

## Extrapolate - FEniCS

Après plusieurs tests sur la partie de correction/certification du modèle où l'on prend la solution analytique comme nouvelle level-set, il semblerait que la méthode avec les meilleurs résultats soit celle où l'on utilise la méthode **extrapolate** de FEniCS. C'est pourquoi, on va ici présenter en détail le code source de cette fonction FEniCS ([Extrapolation](#)).

Pour illustrer les explications, nous considérerons un domaine rectangulaire maillé uniformément par des triangles. Dans la suite, nous considérerons également *cell0* comme étant une des cellules de maillage. On prendra comme exemple une extrapolation de  $\mathbb{P}^1$  vers  $\mathbb{P}^2$ .

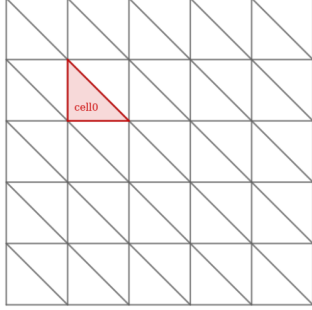
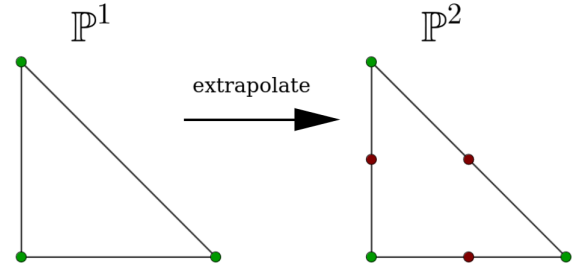


FIGURE 1 – Cell0

FIGURE 2 – Extrapolation de  $\mathbb{P}^1$  vers  $\mathbb{P}^2$ 

Corps de la fonction **extrapolate** qui a pour but d'extrapoler la fonction  $v$  en une fonction  $w$  :

```
void Extrapolation::extrapolate(Function& w, const Function& v)
```

On cherche à calculer la valeur en chacun des degrés de liberté associé à la fonction  $w$ . Pour cela, on va parcourir toutes les cellules du maillage dans le but de construire le tableau *coefficients* (de taille le nombre total de degrés de liberté associés à  $w$ ). On appelle alors sur chacune des cellules la fonction **compute\_coefficients 1** qui va compléter le tableau *coefficients* aux indices associées à ses degrés de liberté. Comme les degrés de liberté d'une cellule peuvent être communs à ceux d'une autre, on finira par faire une moyenne des coefficients en chacun des degrés de liberté en utilisant la fonction **average\_coefficients 5**, ce qui nous donne alors la fonction  $w$ .

On notera que deux espaces de fonctions sont créés dans la fonction *extrapolate* : l'espace  $V$  est notre espace de départ (dans l'exemple l'espace  $\mathbb{P}^1$ ) et  $W$  est l'espace d'arrivée (espace  $\mathbb{P}^2$ ). On prendra  $v \in V$  et  $w \in W$ .

### 1 compute\_coefficients

Corps de la fonction :

```
void Extrapolation::compute_coefficients(
    std::vector<std::vector<double>>& coefficients,
    const Function& v,
    const FunctionSpace& V,
    const FunctionSpace& W,
    const Cell& cell0,
    const std::vector<double>& coordinate_dofs0,
    const ufc::cell& c0,
    const Eigen::Ref<const Eigen::Matrix<dofin::la_index, Eigen::Dynamic, 1>> dofs,
    std::size_t& offset)
```

Cette fonction a pour but de compléter le tableau *coefficients* aux indices associées aux degrés de liberté d'une cellule donnée *cell0*. Autrement dit, on cherche à déterminer les valeurs aux degrés de liberté de la cellule *cell0* en utilisant l'information que nous apporte les cellule voisines à celle-ci.

On commence par construire les tableaux *cell2dof2row* et *unique\_dofs* en utilisant la fonction **build\_unique\_dofs 2**. L'ensemble *unique\_dofs* contient tous les degrés de liberté de notre espace de départ  $V$  (associé à la fonction  $v$ )

des cellules voisines à *cell0*. Le dictionnaire *cell2dof2row* permet d'associer à chaque degré de liberté (unique) d'une cellule donnée un numéro de ligne unique.

