AN ENVIRONMENT FOR SYMBOLIC AND NUMERIC COMPUTATION

G. DOS REIS
CMLA, ENS Cachan - CNRS (UMR 8536)
61, Avenue du Président Wilson, 94235 Cachan - Cedex, France
dosreis@cmla.ens-cachan.fr

B. MOURRAIN* & PH. TRÉBUCHET*
GALAAD, INRIA,
BP 93, 06902 Sophia Antipolis, France
{mourrain,ptrebuc}@sophia.inria.fr

F. ROUILLIER
SPACES, INRIA,
615, rue du Jardin Botanique, B.P. 101, 54602 Villers-lès-Nancy Cedex France
Fabrice.Rouillier@loria.fr

We describe the environment for symbolic and numeric computations, called SYNAPS (Symbolic and Numeric A PPlicationS) and developed in C++. Its aim is to provide a coherent platform integrating many of the nowadays freely available software in scientific computing. The approach taken here is inspired by the recent paradigm of software developments called active library. In this paper, we explain the design choices of the kernel and their impact on the development of generic and efficient codes for the treatment of algebraic objects, such as vectors, matrices, univariate and multivariate polynomials. Implementation details illustrate the performance of the approach.

1 Introduction
The need to combine symbolic and numeric computations is ubiquitous in many problems. Starting with an exact description of the equations, in most cases, we will eventually have to compute an approximation of the solutions. Even more, in many problems the coefficients of the equations may only be known with some inaccuracy (due, for instance, to measurement errors).

The aim of the SYNAPS* project is to provide a coherent platform for handling such problems, and integrating many of the nowadays freely available software in scientific computing. The kernel of this platform is based on the former library ALPb to which we incorporate new genericity and spe-

*PARTIALLY SUPPORTED BY THE PROJECT ECG IST-2000-26473
*http://www-sop.inria.fr/galaad/logiciels/synaps/
*bhttp://www-sop.inria.fr/galaad/logiciels/ALP/

paperproc: submitted to World Scientific on May 3, 2002
cialization mechanisms. This kernel provides data-structures and classes for the manipulation of basic objects, such as (dense, sparse, structured) vectors, matrices, univariate and multivariate polynomials. The aim of this paper is to describe the design of this kernel.

As SYNAPS is targeting different domains of application, it has to provide various internal representations for the abstract data-types required by algorithms specifications. Thus, this library makes it possible to define parameterized but efficient data structures for fundamental algebraic objects such as vectors, matrices, monomials and polynomials...which can be used easily in the construction of more elaborated algorithms. We pay a special attention to genericity, or more precisely to parameterized types (templates in C++ parlance) which allow us to tune easily the data structure to each specific problem at hand. The approach taken here is inspired by a recent paradigm of software developments called active library\(^5,6,7\) and illustrated by the library STL\(^8\). The software components of such libraries take active parts of the configuration and generation of codes (at compile time), which combine parameterization and efficiency. Projects like BLITZ, SCITL, MTL, NTL successfully use those ideas.

We also carefully consider the problem of reusability of external or third-party libraries. Currently, we have connected LAPACK (Fortran library for numerical linear algebra), UMFPACK, SPARSELIB (respectively Fortran and C++ libraries for sparse matrices), SUPERLU (C library for solving sparse linear system), GMP\(^9\)(C library for extended arithmetics), GB\(^4\), RS\(^6\)...Specialized implementations — for instance LAPACK routines — can coexist with generic ones, the choice being determined by the internal representation.

2 Tools for an Active Library Design

The fundamental characteristic of an active library is to provide general purpose abstractions along with domain-specific specializations, tuned to take advantage of configuration knowledge necessary to deliver optimal efficiency. The configuration knowledge of an active library must be bundled in a uniform and extensible framework.

2.1 Types Are Our Friends

SYNAPS extensively uses the typefull programming paradigm\(^9\), whereby the motto is: laws should be enforceable. By that we mean, whenever some con-

---

\(^{9}\)http://www.swx.se/gmp/
straints are known to hold, we use the type-system to express them\(^{\dagger}\) so that much errors can be detected very early in the development stage (i.e. compile-time). In doing so, the program translator (i.e. the compiler) can keep us from breaking hard-to-find constraints violations and save us considerable time. Another immediate benefit is that, when a program construct fails to type-check, then the reasons of the failure are apparent. Last but not least, as compilers are improving, information gathered from static analysis of a program text may be profitably used by the compiler to emit efficient executable images (see e.g. \(^{10}\)). Template expressions (see §2.4) illustrate typical uses of the type-system to implement (in library) domain-specific optimizations not available in common-place compilers.

