Advanced Expression Template Concepts

Christoph Pflaum

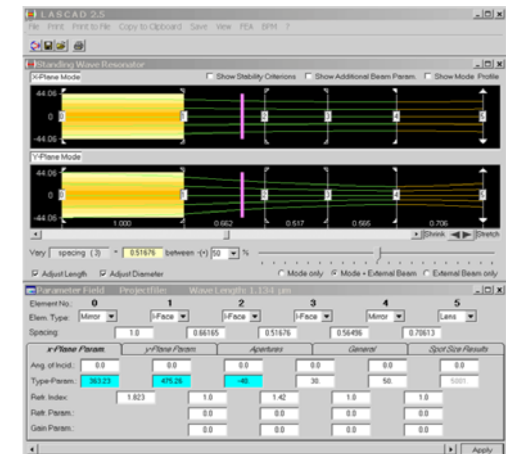
High Performance Computing Group
Graduate School of Advanced Optical Technologies (SAOT)
at the University Erlangen-Nürnberg

Simulation Projects of Optical Waves at the LSS

SAOT

- simulation of laser resonators

  (cooperation with LASCAD, Infineon, …)



- solar cell simulation

  (cooperation with Intitute of Energy Research, Jülich)

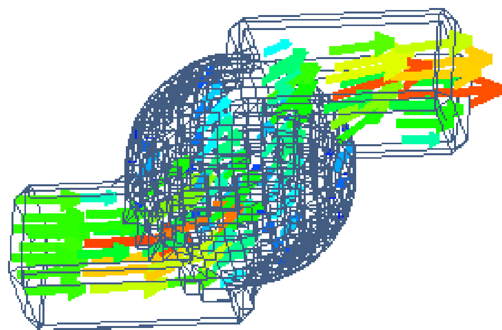
- …

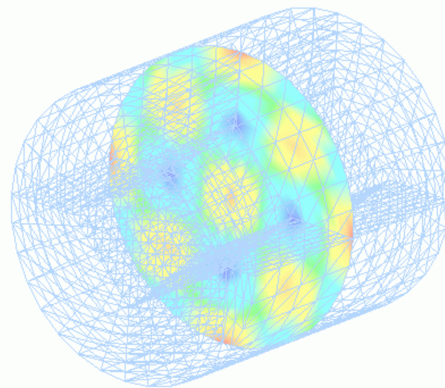
Simulation Software of the HPC Group in Erlangen

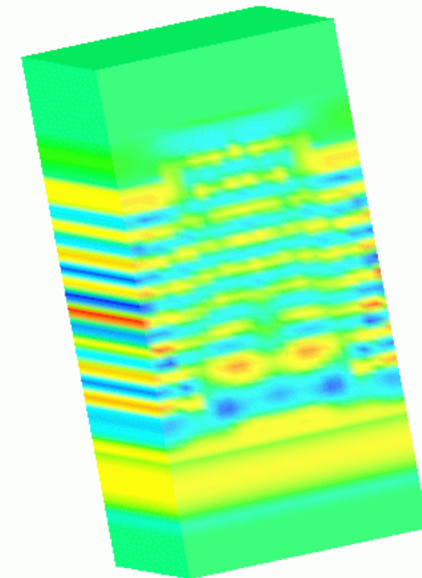
SAOT

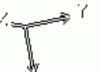
<u>Expression Template Libraries</u>

- FE on block structured grids
- FD on staggered grids
- FE on special semi-unstructured grids
- FE local stiffness matrices









Expression Templates

<u>Aims of Expression Templates:</u>

- Implement algorithms in a language which is close to the mathematical language!
- Obtain optimal efficiency by template constructions!

(Expression templates were invented independent from each other by T. Veldhuizen and D.Vandevoorde .)

Expression Templates

Basic expressions in a mathematical language are:

```
d = a+b;
d = a+b+c;
```

How can these expressions be implemented in an efficient way, if a,b,c,d are large vectors?

