

EMT: a DGA-based tool for (electromagnetic) simulations

Matteo Cicuttin

Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica

`std::meeting` @ University of Udine

October 21, 2014

Outline

This talk is about EMT, a simulation tool based on the Discrete Geometric Approach. Currently it can solve electromagnetic wave propagation problems in the frequency domain, but it can be extended to solve various physical problems. Today we will talk about

- Some theory about the problem we want to solve
- Something about the internals of the code
- Why C++ helped me in building it

Time Harmonic Maxwell Equations

The whole electromagnetism is described by the Maxwell's equations:

- Ampère-Maxwell equation: $\nabla \times \mathbf{h} = i\omega\mathbf{d} + \mathbf{j}_s$
- Faraday-Neumann equation: $\nabla \times \mathbf{e} = -i\omega\mathbf{b}$
- Electric Gauss law: $\nabla \cdot \mathbf{d} = \rho$
- Magnetic Gauss law: $\nabla \cdot \mathbf{b} = 0$

Moreover, there are the constitutive relations

- Electric constitutive relation: $\mathbf{d} = \epsilon\mathbf{e}$
- Magnetic constitutive relation: $\mathbf{b} = \mu\mathbf{h}$
- Ohm's law: $\mathbf{j} = \sigma\mathbf{e}$

Electromagnetic wave propagation (in frequency domain)

From Maxwell equations we can obtain the wave propagation equation in the frequency domain:

- Take the Ampère-Maxwell equation: $\nabla \times \mathbf{h} = i\omega\mathbf{d} + \mathbf{j}_s$.
- Substitute $\mathbf{h} = \nu\mathbf{b}$: $\nabla \times (\nu\mathbf{b}) = i\omega\epsilon\mathbf{e} + \mathbf{j}_s$
- Using Faraday-Neumann: $\nabla \times (\nu\nabla \times \mathbf{e}) = -i\omega(i\omega\epsilon\mathbf{e} + \mathbf{j}_s)$
- Rearrange terms: $\nabla \times (\nu\nabla \times \mathbf{e}) - \omega^2\epsilon\mathbf{e} = -i\omega\mathbf{j}_s$

We are not interested in the imposed currents, so we could write the propagation problem as follows:

$$\nabla \times (\nu\nabla \times \mathbf{e}) - \omega^2\epsilon\mathbf{e} = 0$$

Swapping the equations and repeating the same procedure we obtain the *complementary formulation*:

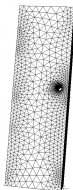
$$\nabla \times (\xi\nabla \times \mathbf{h}) - \omega^2\mu\mathbf{h} = 0$$

How do we solve that problem?

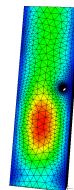
Discrete Geometric Approach

The general idea is:

- Discretize the domain of interest Ω in tetrahedral elements
- Solve the *discrete* Maxwell equations s.t. boundary conditions
- Interpolate quantities on volume elements to obtain the fields



Linear system
 $\mathbf{Ax} = \mathbf{b}$
obtained from
discretization

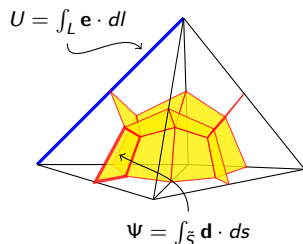


Discretizing the domain

The discretization of Ω is done by means of tetrahedral elements, which form the **primal complex** \mathcal{G} . The barycentric subdivision of \mathcal{G} induces another grid, the **dual complex** $\tilde{\mathcal{G}}$.