One feature of symbolic and numeric computations is the ubiquity of parameterized data-types and algorithms. From a symbolic perspective, the notion of polynomial may be defined for any coefficient data type that happens to meet the ring axioms. Therefore, symbolic computation naturally calls for polymorphism\(^{11}\) as an effective programming tool. The C++ programming language\(^{12}\) supports polymorphic data type and algorithms through class inheritance, virtual functions, template classes and template functions. Both programming paradigms are not exclusive even though the stress on parametric polymorphism is rather recent, compared to the conventional use of inclusion polymorphism, which through the use of virtual functions and dynamic binding yields a very flexible mechanism to plug new components in an existing environment. However, inclusion polymorphism may become cumbersome or even conflicts with situations where value semantics, efficiency and static-type constraints are premium.

The template mechanism is the way the C++ programming language implements parametric polymorphism. By definition, it is static-checking oriented; the program-translator uses static type information to generate program components which, in turn, are integrated into the program being translated. That means templates are the feature of choice where polymorphism, value semantics, execution efficiency and static constraints are discriminating criterions. Because templates operate on two levels, they embody a form of meta-language which has proven to be handy when it comes to use the type-system to express constraints, perform compile-time computations, or implement features (such as lazy evaluation) not directly supported by the language.

\(^{\dagger}\)Obviously, not all constraints are expressible in the type-system.
2.2 How Do We Find Our Algorithms?

A distinctive feature of the C++ programming language is the ability to name *scopes* which yields a powerful way of associating a declaration with the name of a function or variable. Understanding how this process, called *name lookup* works, what rules govern it, is crucial for large libraries and programs organization. There are three categories of name lookup rules: qualified name lookup, unqualified name lookup, and function argument dependent name lookup also known as Koenig lookup.

Qualified names are used when a particular scope (a class-scope or namespace-scope) is specified to designate where the program-compiler should look for a name. When the qualified name has the form \( N::m \), where \( N \) designates a namespace, then the set of declarations of \( m \) comprises all declarations for \( m \) in \( N \) and in the transitive closure of all namespaces nominated by using-directives in \( N \) and its used namespaces. Names brought from foreign namespaces through using-declarations are considered during qualified name lookup.

The rules for unqualified name lookup are rather simple. In general, a name shall have a declaration “in scope” before use (an exception to that general rule is that covered by the Koenig lookup to be discussed in the next paragraph). For that matter, declarations from namespaces nominated by using-directives are visible in the namespace enclosing the using-directives (e.g. using **DOMAIN**). For the name lookup machinery, names declared in namespaces nominated by using-directives are considered members of the enclosing namespace. That mechanism is known as *namespace composition*.

The third group of lookup rules, called Koenig lookup, works as follows. When looking up an unqualified name as a function-name in a function call, namespaces associated with the type of the arguments are also examined; in those namespaces the function is looked up as a qualified name where the qualifier is the associated namespace. Koenig lookup is applicable not only to operator names but also to any function-name used in an unqualified manner. Concretely, that means that given the following:

```cpp
namespace Our {
    template<typename T> class Matrix { /* ... */ };
    template<typename T> T frobenius_norm(const Matrix<T>& m) { /* ... */ }
}
int main() {
    Our::Matrix<double> m; double n = frobenius_norm(m); // ... 
}
```

the function `Our::frobenius_norm` is found by Koenig lookup which considers the namespace `Our` associated with the type of the argument
m. This may require the use of additional using-declarations such as using std::function; to make sure that the generic function (e.g., frobeniusNorm) will also apply to built-in types.

2.3 Classes versus Namespaces

There are two kinds of “nameable” scope: class-scoped and namespace-scoped. These scopes, while sharing some properties, work in fundamentally different ways. Whereas namespaces are “open”, classes are “closed”: A namespace can be “re-open” for the purpose of adding new members but classes cannot, once the right-brace terminating their definitions have been seen. Another distinctive difference is the fact that namespaces cannot be used as template-arguments whereas classes can. Thus, the decision of using a class or a namespace to implement a particular instance of scope is usually guided by the intended use.