SAOT

Easy Expression Templates

Expressions are a

a+b

a+b+b

```
// --------- vector class ----------
class Vector : public Expr<Vector> {
  int n;
  double *data;
public:
 Vector ( … ) { … }
 …
 double operator[](int i) const {
    return data[i]; }
 …
};
```

```
//-- wrapper class --
template <class A>
struct Expr {
  …
};
```

Easy Implementation of Expression Templates

SAOT

Barton-Nackman-Trick:

```
template <class Child>
struct Parent{
  const Child& to() const {
    return static_cast<const Child&> (*this);}
};

class Son : public Parent<Son> …
```

The idea of this trick is to remove the base class **Parent** *by the member function* ***operator*****().**

Easy Implementation of ET with Barton-Nackman Trick

```
// -------- wrapper class -----------
template <class A>
struct Expr {
  operator const A&() const {
    return *static_cast<const A*>(this);
  }
};
```

```
template <class A, class B>
class Add  : public Expr < Add<A,B> > {
  const A& a_;
  const B& b_;
public:
 Add (const A& a, const B& b) : a_(a), b_(b) {}
 double operator[](int i) const {
        return a_[i] + b_[i];  }
};
```

```
class Vector : public Expr<Vector> {
  int n;
  double *data;
public:
 Vector ( … ) { … }
 …
 double operator[](int i) const { return data[i]; }

 template <class A>
 void operator= (const Expr<A> &a_){
     const A& a(a_);
     for(int i=0; i<n; ++i)
       data[i] = a[i];
  }
};
```

Easy Implementation of ET with Barton-Nackman Trick

SAOT

```
template <class A, class B>
Add<A,B>
operator+ (const Expr<A>& a, const Expr<B>& b){
  return Add<A,B> (a,b);
}
```

- *Homework: Implement:*

```
? operator+ (const Expr<A>& a, double x);

? operator* (const Expr<A>& a, double x);

? operator* (double x, const Expr<A>& a);
```

Implementation of the Operator +

The evaluation of an expression

c = a+b+b;

is performed by the operator = :

```
class Vector : public Expr<Vector> {   ...
   public:
     template < class Right_Expression >
     void operator=(const Expr<Right_Expression>& s)  {
       for(int i=0;i<length;++i)
              data[i] = s[i];
     }
};
```

The member functions

inline double operator[int i] const;

have to be defined as inline functions.

Easy Expression Templates

- easy use of expression templates
- extendable expression templates
- short expression template code
- high performence

- We teach expression templates in a bachelor program of computational engineering.

Further ET Concepts

- return type minimization
- specialization of the operator =
  (for GPU’s)
- automatic parallelization with MPI and OpenMP
- fast expression templates
  (for vector machines)
- storage of expression templates
- blocking with ET

“Aliasing” Problems with ET

- Basetti F, Davis K, Quinlan D :
  - C++ Expression Templates Performance Issues in Scientific Computing
- For example,

```
c = a + b * a;
```

```
Expr<Add<Vector,Expr<Mult<Vector,Vector> > > >
```

- should perform like

```
c.data[i] = a.data[i] + b.data[i] * a.data[i];
```

- but reaches the performance of

```
c.data[i] = z.data[i] + y.data[i] * ž.data[i];
```

Fast Expression Templates

SAOT

- Easy Expression Templates:

```
Vector a(n);
Vector b(n);
Vector c(n);

a =  b*b + c;
```

- Fast Expression Templates:

```
FastVector<1> a(n);
FastVector<2> b(n);
FastVector<3> c(n);

a = b*b + c;
```

Fast Expression Templates

SAOT

- <u>Fast Expression Templates:</u>
  - Enumeration of vectors by template integer
  - Declare all data and functions as static
  - Call evaluation on expression types
- <u>Changes for users</u>
  - In some cases: template enumeration of all variables

Fast Expression Templates

SAOT

- Easy Expression Templates:

```
template <class A>
struct Expr {
  operator const A&() const {
    return *static_cast<const A*>(this);
  }
};
```

- Fast Expression Templates:

```
template <class A>
struct FastExpr {};
```

Fast Expression Templates

SAOT

- Easy Expression Templates:

```
template <class A, class B>
Add<A,B>
operator+ (const Expr<A>& a, const Expr<B>& b){
  return Add<A,B> (a,b);
}
```

- Fast Expression Templates:

```
template <class A, class B>
inline FastAdd<A,B>
operator+ (const FastExpr<A>& a, const FastExpr<B> b){
 return FastAdd<A,B>();
}
```

Fast Expression Templates

- Easy Expression Templates:

```
template <class A, class B>
class Add   : public Expr < Add<A,B> > {
  const A& a;
  const B& b;
public:
 Add (const A& a_, const B& b_) : a(a_), (b_) {}
 double operator[](int i) const {
        return a[i] + b[i];   }
};
```

```
template <class A, class B>
class FastAdd   : public FastExpr < Add<A,B> > {
public:
  FastAdd() {}
  static double get(int i) const {
    return A::get(i) + B::get(i);   }
};
```