Different integral quantities are associated to different geometric entities, on both \mathcal{G} and $\tilde{\mathcal{G}}$. For example:

- Primal edges: electromotive force
- Primal faces: magnetic fluxes
- Dual edges: magnetomotive forces
- Dual faces: electric fluxes



Discretizing the equations

We write the Maxwell equations in the discrete domain as follows:

- F-N: $\mathbf{C}\mathbf{U} = -i\omega\mathbf{\Phi}$
- A-M: $\mathbf{C}^T\mathbf{F} = i\omega\mathbf{\Psi} + \mathbf{I}_s$

\mathbf{C} is the face-edge incidence matrix

\mathbf{U} : electromotive force

$\mathbf{\Phi}$: magnetic flux

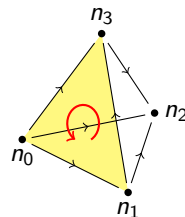
\mathbf{F} : magnetomotive force

$\mathbf{\Psi}$: electric flux

Note the perfect similarity to the continuous ones:

$$\begin{aligned}\nabla \times \mathbf{e} &= -i\omega\mathbf{b} \\ \nabla \times \mathbf{h} &= i\omega\mathbf{d} + \mathbf{j}_s\end{aligned}$$

$$u_0 - u_3 + u_4 = i\omega\Phi_2$$



A very interesting fact is that the discrete equations are **exact**: we haven't introduced any approximation yet!

Discretizing the constitutive relations

In addition to the discrete Maxwell equations, we need the discrete constitutive relations:

Discrete form	Continuos form
$\mathbf{F} \approx M_\nu \Phi$	$\mathbf{h} = \nu \mathbf{b}$
$\Psi \approx M_\epsilon \mathbf{U}$	$\mathbf{d} = \epsilon \mathbf{e}$

The matrices M_ϵ and M_ν are the *constitutive matrices* which relate quantities on the primal grid and quantities on the dual grid. They are the approximate counterparts of ν and ϵ (they are exact *only* for constant fields in the tetrahedron)¹, so this is the place where we introduce an approximation. I will not discuss them further, since it is an extensive topic.

¹L. Codecasa and F. Trevisan, “Piecewise uniform bases and energetic approach for discrete constitutive matrices in electromagnetic problems”

Discrete geometric equations

Now we are ready to solve the DGA equations to obtain the discrete propagation problem:

- Take the Ampère-Maxwell equation: $\mathbf{C}^T \mathbf{F} = i\omega \mathbf{\Psi} + \mathbf{I}_s$
- Substitute $\mathbf{F} = M_\nu \mathbf{\Phi}$
- Using Faraday-Neumann: $\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} = -i\omega (i\omega M_\epsilon \mathbf{U} + \mathbf{I}_s)$
- Rearrange terms: $\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} - \omega^2 M_\epsilon \mathbf{U} = -i\omega \mathbf{I}_s$

We are not interested in the imposed currents, so we could write the propagation problem as follows:

$$\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} - \omega^2 M_\epsilon \mathbf{U} = 0$$

Swapping the equations and repeating the same procedure we obtain the *complementary formulation*:

$$\mathbf{C}^T M_\xi \mathbf{C} \mathbf{F} - \omega^2 M_\mu \mathbf{F} = 0$$

Boundary conditions

The propagation problem in the form we obtained in the previous slide is not very interesting, we need the *boundary conditions* which, for now, are of four types:

- Perfect Electric Conductor: the electric field on a given boundary is zero (Shorted transmission line)
- Perfect Magnetic Conductor: the magnetic field on a given boundary is zero (Open transmission line)
- Admittance: a surface with a given wave admittance (transmission line closed on a load)
- Port: a surface where an electromagnetic wave can enter (the generator)

But to introduce them we need a slight generalization of the problem!

Admittance boundary conditions

The admittance boundary condition is imposed by considering the contribution of the boundary to the circulation of \mathbf{h} :

Our aim is achieved by modifying the Ampère-Maxwell law

$$\mathbf{C}^T \mathbf{F} - \mathbf{F}^b = i\omega \boldsymbol{\Psi} + \mathbf{I}_s$$

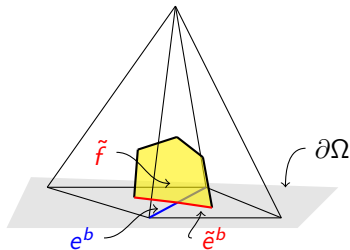
Neglecting I_s our problem becomes

$$\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} - \omega^2 M_\epsilon \mathbf{U} + i\omega \mathbf{F}^b = 0$$

where “ b ” remind us that we are dealing with boundary edges.