The first approach based on class-scoped is used for instance in\textsuperscript{13,14}. The algorithms are enclosed in so-called kernels. A kernel is a class, containing other sub-classes representing data or function-objects or methods operating on these types. The extension of a given kernel is performed by the derivation of classes, but with some constraints.

In SYNAPS, the second approach based on namespace has been chosen, for its easy extension capability. Moreover, using Koenig lookup, this allows us to combine generivity and specialization, exploiting natural conventions of developments.

2.4 Template expression

Template expressions are type manipulations used to guide the compiler to produce optimized code. This technique is particularly interesting for some linear algebra operations, where intensive but independent instructions are performed. We illustrate it on the addition of vectors. Consider for instance the following instruction: \( v = v_1 + v_2 + v_3 \); where \( v, v_1, v_2, v_3 \) are vectors (say of type \texttt{Vector}). A traditional implementation would define the operator \texttt{Vector operator+\texttt{(}const Vector & v1, Vector & v2\texttt{)}}; so that the previous instruction will produce two temporary vectors which is time and memory consuming, especially if the vectors are of large size. This can be avoided by using template expression techniques, which we implement as:
\[ \texttt{val<Op'+,Vector,Vector> \ operator+\texttt{(}const Vector & v1, Vector & v2\texttt{)}}; \]
where \texttt{val<Op'+,Vector,Vector> \ } is the type of a data-structure which contains references to the two vectors \( v_1, v_2 \), but which does not

\texttt{http://www-sop.inria.fr/prisme/CGAi/index.html}
compute the sum of these two vectors. The computation is effectively performed in the assignement operator, which does not involve temporary objects. A complete set of arithmetic unevaluated operations is available in the SYNAPS kernel, namely types of the form $0P<c,T1,T2>$ with $c \in \{\,'+',\,'-',\,'\times',\,'/',\,'\%','\,'\}$. Such arithmetic operations will return an unevaluated arithmetic tree and the computation will be effectively performed only when the assignment operator will be used.

3 The Design

We have distinguished three levels of abstractions: containers attached to domains, modules and views, that we describe hereafter.

3.1 Containers and domains

A container specifies the internal representation of a SYNAPS object. It provides abstractions to access, scan, create, or transform that concrete representation and thus depends closely on the data-type being modeled. Containers are implemented as classes (often parameterized by coefficient types or index types), with few functions so that they can be easily rewritten or extended. These structures are adapted to the problem they aim to solve. The algorithm will be provided by other classes or namespaces.

Our approach is in the spirit of the STL library, which implements classical data structures. SYNAPS add its own containers like one-dimensional arrays $\text{rep1d}<C>$ with generic coefficients, two-dimensional arrays $\text{rep2d}<C>$ for dense matrices ...Specialized implementations are usually found in domains (concretely implemented by namespaces). For instance, the container $\text{rep1d}<C>$ is in the namespace linalg and the usual way to call it, is $\text{linalg::rep1d}<C>$. Similarly, we have defined 2-dimensional containers $\text{linalg::rep2d}<C>$ and $\text{lapack::rep2d}<C>$, for dense matrices. The last container is adapted to the internal representation of dense matrices in the LAPACK library. Adapted containers for the manipulations of structured matrices such as Toeplitz matrices ($\text{linalg::toeplitz}<C>$), Hankel matrices ($\text{linalg::hankel}<C>$) or sparse matrices ($\text{linalg::sparse2d}<C>$) are also available in the library. The job of iterating over data-structures is delegated to iterators.

Because namespaces are open, using this domain mechanism allows us to add easily specialized functions, and to transparently use them in elaborated algorithms. As we will see, using the Koenig lookup mechanism described in 2.2, these specialized functions are searched in the domain of the container, if they exist. Another interesting feature of this approach, is that functions
and classes provided in a domain, can be compiled separately.

3.2 Modules

A module is a collection of generic implementations which apply to a family of objects sharing common properties (such as vectors, matrices, univariate polynomials, ...) implemented as *namespaces* and thus can be extended easily. For instance in the module (or namespace) VECTOR, we have gathered generic implementations for vectors, which are independent of the internal representation, such as Print, Norm, ... Another interesting feature of the approach, is that namespaces can be combined or extend naturally: *namespace UPOLY { using namespace VECTOR; ... }*. This yields a great flexibility in the construction of the set of functions which applies for a families of objects. The main modules of the library are VECTOR, MATRIX, UPOLY (univariate polynomials), MPOLY (multivariate polynomials).