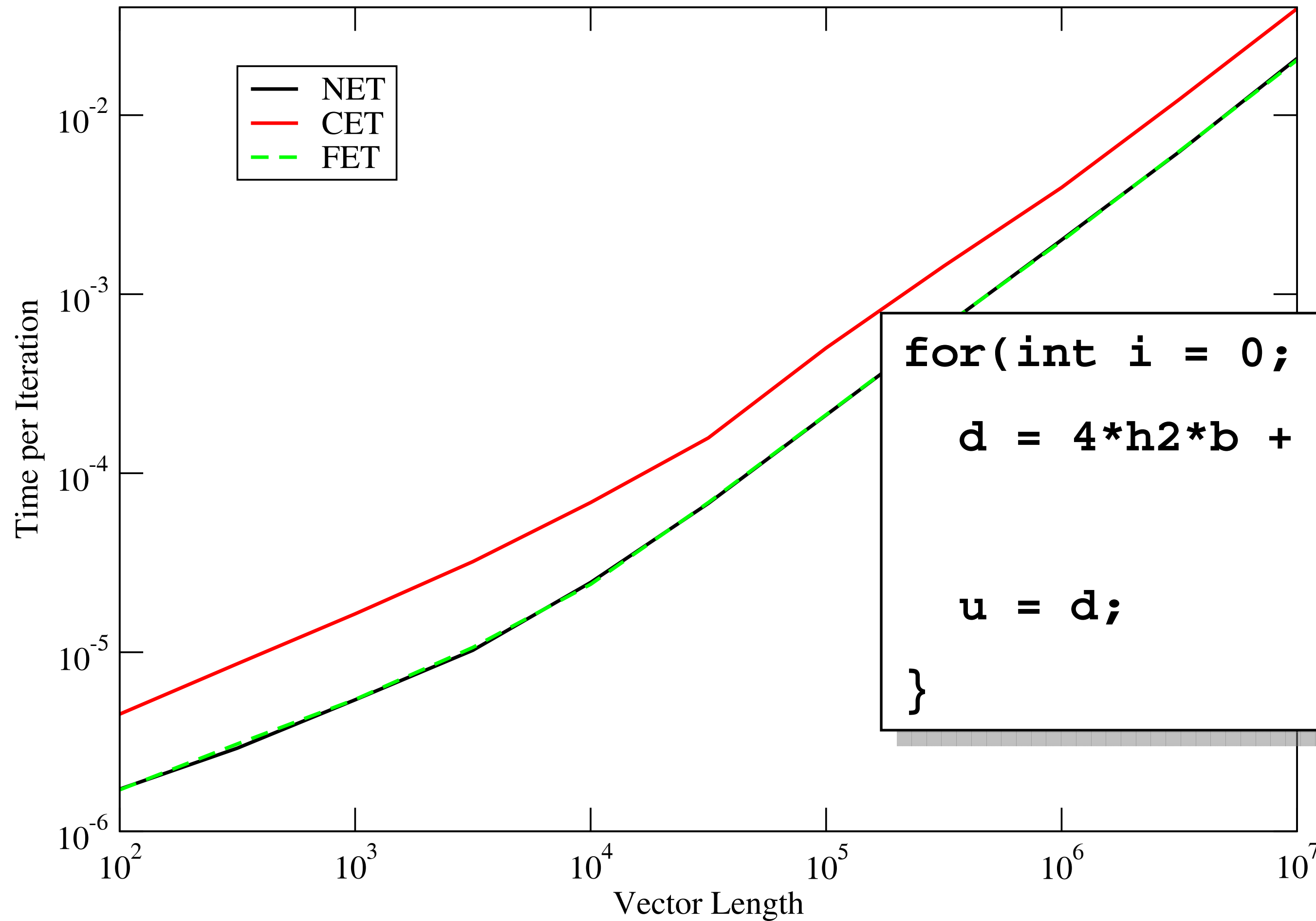
Fast Expression Templates

- Fast Expression Templates:

```
Template<int i>
class FVector : public FastExpr<FVector<id> > {
  int n;
  static double *data;
public:
 Vector ( … ) { … }
 …
 double get(int i) const { return data[i]; }

 template <class A>
 void operator= (const FastExpr<A> &a_){
     for(int i=0; i<n; ++i)
       data[i] = E::get(i);
  }
};

Template<int id> double* FVector<id>::data;
```

