SYNAPS

Symbolic and Numeric APplicationS

B. Mourrain, Ph. Trébuchet, F. Rouillier
INRIA, BP 93, 06902 Sophia Antipolis

22nd May

What is SYNAPS ?

What is SYNAPS ?

It is a library

What is SYNAPS ?

It is a library

• containing basic *parameterized* data structures : **vectors, matrices (dense, Toeplitz, Hankel, sparse, . . .), univariate polynomials, multivariate polynomials.**

What is SYNAPS ?

It is a library

• containing basic *parameterized* data structures : **vectors, matrices (dense, Toeplitz, Hankel, sparse, . . .), univariate polynomials, multivariate polynomials.**

• integrating several **specialized**, **efficient**, freely available softwares.

SYNAPS

What is SYNAPS ?

It is a library

• containing basic *parameterized* data structures : **vectors, matrices (dense, Toeplitz, Hankel, sparse, . . .), univariate polynomials, multivariate polynomials.**

• integrating several **specialized**, **efficient**, freely available softwares.

LAPACK

SPARSELU

GMP

TORIC

SYNAPS

MPSOLVE

UDX

RS/GB

CGAL

Objectives

Objectives

- Combine **symbolic and numeric** computation.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.
- A kernel to be used for the development of **elaborated algorithms**.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.
- A kernel to be used for the development of **elaborated algorithms**.
- **Easy to use**, from inside and outside.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.
- A kernel to be used for the development of **elaborated algorithms**.
- **Easy to use**, from inside and outside.
- **Reusability** as much as possible of efficient external tools.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.
- A kernel to be used for the development of **elaborated algorithms**.
- **Easy to use**, from inside and outside.
- **Reusability** as much as possible of efficient external tools.
- Provide a **coherent platform** of specialized softwares, connected and configured together.

Objectives

- Combine **symbolic and numeric** computation.
- A kernel implementing **basic data structures**.
- A kernel to be used for the development of **elaborated algorithms**.
- **Easy to use**, from inside and outside.
- **Reusability** as much as possible of efficient external tools.
- Provide a **coherent platform** of specialized softwares, connected and configured together.
- Incorporate transparently **specialisations**, without penalty.

Tools for an Active Library Design

Tools for an Active Library Design

□ Ubiquity of **parameterised type** in algebra (coefficients, internal representation, . . .)

Tools for an Active Library Design

□ Ubiquity of **parameterised type** in algebra (coefficients, internal representation, . . .)

□ **Generic algorithms** which apply for a large class of data-structures.

Tools for an Active Library Design

- Ubiquity of **parameterised type** in algebra (coefficients, internal representation, . . .)
- **Generic algorithms** which apply for a large class of data-structures.
- Choose the **right implementation for the right job**.
- Combine **generic** implementations with **specialised** functions.

The C++ technology

The C++ technology

Types:

- ☐ express constraints;
- ☐ help to detect errors at compile-time;
- ☐ help to optimise code in the static analysis step;

The C++ technology

Types:

- □ express constraints;
- □ help to detect errors at compile-time;
- □ help to optimise code in the static analysis step;

Class: Entity collecting data-structures and functions in a new type.

The C++ technology

Types:

- ☐ express constraints;
- ☐ help to detect errors at compile-time;
- ☐ help to optimise code in the static analysis step;

Class: Entity collecting data-structures and functions in a new type.

Namespace: Naming extension, for data-structures and functions.

The C++ technology

Types:

- ☐ express constraints;
- ☐ help to detect errors at compile-time;
- ☐ help to optimise code in the static analysis step;

Class: Entity collecting data-structures and functions in a new type.

Namespace: Naming extension, for data-structures and functions.

Derivation:

- ☐ Allow to extend class definitions: **class B : class A {...}**
- ☐ The functions and data-structures of `B` contains those of `A` (except the constructors).

The C++ technology

Types:

☐ express constraints;

☐ help to detect errors at compile-time;

☐ help to optimise code in the static analysis step;

Class: Entity collecting data-structures and functions in a new type.

Namespace: Naming extension, for data-structures and functions.

Derivation:

☐ Allow to extend class definitions: **class B : class A {...}**

☐ The functions and data-structures of `B` contains those of `A` (except the constructors).

Parameterized type:

```
template <class X> X F(const X &) { ... }
```

- Description of how a code depends on type parameters.
- Allow static-checking.

Parameterized type:

```
template <class X> X F(const X &) { ... }
```

- Description of how a code depends on type parameters.
- Allow static-checking.

Virtual functions: Mechanism of dynamic binding, at run time, through function pointers.

Parameterized type:

```
template <class X> X F(const X &) { ... }
```

- ☐ Description of how a code depends on type parameters.
- ☐ Allow static-checking.

