ASC Report No. 30/2014

C++11 Implementation of Finite Elements in NGSolve

J. Schöberl

Institute for Analysis and Scientific Computing —
Vienna University of Technology — TU Wien
www.asc.tuwien.ac.at ISBN 978-3-902627-05-6

Most recent ASC Reports

29/2014 *A. Arnold and J. Erb*
Sharp entropy decay for hypocoercive and non-symmetric Fokker-Planck equations with linear drift

28/2014 *G. Kitzler and J. Schöberl*
A high order space momentum discontinuous Galerkin method for the Boltzmann equation

27/2014 *W. Auzinger, T. Kassebacher, O. Koch, and M. Thalhammer*
Adaptive splitting methods for nonlinear Schrödinger equations in the semiclassical regime

26/2014 *W. Auzinger, R. Stolyarchuk, and M. Tutz*
Defect correction methods, classic and new (in Ukrainian)

25/2014 *J.M. Melenk and T.P. Wihler*
A posteriori error analysis of hp-FEM for singularly perturbed problems

24/2014 *J.M. Melenk and C. Xenophontos*
Robust exponential convergence of hp-FEM in balanced norms for singularly perturbed reaction-diffusion equations

23/2014 *M. Feischl, G. Gantner, and D. Praetorius*
Reliable and efficient a posteriori error estimation for adaptive IGA boundary element methods for weakly-singular integral equations

22/2014 *W. Auzinger, O. Koch, and M. Thalhammer*
Defect-based local error estimators for high-order splitting methods involving three linear operators

21/2014 *A. Jüngel and N. Zamponi*
Boundedness of weak solutions to cross-diffusion systems from population dynamics

20/2014 *A. Jüngel*
The boundedness-by-entropy principle for cross-diffusion systems

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8–10
1040 Wien, Austria

E-Mail: admin@asc.tuwien.ac.at
WWW: http://www.asc.tuwien.ac.at
FAX: +43-1-58801-10196

ISBN 978-3-902627-05-6

ASC TU WIEN

C++11 Implementation of Finite Elements in NGSolve

Joachim Schöberl

September 26, 2014

Abstract

We discuss an object oriented design of finite element core functionality. It allows to separate the mathematical definition of the finite element basis functions, the efficient implementation of operations, and the calculation of stiffness matrices and residual vectors.

We show how features of the C++11 programming language help to reduce code complexity and thus allow for additional performance optimization such as vectorization.

The presented techniques are implemented in the open source finite element package NGSolve.

1 Introduction

The finite element method is the major numerical method for the numerical approximation of partial differential equations [6]. While continuous finite element methods fit well for elliptic and parabolic equations, discontinuous Galerkin methods are particularly efficient for hyperbolic equations [12]. The combination of local mesh refinement and variable distribution of polynomial orders lead to highly accurate hp-finite element methods, see [29, 27, 14, 8, 12].

Efficient coding of hp-FEM is a challenging endeavor. Here, the sophisticated use of modern programming languages such as C++ helps a lot. It allows to express the mathematical concepts directly in computer programs. Since its beginning, the C++ programming language allows to combine high performance computing with object oriented design. In particular, template-based compile time polymorphism [30, 31, 3] allows generic programming without performance penalties. Recent C++ language developments have been combined in the new standard C++11. It is included in the latest version of Stroustrup's textbook [28]. Citing Stroustrup from http://www.stroustrup.com/C++11FAQ.html: *Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.*

There are several widely used open source finite element packages available: DUNE [4] with its module DUNE-FEM [7], deal.II [2], Life [23] which evolved into Feel++ [24], Elmer [19], FEMSTER [5], Freefem++ [11], 3Dhp[9]. The techniques presented in the current work are implemented in the finite element library NGSolve [25], version 5.3. We mention the special purpose compilers FIAT [15] and the framework FEniCS [18], which produce machine code directly from the mathematical formulation of finite elements.

This paper discusses the core functionality of finite element routines, namely the evaluation of and operations with shape functions and its derivatives. The element matrix and element vector calculation routines have to compute all basis functions. When computing residuals of non-linear problems, evaluating operators for instationary equations, or even when solving linear problems in a matrix free way, one needs the evaluation of finite element functions in

```
class FiniteElement {
  int ndof;     // number of basis functions
  int order;    // highest polynomial order
public:
  int GetOrder() { return order; }
  int GetNDof() { return ndof; }
  virtual ELEMENT_TYPE GetElementType() = 0;
  virtual string GetElementName() = 0;
};
```

Listing 1: class FiniteElement

the integration points. In addition, gradients are needed, and the transpose operations. We aim in optimizing these functions, while keeping the code structure transparent.

One new key feature of C++11 are lambda functions, which are used here as follows: The specific finite element class implements shape functions. The functionality layer classes implement what to do with these shape funcitons. These operations are specified as lambda functions, and are applied to the shape functions. Mathematically speaking, a lambda function is an element of the dual space. The techniques presented in the current work have been implemented within the standard C++ programming language using classes before. But, the new C++11 syntax allowed the simplification of a large portion of the code. The C++11 syntax is supported by the major compilers at time of writing this work.

Modern microprocessors can perform several operations of the same type simultaneously (SIMD - single instruction multiple data), even within one core. Now very popular and low-cost general purpose GPU devices can even compute with typically 32 synchronous threads per processor core. These processor architecture is very well suited for performing equivalent operations in many integration points. But, while computing power is increasing very fast, the access to memory, and even caches, becomes more and more a bottleneck. The GPU cores even have only a small number of registers as fast local memory. Thus, it is a primary goal to eliminate the need of temporary memory.

2 Finite elements and element matrix integration

The NGSolve finite element class hierarchy is drawn in Figure 1. The base class for all finite elements is `FiniteElement`, see Listing 1. It provides the common functionality of all finite elements. It stores the number of basis functions, and the maximal polynomial degree as class members and provides query functions to them. In addition, it provides a query to the element-type (`ET_QUAD, ET_HEX` etc.) as well as the name of the element which is useful for debugging.