3.3 Views

The views specify how to interpret containers as mathematical objects and provide operations on the objects which are independent of the container. They are implemented as classes parameterized by the container type and sometimes by *trait* classes which precise the implementation. The only data available in such a class, via the member function *rep()*, is of container type.

The implementations of a view are searched in a module as it is illustrated below:

```cpp
template <class C, class R>
UPolDse<C,R> operator*(const UPolDse<C,R>& v1, const UPolDse<C,R>& v2)
{
    UPolDse<C,R> w(Degree(v1)+Degree(v2)+1,AsSize());
    using namespace UPOLY; mul(w.rep(),v1.rep(),v2.rep());
    return w;
}
```

An eventual declaration of *mul* in UPOLY is added to the possible overload set constructed by Koenig lookup. UPOLY::mul is the naive implementation that works on all random-access containers. In the case the container Domain::R is specialized for an external library providing its notion of multiplication, the forwarding *mul* function can be declared as:

```cpp
namespace Domain { void mul(R & s, const R & r1, const R & r2); }
```

This is the scheme that has been used to connect the external libraries LAPACK, GMP, SUPERLU, ...

Here are the main examples of view classes currently implemented in the library, R stands for the container type: *VectDse<C,R>* (standard vectors), *MatrDse<C,R>* (dense matrices), *MatrStr<C,R>* (structured matrices),

*paperproc: submitted to World Scientific on May 3, 2002*
3.4 Large Objects Management: Copy On Write

Because objects management in the C++ programming language defaults to value semantics, we have to take explicit actions to avoid copying large objects — the time spent in copying large objects may be shown to be non negligible under usual circumstances. A popular way to solve this problem is to use reference counting coupled with a copy-on-write strategy. This scheme is implemented by the class

\texttt{shared\_object\langle R\rangle}. Therefore, returning such objects (of type \texttt{VectDse}, \texttt{MatrDse}, \texttt{MatrStr}, \texttt{MatrSps}, \texttt{UPolDse} or \texttt{MPol}, ...) by value in a function does not induce a performance penalty. Indeed a copy and destruction of these objects will in this case just increment or decrement a counter.

4 Available packages in the current distribution

We describe briefly the functions, algorithms, libraries which are available in the current distribution\footnote{accessible via \texttt{cvs -d:server:cvs@op.sop.inria.fr/CVS/galaad co synaps} or \texttt{wget http://www-sop.inria.fr/galaad/logiciels/synaps.tgz}}.

\textit{Input/output}: Streams for printing objects in maple and latex format are available. An output stream for \texttt{vrml} format is under construction as well as a connection to \texttt{povray}.

\textit{Arithmetic}: The extended arithmetic classes are based on the \texttt{GMP} and \texttt{MPFR}\footnote{http://www.mpfr.org/} libraries, and correspond to the predefined types \texttt{ZZ}, \texttt{QQ}, \texttt{RR}, \texttt{RRR}, \texttt{CC}. A simple implementation of modular numbers \texttt{Z<p>} is also available.

\textit{Linear algebra}: Basic linear algebra operations such as \texttt{LU}, \texttt{QR}, Bareiss decomposition, determinants, svd, rank, eigenvalues and eigenvectors computations, \texttt{FFT} are available, either using the \texttt{LAPACK} routines for double number type or generic ones. In particular, the implementation of structured \texttt{LU} decomposition of sparse matrices\footnote{implemented in \texttt{UMFPACK} and \texttt{SuperLU} by J. Demmel, J.R. Gilbert and X.S. Li, in C for \texttt{float} and \texttt{double} number types has been extended to generic coefficients.

\textit{Resultants}: A package devoted to resultant constructions including the Sylvester and Bezout matrices for univariate polynomials, the Macaulay, Bezoutian, Toric (integrating the C-library developed by I. Emiris) formulations for multivariate polynomials\footnote{\texttt{MatSparse\langle R\rangle} (sparse matrices), \texttt{UPolDse\langle C,R\rangle} (dense univariate polynomials), \texttt{UPolQuot\langle R\rangle} (quotiented univariate polynomials), \texttt{Monom\langle C,E\rangle} (monomials with coefficients in C and exponents in E), \texttt{MPol\langle C,0,R\rangle} (multivariate polynomials where 0 defines the ordering on the monomials).}.