But we have an equation in \mathbf{U} and \mathbf{F}^b ! We need to introduce a new matrix, which relates MMFs and EMFs on the boundary edges: $\mathbf{F}^b = M_Y \mathbf{U}^b$, the *admittance matrix*. Our new problem is

$$\mathbf{C}^T \mathbf{M}_\nu \mathbf{C} \mathbf{U} - \omega^2 \mathbf{M}_\epsilon \mathbf{U} + i\omega \mathbf{M}_\gamma \mathbf{U}^b = 0$$



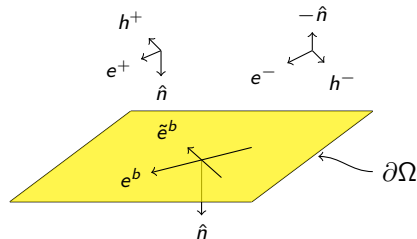
Port boundary conditions

Consider the equation of previous slide:

$$\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} - \omega^2 M_\epsilon \mathbf{U} + i\omega \mathbf{F}^b = 0$$

instead of writing $\mathbf{F}^b = M_Y \mathbf{U}^b$ we could write $\mathbf{F}^b = M_Y \mathbf{U}^b + 2\mathbf{F}^{b-}$.

This allows us to separate the components *entering* Ω from the components *leaving* Ω .



The problem then becomes

$$\mathbf{C}^T M_\nu \mathbf{C} \mathbf{U} - \omega^2 M_\epsilon \mathbf{U} + i\omega M_Y \mathbf{U}^b = -2i\omega \mathbf{F}^b$$

which is the *full electromagnetic wave propagation problem in the frequency domain*.

Why a new code?

Goal: study electromagnetic phenomena inside anechoic chambers

Obstacles:

- matrices resulting from discretization have bad convergence properties (they are indefinite) \implies direct solvers
- anechoic chambers are big \implies *lots* of elements
- high frequency \implies *tons* of elements
- presence of antennas and other objects \implies fine geometrical details \implies *too many* elements

Necessity: reducing the number of elements in the discretization, having a tool where to implement and study equivalent models of the objects inside the chamber.

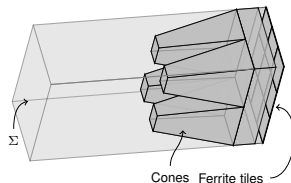
There isn't any code providing such capabilities.

Equivalent model for anechoic walls

The port boundary condition is crucial to study the *unitary cell* (figure), the basic unit of an anechoic wall.

- 2×2 cones
- 3×3 ferrite tiles

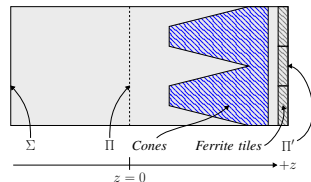
Our goal is to try to substitute the cones and the ferrites with a 2D surface.



Equivalent model for anechoic walls

The port boundary condition is crucial to study the *unitary cell*, to substitute cones and ferrite tiles with a 2D surface. Idea:

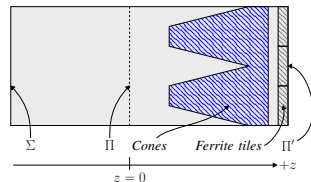
- use the port to apply a plane wave on Σ



Equivalent model for anechoic walls

The port boundary condition is crucial to study the *unitary cell*, to substitute cones and ferrite tiles with a 2D surface. Idea:

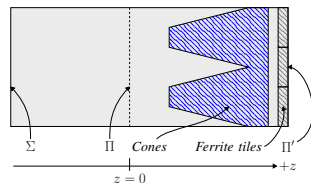
- use the port to apply a plane wave on Σ
- calculate wave impedance on a plane far away from cones