The next level in the class hierarchy specifies the type of the finite element, and the space dimension via a template argument. For second order elliptic equations we need continuous, scalar valued basis functions. The element provides the basis functions on the reference element, and also the gradients of them. Other types of elements are vectorial finite elements, which are conforming in $H(\mathrm{curl})$ or $H(\mathrm{div})$, and are needed for solving electromagnetic field problems and some flow models. These elements must compute the vector-valued shape

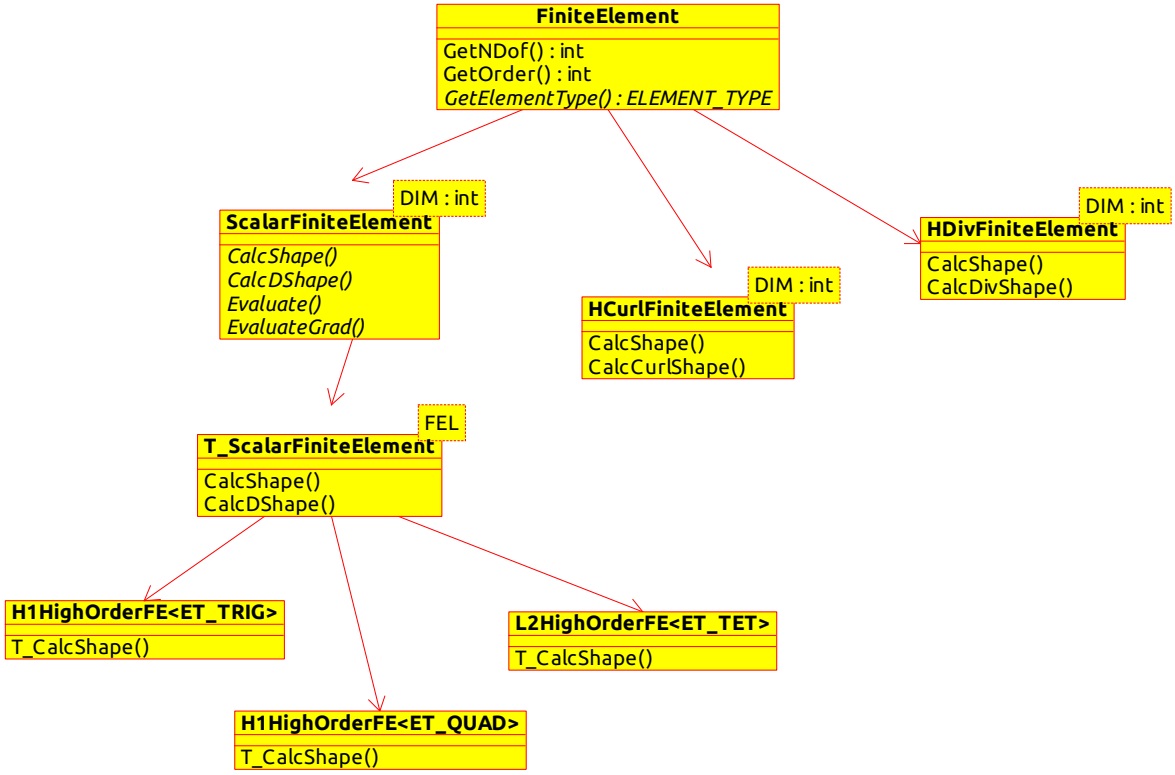

Figure 1: *FiniteElement class-diagram* The root is the common base class `FiniteElement`. The user works with the second level, for example a three-dimensional $H(curl)$ finite element `HCurlFiniteElement<3>`. The third level is the implementation of functionality, including special hardware tuning. The fourth level implements the specific finite element shape functions.

```
template <int DIM>
class ScalarFiniteElement : public FiniteElement {
public:
  virtual void CalcShape (const IntegrationPoint & ip,
                          FlatVector<double> shape) = 0;
  virtual void CalcDShape (const IntegrationPoint & ip,
                           FlatMatrixFixWidth<DIM,double> dshape)=0;
  ...
};
```

Listing 2: ScalarFiniteElement

functions, and the curl or divergence of them.

Element matrix calculation and similar functions work with the abstract intermediate class `ScalarFiniteElement<DIM>` (see Listing 2) without knowing the particular finite element class provided. Here, we use the traditional C++ run-time polymorphism via virtual functions. The `FlatVector` template class is a cheap implementation of a vector in the sense of linear algebra. It only contains the vector-size and the data pointer. The constructor just copies the pointer (and no data), the destructor does not free any memory. Thus, a call by value argument is efficient for a FlatVector, and similar for the `FlatMatrixFixWidth`, a matrix with fixed width. Fixing the width at compile-time helps the compiler optimizing the index calculation.

A simple implementation of element matrix calculation for the Laplace operator is given in Listing 3. The input is a finite element object, and the transformation from reference element to physical element. It computes the element matrix, which is already allocated by the caller. The *LocalHeap* is a class providing fast allocation of temporary small memory blocks. The whole memory is freed after finishing each element.

Since the common argument of the virtual function `CalcElementMatrix` must be the base class `FiniteElement`, and we know we are dealing with scalar elements in `DIM` dimensions, we can cast up to that class. The `IntegrationRule` provides a numerical integration formula on the reference element of given geometry and order. The bracket operator applied to it delivers an integration-point, which contains the coordinates on the reference-element as well as the integration weight. In the loop over integration points we compute the mapped integration point and the Jacobian from the abstract element transformation. Next we compute the matrix of shape function gradients on the reference element. The push-forward is computed by a matrix-matrix product with the inverse Jacobian. Finally, we update the matrix by a matrix-matrix product. The linear algebra is based on an own implementation of expression templates [30, 31, 10]. The dominant costs are the last matrix-matrix product. It is reduced by unrolling the integration loop and combining a few integration points such that the inner dimension of the matrix-matrix product increases.

The `FESpace` is the instance generating the specific finite element object, see Listing 4. It has access to the mesh data structure. A particular class derived from `FESpace` is `H1HighOrderFESpace`, which provides an arbitrary order finite element sub-space of the Sobolev-Space H^1. The virtual function `GetFE` allocates an object of the specific H^1-conforming finite element of given order. For performance reasons, the allocation is on the LocalHeap. Global element vertex numbers are provided to the element for consistent ori-

```
template <int DIM> void LaplaceIntegrator ::
CalcElementMatrix (const FiniteElement & base_fel,
                   const ElementTransformation & trafo,
                   Matrix<> & elmat, LocalHeap & lh)
{
  const ScalarFiniteElement<DIM> & fel =
    static_cast<const ScalarFiniteElement<DIM>&> (base_fel);
  IntegrationRule ir (fel.GetElementType(), 2*fel.GetOrder());
  MatrixFixWidth<DIM,double> dshape(fel.GetNDof(),lh);
  MatrixFixWidth<DIM,double> dshape_ref(fel.GetNDof(),lh);

  elmat = 0.0;
  for (int i = 0; i < ir.Size(); i++)
    {
      MappedIntegrationPoint<DIM,DIM> mip(ir[i], trafo);
      fel.CalcDShape (ir[i], dshape_ref);
      dshape = dhape_ref * mip.GetJacobianInverse();
      double factor = ir[i].Weight()*mip.GetMeasure();
      elmat += fac * dshape*Trans(dshape);
    }
}
```

Listing 3: Element matrix calculation

entation of edge- and face-bubble functions. The template class `H1HighOrderFE` is a derived class from `ScalarFiniteElement`.

3 Element functionality

The template class `H1HighOrderFE<ELEMENT_TYPE>` implements shape functions for the particular element geometries. The `CalcShape` method computes all shape functions in a given point, and `CalcDShape` computes the matrix of shape function gradients. Additional functions help to speed up certain finite element computations.