Ensuite, on définit  $N$  le nombre de degré de liberté associé à un élément de  $W$  (dans notre cas  $N = 6$ ) et  $M$  le nombre de degré de liberté (unique) des cellules voisines à la cellule courante *cell0* (dans notre cas les nœuds des cellules voisines et donc  $M = 12$ ). Attention : il faut que  $M \geq N$  pour avoir suffisamment de degré de libertés pour pouvoir construire l'extrapolation.

On peut maintenant créer la matrice  $A$  (de taille  $M \times N$ ) et le vecteur  $b$  (de taille  $M$ ). En parcourant les cellules voisines de la cellule courante *cell0*, on va compléter la matrice  $A$  et le vecteur  $b$  en utilisant la fonction **add\_cell\_equations** 4. A noter que la cellule courante *cell0* est incluse dans ses cellules voisines.

On pourra ensuite résoudre le système linéaire  $Ax = b$  qui nous donnera la valeur en chacun des degrés de liberté de la cellule courante *cell0*. Ces valeurs sont alors ajoutées au tableau global *coefficients* qui nous fournit après avoir utilisé la fonction **average\_coefficients** 5 les valeurs en chaque degré de liberté de  $w$ .

## 2 build\_unique\_dofs

Corps de la fonction :

```
void Extrapolation::build_unique_dofs(
    std::set<std::size_t>& unique_dofs,
    std::map<std::size_t, std::map<std::size_t, std::size_t>>& cell2dof2row,
    const Cell& cell0,
    const FunctionSpace& V)
```

Cette fonction a pour but de compléter les tableaux *cell2dof2row* et *unique\_dofs* donnés en entrée. A noter que au total, on a le même nombre de degré de liberté dans *cell2dof2row* et *unique\_dofs*.

On commence par remplir un ensemble contenant les cellules voisines à *cell0*. Pour être plus précis, les cellules voisines à *cell0* sont les cellules ayant un nœud commun avec la cellule courante *cell0* (Figure 3).

En parcourant ensuite chacune de ces cellules, on va pouvoir compléter les tableaux *cell2dof2row* et *unique\_dofs* donnés en entrée en appelant la fonction **compute\_unique\_dofs** 3.

Dans le cas de notre exemple, on va numéroter tous les nœuds des cellules voisines à *cell0* et on supposera que le parcours des cellules est effectuées dans un ordre précis (Figure 4).

Alors le set *unique\_dofs* contiendra tous les noeuds des cellules voisines à *cell0* :

$$unique\_dofs = \{n1, n2, n3, \dots, n12\}$$

Et le dictionnaire *cell2dof2row* associé à *cell0* est construit de la manière suivante :

$$cell2dof2row = \left\{ \begin{array}{l} "cell0" : \{ "n1" : 0, "n2" : 1, "n3" : 2 \}, \\ "cell1" : \{ "n4" : 3 \}, \\ "cell2" : \{ "n5" : 4 \}, \\ \dots \end{array} \right\}$$

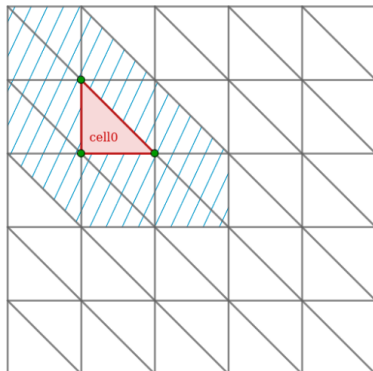


FIGURE 3 – Cellules voisines à Cell0

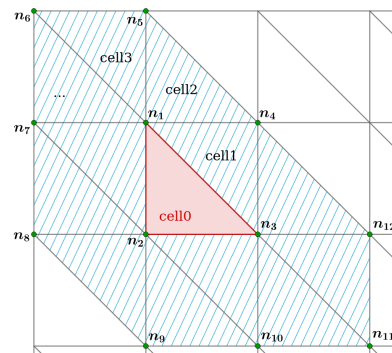


FIGURE 4 – Parcours des cellules voisines

### 3 compute\_unique\_dofs

Corps de la fonction :

```
std::map<std::size_t, std::size_t> Extrapolation::compute_unique_dofs(  
    const Cell& cell,  
    const FunctionSpace& V,  
    std::size_t& row,  
    std::set<std::size_t>& unique_dofs)
```

Cette fonction a pour but de traiter chacune des cellules voisines afin de compléter les tableaux *unique\_dofs* et *cell2dof2row* créés dans **compute\_coefficients 1**.