Equivalent model for anechoic walls

The port boundary condition is crucial to study the *unitary cell*, to substitute cones and ferrite tiles with a 2D surface. Idea:

- use the port to apply a plane wave on Σ
- calculate wave impedance on a plane far away from cones
- translate impedance on the rightmost end of the cell

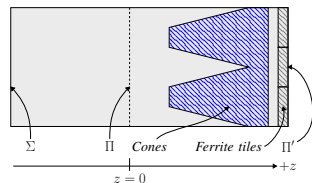


$$Z_{\Pi'}(z) = Z_c \frac{Z_{\Pi} - iZ_c \tan(\beta z)}{Z_c - iZ_{\Pi} \tan(\beta z)},$$

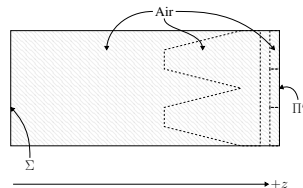
Equivalent model for anechoic walls

The port boundary condition is crucial to study the *unitary cell*, to substitute cones and ferrite tiles with a 2D surface. Idea:

- use the port to apply a plane wave on Σ
- calculate wave impedance on a plane far away from cones
- translate impedance on the rightmost end of the cell
- substitute cones and ferrites with that equivalent impedance



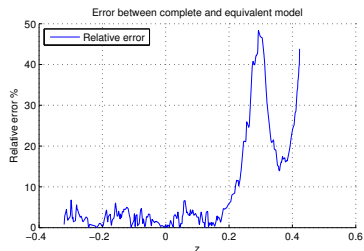
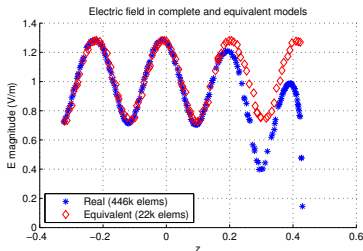
$$Z_{\Pi'}(z) = Z_c \frac{Z_{\Pi} - iZ_c \tan(\beta z)}{Z_c - iZ_{\Pi} \tan(\beta z)},$$



Equivalent model for anechoic walls: results

The proposed equivalent model gave very good results

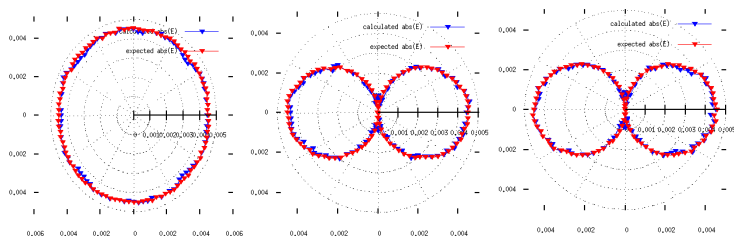
- it allowed 20x reduction of mesh elements
- it allowed 60x reduction of computation times
- in the whole area of interest the relative error was below 5%



Equivalent radiating elements: results

Applicability conditions of the model

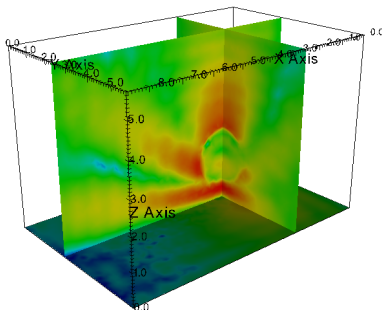
- the sphere must have dimensions similar to λ
- the field must be calculated sufficiently far away from the sphere ($> \lambda/2$)



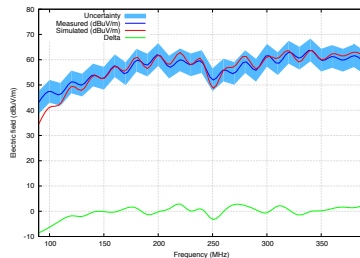
Under these conditions, the model gives very good results!

Anechoic chamber models

But the code was built to simulate entire anechoic chambers...