NEC SX 6 vector machine :

Differences




```
for(int i = 0; i < iter_max; ++i){
  d = 4*h2*b + North(u) + South(u) +
                 East(u) + West(u);
  u = d;
}
```

SAOT

Parallel Computing

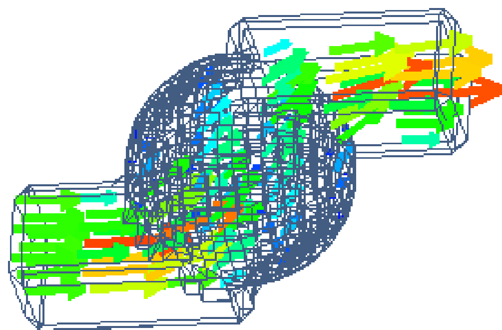
- Vector machines : Fast Expression Templates
- OpenMP Parallel: hide in operator=
- MPI Parallel: hide in operator=
- GPU: specialization of a certain class of expressions??

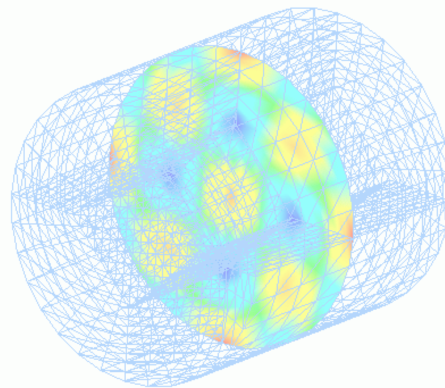
Simulation Software of the HPC Group in Erlangen

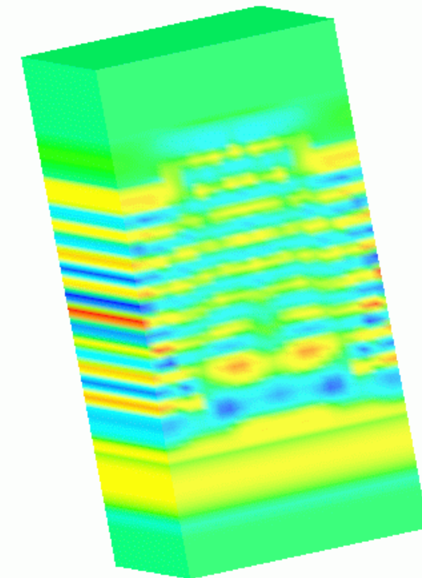
SAOT

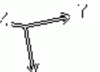
<u>Expression Template Libraries</u>

- FE on block structured grids
- FD on staggered grids
- FE on special semi-unstructured grids
- FE local stiffness matrices









Expression Templates for Finite Elements

- Operators for FE-operators:

```
F = Laplace_FE(U);
F = DX_FE(U);
```

- Elementary operators like scalar product:

```
S = product(u,f);
```

- Combination of geometric and algebraic objects:

```
Boundary_Neumann = …  ;
FE_space = Boundary_Neumann || interior_points;

F = Laplace_FE(U) + k * Int_boundary(U) | space_FE;
```

The last operator corresponds to the bilinear form:

$$a(u,v) = \int_{\Omega} \nabla u \, \nabla v \, dz + k \int_{\Gamma_N} uv \, dz$$

cg - Algorithm

```
u = 0.0 | points_cooling;                       // for Dirichlet-Neumann
u = 0.0 | points_in_space;                      // boundary
f  = f  | points_in_space;                      // conditions
g = Laplace_FE(u) - f;                          //  cg-iteration
d = -g;
delta = product(g,g);
for(int i=1;i<=k;++i) {
   r= Laplace_FE(d);
   tau = delta /product(d,r);
   u = u+tau*d;
   g = g+tau*r;
   delta_prime = product(g,g);
   beta = delta_prime / delta;
   delta = delta_prime;
  d = beta*d-g;
}
```

Parallelization with ET

- Shared memory parallelization:

```
template <class A>
 void Vector::operator= (const Expr<A> &a_){
     const A& a(a_);
     #pragma omp parallel for
     for(int i=0; i<n; ++i)
       data[i] = a.get(i);
  };
```

- MPI parallelization:

  Store update information in every variable.
  Do necessary update of variables, before the evaluation of the expression.

```
F = Laplace_FE(u) + g;
```

Application of ET to Finite Differences

- The finite difference operator

$$f_M = \Delta u(M) = \frac{1}{4}(u_N + u_S + u_E + u_W - 4u_M)$$

- is implemented as follows:

```
F = 0.25 * (N(U)+S(U)+E(U)+W(U)-4.0*M(u));
```

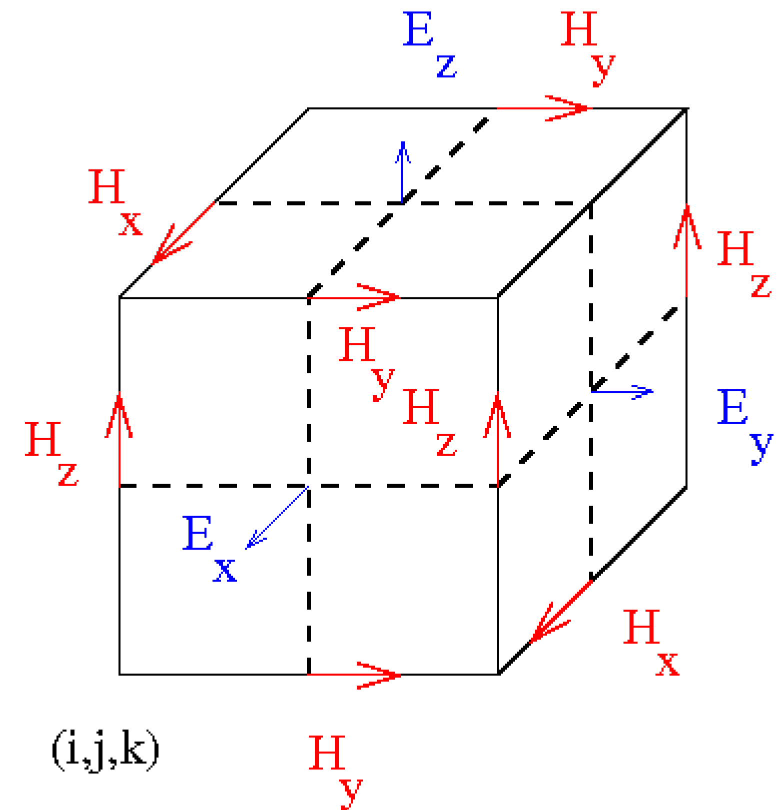
FDTD for Maxwell´s Equations

- The finite difference equation

$$E_x^{t_{n+1/2}}(M) = E_x^{t_{n-1/2}}(M) \quad + \frac{\tau}{\varepsilon h}\left(H_z^{t_n}(N) - H_z^{t_n}(S) - H_y^{t_n}(T) + H_y^{t_n}(D)\right)$$

- is implemented as follows:

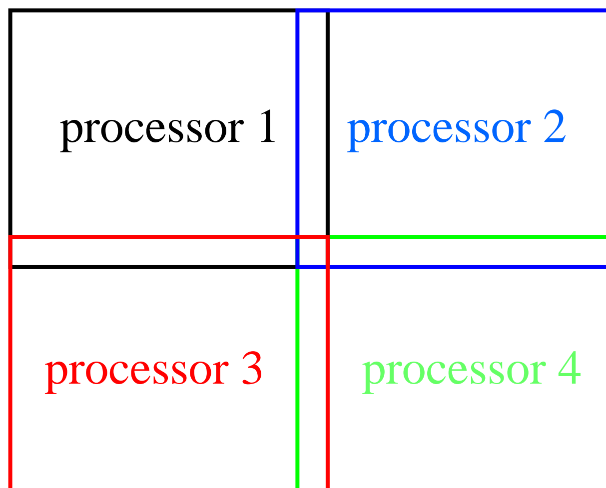
```
Variable<Dcomplex, not_staggered,
                   staggered,
                   staggered>    Ex(grid);
Variable<Dcomplex, not_staggered,
                   staggered,
               not_staggered>   Hy(grid);
Variable<Dcomplex, not_staggered,
               not_staggered,
                    staggered>  Hz(grid);

    Ex_new = Ex + tau / eps / h *
             ( N(Hz) – S(Hz) - T(Hy) + D(Hy) );
```



