C++ TOOLS TO CONSTRUCT OUR USER-LEVEL LANGUAGE

FRÉDÉRIC HECHT

Abstract. The aim of this paper is to present how to make a dedicated computed language polymorphic and multi type, in C++ to solve partial differential equations with the finite element method. The driving idea is to make the language as close as possible to the mathematical notation.

Mathematics Subject Classification. 68N15, 78M10, 80M10.

Received: January 10, 2001. Revised: May 23, 2002.

1. Introduction

A partial differential equation is a relation between a function of several variables and its (partial) derivatives. Many problems in physics, engineering, mathematics and even banking are model by one or several partial differential equations.

Many phenomena involve several different fields. Fluid-structure interactions, Lorentz forces in liquid aluminum and ocean-atmosphere problems are three such systems. These require approximations of different degrees, possibly on different meshes. Some algorithms such as Schwarz’ domain decomposition method require also the interpolation of data on different meshes within one program. Thus the software freefem++ [6] can handle these difficulties: arbitrary finite element spaces on arbitrary unstructured and adaptive meshes (see [2,9] for the finite elements method).

We present the kernel of the freefem++ user-level language. First, this language as to be able to use functions. In Section 2, we explain how to construct an algebra of functions, in Section 2.2 we generalize this construction to $C^\infty$ functions, and in Section 3, we explain how to modelised elliptic PDE’s in variational form, and how to compute integrals.

To construct our user-level language, we need a virtual machine to execute the instructions. Wirth’s book [11] explains clearly the way. The way to build the virtual machine is based on the the following remark: the algebra of functions is a simple version of a virtual machine with just $+, -, \times, /, ^$ operator, where the “run” action is the evaluation of the function. To construct a virtual machine, we just add to the algebra of functions, some operators and functions like:

- sequencing operator
- affectation operator
- while while loop construction
- getvalue get the value of local variable
- call call a virtual machine function

Keywords and phrases. Finite element method, grammars, languages.

1 Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, Paris, France. e-mail: hecht@ann.jussieu.fr

© EDP Sciences, SMAI 2002
The virtual machine is an extended algebra of "function instructions", with lot of operators and functions. All the local variables of a procedure are store in a stack memory. We want a reentrant language so the memory stack must be a parameter of the "function instructions" and this stack must be allocated when we evaluated the operator "call" a function.

We describe how to change the algebra, to be able to handle many data types, with or without dynamic type checking. We explain how to use the STL to make symbol tables and how to handle easily the polymorphism in the language. We finish with the description of how add operators to the user language, and how to use simply yacc or bison processor.

Finally, we give two example from the freefem++ language, one to solve the Navier-Stokes equation (Chap. 5 in [10]) by the Taylod-Hood approximation, and another for the Schwarz’ domain decomposition method [8].

2. An Algebra of Functions

In this section we show how to construct an algebra of function in C++. We start, with functions from \( \mathbb{R} \) to \( \mathbb{R} \), a mathematical function from \( \mathbb{R} \) to \( \mathbb{R} \), is just an object with a name, \( f \) for example, with the evaluation operator \( f(x) \) for any \( x \in \mathbb{R} \).

2.1. Basic version

First, we just make an algebra of functions with the classical mathematical operators \( +, -, *, /, ^ \). The model function is just a pure virtual class \( C_{virt} \) with only the evaluation operator: \( R \ operator() (R) \ const; \).

```cpp
typedef double R; // definition of the field R
class Cvirt { public: virtual R operator() (R ) const =0;};
```

Now we use inheritance, to create the classical functions (C++ function, constant function, combination of two functions).

```cpp
class Cfonc : public Cvirt { public:
  R (*f)(R);
  R operator() (R x) const { return (*f)(x);}
  Cfonc(R (*ff)(R)) : f(ff) { } ;
};
class Cconst : public Cvirt { public:
  R a;
  R operator() (R ) const { return a;}
  Cconst(R aa) : a(aa) { } ;
};
class Coper : public Cvirt { public:
  const Cvirt *g, *d;
  R (*op)(R,R);
  R operator() (R x) const { return (*op)((*g)(x),(*d)(x));}
  Coper(R (*opp)(R,R), const Cvirt *gg, const Cvirt *dd) : op(opp),g(gg),d(dd){}
};
```

We define the five classical operators.

```cpp
// the 5 binary operators
```

\(^1\)ftp://ftp.inria.fr/INRIA/Projects/Gamma/hecht/cpp/lg/fonctionsimple.cpp
Now we encapsulate all this in a class `Function` because it is not possible to overload operators on pointers. The class contains just a pointer to a `Cvirt`, plus some constructors of all the different functions (from constant, a pointer to a C++ function, or from a `Cvirt` pointer,...), and the five basic binary operators.

```cpp
class Function { public:
  const Cvirt *f;
  R operator() (const R x) { return (*f)(x); }
  Function(const Cvirt *ff) : f(ff) {}
  Function(R (*ff)(R)) : f(new Cfonc(ff)) {}
  Function(R a) : f(new Cconst(a)) {}
  operator const Cvirt * () const { return f; }
  Function operator+(const Function & dd) const { return new Coper(Add,f,dd.f); }
  Function operator-(const Function & dd) const { return new Coper(Sub,f,dd.f); }
  Function operator*(const Function & dd) const { return new Coper(Mul,f,dd.f); }
  Function operator^ (const Function & dd) const { return new Coper(Pow,f,dd.f); }
};
```

Here is an example on how to use this class.

```cpp
inline R Identite(R x){ return x; }
int main()
{
  Function Cos(cos),Sin(sin),x(Identite);
  Function f((Cos^2)+Sin*Sin+(x^4)); // warning ^ operator 
  // has the wrong precedence
  cout << f(2) << endl;
  cout << (Cos^2)+Sin*Sin+(x^4))(1) << endl;
  return 0;
}
```

In this example, the function \( f = \cos^2 + (\sin \cdot \sin + x^4) \) will be defined by a tree of objects. We can represent it with:

```plaintext
f.f= Coper (Add,t1,t2); t1=(cos^2) + t2=(sin*sin + x^4)
t1.f= Coper (Pow,t3,t4); t3=(cos) ^ t4=(2)
t2.f= Coper (Add,t5,t6); t5=(sin*sin) + t6=x^4
t3.f= Ffonc (cos);
t4.f= Fconst (2.0);
t5.f= Coper (Mul,t7,t8); t7=(sin) * t8=(sin)
```
2.2. The $C^\infty$ function

Now in this version\(^2\), we want to differentiate the function recursively, this tool is also use in FreeFem+ software [1]. We focus on the problem of infinite recursively, if the function constructor build also the derivative, recursively all the derivative of the function will be effectively constructed. The n-derivative of function $x \to 1/x$ is $(-1)^n(n!)(x^{-n-1})$, when we construct the function $1/x$, all the derivative will be also constructed, we get a “infinite program” (no bug at the compilation time, just a stack overflow at execution).