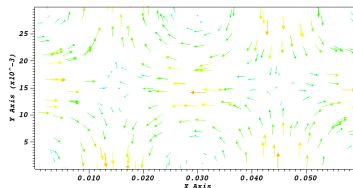
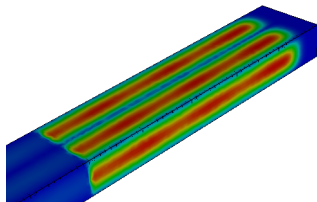
Experiments have shown that the predictions of the code are very accurate!



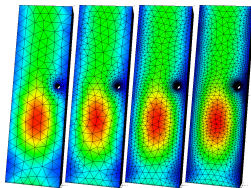
Other applications and features

The developed tools (both theoretical and software) allow to do other interesting things:

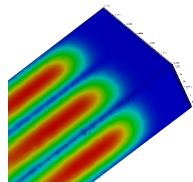
Waveguide propagation



Adaptive mesh refinement



Perfectly matched layers



The code (1)

How do we translate all this theory in code? The *general strategy* to solve a numerical problem with DGA is

- Read the geometry
- Read simulation parameters (frequency, boundary conditions, materials, ...)
- Assemble the linear system of the discretized problem
- Solve it
- Interpolate quantities (for example to get **E** and **H**)
- Output them in some format

The code (2)

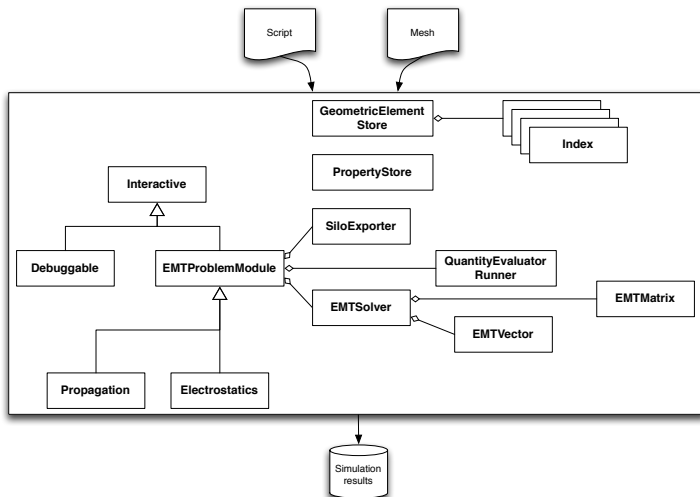
Writing a Matlab script that solves our problem is not too difficult, but we would like to have something more flexible. In particular we want a code that is:

- Expandable and understandable. It should be easy to:
 - code new problems
 - add new numerical solvers
 - add new data import/export procedures
- Modular. If something breaks it must not break surrounding things
- Debuggable and correct.
- Efficient. We want to be able to scale from small problems to very big ones

To achieve these goals some ingredients are needed:

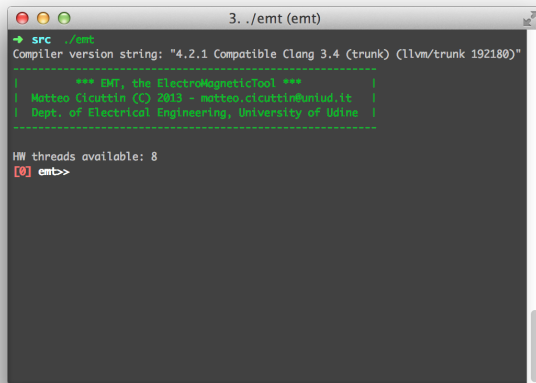
- A serious implementation language: C++11
- A good design: no code is written “on the fly”, without thinking about the structure
- Efficient memory usage and parallelism

The big picture



How does it look?