Our approach is to implement just one template function `T_CalcShape` which is able to compute shapes, gradients, etc. This is obtained by keeping the variable types generic. Substituting the template arguments defines the particular operation. The operation is provided by the intermediate class `T_ScalarFiniteElement` in Listing 7.

Finite elements for H^1 require basis functions connected with vertices, edges, faces (3D only), and cells. Typically, these blocks are built from Legendre and Jacobi orthogonal polynomials. An L_2-orthogonal basis on simplices is provided via the Dubiner-Basis [14]

Basis functions of total order p for the triangle are given in terms of barycentric coordinates λ:

$$\begin{array}{lrcll}
\text{Vertex basis:} & \varphi_V & = & \lambda_V & \\
\text{Edge basis:} & \varphi_{E,i} & = & \lambda_{E_1}\lambda_{E_2} P_i^S(\lambda_{E_1}-\lambda_{E_2},\lambda_{E_1}+\lambda_{E_2}) & 0 \le i \le p-2 \\
\text{Cell basis:} & \varphi_{T,ij} & = & \lambda_1\lambda_2\lambda_3 P_i^S(\lambda_1-\lambda_2,\lambda_1+\lambda_2) P_j^{(0,2i+1)}(2\lambda_3-1) & i+j \le p-3
\end{array}$$

```
FiniteElement & H1HighOrderFESpace::GetFE(int elnr, LocalHeap & lh)
{
  ELEMENT_TYPE et = mesh.GetElementType (elnr);
  Array<int> vnums = mesh.GetElementVertices (elnr);

  switch (et) {
    case ET_TRIG:
      return new (lh) H1HighOrderFE<ET_TRIG> (order, vnums);
    case ET_QUAD:
      return new (lh) H1HighOrderFE<ET_QUAD> (order, vnums);
    case ET_TET:
      return new (lh) H1HighOrderFE<ET_TET> (order, vnums);
    ...
  }
}
```

Listing 4: Element construction

```
template <ELEMENT_TYPE ET>
class H1HighOrderFE : public T_ScalarFiniteElement<H1HighOrderFE<ET>>
{
  int vnums[ElementTrait<ET>::N_VERTEX];
public:
  H1HighOrderFE (int o, FlatArray<int> v)
  { vnums = ...; order = o; ndof = ... }

  template <typename T, typename TSHAPE>
  T_CalcShape (T x[], TSHAPE shape) const;
}
```

Listing 5: High order finite element template

Here, the scaled Legendre polynomial is defined as

$$P_i^S(x,t) = P_i(x/t)t^i.$$

It is a bivariate polynomial of total polynomial degree i, and can be evaluated by a very similar three-term recurrence as the usual Legendre Polynomial [26, 32].

Orthogonal polynomial functions in NGSolve evaluate all polynomials up to the given order `n` in one point `x`. The result is stored in the `values` array:

```
template <typename T, typename TVAL>
void LegendrePolynomial::Eval (int n, T x, TVAL && values);
```

Note, the `values` array is defined as a C++11 right-value object which allows to provide automatic objects. Its elements `values[i]` must be left-values. Multiplying all polynomials by a factor c can be optimized by multiplying just the two initial polynomials by the factor, the three-term recurrence delivers the multiplied polynomials for all other orders. This is implemented in:

```
template <typename T, typename Tc, typename TVAL>
void LegendrePolynomial::EvalMult(int n, T x, Tc c, TVAL && values);
```

Note that without this function one had to compute the usual polynomials first, store them, and multiply all polynomials later. This additional function eliminates the need for the local temporary memory.

The implementation of the basis functions for the H^1-conforming triangular element is given in Listing 6. It allows also to give individual polynomial orders for edges and the interior. The `GetEdgeSort` method gives the local vertex numbers of the i^{th} edge such that `vnums[e[0]] < vnums[e[1]]`. This ensures a consistent parametrization of the edge across neighboring elements. We note that the generic object `shape` is supposed to behave like a simple C - array. It needs the bracket access operator for its elements, and the usual pointer arithmetic *pointer+int* to give an offset. This allows to store the orthogonal polynomials for edges and the cell directly in the corresponding sub-arrays.

We repeat that we only have to implement this one function defining the basis-functions, and use C++ template mechanism to generate code for computing gradients, and other operations.

The actual operations are triggered from the intermediate class `T_ScalarFiniteElement`. We use the Barton-Nackman trick: The specific element is a derived class, which hands over itself as a template argument (`FEL`) to its base class. To call functions of the specific class, the intermediate class can static up-cast itself to `FEL`, i.e. the derived class. This kind of compile-time polymorphism avoids the performance penalty of virtual functions, and allows aggressive inlining of the compiler.

The gradients are calculated by automatic differentiation implemented via the class `AutoDiff`. In contrast to numerical differentiation, automatic differentiation calculates exact derivatives. Precisely, we use the forward differentiation technique based on operator overloading. An object of type `AutoDiff<DIM>` stores a value, and `DIM` partial derivatives. Adding two AutoDiff variables (via the overloaded +-operator), just adds values and derivatives. Multiplication multiplies values, and implements the product rule, i.e.

$$product.deriv[i] = a.value * b.deriv[i] + b.value * a.deriv[i].$$

The `AutoDiff` class has two constructors:

```
template<> template<typename T, typename TSHAPE>
void H1HighOrderFE_Shape<ET_TRIG> ::
T_CalcShape (T x[], TSHAPE & shape) const
{
  // barycentric coordinates
  T lam[3] = { x[0], x[1], 1-x[0]-x[1] };

  // vertex-based basis functions
  for (int i = 0; i < N_VERTEX; i++) shape[i] = lam[i];

  int ii = N_VERTEX;
  // edge-based basis functions
  for (int i = 0; i < N_EDGE; i++)
    if (order_edge[i] >= 2)
      {
        INT<2> e = GetEdgeSort (i, vnums);
        LegendrePolynomial::
          EvalScaledMult (order_edge[i]-2,
                          lam[e[1]]-lam[e[0]], lam[e[0]]+lam[e[1]],
                          lam[e[0]]*lam[e[1]], shape+ii);
        ii += order_edge[i]-1;
      }

  // cell-based basis functions
  if (order_face[0] >= 3)
    {
      INT<3> f = GetFaceSort (0, vnums);
      DubinerBasis::EvalMult (order_face[0]-3,
                              lam[f[0]], lam[f[1]],
                              lam[f[0]]*lam[f[1]]*lam[f[2]],
                              shape+ii);
    }
}
```