Virtual functions: Mechanism of dynamic binding, at run time, through function pointers.

Genericity

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

- *explicitly*: **struct A { void f(); }; A a; a.f(); a.A::f(); D::g(a);**

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

- *explicitly*: **struct A { void f(); }; A a; a.f(); a.A::f(); D::g(a);**

- *by polymorphism*, that is by searching in a given scope, for a function (or operator) with a given name, whose signature matches: **20/3; 20./3;**

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

• *explicitly*: **struct A { void f(); }; A a; a.f(); a.A::f(); D::g(a);**

• *by polymorphism*, that is by searching in a given scope, for a function (or operator) with a given name, whose signature matches: **20/3; 20./3;**

• *by derivation*, that is by coercing a type into a subtype, in order to match a given signature: **struct B : public A {}; B b; b.f();**

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

- *explicitly*: **struct A { void f(); }; A a; a.f(); a.A::f(); D::g(a);**

- *by polymorphism*, that is by searching in a given scope, for a function (or operator) with a given name, whose signature matches: **20/3; 20./3;**

- *by derivation*, that is by coercing a type into a subtype, in order to match a given signature: **struct B : public A {}; B b; b.f();**

- *by specialisation*, that is by searching a **template** function or class that can be specialized to match a given signature: **F(2); F(2.);**

Genericity

The way to write a code so that the missing definitions (types, functions) are filled automatically ?

How do we find the missing definitions ?

- ***explicitly***: **struct A { void f(); }; A a; a.f(); a.A::f(); D::g(a);**

- ***by polymorphism***, that is by searching in a given scope, for a function (or operator) with a given name, whose signature matches: **20/3; 20./3;**

- ***by derivation***, that is by coercing a type into a subtype, in order to match a given signature: **struct B : public A {}; B b; b.f();**

- ***by specialisation***, that is by searching a **template** function or class that can be specialized to match a given signature: **F(2); F(2.);**

- ***by Koenig lookup***, that is by searching in the ***namespace*** associated with the types of the signature, the function that matches.

```
namespace Domain {  struct A {}; void f(const A & a) {}; }
Domain::A a; f(a);
```

How can we specify a collection of algorithms

How can we specify a collection of algorithms

By class scope

```
struct View
{
    struct atype;
    void f(const atype & r) {...}
}
View::atype a; View W; W.f(a);
```

- ☐ Extension by derivation, but the name change;
- ☐ It is closed;
- ☐ Can be used as parameter (in template classes);
- ☐ Allows parameters `View(p)`;

By namespace scope

```
namespace Domain
{
    struct atype;
    void f(const atype & r) {...}
}
Domain::atype a; Domain::f(a);
```

- ☐ Extension by `using` directives;
- ☐ It is open;
- ☐ Cannot be used as parameter.
- ☐ Do not allow parameters.
- ☐ Koening lookup allow to select the functions of the domain:

`f(a)` is equivalent to `Domain::f(a)` because `a` is of type `Domain::atype`.

The Design

Three levels of objects:

The Design

Three levels of objects:

□ **Container and domains:**

Internal representation, associated with iterators, and access/modification methods.

□ **View:** ***How we see the container***.

eg. as a `Vector<R>` or as a univariate polynomial `UPolDse<R>`.

Allow local views sharing datas.

□ **Module:**. ***Set of (generic) functions which apply to a category of objects.***

eg. `VECTOR::Print`, `MATRIX::mult`.

Allow easy specialisation.

Containers and domains

The containers

- They specify the internal representation,

- They provide methods or functions for accessing, scanning, creating, transforming this representation. Exemple from the STL library:

`list<R>, vector<R>, set<R>, deque<R>, ...`
`rep1d<C>` (one-dimensional arrays with generic coefficients)
`rep2d<C>` (two-dimensional arrays for dense matrices)

How to use iterators to scan such structures:

```
for(R::iterator it=r.begin(); it !=r.end(); ++it) *it = ...
```

The domains

- They are `namespaces` to which, are attached the **containers**:

```
linalg::rep1d<C>    linalg::toeplitz<C>    linalg::sparse2d<C>
linalg::rep2d<C>    linalg::hankel<C>
```

- Specialized algorithms for a container are defined in the corresponding domain:

```
namespace lapack {
  template<class C>
  void solve(LU,linalg::rep1d<C> & x,
                 const rep2d<C> & A, const linalg::rep1d<C> & b);
}
```

- Specialisation or extension can be achieved either like:

```
namespace lapack {
  template<class C>  void my_new_function(const rep2d<C> & r) {}
}
```

or by derivation and redefinition of functions.