Time for a little demo!



```
3. ./emt (emt)
→ src ./emt
Compiler version string: "4.2.1 Compatible Clang 3.4 (trunk) (llvm/trunk 192180)"
-----
|      *** EMT, the ElectroMagneticTool ***      |
| Matteo Cicuttin (C) 2013 - matteo.cicuttin@uniud.it |
| Dept. of Electrical Engineering, University of Udine |
|-----|
HW threads available: 8
[0] emt->
```

EMT for programmers

An important goal of EMT is to be programmer-friendly. I'll try to show you some examples on how C++ helped me in giving a clear structure to the code, leading to easy growth of the whole project:

- What is an EMT module and how you write one
- How the numerical solvers are interfaced
- How poorly written Fortran code is hidden behind simple interfaces
- The geometry representation and how it leads to safe code and simplifies parallel execution

The EMT modules

The code is composed by modules: if you want to solve a propagation problem, you enter the propagation module. The modules are one example of the extendability of EMT.

```
→ src ./emt
Compiler version string: "4.2.1 Compatible Clang 3.4 (trunk) (llvm/trunk 192180)"
-----
|      *** EMT, the ElectroMagneticTool ***      |
| Matteo Cicuttin (C) 2013 - matteo.cicuttin@uniud.it |
| Dept. of Electrical Engineering, University of Udine |
|-----|
HW threads available: 8
[0] emt>> enter propagation
[0] emt/propagation>>
```

Modules are “templates” (not in the C++ sense!) that allow the programmer to code new problems in a simple way. All the modules (even an empty one) provide access

- to the geometry
- to all the numerical solvers
- to the data I/O facilities

Coding a new problem

Modules encode in some way the *general strategy* we mentioned some slides ago. Coding a new problem is simply matter of

- creating a new module by subclassing `EMTProblemModule` (some 10s LOCs)
- writing a system assembler (the hardest part)
- telling the solver to run (1 LOC)
- choosing which data to export (some 10s LOCs)
- registering your brand new module with the system (1 LOC)

Let's see some code, the electrostatics module.

The numerical solvers

Another example of extendability is how the numerical solvers are interfaced. Each problem benefits from specific solvers (BiCGstab vs. multigrid vs...) and there are plenty of libraries implementing them, each with its own interface.

In EMT to call **all** of them you need only to know that some solver exists.

```

/* Ask the solver to run */
std::cout << "Solving..." << std::endl;
_owner->solver()->solve();

EMTVector *b = _owner->solver()->get_knowns();
b->set_value(ni_comp, -u, true);
continue;
}

EMTMatrix *A = _owner->solver()->get_matrix();
nj_comp = _owner->_sc.toCompressed(nj_orig);
A->set_value(ni_comp, nj_comp, mGtEG(i,j), true);

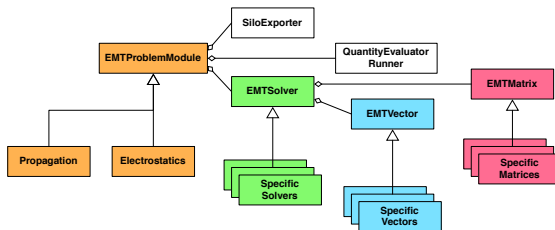
```

This is possible with the *Factory*, an OOP design pattern.

How solvers are interfaced

Whichever solver you get, and whichever matrix/vector format it supports, you only need to

- make it *look like* the generic solver (by subclassing EMTSolver), EMTMatrix and EMTVector
- register it with the system



After this, **all the modules** of EMT are aware of the new solver and able to use it. How? You make your choice **at runtime** with the solver command we already seen.

Representing the geometry

The problem domain Ω , as we already know, is discretized in a tetrahedral grid which could be composed of **millions** of elements. A good representation of the geometry has direct impact on

- the performance and the scalability of the code
- the safety of the code

Let's see some requirements

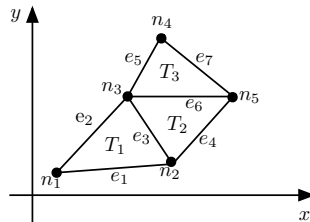
- Keep track of single nodes, edges, triangles and tetrahedrons
- Each element is composed by the indices of its points
- Each element must have an identifier
- Discard duplicates while building data structure
- Fast lookup of elements in both directions (index \rightarrow element and element \rightarrow index)
- Add operations to the elements of the geometry easily

