic form*. From the residual form or the energetic form, the software automatically constructs the system matrices in parallel using both MPI and multithreaded parallelizations. The aim of this software is to deal with multiphysics equations with non linear terms in a easy and intuitive way: a system of PDEs coupled together. An example are the Navier-Stokes equations for compressible fluid coupled with some costitutive equations for the density, the viscosity, or the temperature.

First of all it is better to explain what we mean for *residual* and *energetic* formulation. Let us start from the easier:

(III.2.1) DEFINITION: we say that a differential equation

$$\mathcal{L}[\dot{u}, u, t] = f \tag{III.1}$$

admits *an energetic formulation* if there is a functional

$$\begin{array}{rccc} \mathcal{E}: & \mathbb{R}^n & \to & \mathbb{R} \\ & u & \mapsto & \mathcal{E}[u] \end{array} \tag{III.2}$$

depending on u such that every solution of (III.1) is a stationary point of (III.2).

Notice that we are minimizing only with respect to u contrary to the classical sense of stationary point of energy.

A lot of PDEs (linear PDEs like Laplace equation, Stokes equation, and Heat Equation) admit an energetic formulation in the sense of III.2.1. Furthermore, there are important PDEs that do not admits such a formulation. Among these equation we could list a lot of multiphysics equations, such as the equations related to Neo-Hookean hyperelastic materials, and the well known Navier-Stokes Equations. In order to recover generality in our theory we reformulate the code in π-`DoMUS` to tackle general *residual* formulations:

(III.2.2) DEFINITION: Consider the following PDE:

$$\mathcal{L}[\dot{u}, u, t] = f. \tag{III.3}$$

The *residual formulation* of (III.3) is a functional

$$\begin{array}{rccc} \mathcal{R}: & \mathbb{R}^n \times \mathbb{R}^n & \to & \mathbb{R} \\ & (u, v) & \mapsto & \mathcal{L}[\dot{u}, u, t] \cdot v - f \cdot v \end{array} \tag{III.4}$$

such that if $\bar{u}$ is solution of III.3 then for every $v \in \mathbb{R}^n$, $(\bar{u}, v)$ is a solution for III.4.

III.2.1. π-DOMUS

π-`DoMUS` uses III.2.1 and III.3 to implement a multiphysics solver. It implement two different interfaces: `ConservativeInterface` for energetic problems and `NonConservativeInterface` for residual formulations. Every problem is an interface derived from the `ConservativeInterface` or the `NonConservativeInterface`. The user should implement in this file the energetic/residual formulation of the system and of the preconditioner. For the preconditioner it is possible to assemble more than one enery/residual: a priori one can combines different preconditioner or to choose different preconditioner according to the physic parameters.

An important feature of π-`DoMUS` is the complete integration with `deal2lkit`. This makes the code really flexible: every parameter can be changed once the code is compiled. Test different preconditioners or solve a family of PDEs can be done without recompile the code but changing a text file.

III.3. HPC

The main task of π-`DoMUS` is to provide a flexible *HPC* solver for multiphysics PDEs and parallelism is essential in this framework. Indeed, we are interested in solving huge linear systems with milions and even bilions degrees of freedom and geneally, a code that does not scale properly has no hope to solve such kind of problems. A well tested tool able to answer efficiently to this problem is the so called `WorkStream` of `deal.II`.

III.3.1. WORKSTREAM

Modern Finite Element codes usually presents the following pattern: a stream of local independent operation followed by a reduction into a global data structure. In [35] we see how such a software pattern, called `WorkStream` inside the `deal.II` library, can be efficiently implemented. An explicit synchronisation of the tasks would be quite inefficient and may not scale well. More importantly we must remember that, in floating point arithmetic, the order of a summation may lead to significant change in the result. Since we can't make any assumption on the order of creation of single task we conclude that with this manual synchronisation we can't obtain the same result two times in a row. To overcome such difficulties WorkStream separates

– the embarrassingly parallel local computations;

– the reduction operation.

The local computation can run in any order and in parallel, while the reduction operation must run on a single thread, it should avoid manual synchronisations and must perform the summation always in the same order. These constraints assures that the results are both reliable and repeatable. We stress that, for what concerns the local parallel computations, WorkStream usually schedules more tasks to the same thread in order to optimize computation time.