Modules

It is a collection of implementations which apply to a family of objects sharing common properties.

```
namespace VECTOR {
   template<class V>
   ostream & Print(ostream & os, const V& v)
   {
     typename V::const_iterator it =v.begin();
     os<<"["<<*it; ++it;
     for( ; it !=v.end(); ++it) os<<","<< *it;
     os<<"]";
     return os;
   }
}
```

- No constraints on the parameter type `V`, except a `const_iterator`.

- They can be combined or extended naturally:

```
namespace UPOLY { using namespace VECTOR; ... }
```

- The main modules of the library are

```
VECTOR, MATRIX, UPOLY, MPOLY.
```

Views

They specify how to manipulate or to see the containers as mathematical objects.

• The internal data is available, via the method `rep()`.

• They are usualy classes, parameterised by the container type and sometimes by ***trait*** classes which precise the implementation.

```
VectDse<double,linalg::rep1d<double> > <=>   VectDse<double>
```

If we want to see it as a univariate polynomial:

```
UPolDse<double,linalg::rep1d<double> >
```

- The implementations of these views are based on modules:

```
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;
}
```

- If defined for the container `D::R`, the following specialized function is used:

```
namespace D {  void mul(R & r, const R & p1, const R & p2);  }
```

- Views on subobjects:

```
VectDse<double> V(5,"1 2 3 4"), W1(3,"0 1 0"), W2(3,"1 0 0");
V[Range(1,3)]=W1+W2;
```

Other features

Reference counting

- Implemented for the views.

- Principle: a counter attached to the data, counting the number of objects pointing to the data.

- Usefull when the copy by value is invoqued:

```
View<R> f(const View<R> & V) {View<R> W(V); ...; return W;}
```

Other features

Reference counting

• Implemented for the views.

• Principle: a counter attached to the data, counting the number of objects pointing to the data.

• Usefull when the copy by value is invoqued:

```
View<R> f(const View<R> & V) {View<R> W(V); ...; return W;}
```

Template expressions

• They are type manipulations used to guide the compiler to produce optimised code.

```
v = v1 + v2 + v3;
```

- A usual implementation, involving 2 temporary vectors:

```
Vector operator+(const Vector & v1, Vector & v2);
```

and the use of the assignement operator:

```
Vector & Vector::operator=(const Vector & v);
```

- With **template** expression:

```
VAL<Op<'+',Vector,Vector> >
        operator+(const Vector & v1, Vector & v2);
```

In our case, it builds an object of type

```
VAL<Op<'+', Vector, VAL<Op<'+', Vector, Vector> > > >;
```

The expansion of the code at compile time yields

```
for(index_type i = 0; i <v.size(); i++) v[i] = v1[i] + v2[i] + v
```

- Complete set of arithmetic unevaluated operations: **OP⟨c,T1,T2⟩ with c in {'+','-','*','.','/','%','^' }**

Historic

• **1996** : A version based on abstract classes, derivation, and virtual functions (Internship of F. Livigni).

Slow and heavy.

Historic

• **1996** : A version based on abstract classes, derivation, and virtual functions (Internship of F. Livigni).

Slow and heavy.

• **1999** : Appear to be a deliverable of the europeen project Frisco (LTR 21.024)

Classes based on containers, parameterisation of the container type, the coefficient type.

The g++ compiler was improving on template classes.

Historic

• **1996** : A version based on abstract classes, derivation, and virtual functions (Internship of F. Livigni).

Slow and heavy.

• **1999** : Appear to be a deliverable of the europeen project Frisco (LTR 21.024)

Classes based on containers, parameterisation of the container type, the coefficient type.

The g++ compiler was improving on template classes.

• **2000** : Development of resultant and solver modules.

Application in robotic, Signal processing, and biology problems (work of H. Prieto).

Historic

• **1996** : A version based on abstract classes, derivation, and virtual functions (Internship of F. Livigni).

Slow and heavy.

• **1999** : Appear to be a deliverable of the europeen project Frisco (LTR 21.024)

Classes based on containers, parameterisation of the container type, the coefficient type.

The g++ compiler was improving on template classes.

• **2000** : Development of resultant and solver modules.

Application in robotic, Signal processing, and biology problems (work of H. Prieto).

- **2001** : Evolution as a platform. Namespace and Koenig lookup.

  Collaborative work with SPACES, PRISME.

