Proceedings of Equadiff-11
2005, pp. 1–10

FLENS — A FLEXIBLE LIBRARY FOR EFFICIENT NUMERICAL SOLUTIONS

MICHAEL LEHN , ALEXANDER STIPPLER , AND KARSTEN URBAN *

Abstract. In this paper we describe the main design and realization principles of our software library FLENS (A Flexible Library for Efficient Numerical Solutions). FLENS is a C++ library allowing easy and straightforward coding while providing a maximum extend of efficiency. FLENS is in particular suited as a platform for the realization of fast solvers for differential equations.

AMS subject classifications.

1. Introduction. Nowadays, a whole variety of software packages for numerically solving differential equations is available. Commercial tools offer solvers even for complex industrial problems. Why the need for a new one?

Our software package FLENS is motivated by several aims and scopes, in particular it should be

- used both for teaching and research;
- flexible in the sense that is easily possible for a user to extend the library. In particular, it should be easy to realize and test new numerical methods;
- open source and free for students and researchers;
- highly efficient providing a platform also for complex industrial applications.

The use of commercial numerical software in particular undergraduate classes may conflict with high fees for the corresponding licences. Thus it was one source of motivation to provide our students with a free but easy to use package. This also includes graduate and PhD students writing the particular thesis in our group. Those students enter the group and should be able to become familiar with a software package within short time. Moreover, such persons typically leave the group after a certain amount of time after finishing the particular thesis. The software that has been produced in the framework of the thesis should be written in such a way that is can be used for further research.

On the other hand, we work on numerical methods for pde's since a couple of years, in particular on adaptive wavelet methods. It has become clear that the full potential of such methods cannot be shown by using standard existing software. The new theoretical paradigm also requires a corresponding new software design.

The third issue is that we have a variety of third party research projects, some of them in close cooperation with industrial partners. For these purposes, we need a flexible and efficient software platform that enables us also to provide numerical schemes for such applications.

These goals seem to be conflicting. In particular, known packages are typically *either* highly efficient *or* easy to handle. We introduce some of the design principles of FLENS that bridge these two demands. The framework of FLENS is suitable both for academic and commercial applications. It my be downloaded via the webpage `flens.sourceforge.net`.

This paper is organized as follows. In Section 2 we describe the design of our matrix and vector types allowing a clear separation of data and algorithm, an easy

*University of Ulm, Department of Numerical Analysis, 89069 Ulm, Germany

usage without loss of efficiency. This is in particular achieved by avoiding virtual functions. Section 3 is devoted to the description of the efficient realization of basic matrix-vector operations. Here, we also make use of BLAS routines, [4]. In Section 4, we show the possible interaction of FLENS with other numerical libraries such as LAPACK. One feature particularly useful for adaptive numerical schemes, namely *slices* of matrices and vectors, are described in Section 5. In Section 6, we show one short example how to solve pde's with FLENS and we end with some comments on efficiency and benchmarks in Section 7.

2. Implementation of matrix and vector types. Polymorphism means that a particular interface is provided by entities of different types [1]. In C++ this is realized through base classes and thereof derived classes. The base class defines the interface that has to be provided by all derived classes. For our purpose minimal interfaces for matrix/vector types are defined in the corresponding base classes `Matrix` and `Vector`. For the sake of simplicity we consider in this section just an interface that enforces that all concrete implementations support index based element access and have methods to retrieve row and column dimensions. Based upon this, a function `minij` that initializes a matrix A with $(A)_{ij} = \min\{i, j\}$ can obviously be implemented like this:

```
void minij(Matrix &A)
{
    for (int i=1; i<=A.numRows(); ++i) {
        for (int j=1; j<=A.numCols(); ++j) {
            A(i,j) = min(i,j);
        }
    }
}
```

This function will work with *any* matrix implementation derived from `Matrix`. With this example we address an issue that usually comes along with polymorphism: virtual function calls [2].

While the function `minij` is written for the matrix base type, the specialized member functions have to be dispatched for derived classes. In C++, this is usually achieved by declaring *virtual* member functions in the base class. Calls to such a virtual function are dispatched using a lookup table which contains pointers to the specialized versions of the method. Due to this indirection, using virtual functions inside loops may lead to a considerable loss of efficiency. As the actual type of the matrix object is not known at compile time, the indirection can not be avoided through inlining.