The typical application of such a class is the assembling a matrix in a FiniteElementMethod. In such a case we need to perform computation on each cell and then add together all the contributions inside the global matrix. The assemblage of the local contribution can be done on all the cells simultaneously. However we can't allow all the local contributions to be written at the same time on the global matrix because we would have race conditions and we would corrupt the data. Consequently, we want to ensure that only one thread at a time writes into the global matrix, and that results are copied in a stable and reproducible order. We need the following ingredients to use WorkStream

- A stream of object that is used by the scheduler to spawn the needed Tasks.
- A worker function to be run in parallel on all the objects to perform the local computations
- A copier that reduces the local contributions.

Generated code

```
typedef
FilteredIterator<typename DoFHandler<dim, spacedim>::active_cell_iterator>
CellFilter;
WorkStream::
run (CellFilter (IteratorFilters::LocallyOwnedCell(),
                 dof_handler->begin_active()),
     CellFilter (IteratorFilters::LocallyOwnedCell(),
                 dof_handler->end()),
     local_assemble,
     local_copy,
     Scratch(*mapping,
             *fe,
             quadrature_formula,
             energy.get_jacobian_flags(),
             face_quadrature_formula,
             energy.get_face_flags()),
     Assembly::CopyData::
     piDoMUSSystem<dim, spacedim> (*fe,n_aux_matrices));

compress(jacobian_matrix, VectorOperation::add);
```

In the following we analyze the basic features of the two functions needed by WorkStream.

Worker function

WorkStream assigns one worker function per threads. We need to keep in mind that all the workers must be able to run in parallel since WorkStream does not check for any race condition at this level. In the following we list the argument of the worker object.

- Input: a ScratchData object. The worker uses this object to make its computation. The object needs to have a working copy constructor since it is copied among all threads when WorkStream is run.
- Output: a CopyData object. Every worker fills its own CopyData. This object will be pass down by reference to the copier function.

- Additional parameters: if we need to pass more informations to the work stream we have to make sure that this information is the same for all the threads. Then we have two strategies.
  - We can use a std::bind function coupled with placeholder mechanism to obtain a function that takes only ScratchData and CopyData, in this way we are binding the additional parameters to some known values
  - We can exploit a functionality of C++11, the *lambda function*. A lambda function is a function that you can write inline in your source code. It uses a so-called capture that allows the user to pass additional information to the function. In this way we can straightforwardly limit the number of parameter of the function erasing the need of a binding.

Generated code

```
auto local_assemble = [ this ]
                      (const typename DoFHandler<dim, spacedim>::active_cell_iterator &
                           cell,
                       Scratch & scratch,
                       SystemCopyData & data)
{
  this->energy.assemble_local_system(cell, scratch, data);
};
```

Copier function

The copier function is the object that takes care of copy back the result of the worker computation back to the global memory. It takes as argument the reference to the CopyData object computed by the worker. For any additional parameter we can follow the same strategies of the worker function. The copier function automatically handles any race conditions, this means that it is responsible for any synchronisation overhead of the WorkStream class.

Generated code

```
auto local_copy = [ this ]
                  (const SystemCopyData & data)
{
  this->constraints.distribute_local_to_global (data.local_matrix,
                                                data.local_dof_indices,
                                                this->jacobian_matrix);
};
```

We recall here that all local matrices are assembled automatically using the Sacado library of Trilinos. Such implementation allows easy and fast prototyping of both system matrices and preconditioners.

III.4. Interfaces

In this Section we use the Stokes equation to show a possible interface for an energetic problem. The equation is the following:

$$\begin{cases} -\operatorname{div}\varepsilon(u) + \nabla p = f \\ \operatorname{div} u = 0 \end{cases}$$

where u is the velocity field, p is the pressure, f is the force, and $\varepsilon(u) = \frac{\nabla u + [\nabla u]^t}{2}$.