To solve this, we use a construction in two stages. First choose a variable ($md$, initialize to 0) to store the derivative and secondly use the virtual method virtual CVirt *de () {return zero;} construct the the derivative function. Now, the method CVirt *d (), can construct the derivative function once only when we need it.

So the class is:

```cpp
class CVirt {
friend class Function;
mutable CVirt *md; // pointer to the derivative function
protected:
    virtual CVirt *de () {return zero;} // function to construct f'
static CVirt *zero; // pointer to zero function (global)
public:
    CVirt () : md (0) {} // constructor to be sure md is 0
    virtual CVirt *d () {
        if (md == 0) md = de (); // construct if unset
        return md;} // return the derivative
};
```

We also introduce the functions Function2 from $\mathbb{R}^2$ to $\mathbb{R}$. and we use the composition operator to defined all the derivative of the binaries operators (example: operator $f+g = \text{Add}(f,g)$).

```cpp
class Function { // one variable Function
friend class Function2;
CVirt *p;
public:
    operator CVirt *() const {return p;}
Function (CVirt *pp) : p(pp) {} // Creation from a C Function
Function (R x); // constant Function
Function (const Function& f) : p(f.p) {} // Copy constructor
Function d () {return p->d ();} // derivative
void setd (Function f) {p->md = f;} // set the derivative
R operator() (R x) {return (*p)(x);} // get value at point x
Function operator() (Function); // compose of Functions
```

\(^2\)ftp://ftp.inria.fr/INRIA/Projects/Gamma/hecht/cpp/lg/Function.[ch]pp
Function2 operator() (Function2); // compose of Function Function2
static Function monome (R c, int n); // c x^n

With the same technique, the virtual class for functions from $\mathbb{R}^2$ to $\mathbb{R}$ is:

```cpp
struct CVirt2 {
    CVirt2 (): mdx (0), mdy (0) {}
    virtual R operator () (R, R) = 0;
    virtual CVirt2 *dex () {return zero;}
    virtual CVirt2 *dey () {return zero;}
    CVirt2 *mdx, *mdy;
    CVirt2 *dx () {
        if (mdx == 0) mdx = dex ();
        return mdx;
    }
    CVirt2 *dy () {
        if (mdy == 0) mdy = dey ();
        return mdy;
    }
    static CVirt2 *zero;
};
```

Now the class of functions from $\mathbb{R}^2$ to $\mathbb{R}$ is:

```cpp
class Function2 { // two variable Function
    CVirt2 *p;
public:
    operator CVirt2 () const {return p;} // cast operator to CVirt2
    Function2 (CVirt2 *pp) : p(pp) {}
    Function2 (R (*) (R, R)); // create from Function C
    Function2 (R x); // constante Function
    Function2 (const Function2& f) : p(f.p) {} // Copy constructor
    Function2 dx () {return p->dx ();} // $\partial_x$ derivative
    Function2 dy () {return p->dy ();} // $\partial_y$ derivative
    void setd (Function2 f1, Function2 f2) // set $\partial_y, \partial_x$ derivatives
        {p->mdx = f1; p->mdy = f2;}
    R operator () (R x, R y) {return (*p)(x, y);}
    Function operator () (Function, Function); // compose of function
    Function2 operator () (Function2, Function2); // compose of function
    static Function2 monome (R c, int n1, int n2); // to construct $c x^{n1} y^{n2}$
    void operator += (Function2 a);
    void operator *= (Function2 a);
friend class Function;
};
```

Some useful global functions:

```cpp
extern Function Chs, Identity, Log;
extern Function2 Add, Sub, Mul, Div, Pow, CoordinateX, CoordinateY;
```

All the operators are:

```cpp
inline Function operator+ (Function f, Function g) {return Add(f, g);}
inline Function operator- (Function f, Function g) {return Sub(f, g);}
```
inline Function operator* (Function f, Function g) {return Mul(f, g);} 
inline Function operator/ (Function f, Function g) {return Div(f, g);} 
inline Function operator^ (Function f, Function g) {return Pow(f, g);} 
inline Function operator- (Function f) {return Chs(f);} 
inline Function2 operator+ (Function2 f, Function2 g) {return Add(f, g);} 
inline Function2 operator- (Function2 f, Function2 g) {return Sub(f, g);} 
inline Function2 operator* (Function2 f, Function2 g) {return Mul(f, g);} 
inline Function2 operator/ (Function2 f, Function2 g) {return Div(f, g);} 
inline Function2 operator- (Function2 f) {return Chs(f);} 
inline void Function2 operator+= (Function2 b) {f = Add(*this, b);} 
inline void Function2 operator*= (Function2 b) {f = Mul(*this, b);} 

To construct effectively the classical functions, we define the class CMonome for $cx^n$ and the class CMonome2 for $cx^{n1}y^{n2}$.

class CMonome : public CVirt {
    R c; int n;
public:
    CMonome (R cc = 0, int k = 0) : c (cc), n (k) {}
    R operator () (R x) ; // return $c x^n$
    CVirt *de () {return n ? new CMonome (c * n, n - 1) : zero;}
};
Function Function::monome (R x, int k) {return new CMonome (x, k);} 

class CMonome2 : public CVirt2 {
    R c; int n1, n2;
public:
    CMonome2 (R cc = 0, int k1 = 0, int k2 = 0) : c (cc), n1 (k1), n2 (k2) {}
    R operator () (R x, R y) ; // return $c x^{n1} y^{n2}$
    CVirt2 *dex () {return n1 ? new CMonome2 (c * n1, n1 - 1, n2) : zero;}
    CVirt2 *dey () {return n2 ? new CMonome2 (c * n2, n1, n2 - 1) : zero;}
};
Function2 Function2::monome (R x, int k, int l) {return new CMonome2 (x, k, l);} 

With the same technique, we construct the classes using the same rules:

CFunc a classical function defined with a pointer to a function (of type R (*)(R)). This class derives from Cvirt 
CComp a class to handle the composition of $f,g$, like $(x) \rightarrow f(g(x))$ 
CComb a class to handle the composition of $f,g,h$, like $(x) \rightarrow f(g(x),h(x))$. 
CFunc2 a classical function defined with a pointer to a function (of type R (*)(R,R)). This class derives from Cvirt2 
CComp2 a class to handle the composition of $f,g$, like $(x,y) \rightarrow f(g(x,y))$ 
CComb2 a class to handle the composition of $f,g,h$, like $(x,y) \rightarrow f(g(x,y),h(x,y))$
Now to finish, we just give the definitions of the usual global functions.

```c++
CVirt *CVirt::zero = new CMonome;
CVirt2 *CVirt2::zero = new CMonome2;
```

```
Function::Function (R (*f) (R)) : p(new CFunc(f)) {}
Function::Function (R x) : p(new CMonome(x)) {}
Function Function::operator() (Function f) {return new CComp (p, f.p);}
Function2 Function::operator() (Function2 f) {return new CComp2 (p, f.p);}
```

```
Function2::Function2 (R (*f) (R, R)) : p(new CFunc2(f)) {}
Function2::Function2 (R x) : p(new CMonome2(x)) {}
Function2 Function2::operator() (Function f, Function g)
{ return new CComb (f.p, g.p, p); }
Function2 Function2::operator() (Function2 f, Function2 g)
{ return new CComb2 (f.p, g.p, p); }
```

```
static R add (R x, R y) {return x + y;}
static R sub (R x, R y) {return x - y;}
```

```
Function Log = new CFunc1(log, new CMonome1(1,-1));
Function Chs = new CMonome (-1., 1);
Function Identity = new CMonome (-1., 1);
```

```
Function2 CoordinateX = new CMonome2 (1., 1, 0);// x
Function2 CoordinateY = new CMonome2 (1., 0, 1);// y
Function2 One2 = new CMonome2 (1., 0, 0);// 1
Function2 Add = new CFunc2 (add, Function2(1.), Function2(1.));
Function2 Sub = new CFunc2 (sub, Function2(1.), Function2(-1.));
Function2 Mul = new CMonome2 (1., 1, 1);
Function2 Div = new CMonome2 (1., 1, -1);
//     pow(x,y) = x^y = e^{log(x) y}
Function2 Pow = new CFunc2 (pow, CoordinateY*Pow(CoordinateX, CoordinateY-One2)),Log(CoordinateX)*Pow);
```

And for example to construct, the $\sin, \cos$ functions, we just write:

```
Function Cos(cos),Sin(sin);
Cos.setd(-Sin); // set the derivative of cos
Sin.setd(Cos); // set the derivative of sin
Function d4thCos=Cos.d().d().d().d(); // construct the fourth derivative
```

3. Design of variationnal forms

Consider a PDE problem on a domain $\Omega$, for example, we take the Stokes problem:
Find the velocity vector $u$ and the pressure $p$ of a fluid defined in the domain $\Omega$, such that

$$\begin{align*}
\alpha u - \nu \Delta u + \nabla p &= 0 \\
\nabla \cdot u &= 0
\end{align*}$$

in $\Omega$. (1)
For simplicity, let us choose Dirichlet boundary conditions on the velocity: \( u = u_\Gamma \) on \( \Gamma = \partial \Omega \). Here \( \alpha \) and \( \nu \) can be given positive functions (\( \alpha \) can be zero but \( \nu \) must be greater than a constant number \( \beta > 0 \).

To solve this problem, we introduce the variational form of equation (1). Take a test velocity vector \( v \), null on \( \Gamma \), and take a test pressure function \( q \), and multiply the first equation by \( v \) and integrate; do the same on the second equation with function \( q \). We get:

\[
- \int_{\Omega} \alpha uv + (\nu \Delta u) v + (\nabla p) v = \int_{\Omega} fv, \\
\int_{\Omega} (\nabla \cdot u) q = 0.
\]

After integration by parts the variational formulation of the Stokes problem is: find \( u \in H^1(\Omega)^2 \) and \( p \in L^2(\Omega) \), with \( u = u_\Gamma \) on \( \Gamma \), such that:

\[
\begin{align*}
\int_{\Omega} \alpha uv + \nu \nabla u \cdot \nabla v + \nabla p \cdot v &= \int_{\Omega} fv, \\
\int_{\Omega} \nabla \cdot u q &= 0.
\end{align*}
\]

The variational form is the sum of a bilinear form \( a \), and a linear form \( l \).

\[
a((u, p), (v, q)) = \int_{\Omega} \alpha uv + \nu \nabla u \cdot \nabla v + \nabla p \cdot v + \int_{\Omega} (\nabla \cdot u) q;
\]

\[
l((v, q)) = - \int_{\Omega} fv = 0.
\]

The problem is: find \( u \in H^1(\Omega)^2 \) and \( p \in L^2(\Omega) \), with \( u = u_\Gamma \) on \( \Gamma \), such that:

\[
a((u, p), (v, q)) + l((v, q)) = 0 \quad \forall (v, q) \in H^1_0(\Omega)^2 \times L^2(\Omega).
\]

The bilinear form is the integral of sums of product an unknown functions or their derivative and test function or their derivative. Formally, we have in this case \( n = 3 \) types of unknowns functions \( u_1, u_2, p \) associated respectively to the number 0, 1, 2, and the function unknowns \( w \) can appear in three states: the function \( w \) (number 0), \( \partial_x w \) (number 1), or \( \partial_y w \) (number 2). We can do exactly the same numbering for the test functions part. So the bilinear form can be written as a sum of terms in \( C^0 \times \{0, \ldots, n-1\} \times \{0, 1\} \) where \( C^0 \) is the set of functions, \( n \) is the number of unknown functions, the first \( \{0, \ldots, n-1\} \times \{0, 1\} \) term define the unknowns and the last \( \{0, \ldots, n-1\} \times \{0, 1\} \) term define the test function part.

For example the terms of the bilinear part of the variational form can be translated in:

\[
\begin{align*}
\alpha uv & \equiv (\alpha, [0, 0], [0, 0]) + (\alpha, [1, 0], [1, 0]) \\
\nu \nabla u \cdot \nabla v & \equiv (\nu, [0, 1], [0, 1]) + (\nu, [0, 0], [0, 2]) + (\nu, [0, 2], [0, 0]) + (\nu, [1, 1], [1, 1]) + (\nu, [1, 2], [1, 2]) \\
\nabla p \cdot v & \equiv (1, [2, 1], [0, 0]) + (1, [2, 2], [1, 0]) \\
\n\nabla \cdot u q & \equiv (1, [0, 1], [2, 0]) + (1, [1, 2], [2, 0]).
\end{align*}
\]

We can do the same analysis for the linear form, and to close the analysis we have just to remark that in general we have only two types of integrals: over the domain or on its boundary.

To make the implementation easier we introduce, a kind of linear combination, or a sparse vector of \( R^I \) like:

\[
\sum_{k=1}^{n} r_k i_k, \quad \forall k = 1 \ldots n, r_k \in R \text{ and } i_k \in I.
\]
Where $R$ here, is an algebra of functions, and where $I$ is the set of indices (in case of linear form an index is just a pair of integers).

A simple member function $\text{add}(r, i)$ adds a term $(r, i)$ to the linear combination, plus similarly the classical operator $+$ and external product $\ast$ of $R^{(I)}$.

