Galerkin/Linear Finite Elements Method in 1d, with non uniform coefficients

May 6, 2016

Problem Statement

We want to implement a solver for the following boundary value problem:

$$\begin{cases} -(a(x)u')' = f(x) & \text{in } (0,1) \\ u(0) = u(1) = 0 \end{cases} \tag{1}$$

where the coefficients $a(x)$ and $f(x)$ are to be specified by the user at run time

Weak Formulation

The weak form of equation (**??**) reads:

$$\begin{cases} \text{find } u \in V = H_0^1(0,1) s.t. \\ -\int\limits_0^1 (a(x)\, u')' \varphi \, dx = \int\limits_0^1 f(x) \cdot \varphi \, dx \quad \forall \varphi \in V \end{cases} \tag{2}$$

Using integration by parts we get

$$-\int\limits_0^1 (a(x)\, u')' \varphi \, dx = \int\limits_0^1 a(x)\, u' \varphi' \, dx - \left. a(x)\, \varphi u' \right|_0^1 = \int\limits_0^1 a(x)\, u' \varphi' \, dx$$

Galerkin Method

Let $V_h \subset V$ be a subspace of finite dimension N_h and let $\{\varphi_i\} \subseteq V_h$ be a basis of V_h

$$\begin{cases} \text{find } u_h \in V_h s.t. \\ \int\limits_0^1 a(x) u_h' \varphi_i' \, dx = \int\limits_0^1 f(x) \cdot \varphi_i \, dx \quad \forall \varphi_i \in V_h \end{cases} \tag{3}$$

We can express u_h as a linear combination of basis vectors:

$$u_h = \sum_{j=1}^{N_h} u_j \varphi_j \Rightarrow u_h' = \sum_{j=1}^{N_h} u_j \varphi_j'$$

Galerkin Method

Equation **??**$_2$ becomes:

$$\sum_{j=1}^{N_h} u_j \int_0^1 a(x)\, \varphi_i' \varphi_j' \, dx = \int_0^1 f(x)\, \varphi_i \, dx \quad i = 1, \ldots, N_h$$

We therefore get a linear algebraic system of the form

$$\mathbf{Au} = \mathbf{f},$$

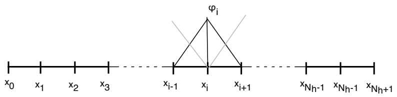
where

$$A_{ij} = \int_0^1 a(x)\, \varphi_i' \varphi_j' \, dx, \;\; f_i = \int_0^1 f(x)\, \varphi_i \, dx$$

and the vector of unknowns $\mathbf{u}$ is formed by the coefficients of the expansion of u_h w.r.t. the basis $\{\varphi_i\}$

Linear Finite Elements

Let us define a *triangulation* of the interval $(0, 1)$



Let as choose as the finite dimensional space V_h the set of functions that are continuous in $(0, 1)$ and are degree-1 polynomials (*i.e.*, affine functions) in each subintervall.
the canonical basis for V_h is gicen by:

$$\{\varphi_i\} = \{\varphi_i \in V_h \, t.c. \, \varphi_i(x_j) = \delta_{ij}\}$$

Implementation of the Linear FEM I

Using the local support property of the Finite Element basis $\varphi_i(x) \neq 0$ solo se $x \in (x_{i-1}, x_{i+1})$ we get

$$A_{ij} = \begin{cases} 0 \text{ if } |i-j| > 1 \\ \int\limits_{x_{i-1}}^{x_i} a(x)\, \varphi_i'^2 + \int\limits_{x_i}^{x_{i+1}} a(x)\, \varphi_i'^2 \text{ if } i = j \\ \int\limits_{x_{i-1}}^{x_i} a(x)\, \varphi_i' \varphi_{i-1}' \text{ if } j = i-1 \\ \int\limits_{x_i}^{x_{i+1}} a(x)\, \varphi_i' \varphi_{i+1}' \text{ if } j = i+1 \end{cases}$$

- the non zero terms in the matrix are $N_h + 2 * (N_h - 1)$ (the matrix is therefore sparse and tridiagonal)