This problem has an "energetic" formulation:

$$\mathcal{E}(u,p) = \frac{1}{2}|\nabla(u)|^2 - p\operatorname{div}(u).$$

and therefore we use the *conservative interface*. The code starts with the inclusion of `interfaces/conservative.h` and the consequently derivation of the class `Stokes` form `ConservativeInterface`.

```
#include "interfaces/conservative.h"
...
template <int dim>
class Stokes : public ConservativeInterface<dim,dim,dim+1, Stokes<dim> >
{
public:
  ...
  void declare_parameters (ParameterHandler &prm);
  void parse_parameters_call_back ();
  ...
  template<typename Number>
  void preconditioner_energy( ... ) const;

  template<typename Number>
  void system_energy( ... ) const;

  virtual void compute_system_operators( ... ) const;
};
```

Notice that this code has mainly two kind of function: a group is composed by `declare_parameters` and `parse_parameters_call_back` and an other by `preconditioner_energy`, `system_energy`, and `compute_system_operators`.

The first group concerns parameter files while the second the math of the problem.

π-DoMUS it is designed in such a way if you need to change any parameters of the problem you will have to change just the parameter file without recompile anything. `declare_parameters` and `parse_parameters_call_back` take all the parameters that the user might want to change after different tests of the same code (e.g. density, velocity, boundary conditios, or initial conditions). These feature are inherit from the full integration of π-DoMUS in `deal2lkit` (Section II.2.1).

```
template <int dim>
void Stokes<dim>::declare_parameters (ParameterHandler &prm)
{
  ConservativeInterface<dim,dim,dim+1, Stokes<dim> >::declare_parameters(prm);
  this->add_parameter(prm, &eta, "eta [Pa s]", "1.0", Patterns::Double(0.0));
  this->add_parameter(prm, &block_back_substitution_bool, "use
      block_back_substitution", "false", Patterns::Bool());
}

template <int dim>
void Stokes<dim>::parse_parameters_call_back ()
{
  ConservativeInterface<dim,dim,dim+1, Stokes<dim> >::
      parse_parameters_call_back();
}
```

Finite elements, coupling, and all the numerical structure of the problem is passed through the constructor.

```
template<int dim>
Stokes<dim>::Stokes() :
  ConservativeInterface<dim,dim,dim+1,Stokes<dim> >
  ("Stokes",
    "FESystem[FE_Q(2)^d-FE_Q(1)]",
    "u,u,p", "1,1; 1,0", "1,0; 0,1","0,0")
{};
```

In `system_energy` you write the energy of the system. A lot of mathematical functions (e.g. `scalar_product`, `grad`, `grad_sym`, `transpose`, etc ...) are implemented in order to simplify this routine:

```
void Stokes<dim>::system_energy(...) const
{
  ...
  energy = 0;
  for (unsigned int q=0; q<n_q_points; ++q)
    {
      ...
      Number psi =  eta * .5 * scalar_product(sym_grad_u,sym_grad_u)
                    - p*div_u;
      energy += psi*JxW[q];
    }
}
```

and the same procedure has to be done for the preconditioner:

```
void Stokes<dim>::preconditioner_energy(...) const
{
  ...
  energy = 0;
  for (unsigned int q=0; q<n_q_points; ++q)
    {
      ...
      Number psi =  (1./eta)*p*p
      energy += psi*JxW[q];
    }
}
```

At this point we are ready to assemble all the blocks of the system matrix and of the preconditioner matrix. In this part the `LinearOperator` class (Section I.3) is crucial to have flexibility and do not lose performances:

```
template <int dim>
void
Stokes<dim>::compute_system_operators(...) const
{
  auto A      = linear_operator< VEC >( matrix.block(0,0) );
  auto Bt     = linear_operator< VEC >( matrix.block(0,1) );
  auto B      =  transpose_operator(Bt);
  auto ZeroP  = null_operator< VEC >( matrix.block(1,1) );

  auto Mp     = linear_operator< VEC >( preconditioner_matrix.block(1,1) );

  auto A_inv     = inverse_operator( A, solver, Amg_preconditioner);
  auto Schur_inv = inverse_operator( Mp, solver, Mp_preconditioner);

  auto P00 = A_inv;
  auto P01 = null_operator(Bt);
  auto P10 = Schur_inv * B * A_inv;
  auto P11 = -1 * Schur_inv;

  system_op  = block_operator<2, 2, VEC >({{
      {{ A, Bt }} ,
      {{ B, ZeroP }}
    }
  });

  prec_op = block_operator<2, 2, VEC >({{
      {{ P00, P01 }} ,
      {{ P10, P11 }}
    }
  });
}
```

III.5. Examples

The research of a *Schur* complement for a *saddle point system* is a well know problem of Numerical Analysis. In few words, dividing the block of the velocity from the block of the pressure, many PDEs has the same representation in term of linear system:

$$\begin{pmatrix} A & B^t \\ 0 & B \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}. \tag{III.5}$$

A problem that admits a numerical representation in the form of (III.5) is called a *saddle point problem*.