Thanks to the STL the class $\text{LinearComb}$ is easy to construct with $\text{vector}$ and $\text{pair}$ where a $\text{pair}$ is just the following structure:

```cpp
#include <vector>
#include <utility>

template<
class A,
class B>
struct pair
{ // the struct pair from STL
  A first; B second;
  pair(A a, B b) : first(a), second(b) {} }

template<class R, class I>
class LinearComb : public E_F0
{ public:
  typedef I TI;
  typedef R TR;
  typedef pair<I, R> K; // to store 1 element
  typedef vector<K> array;
  typedef array::const_iterator const_iterator;
  typedef array::iterator iterator;

  array v;

  LinearComb() : v() {}
  LinearComb(const R & r, const I & i) : v(1)
  { v[0] = make_pair<I, R>(i, r); }

  LinearComb(const LinearComb & l) : v(l.v) {}

  void operator=(const LinearComb & l) { v = l.v; }

  void add(const I & i, const R & r) { // add of term
    for (iterator k = v.begin(); k != v.end(); k++)
      if (k->first == i) { k->second += r; return; }
    v.push_back(make_pair<I, R>(i, r)); }

  void operator+=(const LinearComb & l) {
    for (const_iterator k = l.v.begin(); k != l.v.end(); k++)
      add(kk->first, kk->second); }

  void operator*=(const R & r) {
    for (iterator k = v.begin(); k != v.end(); k++)
      k->second = k->second*r; }

};
```

```cpp
template<class R, class I>
LinearComb<R, I> operator+(const LinearComb<R, I> & a,
                           const LinearComb<R, I> & b)
{ LinearComb<R, I> r(a);
  r += b;
  return r; }
```
template<class R, class I>
LinearComb<R, I> operator*(const R & v, const LinearComb<R, I> & l)
{
    LinearComb<R, I> r(l);
    r *= v;
    return r;
}

template<class R, class I>
LinearComb<R, I> operator*(const LinearComb<R, I> & l, const R & v)
{
    LinearComb<R, I> r(l);
    r *= v;
    return r;
}

with:

definitions:
enum operator type { op_id=0, op_dx=1, op_dy=2, op_dz=3};

template<class testfunction>
struct FunctionNumber {
    const int i;
    FunctionNumber(int j) : i(j) {}  
    operator int() const {return i;}
    void operator=(int j) { i=j; }
};

The C++ types corresponding to the two kinds of linear variational forms are:

// the linear variational form with unknown function
typedef LinearComb<Function2, pair<FunctionNumber<false>, operator type>> LVOunknown;

// the linear variational form with test function
typedef LinearComb<Function2, pair<FunctionNumber<true>, operator type>, pair<FunctionNumber<false>, operator type>>> LVOtest;

The C++ types to handle bilinear forms is:

typedef LinearComb<Function2, pair< pair<FunctionNumber<false>, operator type>, pair<FunctionNumber<true>, operator type>, operator type>>> BVO;

A BVO is a construct with a product of LVOtest and LVOunknown. We have used the two following product operators:

inline BVO operator*(const LVOunknown & a, const LVOtest & b)
{
    BVO r;
}
for (LVOunknown::const_iterator i=a.v.begin();i!=a.v.end();i++)
for (LVOtest::const_iterator j=b.v.begin();j!=b.v.end();j++)
{
    const LVOunknown::K vi(*i);
    const LVOtest::K vj(*j);
    r.add(make_pair(vi.first,vj.first),vi.second*vj.second)
}
return r;
}

inline BVO operator*(const LVOtest & a,const LVOunknown & b)
{ return b*a;}
The function to differentiate a linear combination is:

template<class L>
L Diff(const L & u, const operator* & d) {
    L r= L(u); // copy
    for (typename L::iterator i=r->v.begin();i!=r->v.end();i++)
    { 
        operator* & dd=i->first.second;
        Function2 f= i->second;
        if (dd != op_id)
            Error(" Sorry diff(diff(..)) too complex ");
        else
            { 
                if (CVirt2::zero != f.d() )
                    L->Add(i->first,f.d());
                dd = d ; } 
    }
    return r; }

The variational forms for test and unknown functions are:

inline LVOtest test(int i) \{ return LVOtest(i, op_id);\}
inline LVOunknown unknown(int i) \{ return LVOunknown(i, op_id);\}
and the two functions \( dx= \partial_x \) and \( dy= \partial_y \) to differentiate are:

template<class L> L dx(const L & u) \{ return Diff(u,id_dx);\}
template<class L> L dy(const L & u) \{ return Diff(u,id_dy);\}

Now we use all the classes to construct the variational form of the Stokes equation. First, construct all the linear forms corresponding to the unknown functions \( u_1, u_2, p \) and the corresponding to the test functions \( v_1, v_2, q \):

LVOunknown u1(unknown(0)),u2(unknown(1)),p(unknown(2))
LVOunknown v1(test(0)),v2(test(1)),q(test(2));
double alpha(1./dt);
double nu(1./400.) ;

The translation of the bilinear form (3) and the linear form (4) are:

BVO a = (u1*v1+u2*v2)*alpha
    + (dx(u1)*dx(v1)+dy(u1)*dy(v1)
As we can see, this writing is very close to the mathematical formulation.

To compute the elementary matrix $a$ on an element $K$, we can use this kind of lines:

```cpp
for (npi=0; npi<FI.n; npi++) { // loop on the Quadrature Point
    QuadraturePoint pi(FI[npi]);
    R coef = K.area*pi.a;
    R2 PHat(pi); // QuadraturePoint on $\hat{K}$
    R2 P(T(PHat)); // QuadraturePoint on $K$
    Ku.BF(PHat,fu); // componate of basic unkown function and derivative
    Kv.BF(PHat,fv); // componate of basic test function and derivative
    for ( i=0; i<n; i++ ) {
        for ( j=0; j<m; j++ ) {
            for (BOV::const_iterator l=Op.v.begin(); l!=Op.v.end(); l++) {
                BOV::K ll(*l);
                pair<int,int> jj(ll.first.first),ii(ll.first.second);
                R w_i = fu(i,ii.first,ii.second); // get componate of
                    // unknown func. or derivative
                R w_j = fv(j,jj.first,jj.second); // get componate of test func.
                    // or derivative
                R ccc = ll.second(P.x;P.y);
                a[i][j] += coef * ccc * w_i*w_j;
            }
        }
    }
}
```

where $Ku$ and $Kv$ are the objects associed the finite element space on the element $K$.

4. DESIGNING A MULTITYPE VIRTUAL MACHINE

In this section, we explain the design of the virtual machine for the language. This language can handle lots of types, as can be seen in Section 4.1; it is relatively easy to handle functions with the virtual mechanism. As we remark previously, the virtual machine is based on an extended algebra of “instruction function” with one parameter the stack memory. The stack memory is used to store all the local variable of the language. The “instruction function” can return any type of data. We have two kinds of data the small ones (example: bool, long, double, complex) and the large ones (example: string, Mesh, ...), classically the small data are stored directly, and the large ones are stored via a pointer.