Implementation of the Linear FEM I

Using the local support property of the Finite Element basis $\varphi_i(x) \neq 0$ solo se $x \in (x_{i-1}, x_{i+1})$ we get

$$A_{ij} = \begin{cases} 0 \text{ if } |i-j| > 1 \\ \int_{x_{i-1}}^{x_i} a(x)\, \varphi_i'^2 + \int_{x_i}^{x_{i+1}} a(x)\, \varphi_i'^2 \text{ if } i = j \\ \int_{x_{i-1}}^{x_i} a(x)\, \varphi_i' \varphi_{i-1}' \text{ if } j = i-1 \\ \int_{x_i}^{x_{i+1}} a(x)\, \varphi_i' \varphi_{i+1}' \text{ if } j = i+1 \end{cases}$$

- each entry in the matrix is given by a sum of (few) integrals each computed on only one subinterval

Implementation of the Linear FEM I

Using the local support property of the Finite Element basis $\varphi_i(x) \neq 0$ solo se $x \in (x_{i-1}, x_{i+1})$ we get

$$A_{ij} = \begin{cases} 0 \text{ if } |i-j| > 1 \\ \int\limits_{x_{i-1}}^{x_i} a(x)\, \varphi_i'^2 + \int\limits_{x_i}^{x_{i+1}} a(x)\, \varphi_i'^2 \text{ if } i = j \\ \int\limits_{x_{i-1}}^{x_i} a(x)\, \varphi_i' \varphi_{i-1}' \text{ if } j = i-1 \\ \int\limits_{x_i}^{x_{i+1}} a(x)\, \varphi_i' \varphi_{i+1}' \text{ if } j = i+1 \end{cases}$$

- the integrals cannot be computed exactly in general, we need to use a *quadrature rule*

Implementation of the Linear FEM II

To assemble **A**, we decompose the intervals of $(0, 1)$ into integrals on subintervals:

$$A_{ij} = \int_0^1 a(x)\ \varphi_i' \varphi_j'\, dx = \sum_{k=1}^{N_h+1} \int_{x_{k-1}}^{x_k} a(x)\ \varphi_i' \varphi_j'\, dx$$

We can then use the following algorithm for assembling **A**:

1. Initialize all elements of **A** to 0
2. Compute integrals on subintervals
3. Compute entries od **A** as sums of the partial integrals

This is unnecessary for this simple case but has advantages in more complex situations we will discuss later:

1. Extension to different bases
2. Extension to multiple space dimensions
3. Parallel computing

Implementation of the Linear FEM III

In each subinterval (x_{k-1}, x_k) there are four integrals $\neq 0$ which need to be computed, they are usually arranged into a *local matrix*:

$$\mathbf{A_{loc}} = \begin{bmatrix} i = k-1, j = k-1 & i = k-1, j = k \\ i = k, j = k-1 & i = k, j = k \end{bmatrix}$$

$A_{loc_{11}}$ will then be added to $A_{k-1,k-1}$
$A_{loc_{12}}$ will then be added to $A_{k-1,k}$ etc...

this process is called assembly of the coefficient matrix

Exercise

- adapt the fem1d code to allow the user to specify the coefficients $a(x)$ and $f(x)$ at runtime
  - use the trapezoidal rule to compute integrals
  - use muparser to parse the function provided by the user
- adapt the fem1d code to allow the user to specify the quadrature rule at runtime