Geometry data structure example

Point	X	Y	Z	Edge	Points
1	x_1	y_1	z_1	1	1 2
2	x_2	y_2	z_2	2	1 3
3	x_3	y_3	z_3	3	2 3
4	x_4	y_4	z_4	4	2 5
5	x_5	y_5	z_5	5	3 4
				6	3 5
				7	4 5

Node	Points
1	1
2	2
3	3
4	4
5	5

Triangle	Points
1	1 2 3
2	2 3 5
3	3 4 5



Geometry data structure implementation

Each geometric object (Node, Edge...) has its own class Node, Edge, ...
Let's see some code \implies simplex.hpp

Geometry data structure implementation

Each geometric object (Node, Edge...) has its own class Node, Edge, ...
Let's see some code \Rightarrow simplex.hpp

- One implementation, all possible dimensions
- Only one place to touch when maintenance is needed
- `std::array` (C++11) is a wrapper around C arrays \Rightarrow
 - **exact** memory usage known
 - no memory allocator overhead of `std::vector`
 - safe

They are kept in `Index<T>` which are wrappers around `std::vector<T>` and provide $O(1)$ direct lookups and $O(\log n)$ inverse lookups.

Preventing programmer's mistakes

Let's go back to that strange inheritance...

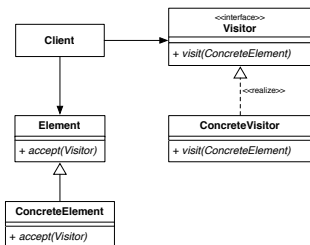
```
1 template<size_t dimension>
2 class Simplex : public IdentifiableObject<Simplex<dimension>>
3
```

Do you remember that each element has its own identifier? How it is possible to be sure that you are not using a triangle index where an edge index is required?

There is a TMP pattern called *Curiously Recurring Template Pattern*: let's see how it allows us making *impossible* to introduce this kind of bug.

Operations on the geometric elements

Operations on the geometric elements are done by means of *Visitors*. A Visitor is a well-known OOP design pattern which allows us to clearly separate *algorithms* from *structures* on which they operate.



There is no structure to traverse, so it is not the classical application of the Visitor, but

- Element “hierarchy” is static and should never require modification
- Adding operations is frequent

The Visitor pattern seems to be a good choice.

Using the visitors

How to get the barycenter of a tetrahedron using a Visitor?

```
1 Tetrahedron t = ges->lookup_tetrahedron(tet_idx);  
2  
3 BarycenterVisitor bv;  
4 t.accept(bv);  
5 Point barycenter = bv.getResult();
```

In the code there are tens of Visitors performing lots of tasks, from the simplest ones (calculation of volumes, barycenters, ...) to the most complex ones (system assembly, field interpolation). It looks complicated, but...

EMT and parallel execution

...an interesting fact about Visitors is that they are designed to *not* share state, so you get parallel execution at no cost (except memory bandwidth):

```
1 std::vector<Point> barycenters;  
2 using PVR = ParallelVisitorRunner<BarycenterVisitor, Tetrahedron>;  
3 PVR pvr(barycenters);  
4 pvr.run();
```

EMT is almost entirely parallel and can use all the cores you throw at it!

Wrapping up Fortran crap...ahem...code!

Historically, Fortran had no pre-processor. So, if you need something that in C++ reads like this \Rightarrow

```
1  template<typename T>
2  T foo(T x)
3  {
4      return x++;
5  }
```

In Fortran you end up having

```
1  FUNCTION foo_f(x)
2      real :: x;
3      foo_f = x+1;
4      RETURN
5  END FUNCTION
```

```
1  FUNCTION foo_d(x)
2      double :: x;
3      foo_d = x+1;
4      RETURN
5  END FUNCTION
```

```
1  FUNCTION foo_c(x)
2      complex :: x;
3      foo_c = x+1;
4      RETURN
5  END FUNCTION
```

This is no joke, but the sad truth! The best numerical solvers around (MUMPS, PARDISO, AGMG) do the nastiest things known to the mankind in order to deal with different data types.