The class `basicForEachType` stores all the data of language's type.

```cpp
class basicForEachType;
typedef const basicForEachType * aType;
```

The size of all small types must be smaller than 24.

```cpp
typedef unsigned char AnyData[24];
```

To make the link between the name of the type and the user language type (a pointer to a `basicForE-`
achType), we use the run time type interface (RTTI) of C++. So to get the user language type we use a map of the STL `map_type`:

```c++
extern map<const string,basicForEachType *> map_type;
```

We can’t use directly the pointer of the `typeid(T)` because the data storage depends on the compilation file. The following template function `atype<T>()` returns the pointer to the internal type.

```c++
template<typename T>
inline basicForEachType * atype() {
    basicForEachType * r=map_type[typeid(T).name()];
    if (! r) { cerr << "Error: aType '" << typeid(T).name() << ", D'ont exist\n";
        assert(0);} 
    return r;}
```

The class to store every small data or pointer with or without dynamic type checking depends on the preprocessor macro `WITHCHECK`:

```c++
class AnyType {
private:
    union {
        AnyData data; // to store any small data.
        void * p; // for debugging to see the value of the data easily
        double r; // for debugging
        long l; }; // for debugging

    #ifdef WITHCHECK
        const basicForEachType *ktype; // the current type of data
    public:
        AnyType(): ktype(0){}
    #else
    public:
        AnyTypeWithOutCheck();{}
    #endif
    void operator = (void *pp) {p=pp;}
};
```

This is because compilers do not work generally with template constructors and template cast operators, we use two template functions `AnyType SetAny<T>(T &)` and `T GetAny<T>(AnyType &)` to transform back and forth between a T and an AnyType.

First, we introduce, a small class `CheckSize` just to perform size checking at C++ compilation time. We use a specialization of this class with a Boolean ok. If this Boolean is false then we use a private constructor to generate a compile error.

```c++
template<class T,bool ok> class CheckSize { } ;
// specialization to generate an error
```
template <class T> class CheckSize<T,false> {
    CheckSize() {} } ; // private to create an error if use ok is false

Now, the function SetAny transforms a type T to AnyType with a bit copy:

template<typename T,bool B= sizeof(T) <= sizeof(AnyData)> AnyType SetAny(const T & x) { // T -> AnyType
    AnyType any;
    CheckSize<T,sizeof(T)<= sizeof(AnyData) > fake;
    #ifdef WITHCHECK
        any.ktype=map_type[typeid(T).name()];
    #endif
    memcpy(&any,&x,sizeof(x));
    return any; }

Similarly, you can write the following functions

template<typename T> T GetAny(AnyType & );
AnyType PtrtoAny(PtrtoAny(void * p, const basicForEachType * r);

We can use, an extend algebra of function to model the instructions of the language. The parameter of the function has to be the stack if we want to produce a reentrant code. In the stack, we store all the local variables of the procedure. Secondly, we have to construct a class basicForEachType to define all the types of the user language.

typedef void *Stack;

So an instruction of the language is an Expression, and an Expression is a pointer to the class E_F0.

class E_F0;
typedef const E_F0 * Expression;
class E_F0 { public:
    virtual AnyType operator()(Stack) const =0;
    // the evaluation operator
    virtual bool Empty() const { return !this; } // is an empty expression to make optimization
    virtual bool EvaluableWithoutStack() const {return false;} // for constant expression
    virtual ~E_F0() {}
};

We use exactly the same technique as in the algebra of functions to construct the virtual machine. Before getting a value of type Tg we use the GetAny<Tg>() function and to return a value of type Ts, we use the SetAny<Tg>() function.
For example, a local variable of the language can be

```cpp
class LocalVariable : public E_F0 {
    size_t offset; // the position in the stack
    aType t; // type of the variable just for check
public:
    AnyType operator ()(Stack s) const {
        return PtrtoAny(static_cast<void*>(static_cast<char*>(s)+offset),t);
    }
    LocalVariable(size_t o,aType tt):offset(o),t(tt) {assert(tt); }
};
```

It is very simple to write an expression to make a “for loop” with the C++ syntax. The return type of a “for loop” expression is clearly a void. The break and the continue instruction are handled by the C++ exception mechanism. So the class is:

```cpp
class ForExpression : public E_F0 {
    public:
    Expression i0,Expression i1,Expression i2,Expression ins;
    AnyType operator ()(Stack s) const { // evaluation
        for ( (*i0)(s);GetAny<bool>((*i1)(s));(*i2)(s)) {
            try {(*ins)(s);} // to catch break and continue instruction
            catch ( E_exception & e) {
                if (e.code == E_exception::e_break) break;
                else if (e.code == E_exception::e_continue) continue;
            }
        }
        return Nothing;
    }
    ForExpression( Expression init,Expression test,
        Expression incr,Expression instr) :
        i0(init),i1(test),i2(incr),ins(instr) {}
};
```

At compilation time, the type of the expression must be known, so we need to define what is the type of the objects of the languages.

Polymorphism for operators and functions is easy to achieve. First, an operator is just a function with a special name and second we just choose the function to call with the type of parameters. But unfortunately, in the language we have four scalar types `bool`, `long`, `double`, `complex<double>`. We want to be able to add every type to every type, so four cases. A lot of operators, the risk of mistake is too important. We have no easy way to do that with templates (an operator parameter); so to solve this problem, we can use automatic casting like in C++. It is clear that one can cast (promote) a numeric type to another type, e.g. a float into a complex. Now we have to solve the problem of ambiguity `((long) + (double))` could mean `(long) + (long)`, `(double) + (double)`, `(complex) + (complex)`, or `(string) + (string)`. The rule to solve this kind of ambiguity is to choose in order to minimize the number of casts (one here), and we add a level of a priority only for binary operator to remove the last ambiguity between `((long) + (long))` and `((double) + (double))` and choose `(double) + (double)`, where the level is 10, 20, 30, 40, 50 for respectively `bool`, `long`, `double`, `complex`, `string` and the level is 0 otherwise.

In the language, we have also some “left expression” like in C++, but expression `a` can be both “left expression” (in `a=x`) and “right expression (in `(a+10)`). In a “left expression”, we need the memory address of the objects so the expression returns a “kind of pointer” to the object, but otherwise in the “right expression”
the expression haves to return the objects in self. To solve, this difficulty, a “kind of pointer” can be a pointer to the value or the value in self, we use the user language cast mechanism by adding one more cast (pointer to pointer value) and this kind of cast is clearly not counted when we solve the problem of ambiguity.

This is the list of the kernel type names:

- OneOperator is a class for a function name with different type of parameter;
- basicForEachType is a class for a type the objects of the language;
- EF0 is a pure virtual class for the instruction expression;
- aType is a pointer to a constant basicForEachType;
- Expression is a pointer to a class EF0;
- Function1 is the type name typedef AnyType (* Function1)(Stack, const AnyType &);
- Polymorphic is a class, which models a polymorphic object. It’s derived from an EF0;
- CF0 is a class for compilation expressions, just 2 pointers. One is an Expression and the other is an aType; These two pointers define the expression and the type of the expression;
- basicACF0 is a class for an array of compilation expressions. This array is used to pass the arguments of an operator;
- TableOfIdentifier is a class for a table of identifiers.