Let A^{-1} be an inverse for A and $S := BA^{-1}B^t$ the so called *Schur complement*. It is straightforward that multiplying (III.5) for the following block matrix

$$\begin{pmatrix} A^{-1} & 0 \\ S^{-1}BA^{-1} & -S^{-1} \end{pmatrix} \tag{III.6}$$

uncouples the pressure block of (III.5) and leads to a simpler problem (the resulting system is upper diagonal). Therefore, the idea is to use (III.6) as starting point for a suitable preconditioner for (III.5). There are two missing ingredients for this recipe: an approximation for A^{-1} and one for S^{-1}. In this Section we focus our attention to the research of S^{-1}.

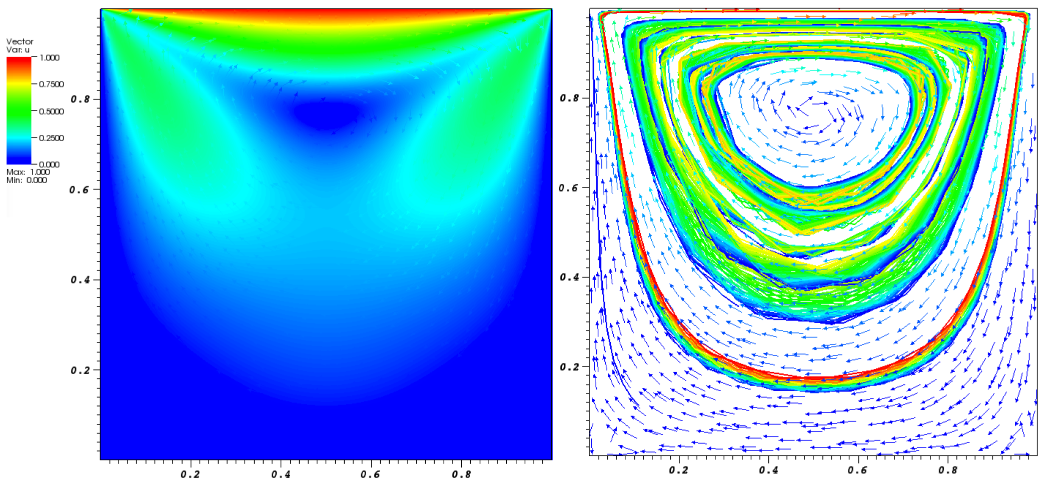
In the following we are going to consider the *Navier-Stokes Equations* as a case study:

$$\begin{cases} \rho\frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v}\cdot\nabla)\mathbf{v} = -\nabla p + \nu\Delta\mathbf{v} + \mathbf{f}(\boldsymbol{x},t) \\ \operatorname{div} v = 0. \end{cases} \tag{III.7}$$

where v is the velocity field, p the pressure, and $\mathbf{f}$ represents the external forces. This problem is behind a lot of problems of fluid dynamic and admits a saddle point representation. Indeed, we are going to look for a Schur complement for (III.7). We approximate S in two different ways: the first is obtained using $\frac{1}{\nu}M_p$ that we call *stokes* and the second using $\frac{1}{\Delta t * \rho}A_p$ that we call *low-nu*. M_p is the mass matrix for the pressure block and A_p is the matrix representing a Laplace Equation solved for the pressure block.