Listing 6: High order triangular finite element

```
template <typename FEL, ELEMENT_TYPE ET>
class T_ScalarFiniteElement :
    public ScalarFiniteElement<ElementTrait<ET>::DIM>
{
  enum { DIM = ElementTrait<ET>::DIM };
public:
  virtual void CalcShape (const IntegrationPoint & ip,
                          FlatVector<double> shape)
  {
    static_cast<const FEL*> (this) -> T_CalcShape (ip, shape);
  }

  virtual void CalcDShape (const IntegrationPoint & ip,
                           FlatMatrixFixWidth<DIM,double> dshape)
  {
    AutoDiff<DIM> adp[DIM];
    for (int i = 0; i < DIM; i++)
      adp[i] = AutoDiff<DIM> (ip(i), i);

    Array<AutoDiff<DIM>> adshape(ndof);

    static_cast<const FEL*> (this) -> T_CalcShape (adp, adshape);

    for (int i = 0; i < ndof; i++)
      adshape[i].StoreGradient (dshape.Row(i));
  }
};
```

Listing 7: T_ScalarfiniteElement using Barton-Nackman trick

```
virtual void CalcDShape (const IntegrationPoint & ip,
                         FlatMatrixFixWidth<DIM,double> dshape)
{
  AutoDiff<DIM> adp[DIM];
  for (int i = 0; i < DIM; i++)
    adp[i] = AutoDiff<DIM> (ip(i), i);

  static_cast<const FEL*> (this) ->
    T_CalcShape (adp, [&](int i, AutoDiff<DIM> val)
                 { val.StoreGradient(dshape.Row(i)); });
}
```

Listing 8: Lambda function

```
  AutoDiff (double val);
  AutoDiff (double val, int i);
```

The first one assigns the value, and zeros the gradient. The second one assigns the value, and sets the gradient to the i^{th} unit-vector.

The CalcDShape method in Listing 7 uses DIM AufoDiff variables for the coordinates. The first one is initialized with the x-component with the 0^{th} unit-vector as gradient. The second one as y-coordinate with the 1^{st} unit-vector as gradient, etc. The generic `T_CalcShape` template method is instantiated with `AutoDiff` variables. This type propagates down through the evaluation of orthogonal polynomials. Whenever a basis-function is calculated and stored in the result array, it contains the value in the point as well as the gradient of the basis function in the point.

One drawback of the CalcDShape method in Listing 7 is the need of temporary memory for storing the full AutoDiff shape function values, while only the gradient part is needed as output. This temporary memory will be eliminated by the lambda-function technique.

4 Lambda functions for finite element operations

The implementation of the finite element classes computes basis functions, and stores the result in the generic shape array, e.g.

```
shape[i] = lam[i];
```

We now develop a mechanism to redefine the array member write access to our needs. The particular operation is provided by a lambda-function. Assume for a moment, we assign shape functions via a 2-parameter function call

```
shape(i, lam[i]);
```

instead of the more intuitive array element assignment. This is now a function call to the `shape` object. The C++11 lambda-function syntax allows to define a function (the lambda-function) directly as a function call argument, see Listing 8.

The new syntax deserves some explanation. The lambda functions is started by the (new C++11) scope operator `[&]`, which allows to access all variables in the enclosing scope from the lambda function. The arguments come next, in our case the number of the shape function

```
template <typename FUNC>
class SBLambdaElement {
  FUNC f;
  int i;
public:
  SBLambdaElement (FUNC af, int hi) : f(af), i(hi) { ; }
  template <typename VAL>
  VAL operator= (VAL v) { f(i, v); return v; }
};

template <typename FUNC>
class Class_SBLambda {
  FUNC func;
  int offset;
public:
  Class_SBLambda (FUNC f, int ao = 0) : func(f), offset(ao) { ; }
  SBLambdaElement<FUNC> operator[] (int i) const
    { return SBLambdaElement<FUNC> (func, offset+i); }
  Class_SBLambda<FUNC> operator+ (int i) const
    { return Class_SBLambda<FUNC> (func, offset+i); }
};

template <typename FUNC>
inline Class_SBLambda<FUNC> SBLambda (FUNC f)
  { return Class_SBLambda<FUNC> (f); }
```

Listing 9: Square bracket lambda function wrapper

and its value. Then, the function body is directly implemented in the calling arguments. In our case it is very simple, just store the gradient part of the i^{th} `AutoDiff` shape-function in the output matrix. Here, no additional temporary array for the `AutoDiff`-shapes is required.

Since all function bodies are visible by the compiler, and the lambda-function is very simple, the compiler certainly decides to inline the lambda-function. Thus, no function call penalty occurs.

We prefer to use the array-element assignment in the finite element methods, instead of the 2-argument function call. This is realized by a square-bracket lambda wrapper `SBLambda` in Listing 9. The `SBLambda` class declares pointer-arithmetic, i.e. the bracket access operator and the pointer offset operator. The access generates an automatic variable representing the array element, which finally calls the provided two-argument function.

```
static_cast<const FEL*> (this) ->
  T_CalcShape (adp, SBLambda ([&] (int i, AutoDiff<DIM> val)
                    { val.StoreGradient (dshape.Row(i)); }));
```

Now it is very simple to add additional finite element functions. For example we want to evaluate the finite element function in some point, represented as `IntegrationPoint` object.

```
double T_ScalarFiniteElement ::
Evaluate (IntegrationPoint & ip, FlatVector<double> coefs)
{
  Vector<double> temp(ndof);
  CalcShape (ip, temp);
  return InnerProduct (temp, coefs);
}
```

Listing 10: Traditional evaluation using temporary memory

```
double T_ScalarFiniteElement ::
Evaluate (IntegrationPoint & ip, FlatVector<double> coefs)
{
  double sum = 0;
  static_cast<const FEL*> (this) ->
    T_CalcShape (ip, SBLambda ([&](int i, double val)
                        { sum += coefs(i) * val;}));
  return sum;
}
```

Listing 11: Evaluation using lambda functions

The common way is to calculate the shape functions in the point (and store them in some temporary memory), and form the inner product with the coefficient vector, see Listing 10. Again, the drawback is the need of temporary memory.

By means of the lambda-functions, we can immediately update the sum of the inner product as soon as a shape function is available. This avoids to move the whole shape function vector to memory. An intuitive implementation is given in Listing 11. Note, adding this one short function immediately adds the functionality for all elements derived from `T_ScalarFiniteElement`.

5 Vectorial finite elements

The discretization of Maxwell's equations requires $H(\mathrm{curl})$ conforming Nédélec elements [21, 20]. The classical approach is to define a polynomial space, and the degrees of freedom as linear functions. The construction of Demkowicz et al [9] relies on projection based interpolation procedures, which is also reflected in the implementation. The approach from [1, 26, 32] directly defines the high order basis functions for the Nédélec space. In [32], basis functions for triangles, quadrilaterals, tetrahedral, prismatic and hexahedral elements are given.