Pour une cellule *cell* (voisine à *cell0*), on va parcourir chacun de ses degrés de liberté associé à l'espace *V*. Autrement dit, on parcourt tous les degrés de liberté dont la valeur est connue (dans notre cas les degrés de liberté  $\mathbb{P}^1$  et donc les noeuds de la cellule). Si ce degré de liberté fait partie de *unique\_dofs*, on ne fait rien. Sinon, on l'ajoute à *unique\_dofs*. On va également créer un tableau *dof2row* qui a pour but d'associer un degré de liberté à un numéro de ligne unique. Ce dictionnaire *dof2row* est retourné par la fonction et permet de remplir le dictionnaire plus général *cell2dof2row* créé dans **compute\_coefficients 1**.

### 4 add\_cell\_equations

Corps de la fonction :

```
void Extrapolation::add_cell_equations(  
    Eigen::MatrixXd& A,  
    Eigen::VectorXd& b,  
    const Cell& cell0,  
    const Cell& cell1,  
    const std::vector<double>& coordinate_dofs0,  
    const std::vector<double>& coordinate_dofs1,  
    const ufc::cell& c0,  
    const ufc::cell& c1,  
    const FunctionSpace& V,  
    const FunctionSpace& W,  
    const Function& v,  
    std::map<std::size_t, std::size_t>& dof2row)
```

Cette fonction a pour but de remplir une partie de la matrice *A* et du vecteur *b* à partir de la cellule courante *cell0* et d'une de ses cellules voisines *cell1*.

On commence par créer les fonctions de base  $\Phi_j$  associées aux degrés de liberté de la cellule courante *cell0*.

On va ensuite parcourir les degrés de liberté associés à la cellule voisine *cell1* dans le dictionnaire *cell2dof2row* (c'est le tableau *dof2row* donné en argument). On évalue alors  $\Phi_j$  en chacun des degrés de liberté de la cellule voisine *cell1*. On peut alors compléter  $A(row, j)$  (où *row* nous ai donné par le dictionnaire *dof2row*). On complète également  $b(row)$  par la valeur aux degrés de liberté associés à l'espace *V* (les noeuds dans notre cas).

PARTIE A COMPLETER/MODIFIER : Détailler la construction de la matrice A (dans compute\_coefficients?)

### 5 average\_coefficients

Corps de la fonction :

```
void Extrapolation::average_coefficients(  
    Function& w,  
    std::vector<std::vector<double>>& coefficients)
```

Cette fonction a pour but de faire la moyenne pour chaque degré de liberté des coefficients calculés. Elle associe ensuite ces valeurs au vecteur *w*.

## A Code source FEniCS

```
// Copyright (C) 2009-2011 Anders Logg
//
// This file is part of DOLFIN.
//
// DOLFIN is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// DOLFIN is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with DOLFIN. If not, see <http://www.gnu.org/licenses/>.
//
// Modified by Marie E. Rognes, 2010
// Modified by Garth N. Wells, 2010
//
// First added: 2009-12-08
// Last changed: 2011-11-12
//

#include <vector>
#include <ufc.h>

#include <dolfin/common/Timer.h>
#include <dolfin/fem/BasisFunction.h>
#include <dolfin/fem/DirichletBC.h>
#include <dolfin/fem/GenericDofMap.h>
#include <dolfin/function/Function.h>
#include <dolfin/function/FunctionSpace.h>
#include <dolfin/la/GenericVector.h>
#include <dolfin/log/log.h>
#include <dolfin/mesh/BoundaryMesh.h>
#include <dolfin/mesh/Cell.h>
#include <dolfin/mesh/Vertex.h>
#include <dolfin/mesh/FacetCell.h>
#include "Extrapolation.h"

using namespace dolfin;

//-----
void Extrapolation::extrapolate(Function& w, const Function& v)
{
    // Using set_local for simplicity here
    not_working_in_parallel("Extrapolation_of_functions");

    // Check that the meshes are the same
    if (w.function_space()->mesh() != v.function_space()->mesh())
    {
        dolfin_error("Extrapolation.cpp",
                     "compute_extrapolation",