Virtual functions can be avoided by a technique that became known as the Barton-Nackman-Trick [3], which is illustrated in the following for the matrix hierarchy:

```
template <typename Impl>
class Matrix
{
    public:
        Impl &
        impl() {
            return static_cast<Impl &>(*this);
        }
```

```
        double &
        operator()(int row, int col)
        {
            impl()(row, col);
        }
};

class GeneralMatrix : public Matrix<GeneralMatrix>
{
    public:
        double &
        operator()(int row, int col)
        {
            // ...
        }
};
```

The type of the derived class is provided to the base class through a template parameter. If we just adopt the declaration part of the above function `minij` we can see that now polymorphism can be achieved without performance penalty:

```
template <typename I>
void
minij(Matrix<I> &A)
{
    // ... as above ...
}
```

Calling a method in the base class, e.g., `A(i,j)` results in a conversion to the derived class and a subsequent call of the specialized version. As the type of the derived class is now known at compile time, this can be inlined by a compiler. Thus, specialized methods in `GeneralMatrix` are called directly. In fact, writing functions in terms of our base classes comes without any additional runtime overhead.

FLENS provides matrix implementations that correspond to those defined in BLAST [6]. These are `GeneralMatrix`, `TriangularMatrix`, `SymmetricMatrix` and `HermitianMatrix`. Those are further parameterized with respect to

1. Element type, which can be:
   `float`, `double`, `complex<float>`, `complex<double>`,
2. row or column oriented storage,
3. full, banded or (except for `GeneralMatrix`) packed storage[1].

In Section 6 we will describe how users can add new matrix/vector types.

3. Basic matrix-/vector operations. FLENS implements an efficient mechanism to evaluate linear algebra expressions. Temporary objects are hereby avoided where possible. For appropriate standard operations BLAS routines are used to achieve high performance. While the core of this mechanism is hidden from the user inside the library, it can easily be extended to support operations on user defined matrix types (see Section 6 below). We first outline drawbacks of a straightforward but naive implementation and conclude with an outline of the mechanism used by FLENS.

[1] See [4] for details

Consider the expression $z = A^T x + y$ where x, y and z are vectors and A is some matrix type of appropriate dimension. We want that this can be coded in the most natural and readable way:

```
z = transpose(A)*x+y;
```

In order to realize such a notation in C++, one has to overload operators `*` and `+` and provide a function `transpose`[2]. In a naive implementation operators and functions would compute and return the result of the operation as follows:

```
const VectorType
operator*(const MatrixType &A, const VectorType &x)
{
    // compute A*x and return result
}
```

Behind the scene computation of $z = A^T x + y$ would trigger in this case

1. the computation of $t_1 = A^T$ (i.e., one temporary matrix),
2. the computation of $t_2 = t_1 x$ (i.e., one temporary vector),
3. the computation of $t_3 = t_2 + y$ (i.e., a second temporary vector) and
4. the assignment $z = t_3$ (i.e., copying a vector).

Hence a lot of CPU time and memory is just consumed for the sake of a readable notation. Now we are going to describe a natural and efficient realization.

For the evaluation of linear algebra expression, BLAS routines can be regarded as building blocks. Single BLAS routines can cover the computation of compound (but still simple) expressions. Let us consider how the above example can be computed using BLAS routines. Only two BLAS routines are needed. For a general matrix A with full storage and dense vectors x and y the routine `gemv` is capable to perform operations of type $y = \alpha \mathrm{op}(A)x + \beta y$, where α, β are scalars and $\mathrm{op}(A)$ represents A, A^T or A^H. Obviously α is a scaling parameter for the product while β can be regarded as an update parameter for the left-hand side of an assignment. For other matrix types routines of similar or same form are provided. For the sum of two dense vectors the routine `axpy`, capable to perform $y = \alpha x + y$, is used. Putting both pieces together, we compute $z = A^T x + y$ in two steps:

1. compute $z = A^T x$ using `gemv` with $\alpha = 1$, $\beta = 0$ and $\mathrm{op}(A) = A^T$,
2. compute $z = z + y$ using `axpy` with $\alpha = 1$.

Hence, no temporary objects are created. Hardware vendor tuned BLAS implementations will typically recognize cache sizes and other architecture specific features providing near optimal performance.