How could C++ help us in isolating that? Let's see the code of the MUMPS wrapper.

Typical objections to C++ in scientific computing

In many years, I heard too many unfounded things about C++. The funniest ones:

- C++ is slower than Fortran
- The level of abstraction of template metaprogramming leads to slow code.
- C++ has no complex numbers
- Memory footprint of C++ programs is larger than Fortran and not controllable
- C++ compilers do a poor job in code optimization
- There are no linear algebra libraries for C++

Myth 1: C++ is slower than Fortran

Fortran makes some assumptions that, in average, allow the code of users unaware of what is going on under the hood run at decent speeds.

Examples:

- Parameter passing: Fortran uses byref, the C++ default is byval. If you want byref you must use `&`. Passing a 100x100 matrix by value, obviously slows down your code a bit!
- Pointer aliasing: Fortran assumes that two pointers point to non-overlapping memory, but C++ assumes that they overlap. If you want to override this, you must use the keyword `restrict`².
- Virtual methods: they have to be correctly dealt with

C++ makes *conservative assumptions* that make your program run safely. So, normally, C++ speed problems are not in the language itself, but between keyboard and chair.

²Only in first approximation: (maybe) C++17 will address the problem correctly.

Myth 2: Template metaprogramming is slow

The quick answer to this objection is *“please go and do some experiments (or read the assembly generated by the compiler)”*.

With template metaprogramming you can give to the compiler lots of useful information about your intentions, so it can make some assumptions that otherwise cannot. Actually, template metaprogramming is a way to make your code run *faster*.

An interesting example comes from some C++ linear algebra libraries: thanks to TMP, these libraries can choose, in a sequence of matrix operations, the order of operations which runs faster. And at *compile time*!

Myth 3: C++ has no complex numbers

Complex numbers are part of the STL.

```
1 #include <complex>
2
3 std::complex<double> z(1,3);
4
```

Thanks to template metaprogramming, complex numbers and their operations are optimized for the particular type on which you instance them.

Myth 4: C++ memory footprint is big and uncontrollable

Well, if you use STL without thinking about what you're doing, it is not too easy to understand how you're using memory...but in C++ memory is controllable at the bit level.

The representation of the geometry in EMT is an example.

Myth 5: C++ compilers do a poor optimization job...

...which in almost all cases translates in “programmers write poor code”.

C++ is the language of choice in every kind of software project, so optimizers inside compilers received lots of attention. Actually there are cases in which they are **really** smart:

<http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array>

Myth 6: There are no linear algebra libraries for C++

I know at least 4 of them:

- Armadillo
- Eigen
- μ blas (in Boost)
- Newmat

Some words about Armadillo

EMT uses Armadillo (<http://arma.sourceforge.net/>).

- It has matlab-like syntax
- It uses automatically BLAS and LAPACK if found
- It is open source
- It is very fast

Conclusion

Maybe this talk was boring for expert C++ programmers, but it was not intended as a showcase of C++ tricks.

The real goal was to show how C++ is a powerful tool also in fields where normally is not considered an option, as in numerical software.

The language, coupled with (minimal) good software engineering practices, allowed me to write from the ground up a software in a domain where my knowledge was close to zero.

C++ helped me in

- organizing the code in a really structured and modular way
- not sacrificing performance, scalability and expandability
- excluding entire classes of stupid but hardly debuggable bugs
- avoiding domino effect when I took the wrong route
- ...

Conclusion

All this doesn't mean that my code is perfect! Some parts of the code could (and should) be rewritten in a better way and there are some modules where code is rotting...but taking care of all the 40000 lines of code requires time!

Thank you!

Thank you for your attention!

I would like to hear your comments, questions and suggestions
`matteo.cicuttin@uniud.it`