```

```

        "Extrapolation must be computed on the same mesh");
    }

    // Extract mesh and function spaces
    const FunctionSpace& V = *v.function_space();
    const FunctionSpace& W = *w.function_space();
    dolfin_assert(V.mesh());
    const Mesh& mesh = *V.mesh();

    // Initialize cell-cell connectivity
    const std::size_t D = mesh.topology().dim();
    mesh.init(D, D);

    // UFC cell view of center cell and vertex coordinate holder
    ufc::cell c0;
    std::vector<double> coordinate_dofs0;

    // List of values for each dof of w (multivalued until we average)
    std::vector<std::vector<double>> coefficients;
    coefficients.resize(W.dim());

    // Iterate over cells in mesh
    dolfin_assert(W.dofmap());
    for (CellIterator cell0(mesh); !cell0.end(); ++cell0)
    {
        // Update UFC view
        cell0->get_coordinate_dofs(coordinate_dofs0);
        cell0->get_cell_data(c0);

        // Tabulate dofs for w on cell and store values
        auto dofs = W.dofmap()->cell_dofs(cell0->index());

        // Compute coefficients on this cell
        std::size_t offset = 0;
        compute_coefficients(coefficients, v, V, W, *cell0, coordinate_dofs0,
                             c0, dofs, offset);
    }

    // Average coefficients
    average_coefficients(w, coefficients);
}

//-----
void Extrapolation::compute_coefficients(
    std::vector<std::vector<double>>& coefficients,
    const Function& v,
    const FunctionSpace& V,
    const FunctionSpace& W,
    const Cell& cell0,
    const std::vector<double>& coordinate_dofs0,
    const ufc::cell& c0,
    const Eigen::Ref<const Eigen::Matrix<dolfin::la_index, Eigen::Dynamic, 1>> dofs,
    std::size_t& offset)
{
    // Call recursively for mixed elements
    dolfin_assert(V.element());
    const std::size_t num_sub_spaces = V.element()->num_sub_elements();

```

```

if (num_sub_spaces > 0)
{
    for (std::size_t k = 0; k < num_sub_spaces; k++)
    {
        compute_coefficients(coefficients, v[k], *V[k], *W[k], cell0,
                             coordinate_dofs0, c0, dofs, offset);
    }
    return;
}

// Build data structures for keeping track of unique dofs
std::map<std::size_t, std::map<std::size_t, std::size_t>> cell2dof2row;
std::set<std::size_t> unique_dofs;
build_unique_dofs(unique_dofs, cell2dof2row, cell0, V);

// Compute size of linear system
dolfin_assert(W.element());
const std::size_t N = W.element()->space_dimension();
const std::size_t M = unique_dofs.size();

// Check size of system
if (M < N)
{
    dolfin_error("Extrapolation.cpp",
                 "compute_extrapolation",
                 "Not enough degrees of freedom on local patch to build extrapolation");
}

// Create matrix and vector for linear system
Eigen::MatrixX<double> A(M, N);
Eigen::VectorX<double> b(M);

// Add equations on cell and neighboring cells
dolfin_assert(V.mesh());
ufc::cell c1;
std::vector<double> coordinate_dofs1;

// Get unique set of surrounding cells (including cell0)
std::set<std::size_t> cell_set;
for (VertexIterator vtx(cell0); !vtx.end(); ++vtx)
{
    for (CellIterator cell1(*vtx); !cell1.end(); ++cell1)
        cell_set.insert(cell1->index());
}

for (auto cell_it : cell_set)
{
    if (cell2dof2row[cell_it].empty())
        continue;

    Cell cell1(cell0.mesh(), cell_it);

    cell1.get_coordinate_dofs(coordinate_dofs1);
    cell1.get_cell_data(c1);
    add_cell_equations(A, b, cell0, cell1,
                       coordinate_dofs0, coordinate_dofs1,