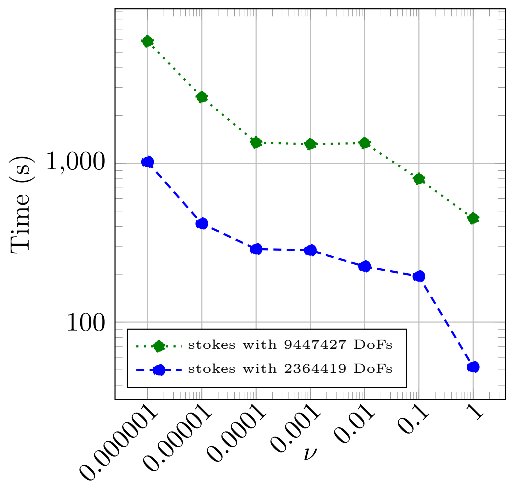
III.5.1. STATIONARY NAVIER-STOKES EQUATIONS

In these subsection we focus on the so called *Lid Cavity*: we solve the Navier-Stokes Equations is a $2D$ $1x1$ square and impose $v = (0,0)$ on the bottom, right, and left wall while we impose $v = (1,0)$ on the top.

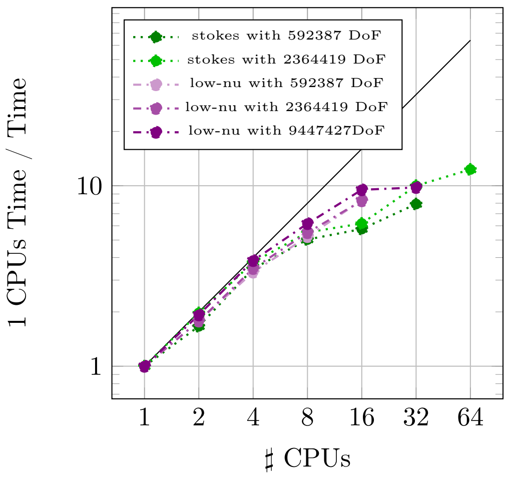


(a) Velocity magnitude (b) Velocity vector field with streamlines

Figure III.1: Lid Cavity



(a) 40 CPUs, 148739 DoF, and Fe=$Q_2 Q_1$



(b) Strong scalability

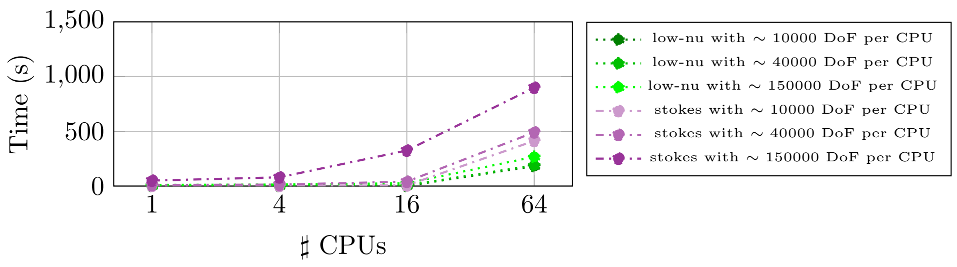
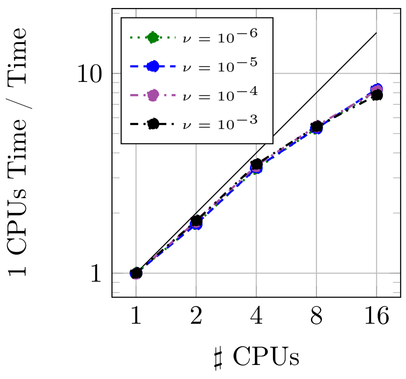
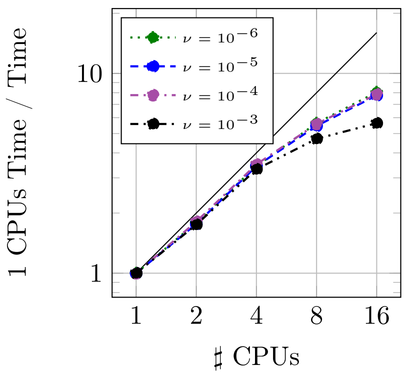


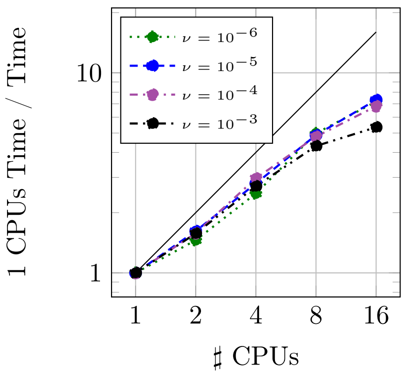
Figure III.3: Weak scalability obtained wth different preconditioners and $\nu = 0.0024$