Parallel Update with ET

```
template <class A>
      NShift<A> N (const Expr<A>& a) {
        return NShift<A> (a);
      }
```

```
 Variable u, w, r;

u = N(w) + S(r) ;
```




```
class Variable … {
  ........
   void Update(int shift_x,
                     int shift_y, int shift_z)  {
       … };  //MPI-update
   private:
     bool  update_x;
     bool  update_y;
     bool  update_z;

          template <class A>
          void operator= (const Expr<A> &a_){
          const A& a(a_);

          a.Update(0,0,0);

          for(int i=0; i<n; ++i)
            data[i] = a[i];
       update_x = update_y =  update_z = false;
        }
};
```

FDTD for Maxwell´s Equations

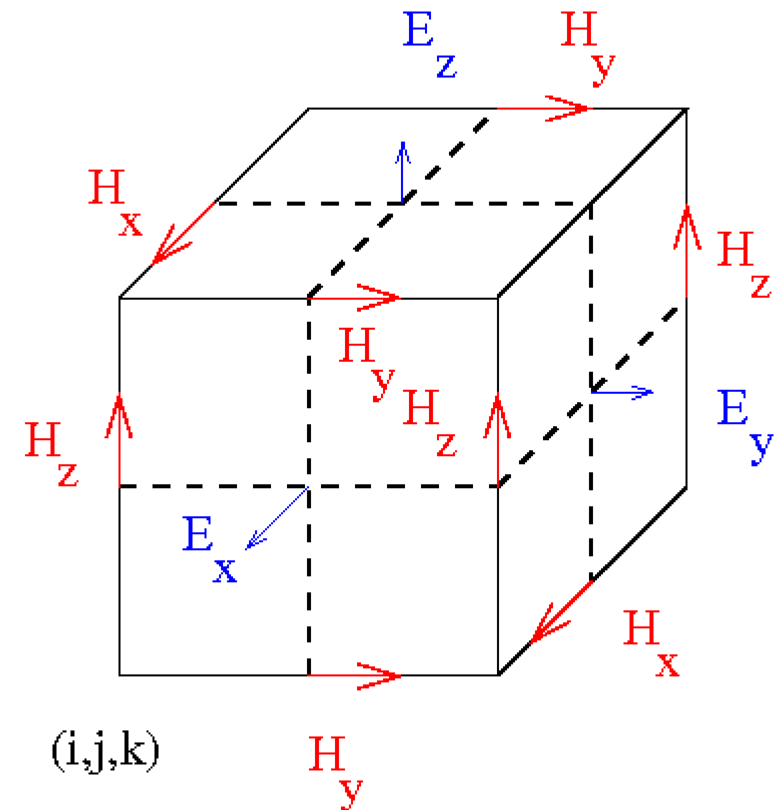
- The equation

$$\frac{dE}{dt}\varepsilon = \nabla \times H$$

- is discretized as follows:

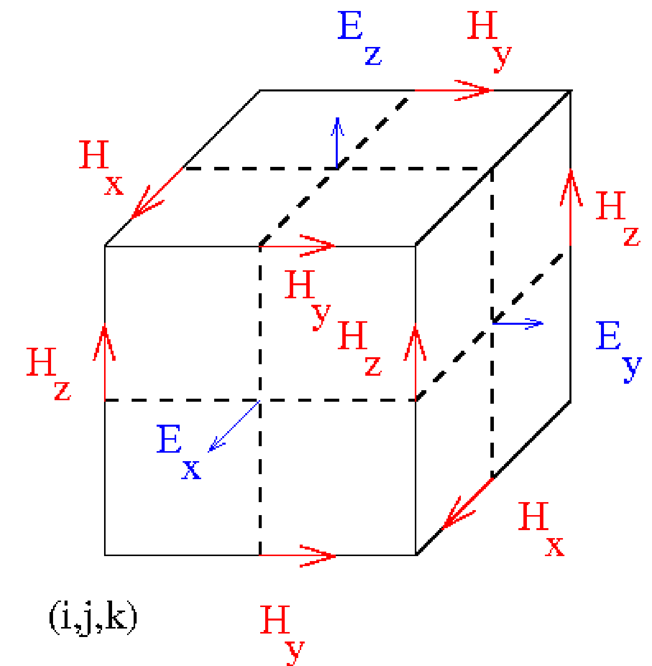
$$\frac{E_x^{t_{n+1/2}}(M) - E_x^{t_{n-1/2}}(M)}{\tau}\varepsilon =$$

$$\frac{H_z^{t_n}(N) - H_z^{t_n}(S)}{h} - \frac{H_y^{t_n}(T) - H_y^{t_n}(D)}{h}$$