4.1. The types of the language

The type definition is the kernel of the language, everything is in the class basicForEachType.

class basicForEachType {
    const type_info * ktype; // the real type_info to get name
public:
    typedef map<aType, CastFunc>::const_iterator const_cast_iterator;
    typedef const char * Key;
    const char * name() const { return this ? ktype->name() : "NULL" ;}
    const size_t size; // the size of the data
    virtual bool CastingFrom(const basicForEachType * t) const ;
    virtual bool SameTypeRight(const basicForEachType * t) const ;
    virtual CF0 CastingFrom(CF0 & e) const ;
    aType right() const;
aType left() const;
    Expression RightValueExpr(Expression f) const;
    Expression LeftValueExpr(Expression f) const;
    virtual CF0 Initialization(CF0 & e) const ;
    virtual Expression Destroy(CF0 & e) const ;
    virtual bool ExistDestroy() const { return destroy;}
    TableOfIdentifier ti; // all operator of the type
Trick: To make the automatic cast easily, we need for all the types a way to promote in this type, see the method CastingFrom to know if this possible or not to create the cast expression.

4.2. One operator

First we use the class OneOperator to construct the compiler expression for one named operator in function of the type of the arguments. We have one kind of construction, and two kinds of query possible to do this operation with or without casting of the parameter. Remark that in the two cases the promotion of a left expression to a right expression is done.

In fact, we just use a OneOperator to define the cast into this type.

class OneOperator : public ArrayOfaType {
  const basicForEachType * r; // return type
  OneOperator *next; // to make a list of OneOperator

public:
  int pref; // to solve ambiguity for binary operator
  OneOperator(aType rr)
    : r(rr),ArrayOfaType(),next(0),pref(0) {assert(r);}
  OneOperator(aType rr,aType a)
    : r(rr),ArrayOfaType(a,false),next(0),pref(0) {}
  OneOperator(aType rr,aType a,aType b)
    : r(rr),ArrayOfaType(a,b,false),next(0),pref(0) {}
  OneOperator(aType rr,aType a,aType b,aType c)
    : r(rr),ArrayOfaType(a,b,c,false),next(0),pref(0) {}
  ...

  // add a new OneOperator
  void operator+=(OneOperator &a)
    {assert(a.next==0);a.next=next;next=&a;}

  virtual ~OneOperator();

typedef pair<const OneOperator *,int> pair_find;
pair_find Find(const ArrayOfaType & at) const ;
pair_find FindWithOutCast(const ArrayOfaType & at) const ;
operator aType () const { return r;} // the return type the operator
  virtual E_F0 * code(const basicAC_F0 &) const =0;
};
where the class ArrayOfaType is just like an array of aType with a basic constructor, and a Boolean to take into account the variable parameter list if any.

The virtual function code constructs effectively the expression (a pointer to E_F0), and the two functions Find and FindWithOutCast return a pair of the OneOperator pointer and the number of ambiguities.

4.3. How to handle a polymorphic object

A polymorphic object is just like a C++ class. On a polymorphic object, we can define any operator, function, member, and method. The operator, member, method or function are defined with a string (const char *). So it is almost a map of string and OneOperator object.

First we define a simple struct to sort in lexical order the map

```cpp
// to sort the a string (const char *)
struct Keyless : binary_function<const char *, const char *, bool>
{
  typedef const char * Key;
  bool operator()(const Key& x, const Key& y) const
  {
    return strcmp(x,y)<0; }
};
```

The class Polymorphic can be:

```cpp
class Polymorphic : public E_F0 {
private:
  typedef const char * Key;
  typedef OneOperator * Value;
  typedef map<Key,Value,Keyless> maptype;
  typedef maptype::const_iterator const_iterator;
  typedef maptype::iterator iterator;
  maptype m; // all polymorphism of the objet
public:
  Polymorphic() : m(), e(0) {}
  virtual AnyType operator()(Stack) const { return Nothing;}
  virtual bool Empty() const { return true; } // by default do nothing
  const OneOperator * Find(const char *op, const ArrayOfaType &at) const;
  const OneOperator * FindWithOutCast(const char *op,
    const ArrayOfaType &at) const;

  // to add a list of operator with same name
  void Add(const char *op,
    OneOperator * p0 ,OneOperator * p1=0,OneOperator * p2=0,... ) ;
  friend ostream & operator<<(ostream & f,const Polymorphic & a);
};
```

The two functions Find and FindWithOutCast are just the encapsulation of the same function of the class OneOperator, if the operator name exists, and generate an error otherwise.

4.4. Identifier tables

The table of Identifier is theoretically just a STL map of names and compiled expression, but when we go outside a block we must delete all local identifier in a right order variable; so we create a new structure Value
which derive from the class $C_F0$, and we add a link to the previous variable, and a Boolean if the variable must be removed.

```cpp
class TableOfIdentifier {
public:
    struct Value {
        pKV * next; // link all the variable in reverse order
        // to call delete on each variable
        bool del;
        Value(const $C_F0$ & vv, pKV * n, bool dd=true) : $C_F0$(vv), next(n), del(dd) {}
        Value(aType t, E_F0 * f, pKV * n, bool dd=true): $C_F0$(t, f), next(n), del(dd) {}
    }; // to store the type and the expression
    pKV * listofvar;

    typedef Value Key;
    typedef map<Key, Value, Keyless> maptype;
    typedef pair<const Key, Value> pKV;
    typedef maptype::iterator iterator;
    typedef maptype::const_iterator const_iterator;

    struct Value : public $C_F0$ {
        pKV * next; // link all the variable in reverse order
        // to call delete on each variable
    }

    C_F0 Find(Key) const;
    C_F0 Find(Key, const basicAC_F0 & ) const;

    C_F0 NewVar(Key k, aType t, const C_F0 & i);
    C_F0 NewVar(Key k, aType t, size_t & top, const basicAC_F0 & args);
    C_F0 NewVar(Key k, aType t, size_t & top);
    C_F0 NewID(aType t, Key k, C_F0 & c, size_t & top);
    C_F0 NewID(aType t, Key k, C_F0 & c, const ListOfId & l, size_t & top);

    friend ostream & operator<<(ostream & f, const TableOfIdentifier & );
    C_F0 destroy();
    TableOfIdentifier() : listofvar(0) {};
};
```
4.5. How to use

Finally, all the operators are in a global variable:

```
Polymorphic * TheOperators = new Polymorphic();
```

And all the globals of the language are stored in a global table of identifiers.

```
TableOfIdentifier Global;
```