(a) DoF=148739 and Fe=$Q_2 Q_1$



(b) DoF=362499 and Fe=$Q_3 Q_2$



(c) 674563 DoF and Fe=$Q_4 Q_3$

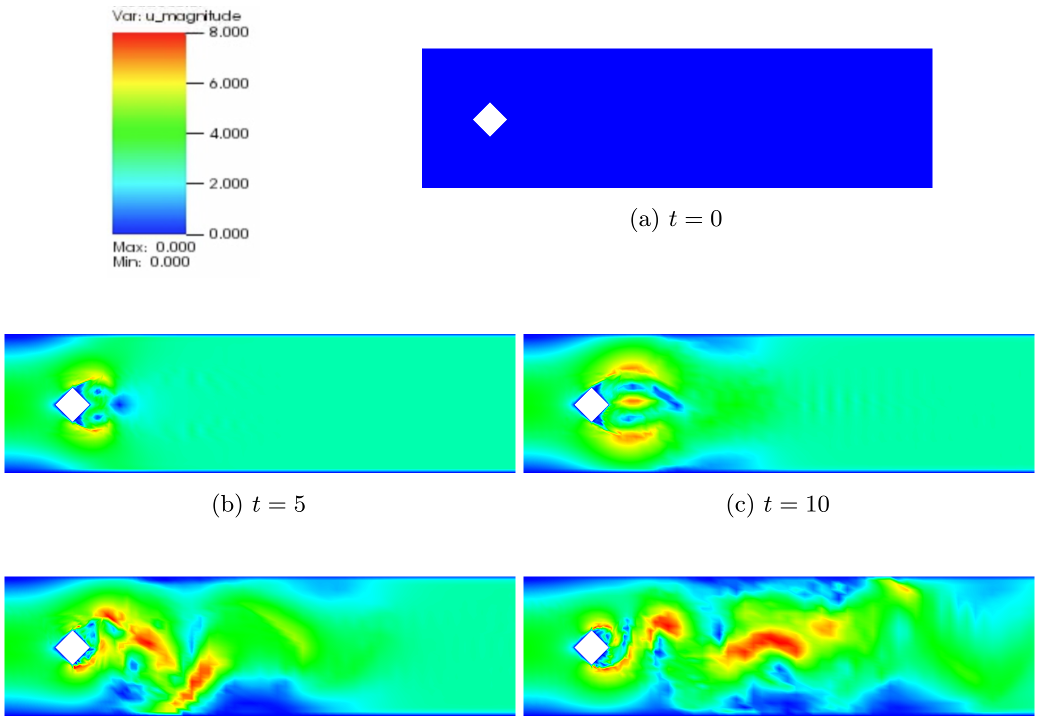
Figure III.4: Strong scalability of different Finite Elements obtained using a "stokes" preconditioner

III.5.2. TIME DEPENDENT NAVIER-STOKES EQUATIONS

This subsection concludes the examples giving a sample of a time depending Navier Stokes simulation. We simulate $20s$ with *Reynolds* number equal to 400 ($\nu \sim 0.0024$). We consider a $2D$ $[0, 20] \times [-1, 1]$ rectangle and impose $v = (0, 0)$ on the top and on the bottom, and on the left side:

$$\begin{cases} v = \big(t(1-y^2), 0\big) & \text{if } t < 1 \\ v = \big((1-y^2), 0\big) & \text{otherwise.} \end{cases}$$

The result is the following:



(a) $t = 0$

(b) $t = 5$

(c) $t = 10$

(d) $t = 15$

(e) $t = 20$

Figure III.5: Navier-Stokes simulation at $t = 0$, $t = 5$, $t = 10$, $t = 15$, and $t = 20$

Bibliography

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

[2] Sacado Web page, 2015. URL https://trilinos.org/packages/sacado.

[3] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, jan 2012.

[4] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[5] W. Bangerth, T. Heister, et al. Aspect: Advanced solver for problems in earth's convection. 2015. `https://aspect.dealii.org/`.