To combine the neat notation through operator overloading with the high performance of BLAS implementations, we adopt the closure concept known from functional programming. In C++ this can be realized by a technique that became known as *expression templates* ([7]). In FLENS, we refined the expression template technique to ease integration of new, user defined matrix/vector types and to simplify maintainability ([8]). For matrix/vector operations that involve types that are not supported by BLAS, user defined BLAS-style functions must be provided (see Section 6 below).

The general idea is that an operator is overloaded such that it merely returns a closure object, i.e., an object containing references of the operands and whose type represents the operation. A closure in turn can be an operand of an operation. So finally the right-hand side of an assignment ends up as a single composite closure object. In the above example this composite closure object would then contain references

[2] Unfortunately C++ does not allow to support the notation `A'` to express A^T.

of A, x and y, while its type would represent the operation $A^T x + y$. For its evaluation, FLENS provides a mechanism that 'breaks' the closure into such parts that can be evaluated by BLAS routines. In our example the evaluation is done through the BLAS routines `gemv` and `axpy` as demonstrated above.

Usage of overloaded operators and expressive functions like `transpose` reduces the risk of error prone code. At the same time, an at most negligible runtime overhead is added. This is mainly due to the fact that the type of a closure object represents the type of the encapsulated operation. Thus, a compiler is capable to map simple expressions directly onto corresponding BLAS routine calls.

4. Using FLENS with other numerical libraries. Matrix/vector types that are implemented in FLENS provide methods to directly access internal data. In particular we explicitly allow the user to receive pointers to underlying storage schemes. It is therefore possible to use FLENS together with other numerical libraries. The storage schemes of our matrix types are conforming to the schemes documented in BLAST ([6]) and can directly be passed to LAPACK routines ([5]). But obviously working with pointers always requires some extra caution even if the structures behind are well documented. Some object oriented libraries completely prohibit direct access for this reason. The compromise in FLENS is kind of a gentlemen's agreement. Other libraries should only be accessed through well-tested wrapper functions and methods accessing internal data only used herein. For many LAPACK routines, FLENS already provides such wrappers.

In addition, FLENS provides a convenient notation for calling numerical routines. The motivation is simply that in every assignment the output parameter should be on the left-hand side whereas input parameter should be on the right-hand side. However, C++ only allows one single object on the left-hand side. Spending some thoughts, the bottom line is that the desired notation can be achieved allowing more than one and even a variable number of output parameter on the left-hand side.

To illustrate functionality and notation, we consider the computation of eigenvalues and right/left eigenvectors. For this purpose, FLENS offers the function `eig`. For real-valued symmetric matrices it acts as wrapper to the LAPACK routine `syev`:

```
DenseVector     d;
SymmetrixMatrix A;
GeneralMatrix   VL, VR;

d           = eig(A);    // only eigenvalues
(d, VR)     = eig(A);    // eigenvalues and right eigenvectors
(VL, d)     = eig(A);    // eigenvalues and left eigenvectors
(VL, d, VR) = eig(A);    // eigenvalues, left and right eigenvectors
```

The neat thing is that types of the output parameter and their ordering control what actually is computed. While the implementation details are beyond the scope of the present paper, what basically gets achieved is that a wrapper routine gets called receiving references of the left hand side objects and const references of the right-hand side objects.

This also solves the problem one usually faces when functions return large objects. In a straightforward implementation, a function would return a temporary object that gets copied. The advantage of our mechanism becomes obvious when we consider the LU-decomposition of a matrix. With `A = lu(A)`, the matrix A is overwritten by the decomposition. This happens while no temporary matrix is created and no matrix

copied.

Following a well-defined pattern, users can write their own wrappers providing the same capabilities. Again, these kind of wrappers can be treated by compilers such that no runtime overhead is added.