To add operator the + operator, we just write:

```
TheOperators->Add("+",
   new OneBinaryOperator<Op2_add<long,long,long>> ,,
   new OneBinaryOperator<Op2_add<double,double,double>> ,,
   new OneBinaryOperator<Op2_add<Complex,Complex,Complex>> ,,
   new OneBinaryOperator<Op2_padd<string,string*,string*>> );
```

where the class OneBinaryOperator is:

```
template<typename C>
class OneBinaryOperator : public OneOperator{
   typedef typename C::result_type R;
   typedef typename C::first_argument_type A;
   typedef typename C::second_argument_type B;

   class Op : public E_F0 { // compiled expression
      typedef typename C::result_type Result;
      Expression a,b;
   public:
      AnyType operator()(Stack s) const
      { return SetAny<R>(static_cast<R>(C::f( GetAny<A>((*a)(s)) ,
                  GetAny<B>((*b)(s)))) );
      }
      Op(Expression aa,Expression bb) : a(aa),b(bb) {}
   };

   public:
   E_F0 * code(const basicAC_F0 & args) const
   { return new Op(t[0]->CastTo(args[0]),t[1]->CastTo(args[1]));}
   OneBinaryOperator():
   OneOperator(map_type[typeid(R).name()]),
   map_type[typeid(A).name()]),
   map_type[typeid(B).name()])
   {pref = SameType<A,B>::OK ;}
};
The class `Op2_add` to make the “add” is just:

```cpp
template<class R, class A=R, class B=A>
struct Op2_add: public binary_function<A,B,R> {
    static R f(const A & a, const B & b) {
        return ((R)a + (R)b);}
};
```

We describe the class `C_F0` and how to construct an instruction of the language.

```cpp
class C_F0 {
    friend class CC_F0;

    protected:
    Expression f; // the expression code
    aType r;     // the expression type

    public:
    C_F0() :f(0),r(0) {}
    C_F0(const C_F0 & c):f(c.f),r(c.r) {}
    C_F0(const C_F0 & e, const char *op, const basicAC_F0 & p);
    C_F0(const C_F0 & e, const char *op, AC_F0 & p);
    C_F0(const C_F0 & e, const char *op, const C_F0 & ee);
    C_F0(const C_F0 & e, const char *nm);
    // without parameter ex f()
    C_F0(const Polymorphic * pop, const char *op);
    // unary operator
    C_F0(const Polymorphic * pop, const char *op, const C_F0 & a);
    // binary operator
    C_F0(const Polymorphic * pop, const char *op, const C_F0 & a,
         const C_F0 & b);
    // n-ary
    C_F0(const Polymorphic * , const char * , const basicAC_F0 & );
};
```

For example, to construct the expression `ab` sum of two expression `a,b`, we have to write:

```cpp
C_F0 ab(TheOperators,"+",a,b);
```

We used the same rule for all the operators.

5. Bison grammar

Now we explain, briefly, how to use this class with the compiler bison or yacc processor [3].

First, we introduce the return type of all the terminal and non-terminal terms of the grammar. But unfortunately, it is impossible to add class with constructor in an union, so we construct another class `CC_F0` with the same data members `r,f`, with the assignment operator form a class `C_F0`, and the cast operator to class `C_F0`. 

After remembering the following construction:

```
C_F0 ab(TheOperators,"+",a,b);
```

build the sum of two expression a,b. The non terminals for “expression” with non set operator are:

```c
no_set_expr:
  unary_expr
    | no_set_expr '*' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr DOTSTAR no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '/' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '%' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '+' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '-' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr LTLT no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr GTGT no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr AND no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr OR no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '<' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr LE no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr '>' no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr GE no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr EQ no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)}
    | no_set_expr NE no_set_expr {$$=C_F0(TheOperators,$2,$1,$3)};
```

where the non terminal `unary_expr` handles all the left unary expressions.

An interesting part is the primary terms of the expression. This is defined with:

```c
primary:
  ID {$$=Find($1);}
  | LNUM {$$= CConstant($1)}
  | DNUM {$$= CConstant($1)}
  | CNUM {$$= CConstant(complex<double>{0,$1})}
  | STRING {$$= CConstant<const char *>($1)}
  | primary '(' parameters ')' {$$=C_F0($1,$2,$3);}
  | primary '[' Expr ']' {$$=C_F0($1,$2,$3)}
```
C++ TOOLS TO CONSTRUCT OUR USER-LEVEL LANGUAGE

831

| primary ' [' ']' { $$=C_F0($1,"["\])} |
| primary ' .' ID { $$=C_F0($1,$3) ;} |
| primary PLUSPLUS { $$=C_F0(The Operators,"right++",$1)} |
| primary MINUSMINUS { $$=C_F0(The Operators,"right--",$1)} |
| TYPE '(' Expr ')' { $$=$1->right()->CastTo($3) } ; |
| '(' Expr ')' { $$=$2} |
| '[' array ']' { $$=C_F0(The Operators,\[",$2)} |

where ID, LNUM, DNUM, CNUM, STRING, TYPE, PLUSPLUS, MINUSMINUS are terminal associated to an indenticator, a numeric number, a type, ++, −−.

The unary operator right ++ (resp. −−) is associated to the string right++ (resp. right--), just to differentiate the left or right unary operator ++ or −−.

6. Examples with FreeFem++

6.1. Navier-Stokes

We give an example to solve the driven cavity flow problem. It is solved first at zero Reynolds number (Stokes flow) and then at Reynolds number 100 (Navier Stokes).

The driven cavity problem computes the flow in the unit box \([0,1]^2\), with a zero velocity on left, right and down sides and \((1,0)\) velocity on the upper side.

The velocity and pressure formulation is used first, and then the calculation is repeated with the stream function vorticity formulation.

The Stokes equations in a domain \(\Omega = [0,1]^2\) are:

\[
-\Delta u + \nabla p = 0 \\
\nabla \cdot u = 0 \\
\]

in \(\Omega\) (6)

where \(u\) is the velocity vector and \(p\) the pressure. For simplicity, let us choose Dirichlet boundary conditions on the velocity, \(u = u_\Gamma\) on \(\Gamma = \partial \Omega\).

A classical way to discretize the Stokes equation with a mixed formulation (see for example Chap. 5 in [10] for details) is to solve the variational problem and then discretize it:

Find \((u_h, p_h) \in X_h \times M_h\) such that \(u_h = u_\Gamma\) on \(\Gamma\), and such that

\[
\int_{\Omega_h} \nabla u_h \cdot \nabla v_h - \int_{\Omega_h} p_h \nabla \cdot v_h = 0, \quad \forall v_h \in X_{oh} \\
- \int_{\Omega_h} \nabla \cdot u_h q_h = 0, \quad \forall q_h \in M_h
\]

(7)

where \(X_{oh}\) is the space of functions of \(X_h\) which are zero on \(\Gamma\). The velocity space is approximated by \(X_h\), and the pressure space is approximated by \(M_h\).

The mesh is constructed by:

```
mesh Th=square(8,8);
```

The labels assigned by \texttt{square} to the bottom, right, up and left edges are respectively 1, 2, 3, 4.

We use a classical Taylor-Hood (Sect. 4.3.3 in [10]) element to solve the problem:

The velocity is approximated with the \(P_2\) elements \((X_h)\), and the the pressure is approximated with the \(P_1\) elements \((M_h)\):

\[
X_h = \{ v \in H^1(\Omega) / \forall K \in T_h \; v|_K \in P_2 \}
\]

\[
M_h = \{ q \in L^2(\Omega) / \forall K \in T_h \; q|_K \in P_1 \}
\]
and
\[ M_h = \{ v \in H^1(\Omega) / \forall K \in T_h, \ v|_K \in P_1 \}. \]

The finite element spaces and functions are constructed by:

```plaintext
fespace Xh(Th,P2);
fespace Mh(Th,P1);
Xh u2,v2;
Xh u1,v1;
Xh p,q;
```

The Stokes operator is implemented as a system-solve for the velocity \((u_1, u_2)\) and the pressure \(p\). The test function for the velocity is \((v_1, v_2)\) and \(q\) for the pressure, so the variational form (7) in FreeFem++ language is:

```plaintext
solve Stokes (u1,u2,p,v1,v2,q,solver=Crout) =
    int2d(Th) ( ( dx(u1)*dx(v1) + dy(u1)*dy(v1) 
                 + dx(u2)*dx(v2) + dy(u2)*dy(v2) )
                 + p*q*(0.000001)
                 - p*dx(v1)+ p*dy(v2)
                 - dx(u1)*q+ dy(u2)*q
             )
    + on(3,u1=1,u2=0)
    + on(1,2,4,u1=0,u2=0);
```

Each unknown has its own boundary conditions. The results of the Stokes problem are shown in Figure 1.

**Technical Remark.** There is some arbitrary decision here as to where to affect the boundary condition within the linear system (see [9] for more details). Basically the Dirichlet operator (on) should be associated with the unknown which contains it so that the penalization appears on the diagonal of the matrix of the underlying discrete linear system, otherwise it will be ill conditioned.

**Remark.** Notice that the term \(p\cdot q\cdot (0.000001)\) is added, because the Crout solver (Chap. 2.6.2 in [7]) needs all the sub-matrices to be invertible.

If the streamlines are required, they can be computed by finding \(\psi\) such that \(\text{rot}\ \psi = u\) or better,

\[-\Delta \psi = \nabla \times u\]

```plaintext
Xh psi,phi;
solve streamlines(psi,phi,solver=GC) =
    int2d(Th) ( dx(psi)*dx(phi) + dy(psi)*dy(phi) )
    + int2d(Th) ( -phi*(dy(u1)-dx(u2)) )
    + on(1,2,3,4,psi=0);
plot(psi,cm="stream line",wait=1); // (Fig. 2, right)
```

Now the time dependant Navier-Stokes equations are solved

\[
\frac{\partial u}{\partial t} + u \cdot \nabla u - \Delta u + \nabla p = 0, \ \nabla \cdot u = 0.
\]
with the same boundary conditions and with initial conditions $u = 0$.

This is implemented by using the convection operator $\text{convect}$ for the term $\frac{\partial u}{\partial t} + u \cdot \nabla u$, giving a discretization in time

$$\frac{1}{\delta t}(u^{n+1} - u^n \circ X^n) - \nu \Delta u^{n+1} + \nabla p^{n+1} = 0, \quad \nabla \cdot u^{n+1} = 0.
$$

(9)

The term, $u^n \circ X^n(x) \approx u^n(x - u^n(x) \delta t)$ will be computed by the operator "convect", so we obtain

```c
int i=0;
real nu=1./100.;
real dt=0.1;
real alpha=1/dt;
Xh up1,up2;

problem NS (u1,u2,p,v1,v2,q,solver=Crout,init=i) =
  int2d(Th)(
    alpha*( u1*v1 + u2*v2)
    + nu * ( dx(u1)*dx(v1) + dy(u1)*dy(v1)
    + dx(u2)*dx(v2) + dy(u2)*dy(v2) )
    + p*q*(0.000001)
    + p*dx(v1) + p*dy(v2)
    + dx(u1)*q+ dy(u2)*q
  )
  + int2d(Th) ( -alpha*
    convect([up1,up2],-dt,up1)*v1 -alpha*convect([up1,up2],-dt,up2)*v2 )
  + on(3,u1=1,u2=0)
  + on(1,2,4,u1=0,u2=0);
```

Figure 1. Stokes solution: the left part is the plot of the isovalues of the pressure and the velocity vector field; the right part is the plot of the streamlines.
Figure 2. Navier-Stokes solution at $Re = 100$: the left part is the plot of the isovalue of the pressure and the plot of velocity; the right part is the plot of the streamlines.

```c
for (i=0;i<10;i++)
{
    up1=u1;
    up2=u2;
    NS;
    if ( !(i % 10)) // plot every 10 iteration
        plot(coef=0.2,cmm=" [u1,u2] et p ",p,[u1,u2]); // (Fig. 2, left)
}
```

6.2. Schwarz domain decomposition

To solve

$$-\Delta u = f, \quad \text{in } \Omega = \Omega_1 \cup \Omega_2 \quad u|_{\Gamma} = 0 \quad (10)$$

the Schwarz algorithm runs like this:

$$-\Delta u_1^{n+1} = f \quad \text{in } \Omega_1 \quad u_1^{n+1}|_{\Gamma_1} = u_2^n \quad (11)$$

$$-\Delta u_2^{n+1} = f \quad \text{in } \Omega_2 \quad u_2^{n+1}|_{\Gamma_2} = u_1^n \quad (12)$$

where $\Gamma_i$ is the boundary of $\Omega_i$ and under the condition that $\Omega_1 \cap \Omega_2 \neq \emptyset$ and that $u_i$ are zero at iteration 1.

Here we take $\Omega_1$ to be a quadrangle, $\Omega_2$ a disk and we apply the algorithm starting from zero.

```c
int inside = 2; // inside boundary
int outside = 1; // outside boundary
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
```
The automatic mesh generator is based on the bamg software [5], and the mathematical background of this mesh generator is available in [4]. The space and problem definition is:

\begin{verbatim}
border e(t=0, pi/2){ x= cos(t); y = sin(t); label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t); label=outside;};
int n=4;
mesh th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh( e(5*n) + e1(25*n) );
plot(th,TH,wait=1); // to see the 2 meshes (Fig. 3)
\end{verbatim}

The space and problem definition is:

\begin{verbatim}
fehspace vh(th,P1);
fehspace VH(TH,P1);
vh u=0,v; VH U,V;
int i=0;

problem PB(U,V,init=i,solver=Cholesky) =
    int2d(TH) ( dx(U)*dx(V)+dy(U)*dy(V) )
    + int2d(TH) (-V) + on(inside,U = u) + on(outside,U = 0);
problem pb(u,v,init=i,solver=Cholesky) =
    int2d(TH) ( dx(U)*dx(V)+dy(U)*dy(V) )
    + int2d(TH) (-V) + on(inside ,u = U) + on(outside,u = 0);
\end{verbatim}
The calculation loop:

```c
for ( i=0 ;i< 10; i++)
{
  PB;
  pb;
  plot(U,u,wait=true); // (Fig. 4)
};
```

Acknowledgements. Acknowledgements to Olivier Pironneau and Dominique Bernardi for an infinite number of coffee talks on these matters.

REFERENCES