```

```

        c0, c1, V, W, v,
        cell2dof2row[cell_it]);
}

// Solve least squares system
const Eigen::VectorXd x
    = A.jacobiSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).solve(b);

// Insert resulting coefficients into global coefficient vector
dolfin_assert(W.dofmap());
for (std::size_t i = 0; i < W.dofmap()->num_element_dofs(cell0.index()); ++i)
    coefficients[dofs[i + offset]].push_back(x[i]);

// Increase offset
offset += W.dofmap()->num_element_dofs(cell0.index());
}

//-----
void Extrapolation::build_unique_dofs(
    std::set<std::size_t>& unique_dofs,
    std::map<std::size_t, std::map<std::size_t, std::size_t>>& cell2dof2row,
    const Cell& cell0,
    const FunctionSpace& V)
{
    // Counter for matrix row index
    std::size_t row = 0;
    dolfin_assert(V.mesh());

    // Get unique set of surrounding cells (including cell0)
    std::set<std::size_t> cell_set;
    for (VertexIterator vtx(cell0); !vtx.end(); ++vtx)
    {
        for (CellIterator cell1(*vtx); !cell1.end(); ++cell1)
            cell_set.insert(cell1->index());
    }

    // Compute unique dofs on patch
    for (auto cell_it : cell_set)
    {
        Cell cell1(cell0.mesh(), cell_it);
        cell2dof2row[cell_it] = compute_unique_dofs(cell1, V, row,
                                                    unique_dofs);
    }
}

//-----
void
Extrapolation::add_cell_equations(Eigen::MatrixXd& A,
    Eigen::VectorXd& b,
    const Cell& cell0,
    const Cell& cell1,
    const std::vector<double>& coordinate_dofs0,
    const std::vector<double>& coordinate_dofs1,
    const ufc::cell& c0,
    const ufc::cell& c1,
    const FunctionSpace& V,
    const FunctionSpace& W,

```

```

        const Function& v,
        std::map<std::size_t, std::size_t>& dof2row)
{
    // Extract coefficients for v on patch cell
    dolfin_assert(V.element());
    std::vector<double> dof_values(V.element()->space_dimension());
    v.restrict(&dof_values[0], *V.element(), cell1, coordinate_dofs1.data(),
               c1);

    // Create basis function
    dolfin_assert(W.element());
    BasisFunction phi(0, W.element(), coordinate_dofs0);

    // Iterate over given local dofs for V on patch cell
    for (auto const &it : dof2row)
    {
        const std::size_t i = it.first;
        const std::size_t row = it.second;

        // Iterate over basis functions for W on center cell
        for (std::size_t j = 0; j < W.element()->space_dimension(); ++j)
        {
            // Create basis function
            phi.update_index(j);

            // Evaluate dof on basis function
            const double dof_value
                = V.element()->evaluate_dof(i, phi, coordinate_dofs1.data(),
                                             c1.orientation, c1);

            // Insert dof_value into matrix
            A(row, j) = dof_value;
        }

        // Insert coefficient into vector
        b[row] = dof_values[i];
    }
}

//-----
std::map<std::size_t, std::size_t>
Extrapolation::compute_unique_dofs(const Cell& cell,
                                   const FunctionSpace& V,
                                   std::size_t& row,
                                   std::set<std::size_t>& unique_dofs)
{
    dolfin_assert(V.dofmap());
    auto dofs = V.dofmap()->cell_dofs(cell.index());

    // Data structure for current cell
    std::map<std::size_t, std::size_t> dof2row;

    for (std::size_t i = 0; i < V.dofmap()->num_element_dofs(cell.index()); ++i)
    {
        // Ignore if this degree of freedom is already considered
        if (unique_dofs.find(dofs[i]) != unique_dofs.end())
            continue;
    }
}

```



```

    // Put global index into unique_dofs
    unique_dofs.insert(dofs[i]);

    // Map local dof index to current matrix row-index
    dof2row[i] = row;

    // Increase row index
    row++;
}

return dof2row;
}
//-----
void Extrapolation::average_coefficients(
    Function& w,
    std::vector<std::vector<double>>& coefficients)
{
    const FunctionSpace& W = *w.function_space();
    std::vector<double> dof_values(W.dim());

    for (std::size_t i = 0; i < W.dim(); i++)
    {
        double s = 0.0;
        for (std::size_t j = 0; j < coefficients[i].size(); ++j)
            s += coefficients[i][j];

        s /= static_cast<double>(coefficients[i].size());
        dof_values[i] = s;
    }

    // Update dofs for w
    dolfin_assert(w.vector());
    w.vector()->set_local(dof_values);
}
//-----

```