C++ object	basis function	curl of basis function
`Du (u)`	$\varphi = \nabla u$	0
`uDv_minus_vDu (u,v)`	$\varphi = u\nabla v - v\nabla u$	$2\,\nabla u \times \nabla v$
`wuDv_minus_wvDu (u,v,w)`	$\varphi = wu\nabla v - wv\nabla u$	$\nabla(uw) \times \nabla v + \nabla u \times \nabla(vw)$

Table 1: $H(\mathrm{curl})$ basis function objects

The triangular element from [26] has basis functions

$$\begin{aligned}
\varphi_{E,0} &:= \lambda_{E_1}\nabla\lambda_{E_2} - \lambda_{E_2}\nabla\lambda_{E_2} \\
\varphi_{E,i} &:= \nabla\big(\lambda_{E_1}\lambda_{E_2}P^S_{i-1}(\lambda_{E_i} - \lambda_{E_j}, \lambda_{E_i} + \lambda_{E_j})\big) \\
&\quad \text{and for } u_i := \lambda_1\lambda_2 P^S_i(\lambda_1 - \lambda_2, \lambda_1 + \lambda_2) \text{ and } v_j := \lambda_3 P_j(2\lambda_3 - 1) \\
\varphi^1_{T,ij} &:= \nabla(u_i v_j), \qquad i + j \le p - 2 \\
\varphi^2_{T,ij} &:= u_i\nabla v_j - v_j\nabla u_i, \qquad i + j \le p - 2 \\
\varphi^3_{T,i} &= (\lambda_1\nabla\lambda_2 - \lambda_2\nabla\lambda_1)v_j, \qquad j \le p - 2
\end{aligned}$$

The lowest order edge-basis functions are the well known Whitney forms. Also all other $H(\mathrm{curl})$ basis functions are defined via scalar polynomials and gradients of scalar polynomials. This is the same also for the other element geometries, as well as in 3D. We implement the basis functions utilizing this structure, see Listing 12. The basis-functions are defined via the objects from Table 1.

Now, the template argument `TSHAPE` in calling `T_CalcShape`, decides whether we want to compute the values of the shape functions, or its curl: If the elements of `TSHAPE` are of type `HCurlShape`, then implicit conversion operators from `Du, uDv_minus_vDu`, etc., calculate the shape function value. If the elements are of type `HCurlCurlShape`, implicit conversion operators calculate the curl, see Listing 13.

6 Vectorization of finite element operations

Modern Intel and Intel-compatible microprocessors, as well as general purpose graphics processing units (GPGPUs or just GPUs) by Nvidia and others are designed to profit from the single-instruction multiple-data (SIMD) paradigm. For example, Intel's recent AVX - technology allows to calculate with four double precision numbers simultaneously in one processor core. Nvidia's GPU multi-processors compute with 32 threads simultaneously. Such SIMD instructions can be used for evaluating shape functions in several integration points simultaneously: data (the coordinates) is different, but the operations are the very same. A more and more serious bottleneck is the access to memory, even to the first level cache. GPUs have only a relatively small number of double precision registers (up to 128 in the Kepler device, and less if higher block-level parallelism is aspired). Thus, eliminating local arrays is an important goal.

A proper interface allowing SIMD parallelization behind the scenes are functions based on whole integration-rules, rather then on individual points. Two such functions are:

```
void ScalarFiniteElement<DIM> ::
  Evaluate (IntegrationRule & ir,
            FlatVector<> coefs, FlatVector<> vals);
```

```
template<> template<typename T, typename TSHAPE>
void HCurlHighOrderFE_Shape<ET_TRIG> ::
T_CalcShape (T x[2], TSHAPE & shape) const {

  T lam[3] = { x[0], x[1], 1-x[0]-x[1] };
  Array<AutoDiff<2>> adpol1(order),adpol2(order);

  int ii = 3;
  for (int i = 0; i < 3; i++) {
     INT<2> e = GetEdgeSort (i, vnums);

     // Whitney lowest order shape function
     shape[i] = uDv_minus_vDu (lam[e[0]], lam[e[1]]);

     // high-order edge-based shape functions (gradients)
     if(order_edge[i] > 0) {
        LegendrePolynomial::
             EvalScaledMult (order_edge[i]-1,
                  lam[e[1]]-lam[e[0]], lam[e[0]]+lam[e[1]],
                  lam[e[0]]*lam[e[1]],
                  SBLambda([&](int i, AutoDiff<2> val)
                       { shape[ii++] = Du (val); }));
      }
    }

  // element-based basis functions
  int p = order_face[0];
  if(p > 1) {
     LegendrePolynomial::
       EvalScaledMult (p-1, lam[0]-lam[1], lam[0]+lam[1],
                        lam[0]*lam[1], adpol1);
     LegendrePolynomial::
       EvalMult (p-1, 2*lam[2]-1, lam[2], adpol2);

     // type 1 - gradients:
     for (int j = 0; j < p-1; j++)
       for (int k = 0; k < p-1-j; k++)
         shape[ii++] = Du (adpol1[j] * adpol2[k]);
     // type 2
     for (int j = 0; j < p-1; j++)
       for (int k = 0; k < p-1-j; k++)
         shape[ii++] = uDv_minus_vDu (adpol2[k], adpol1[j]);
     // type 3
     for (int j = 0; j < p-1; j++)
       shape[ii++] = wuDv_minus_wvDu (lam[1], lam[2], adpol2[j]);
     }
  }
```

Listing 12: $H(\mathrm{curl})$ triangular finite element

```
virtual void CalcShape (const IntegrationPoint & ip,
                        FlatMatrixFixWidth<DIM> shape)
{
  AutoDiff<DIM> adp[DIM];
  for (int i = 0; i < DIM; i++) adp[i] = AutoDiff<DIM> (ip(i), i);

  static_cast<const FEL*> (this) ->
    T_CalcShape (adp, SBLambda ([&](int i, HCurlShape val)
                        { shape.Row(i) = val; })));
}

virtual void CalcCurlShape (const IntegrationPoint & ip,
                            FlatMatrixFixWidth<DIM_CURL> cshape)
{
  AutoDiff<DIM> adp[DIM];
  for (int i = 0; i < DIM; i++) adp[i] = AutoDiff<DIM> (ip(i), i);

  static_cast<const FEL*> (this) ->
    T_CalcShape (adp, SBLambda ([&](int i, HCurlCurlShape val)
                        { shape.Row(i) = val; })));
}
```

Listing 13: $H(\mathrm{curl})$ Shape functions and curls

```
void ScalarFintieElement<DIM> ::
  EvaluateTrans (IntegrationRule & ir,
                 FlatVector<> vals, FlatVector<> coefs);
```

The first one evaluates the the finite element function in all points, i.e.

$$\texttt{vals}(i) = \sum_j \texttt{coefs}(j)\varphi_j(x_i) \qquad \forall \text{ points } x_i,$$

while the second one computes the transposed operation

$$\texttt{coefs}(j) = \sum_i \texttt{vals}(i)\varphi_j(x_i) \qquad \forall\, 0 \le j < ndof.$$

A simple default implementation just loops over all points, and calls the single-point function.

We implemented the vectorization on top of *mdlib*, a C++ wrapper for AVX and SSE data types and compiler intrinsics. *mdlib* uses concepts from [17], but it distinguishes between logical vector size and hardware implementation. For example, a `MD<8,double>` object is implemented using two 256-bit AVX variables, or four 128-bit SSE variables, or just 8 double variables, depending on the hardware.