5. Working with matrix/vector slices. Many numerical applications demand operations on parts of matrices or vectors. Common examples for such parts would be sub-matrices or single rows/columns of a matrix. Creating copies of such slices would in some cases cause an unnecessary waste of memory. Instead, we allow to merely reference them. As this concept bears on similarities with *views* in database systems, we will adopt this term to denote referenced matrix/vector slices. For a convenient way to express index ranges an object named `_` is used. Hereby `_(2,7)` denotes the indices from 2 to 7 and `_(2,2,7)` the indices from 2 to 7 with stride 2, i.e. $\{2, 4, 6\}$. A maximal index range can be expressed by `_` without arguments. Obviously this shows similarities with the MATLAB notation in which `_(k, 1)` is playing the role of `(k:1)`. Here some examples:

FLENS notation	mathematical meaning
`A(_,3)`	$(A_{i,3})_{i=1,\ldots,m}$ (i.e., 3-rd column of A)
`A(3,_)`	$(A_{3,j})_{j=1,\ldots,n}$ (i.e., 3-rd row of A)
`A(_(2,6),_(4,7))`	$(A_{i,j})_{i=2,\ldots,6;\ j=4,\ldots,7}$

Whether matrix views should finally get copied or referenced can be expressed with the usual C++ semantic

```
GeneralMatrix &B = A(_(2,6), _(4,7));
GeneralMatrix C = A(_(2,6), _(4,7));
```

Here, B will be an alias for a sub-matrix while C is a copy of it. For FLENS users the concept of views is completely transparent. Matrix/vector views behave just as regular matrix/vector types.

6. Integration of user defined matrix-/vector types. To illustrate how users can add new matrix/vector types to FLENS, we consider the Poisson problem on the unit square $\Omega = (0, 1)^2$:

$$\begin{cases} -\Delta u &= \\ f & \text{in } \Omega, \\ u &= \\ g & \text{on } \partial\Omega. \end{cases}$$

Using the finite difference method with the standard 5-point stencil for discretization, leads to a linear system of equations $A_h u_h = f_h$. Hereby, the matrix A_h is a block matrix of following structure

$$A_h = \begin{pmatrix} T & -I & & \\ -I & T & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix}, \quad T = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}.$$

Obviously, it would be a waste of memory and CPU time to actually set up this matrix. Instead, one would want to implement a matrix class `Poisson2D` whose instances just stores the dimension of the matrix:

```
class Poisson2D
    : public Matrix<Id>
{
    public:
        Poisson2D(int n);

    private:
        int n;
};
```

Now assume that we want to use the conjugate gradient method to solve the linear system of equations. In FLENS, iterative methods like this are implemented as generic functions and only require that the needed matrix/vector operations are defined for the involved types. If for u_h and f_h FLENS builtin types are used, only a user defined function for the matrix-vector product $A_h u_h$ has to be implemented. Such an implementation has to follow a certain naming and signature convention with the BLAS-style functionality[3]: $y = \alpha \mathrm{op}(A)x + \beta y$.

Without further effort, user defined operations are integrated into the evaluation mechanism described in Section 3. This means in particular that an expression like $z = Ax + y$, where A is of type `Poisson2D`, gets evaluated without creation of temporary objects.

In a similar fashion matrix-matrix products or matrix-matrix sums can be implemented for user defined types.

7. Performance benchmarks. It is not completely obvious what could be a reasonable benchmark for the efficiency of our design and its realization. Since we provide a wrapping mechanism in order to use efficient BLAS or LAPACK routines in a user-friendly way, the optimum we can reach is the performance of the pure BLAS or LAPACK routines. This has been tested for a whole variety of applications clearly showing that we do not lose in this comparison. Of course, the compile time is growing, but not the execution time.

As a second test, we performed comparisons of standard LU-decomposition to solve a linear system of equation for FLENS on the one hand and for MATLAB on the other hand. Depending on the underlying matrix and the size of the problem, FLENS is faster by a factor between 2 and 5.

REFERENCES

[1] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1997.

[2] K. Driesen and U. Holzle, *The direct cost of virtual function calls in C++*, In OOPSLA'96 Conference Proceedings, volume 31, 10 of ACM SIGPLAN Notices, pages 306–323, New York, NY, USA, Oct. 1996. ACM Press.

[3] J. Barton and L. Nackman, *Scientific and engineering C++*, Addison Wesley, 1994.

[4] BLAS, *Basic Linear Algebra Subprograms*, http://www.netlib.org/blas.

[5] LAPACK, *Linear Algebra Package*, http://www.netlib.org/lapack.

[6] BLAST, *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, http://www.netlib.org/blas/blast-forum.

[7] Todd L. Veldhuizen, *Expression templates*, C++ Report, 1995.

[8] M. Lehn, *Implementation of Linear Algebra Packages in C++*, Preprint, Universit"at Ulm, 2005.

[3] As A in this case is a symmetric matrix the implementation can assume that $\mathrm{op}(A) = A$