[6] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, and B. Turcksin. The `deal.II` library, version 8.3. *preprint*, 2015.

[7] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T.D. Young. The `deal.II` Library, Version 8.2. *Archive of Numerical Software*, 3, 2015.

[8] W. Bangerth and O. Kayser-Herold. Data structures and requirements for hp finite element software. *ACM Trans. Math. Softw.*, 36(1):4:1–4:31, mar 2009.

[9] 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.

[10] K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial-value Problems in Differential-algebraic Equations*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1996.

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

[12] The deal2lkit Authors. A toolkit library for deal.ii. `https://github.com/mathLab/deal2lkit`, 2015.

[13] S. Delcourte and J. Delphine. Saddle point preconditioners for linearized navier–stokes equations discretized by a finite volume method. *Applied numerical mathematics*, 60(11):1054–1066, 2010.

[14] H. Elman, W. Howle E, J. Shadid, R. Shuttleworth, and R. Tuminaro. Block preconditioners based on approximate commutators. *SIAM Journal on Scientific Computing*, 27(5):1651–1668, 2006.

[15] G. Gaël, J. Benoît, et al. Eigen v3. 2010.

[16] M.A. Heroux, Ro.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, and K.S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[17] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, September 2005.

[18] A.C. Hindmarsh, R. Serban, and A. Collier. User documentation for ida v2. 8.1 (sundials v2. 6.1). 2015.

[19] B.l Janssen and G. Kanschat. Adaptive multilevel methods with local smoothing for h^1- and h^{curl}-conforming high order finite element methods. *SIAM Journal on Scientific Computing*, 33(4):2095–2114, 2011.

[20] G. Kanschat. Multilevel methods for discontinuous galerkin {FEM} on locally refined meshes. *Computers & Structures*, 82(28):2437 – 2445, 2004. Preconditioning methods: algorithms, applications and software environments.

[21] D. W. Kelly, J.P. De S.R. Gago, O.C. Zienkiewicz, and I. Babuska. A posteriori error analysis and adaptive processes in the finite element method: Part I-error analysis. *International Journal for Numerical Methods in Engineering*, 19(11):1593–1619, 1983.

[22] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods. *Geophysics Journal International*, 191:12–29, 2012.

[23] M. Kronbichler and K. Kormann. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, 63:135 – 147, 2012.

[24] M. Maier, M. B., and L. Heltai. Linearoperator – a generic, high-level expression syntax for linear algebra. Technical report, SISSA, 2015.

[25] A. Mola, L. Heltai, and A. DeSimone. A stable and adaptive semi-lagrangian potential model for unsteady and nonlinear ship-wave interactions. *Engineering Analysis with Boundary Elements*, 37(1):128 – 143, 2013.

[26] M.A. Olshanskii and Y.V. Vassilevski. Pressure schur complement preconditioners for the discrete oseen problem. *SIAM Journal on Scientific Computing*, 29(6):2686–2704, 2007.

[27] Alberto Sartori, Nicola Giuliani, Mauro Bardelloni, and Luca Heltai. Deal2lkit: a toolkit library for high performance programming in deal.ii. 2015.

[28] M.S. Shephard. Linear multipoint constraints applied via transformation as part of a direct stiffness assembly process. *International Journal of Numerical Methods in Engineering*, 20:2107–2112, 1985.

[29] G. Strang. A proposal for toeplitz matrix calculations. *Stud. Appl. Math.*, 74(2):171–176, April 1986.

[30] B. Stroustrup. *C++ Programming Language, The, 4th Edition.* Addison-Wesley Professional, 2013.

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

[32] The deal.II Authors. step-40. `https://www.dealii.org/developer/doxygen/deal.II/step_40.html`.

[33] The deal.II Authors. step-6. `https://www.dealii.org/developer/doxygen/deal.II/step_6.html`.

[34] The π-DoMUS Authors. π-DoMUS. Parallel Deal.II MUlti-physics Solver. In preparation. `https://github.com/mathLab/pi-DoMUS`, 2015.

[35] B. Turcksin, M. Kronbichler, and W. Bangerth. Workstream–a design pattern for multicoreenabled finite element computations.

[36] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide.* 2002.

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

[38] A.J. Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015.