The vectorized integration rule evaluation function is given in Listing 14. The point-vector consists now of four x, y, and z-coordinates. Note, for performance reason we allow the initialization with values possibly out of range of the integration-rule. When constructing the integration-rule we allocate additional memory to ensure valid read access. The `T_CalcShape`

```
template <class FEL, ELEMENT_TYPE ET>
void T_ScalarFiniteElement<FEL,ET> ::
Evaluate (const IntegrationRule & ir, FlatVector<double> coefs,
          FlatVector<double> vals) const
{
  for (int i = 0; i < ir.GetNIP(); i += 4) {

      Vec<DIM, MD<4>> pt;
      for (int k = 0; k < DIM; k++)
        pt[k] = MD<4> (ir[i](k), ir[i+1](k), ir[i+2](k), ir[i+3](k));

      MD<4> sum = 0.0;
      T_CalcShape (&pt(0), SBLambda ( [&] (int j, MD<4> val)
                                      { sum += coefs(j)*val; } ));

      MD<4,mask64> mask = MD<4,int>::FirstInt() < ir.GetNIP() - i;
      sum.Store (&vals(i), mask);
   }
}
```

Listing 14: Evaluate function using SIMD operations

computes now all shape functions for four points simultaneously. The values are added up to the four summation variables. The resulting values are stored to the `vals` result vector. Here, we use a bit-mask to write only into valid memory. The simultaneous integer comparison and masked write are provided by hardware, which is faster than the alternative of four conditional branch instructions.

7 Timings

First, we measure the performance of finite element function evaluation on CPUs and GPUs. Our first test system, referred to as *vector* consists of two Sandy Bridge Intel processors E5-2620, each of them contains 6 cores operating at 2GHz. It supports AVX SIMD - instructions. The peak performance of this system is $2\,\text{processors} \times 6\,\text{cores} \times 4\,\text{AVX} \times 2\,\text{GHz} = 96\,G$ multiplications and additions per second.

First, we measure the performance of shape function evaluation and gradient evaluation for H^1 finite elements. For this, we compute $\sum_{i=1}^{N} u_i \varphi_i(x_j)$ and $\sum_{i=1}^{N} u_i \nabla\varphi_i(x_j)$ in a set of integration points $\{x_j\}$, where the order of the integration rule is twice the element order. We call the function evaluation in an openmp-parallel loop. We use hyper-threading and thus 24 threads are generated. We note that all computations are performed within level 1 caches, so essentially floating point performance is measured.

The inner-most loop for triangles and tetrahedral elements is the evaluation of Jacobi polynomials, which takes 3 multiplication, plus one additional multiplication for the coefficient. The results in Figure 2 show 19 G shape function evaluation for higher polynomial orders, which corresponds to 76 G multiplication. Thus, the floating point efficiency is close to optimum, namely 80%. We use simultaneous evaluation of eight integration points (i.e.

```
template <class FEL, ELEMENT_TYPE ET>
void T_ScalarFiniteElement<FEL,ET> ::
Evaluate (const IntegrationRule & ir, FlatVector<double> coefs,
          FlatVector<double> vals) const
{
  for (int i = threadIdx.x; i < ir.GetNIP(); i += blockDim.x) {
      Vec<DIM> pt = ir[i].Point();
      double sum = 0;
      T_CalcShape (&pt(0), SBLambda ( [&](int j, double shape)
              { sum += coefs(j)*shape; } ));
      vals(i) = sum;
   }
}
```

Listing 15: Evaluate function using CUDA

```
template <class FEL, ELEMENT_TYPE ET>
void T_ScalarFiniteElement<FEL,ET> ::
EvaluateTrans (const IntegrationRule & ir, FlatVector<double> vals,
               FlatVector<double> coefs) const
{
  for (int i0 = 0; i0 < ir.GetNIP(); i0 += blockDim.x) {
      int i = i0 + threadIdx.x;
      Vec<DIM> pt = ir[i].Point();
      double v = (i < ir.GetNIP()) ? vals(i) : 0;
      T_CalcShape (&pt(0), SBLambda ( [&](int j, double shape)
         { double sum = HorizontalSum (v*shape);
           if (threadIdx.x == 0) coefs(j) = sum;  } ));
   }
}
```

Listing 16: Transpose evaluate using CUDA

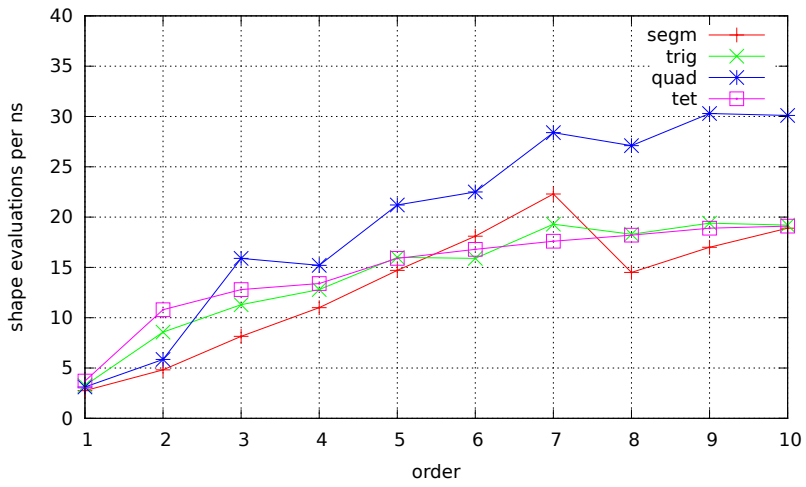

Figure 2: function evaluation for H^1 finite elements using AVX

two AVX data types) to hide operation latency. Figure 3 shows the performance of shape function evaluation without SIMD optimization.

Figure 4 shows the performance of gradient evaluation. For the tetrahedral H^1-element, one gradient evaluation takes asymptotically 18 multiplications. Thus, the maximum is $96/18 = 5.3G$ gradient calculations per second.

Next, we give timings for evaluation and transpose evaluation using NVIDIA's CUDA platform. The test hardware is a Tesla K20 GPU, which contains 13 Cuda multi-processors operating at 705.5 MHz. Each multi-processor can perform 64 double precision multiply-add operations per cycle, which results in a peak performance of 587 G multiply-add operations per second. We use the CUDA 6.5 development tools released August 2014, which is the first version supporting C++11. Thus, the presented results for GPUs are very recent and subject of ongoing research.

We give timings for shape function evaluation and transpose evaluation for quadrilateral elements using tensor product Legendre polynomials. One warp consisting of 32 threads is processing one element. Each thread evaluates either 2 or 4 integration points simultaneously, which reduces memory operations for loading coefficients. The transpose operation requires summation across threads, which is done using warp shuffle operations. Coefficients for the Legendre polynomials are stored in constant memory. Timings for evaluation and transpose evaluation are given in Figure 5. Details on the GPU implementation, results on multiple evaluation, and timings of the full Discontinuous Galerkin code are given in [13].