People involved

D. Amar, P. Aubert, D. Bini, D. Bondyfalat,
L. Carrot, G. Dos Reis, I. Emiris, G. Fiorentino,
G. Gatellier, H. Hirukawa, P. Mario, B. Mourrain, P. Palackel,
H. Prieto, F. Rouillier, O. Ruatta, M. Stillman,
M. Teillaud, N. Thiery, Ph. Trébuchet

People involved

D. Amar, P. Aubert, D. Bini, D. Bondyfalat,
L. Carrot, G. Dos Reis, I. Emiris, G. Fiorentino,
G. Gatellier, H. Hirukawa, P. Mario, B. Mourrain, P. Palackel,
H. Prieto, F. Rouillier, O. Ruatta, M. Stillman,
M. Teillaud, N. Thiery, Ph. Trébuchet

- ☐ discussions,
- ☐ specific contributions,
- ☐ extensions,
- ☐ tests, validation of the environment.

Distribution

- under LGPL
- `cvs@cvs-sop.inria.fr`.
- `http://www-sop.inria.fr/galaad/logiciels/synaps/`,
- Documentation in pdf; automatic processing of the source code and extraction of the documentations.

Experimental evidences

Aim: recovering the roots of a given multivariate polynomial system.

example	Maple float	MacRev float	Maple over $\mathbb{Q}$	MacRev over $\mathbb{Q}$
katsura 6	$4000s$	$0.22s$	$8600s$	$6s$
kruppa	$1000s$	$0.05s$	∞	$72.04s$
signal	$4000s$	$0.20s$	∞	$736.46s$

No Hope to perform usefull computation only using *standards* Computer Algebra systems

Example combining specialized tools

Need for:

Example combining specialized tools

Need for:

- polynomials
  - MPol< > with different fields: Zp<32051>, double, Scl<MPQ>, Scl<MPF>...
  - different monomials representations: Monom<C,dynamicexp<'x'> > >, Monom<C,numexp<'x'> > >, Monom<C,tinyexp<'x'> > >...

Example combining specialized tools

Need for:

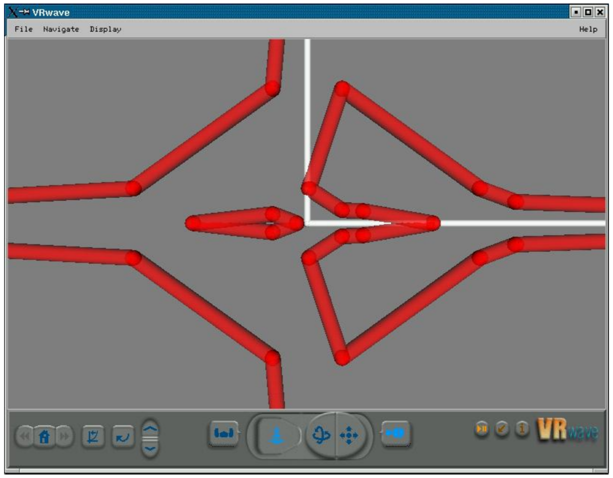
- polynomials
  - MPol< > with different fields: Zp<32051>, double, Scl<MPQ>, Scl<MPF>...
  - different monomials representations: Monom<C,dynamicexp<'x'> > >, Monom<C,numexp<'x'> > >, Monom<C,tinyexp<'x'> > >...

- linear algebra
  - sparse LU decomposition with generic coefficients: MatrSps<CoefFicient_type> and LUdecomp(Matrix, L, U ,perm_r perm_c)
  - numerical linear algebra: MatrDse<double,lapack::rep2d<double> + Eigenvectors, Eigenvalues etc...

Example of external communication

```
#include "linalg.H"
#include "geometry.H"
#include "inout/vrmlstream.H"
#include "mpoly.H"
int main()
{
   //...
   Point<double> O(3,t), E1(3,t+3), E2(3,t+6);
   print(cvrml, Cylinder<double>(O,E1,0.1), txt);
   print(cvrml, Cylinder<double>(O,E2,0.1), txt);
   print(cvrml, Sphere<double>(O,0,1), txt);

   MPol<double> P; cin >> P; cvrml << Draw(P,SPL())<<endl;
}
```



Distributed computation based on UDX

- Binary protocol.
- Data exchange through sockets, files, shared memory.
- Interface for basic objects (native arithmetic data types, extended arithmetic based on GMP or PARI).
- Highter level interface in SYNAPS, based on abstract description and the binary protocol.

Distributed computation based on UDX

- Binary protocol.
- Data exchange through sockets, files, shared memory.
- Interface for basic objects (native arithmetic data types, extended arithmetic based on GMP or PARI).
- Highter level interface in SYNAPS, based on abstract description and the binary protocol.

```
udxstream udx("arz.inria.fr","polynomialsolver");
UPolDse<ZZ> P("x0^320-23453412351254135*x0^123+1;");
udx << P<<endl;
Seq<RR> s; udx >>s;
```

Conclusion

- Interpreter, interface to other systems.
- Compilator issues (windows, . . .).
- Adaptation to the specifications of other libraries.
- Integration of more tools.
- Manage extensions and evolution.