\texttt{paperproc: submitted to World Scientific on May 3, 2002}
Solvers: Regarding solvers for univariate polynomials, we implement subdivision methods based on Bézier representation and Descartes rule, or on exclusion functions, iterative methods such as Sebastio a Sylva method. The solver mp
colve developed by D. Bini and G. Fiorentino is also connected to the environment. For multivariate polynomials, resultant based solvers, hiding a variable are also available. The algorithm for solving zero-dimensional ideals, which computes first a normal form and then the eigenvalues or eigenvectors of the induced matrices of multiplications is implemented and illustrated in the next section.

Geometry: A package on algebraic curves and surfaces is also under constructions, including tools for intersecting them, computing their topology or drawing them.

5 Distributed computation
Our objective is not only to be able to integrate external libraries in a transparent way, but also to use SYNAPS from outside tools such as computer algebra systems like MAPLE, MATHEMATICA, .... Indeed, SYNAPS is an environment which allow to build efficient and dedicated tools, that can be used for distributed computation through the Inter Process Communication tool, called UDX.

It is a protocol for data exchange in binary mode based on a new patent. It is fully customizable, since it can be used with so many supports as one wants: files, sockets, shared memory. Its implementation does not depend on the machine nor on the operating system considered (e.g. no information about the indianness is required). It performs an optimal number of operations (byte permutations), whatever the considered hardware (no operations are needed to exchange data between equivalent hardware), and without any additional information (such as byte ordering) to exchange basic scalar data (machine integers, IEEE floats) between heterogeneous machines. Coupled with a highly optimized buffering system, UDX turns out to be, not only portable, but also several dozen times faster than conventional strategies like XDR.

The communication protocol of SYNAPS uses UDX as a basic tool for exchanging low level data structure and implements a full protocol that takes advantage of the general structure of the library for higher level objects. An UDX non atomic object is described by two components: the descriptor part (independent of the implementation) and the data part. For example, the descriptor for dense vectors which is a flag (DENSE_VECTOR) and the descriptor of the entries (recursive definition) are sent first, followed by the data part constituted by the number of entries, and the sequence of coefficients. The
descriptor part, attached to each kind of objects, is used to facilitate the implementation of the abstract protocol for exchanging efficiently in a binary way, complex mathematical data structures.

6 Illustration
We have already shown in experimentations reported for instance in $^4$, that genericity is compatible with efficiency. We can push further this reasoning: genericity is in fact a matter of performance. Indeed as the code is not attached to any particular implementation of the containers, it is possible to change them to use the most efficient implementation available without any code rewriting. The environment has been used to implement the algorithms described in$^{4,18}$ which compute a normal form for zero-dimensional systems of polynomial equations in order to solve them by eigenvector computation. In this approach, we have two main needs in terms of linear algebra, which are fast sparse LU-decomposition, fast and robust eigenvector computations.

The SYMPS framework allows us to use specialized implementations for these two points, namely superLU$^{19}$ and LAPACK$^{20}$. The results that we obtained when dealing with the Katsura equations ($n$ is the number of variables) on a Pentium III 933 MHz are shown below:

<table>
<thead>
<tr>
<th>n</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF(mod)</td>
<td>0.01s</td>
<td>0.05s</td>
<td>0.17s</td>
<td>0.95s</td>
<td>256.81s</td>
<td>1412s</td>
</tr>
<tr>
<td>NF(double)</td>
<td>0.01s</td>
<td>0.02s</td>
<td>0.13s</td>
<td>0.54s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eg(double)</td>
<td>0.01s</td>
<td>0.02s</td>
<td>0.11s</td>
<td>2.33s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max(</td>
<td>f_i</td>
<td>)</td>
<td>$10^{-13}$</td>
<td>$10^{-10}$</td>
<td>$10^{-8}$</td>
<td>$10^{-8}$</td>
</tr>
</tbody>
</table>

where NF(mod) (resp. NF(double)) is the time spent for the computation of normal form in modular (resp. double floating point) arithmetic, Eg(double) is the time spent for the computation of eigenvectors and Max(|f_i|) is the maximum of the value of the normalized polynomials at the computed roots. During the normal form computations, the superLU library (ported in C++ with template coefficients) is used and then LAPACK is called for the eigenvectors computation.

References

paperproc: submitted to World Scientific on May 3, 2002 11