7.1 Finite element matrix assembling

We provide timings for the matrix assembling for the diffusion equation with variable coefficients. We use a mesh consisting of 212 K elements and finite elements of (runtime) order 3, which leads to 993 K global degrees of freedom. We use openmp multithreading using 12 threads on *vector*. Wall clock time for the assembling is 0.36 sec.

We use Intel's VTune Amplifier XE 2015 performance analysis tool to figure out where the major part of CPU-time is spent. It uses hardware event counters of the Sandy Bridge

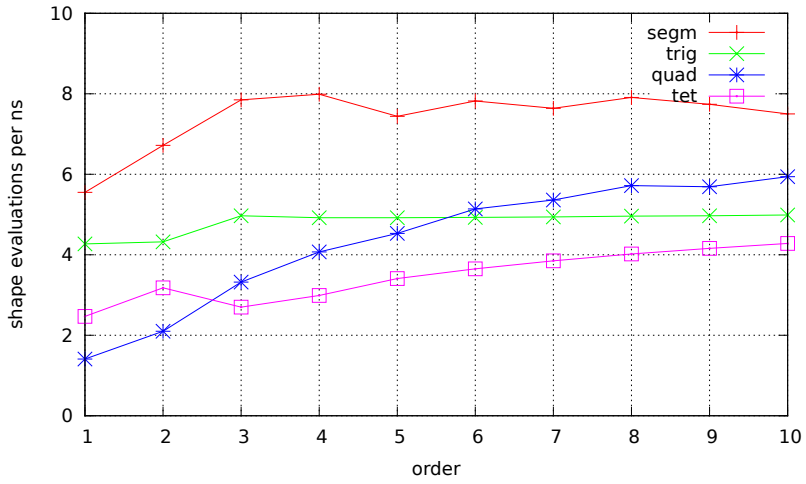

Figure 3: function evaluation for H^1 finite elements without SIMD

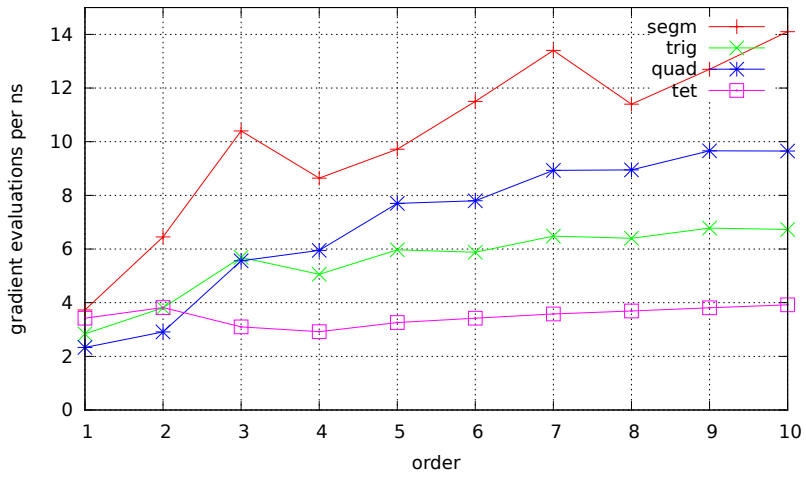

Figure 4: gradient evaluation for H^1 finite elements using AVX

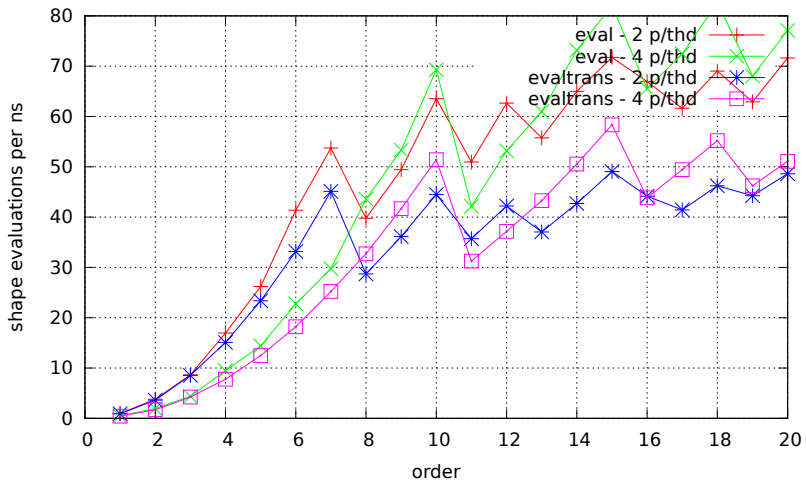

Figure 5: evaluation and transpose for L_2 quadrilateral on Kepler GPU

processor. We give the result of the `CPU_CLK_UNHALTED.THREAD` counter, which gives the active core cycles.

12 active cores at 2 GHz perform 8640 M cycles within 0.36 seconds, which corresponds well to the measured 8370 M cycles in total.

Out of these, 4299 M cylces are spent for the element matrix calculation. Element matrices are of dimension 20 by 20, and 14 integration points are used. Assembling of the element matrices into the global sparse matrix takes 3052 M cycles. Our multithreading algorithm uses coloring, such that parallel assembling can be done without locks (critical sections). The global matrix is stored in shared memory, which is the bottle-neck in the assembling routine. The MPI-parallel version benefits from the NUMA memory architecture, and the assembling can be significantly improved. 583 M cycles are spent for the logic, namely constructing the finite element, and collecting degrees of freedom.

The calculation of one element matrix takes 4299 M / 212 K = 20278 core cycles, where 7382 cycles are spent for computing the mapped gradients. These are $7382/(14 \times 20) = 26$ cycles per gradient evaluation. Thus, all cores compute $12\,\text{cores} \times 4\,\text{AVX} \times 2\text{GHz}/26 = 3.7$ gradients per nano-second, which corresponds well with Figure 4.

The second major part in element calculation is computing all inner products $\sum_{x_k}(w_k\lambda(x_k)\nabla\varphi_i(x_k)) \cdot \nabla\varphi_j(x_k)$. This is done by a matrix-matrix multiplication, again using AVX optimization. The cpu-time for third order elements is approximately the same as calculating the shape function gradients.