Discretization of Maxwell´s Equations

- The finite integration method by Weiland
- Incoming wave
- PML (absorbing boundary conditions)
- Mesh moving
- MPI and OpenMP-Parallel
- Graphical Processor Units (GPU)?
- Accurate approximation of material properties



Automatic Parallelization with ET

- The following code runs in serial and parallel in EXPDE:

```
u = 0.0 | points_cooling;                  // for Dirichlet-Neumann
u = 0.0 | points_in_space;                 // boundary
f   = f    | points_in_space;              // conditions
g = Laplace_FE(u) - f;                     //  cg-iteration
d = -g;
delta = product(g,g);
for(int i=1;i<=k;++i) {
   r= Laplace_FE(d);
   tau = delta /product(d,r);
   u = u+tau*d;
   g = g+tau*r;
   delta_prime = product(g,g);
   beta = delta_prime / delta;
   delta = delta_prime;
  d = beta*d-g;
}
```

- Every variable stores an update information.
- A “parallel update” is performed of the expression requires an update.

Application of ET in Colsamm

SAOT

Colsamm is a library for the calculation of local stiffness matrices.

In the library Colsamm expression templates are used at three stages:

- Description of basis functions on reference element (linear, quadratic, …)
- Description of the mapping from the reference element to the element

  (linear mapping or isoparametric mapping, …).
- Description of the bilinear form of the weak equation.

ET 1 in Colsamm

- Mapping of the reference element to the FE element

```
Define_Element_Transformation
    (
      P_0() +
     (P_1() - P_0()) * _U() +
     (P_2() - P_0()) * _V() +
     (P_3() - P_0()) * _W()
    )
     Tetrahedron_Transformation;
```

Here "**Tetrahedron_Transformation**" contains the type of the expression template. In this construction we apply "fast expression templates".

- Isoparametric transformations are possible.

Finite Element Definition in Colsamm

- Linear elements on a thetrahedron:

```
struct _Tetrahedron_linear
  : public _Domain_<4,D3,
                     Tetrahedron_Transformation,
                     interior,
                     4,0,
                     tetrahedron,
                     Gauss2,
                     double,
                     STLVector>
```

- One can define
  - edge elements (for Maxwell's equations),
  - high order elements and
  - mixed finite elements (for Stokes).

ET 2 in Colsamm

Description of basis functions on a reference element:

- Linear elements on a triangle:

```
_Tetrahedron_linear( )
{
  this->Set( 1. - X_() - Y_() - Z() );
  this->Set( X_() );
  this->Set( Y_() );
  this->Set( Z_() );
};
```

- Quadratic elements on triangle:

```
_Quadrangle_bilinear( )
{
  this->Set( 4*( 1. - X_() - Y_() ) * X_() );
  this->Set( 4*  X_() * Y_() );
  …
};
```

ET 3 in Colsamm

SAOT

Definition of the bilinear form.

- The bilinear form corresponding to Poisson's equation is:

$$a(v, w) = \int_{\Omega} \nabla v \, \nabla w \, dz$$

- Using expression templates this is implemented as follows:

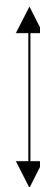
```
Triangle my_element;
std::vector <double> corners(6,0.);
std::vector < std::vector < double > > stencil_a;
…    // set corners of the triangle
my_element( corners );

stencil_a =
  my_element.integrate(grad(v()) * grad(w_()));
```

Interface in a Code for the Numerical solution of PDE`s

SAOT

Library for PDE solvers

- Discretization grid
- Solver for algebraic equations
- Operators
- Parallel



Interface ??

Complex application code:

- Written in a mathematical language

Applications of ET in PDE Codes

SAOT

The expression template concepts have to depend on the discretization and the type of the discretization grid:

- FD discretization: ET and operators for: N,S, …
  (library POOMA, Blitz++),
- FD discretization: library on staggered grids.
- FE: discretization: apply ET to the calculation of local stiffness matrices
  (Pietro and Veneziani , Joachim Härdtlein: Colsamm)
- FE: discretization: apply ET to FE-operators, restriction and prolongation operators
- FV: discretization ???

Further Work

- unstructured grids
- Graphical Processor Units (GPU)?