7.2 Discussion and further improvements

In the present manuscript we have presented a high level implementation of finite element operations. We have demonstrated high floating point performance reaching 80 % of peak on CPUs with AVX operations. Since the evaluation of shape functions take in average 4 multiplications per point and shape function, this factor of 4 is lost in competition with pre-computed matrices, and performing linear algebra operations using vendor BLAS functions. The matrix operations are in particular efficient when equivalent elements are combined, and

element matrix calculation	4299	shape gradient calc matrix matrix product mult coefficients integration rule etc	1565 1552 514 224 444
global matrix assembling	3052		
logic: generate fe, dofs etc	583 373		
total	8307		

Table 2: million core cycles for matrix assembling

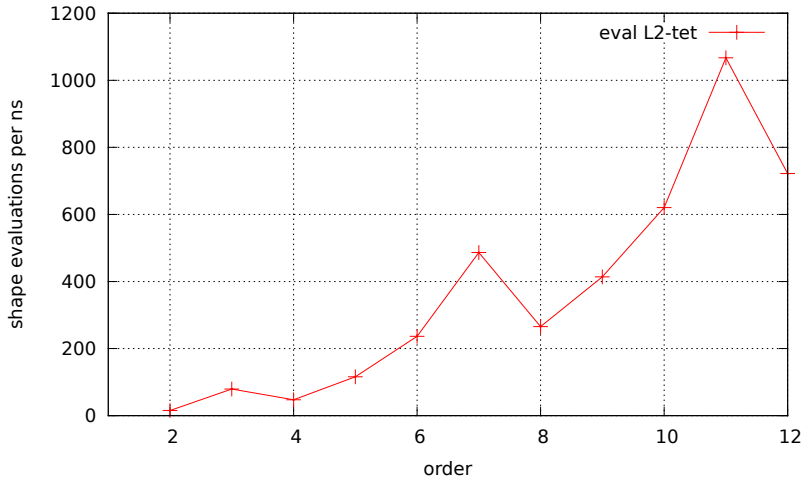

Figure 6: function evaluation by sum-factorization and AVX for L_2 tetrahedral elements

matrix-matrix products are computed. But, this coding technique requires a restructuring of the element loop by grouping equivalent elements. In certain applications we also improve the evaluation by combining evaluation for several vectors, for example when dealing with systems of equations.

Another possibility for improvement is the utilization of tensor product structure of shape functions and integration rules, known as sum-factorization [22, 14]. This allows to reduce the complexity of evaluation from p^{2d} to p^{d+1}. We are currently working on a vectorized implementation of sum-factorization following the programming techniques presented in this paper. A preliminary result for evaluation of tetrahedral L_2-elements using the Dubiner basis is given in Figure 6. Measurements were done on the 12 core system *vector*. At order 5, this version beats evaluation by matrix-multiplication at peak performance, at order 10 the advantage is a factor more than 6.

Acknowledgement: The author wants to thank Matthias Hochsteger for helping with the GPU-version of finite element function evaluation.

References

[1] M. Ainsworth and J. Coyle. Hierarchic Finite Element Bases on Unstructured Tetrahedral Meshes. *Int. J. Num. Meth. Eng.*, 58(14), 2103-2130.

[2] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a General Purpose Object Oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4), 24/1–24/27, 2007

[3] J. Barton, L.R. Nackman. *Scientific and Engineering C++: An Instroduction with Advanced Techniques and Examples*. Addison-Wesley Professional, 1994

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klfkorn, R. Kornhuber, M. Ohlberger, O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2-3), 121–138, 2008

[5] P. Castillo, R. Rieben, D. White FEMSTER : An Object-Oriented Class Library of High-Order Discrete Differential Forms *ACM Trans. Math. Softw.*, 31(4), 425–457, 2005

[6] P.G. Ciarlet. *The finite element method for elliptic problems*. North-Holland, Amsterdam, 1978.

[7] A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module. *Computing* 90(3), 165–196, 2011

[8] L. Demkowicz *Computing with hp-adaptive finite elements. I. One and two dimensional elliptic and maxwell problems* Chapman & Hall / CRC Press, Boca Raton, FL, 2006

[9] L. Demkowicz *Computing with hp-adaptive finite elements. II. Frontiers: Three dimensional elliptic and Maxwell problems with applications* Chapman & Hall / CRC Press, Boca Raton, FL, 2008

[10] G. Guennebaud, B. Jacob. EIGEN - a C++ linear algebra library available from http://eigen.tuxfamily.org

[11] F. Hecht. *Freefem++*. http:/www.freefem.org/ff++

[12] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods—Algorithms, Analysis and Applications*. Text in Applied Mathematics. Springer, 2007.

[13] M. Hochsteger. *High Order Discontinuous Galerkin Methods on GPUs*. Master's Thesis, Inst. Analysis and Scientific Computing, Vienna UT, 2014

[14] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford Science Publications, 2005

[15] R.C. Kirby, Algorithm 839: FIAT, a new paradigm for computing finite element basis functions, *ACM Trans. Math. Software*, 30(4), 502–516, 2004

[16] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven Nodal discontinuous Galerkin methods on graphics processors. *J Comp. Phys.* 228(21), 7863–7882, 2009

[17] M. Kretz and V. Lindenstrutz. Vc: A C++ library for explicit vectorization *Softw. Pract. Exper*,00, 1–18, 2011.

[18] A. Logg, K.-A. Mardal, G.N. Wells (eds) *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Lecture notes in Computational Science and Engineering 84. Springer, 2011

[19] M. Lyly, J. Ruokolainen, and E. Järvinen. ELMER - A finite element solver for multi-physics. CSC-report on scientific computing, 156–159, 1999-2000

[20] P. Monk, Finite Element Methods for Maxwell's Equations. *Oxford University Press*, 2003.

[21] J.C. Nédélec. A new family of mixed finite elements in $\mathbb{R}^3$, *Num. Math.*, 50, 57–81, 1986.

[22] S.A.Orszag. Spectral methods for problems in complex geometries, *J. Comp. Phys.* 37, 70-92, 1980.

[23] C. Prud'homme. *Life: Overview of a Unified C++ Implementation of the Finite and Spectral Element Methods in 1D, 2D and 3D.* in Applied Parallel Computing. Lecture Notes in Computer Science Volume 4699, 712–721, 2007

[24] C. Prud'Homme, V. Chabannes, V. Doyeux; M. Ismail, A. Samake, G. Pena Feel++: A Computational Framework for Galerkin Methods and Advanced Numerical Methods in ESAIM: Proceedings 38, 429–455, 2012

[25] J. Schöberl. NGSolve Finite Element Library http://sourceforge.net/projects/ngsolve

[26] J. Schöberl and S. Zaglmayr, High order Nedelec elements with local complete sequence properties, *COMPEL*, 24, 2, 374–384(11), 2005.

[27] C. Schwab. *p- and hp- Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Clarendon Press, 1998

[28] B. Stroustrup. *The C++ Programming Language,* 4^{th} *edition*. Pearson Education, Ind., 2013

[29] B. Szabó and I. Babuška. *Finite Element Analysis*. Wiley, 1991.

[30] T. Veldhuizen. Expression Templates. *C++ Report* 7(5), 26-31, 1995.

[31] T. Veldhuizen. Techniques for Scientific C++. Indiana University Computer Science Techical Report #542, 2000

[32] S. Zaglmayr, *High Order Finite Element Methods for Electromagnetic Field Computation*, PhD dissertation, Johannes Kepler Universität Linz, Austria, 2006.