Implementation in C++

file `fem1d.h`

```
#ifndef HAVE_FEM1D_H
#define HAVE_FEM1D_H

#include <array>
#include <algorithm>
#include <cmath>
#include <iostream>
#include <string>
#include <Eigen/Dense>
#include <Eigen/Sparse>
#include <Eigen/SparseLU>
#include <muParser.h>
#include <string>

std::string help_text = std::string () +
"fem1d␣command␣line␣options\n" +
  "-h,␣--help␣␣␣␣␣␣␣␣␣␣␣␣␣␣print␣this␣text␣and␣exit\n" +
  "-a␣<value>␣␣␣␣␣␣␣␣␣␣␣␣␣␣first␣end␣of␣interval\n" +
  "-b␣<value>␣␣␣␣␣␣␣␣␣␣␣␣␣␣second␣end␣of␣interval\n" +
  "-d,␣--diffusion␣<value>␣expression␣to␣compute␣diffusion␣coefficient\n" +
  "-f,␣--forcing␣<value>␣␣␣expression␣to␣compute␣forcing␣term␣coefficient\n" +
  "-n,␣--nnodes␣<value>␣␣␣␣number␣of␣triangulation␣nodes\n" +
  "-m,␣--maxit␣<value>␣␣␣␣␣number␣of␣iterations\n" +
  "-t,␣--tol␣<value>␣␣␣␣␣␣␣tolerance\n";

class coeff
{
private :

  std::string expr;
  double var;
  mu::Parser p;

public :
```

Implementation in C++

file `fem1d.h`

```cpp

  coeff (const std::string & s) : expr (s)
  {
    try
      {
        p.DefineVar ("x", &var);
        p.SetExpr (expr.c_str ());
      }
    catch (mu::Parser::exception_type &e)
      {
        std::cerr << e.GetMsg () << std::endl;
      }
  };

  double operator() (double x)
  {
    double y;
    var = x;
    try
      {
        y = p.Eval ();
      }
    catch (mu::Parser::exception_type &e)
      {
        std::cerr << e.GetMsg () << std::endl;
      }
    return (y);
  };
};
#endif
```

Implementation in C++

file fem1d.cpp

```
#include "fem1d.h"
#include "mesh.h"
#include "GetPot"


int main (int argc, char **argv)
{

  GetPot cl (argc, argv);

  if (cl.search (2, "-h", "--help"))
    {
      std::cerr << help_text << std::endl;
      return 0;
    }
  const double a = cl.follow (0.0, "-a");
  const double b = cl.follow (1.0, "-b");
  const unsigned int nnodes = cl.follow (100, 2, "-n", "--nnodes");
  const std::string diffusion = cl.follow ("1.0", 2, "-d", "--diffusion");
  const std::string forcing = cl.follow ("1.0", 2, "-f", "--forcing");

  coeff f_coeff (forcing);
  coeff a_coeff (diffusion);

  std::vector<double> qn;
  std::vector<double> qw;
  const std::string rule = cl.follow ("trapezoidal", 2, "-q", "--quadrature-rule");


  if (rule == "trapezoidal")
    {
      qn.push_back (0.0);
      qn.push_back (1.0);
      qw.push_back (0.5);
      qw.push_back (0.5);
```

Implementation in C++

file fem1d.cpp

```
        qw.push_back (0.5);
      }
    else if (rule == "midpoint")
      {
        qn.push_back (0.5);
        qw.push_back (1.0);
      }
    else
      std::cerr << "unknown␣quadrature␣rule" << std::endl;


    mesh m (a, b, nnodes);
    Eigen::SparseMatrix<double> A(nnodes, nnodes);

    Eigen::Matrix2d mloc;
    mloc << 0, 0, 0, 0;
    for (unsigned int iel = 0; iel < m.nels; ++iel)
      {

        mloc << 0, 0, 0, 0;
        for (unsigned int inode = 0; inode < 2; ++inode)
          {
            double igrad = (inode == 0 ? 1.0 / m.h : -1.0 / m.h);
            for (unsigned int jnode = 0; jnode < 2; ++jnode)
              {
                double jgrad =  (jnode == 0 ? 1.0 / m.h : -1.0 / m.h);
                for (unsigned int qnode = 0; qnode < qn.size (); ++qnode)
                  mloc(inode,jnode) += igrad * jgrad *
                    (a_coeff(m.nodes[m.elements[iel][0]] + m.h * qn[qnode]) *
                     m.h * qw[qnode]);

                A.coeffRef(m.elements[iel][inode],m.elements[iel][jnode]) +=
                  mloc(inode,jnode);
              }
          }
      }
```

Implementation in C++

file fem1d.cpp

```
    }

  Eigen::VectorXd f(nnodes);
  for (unsigned int ii = 0; ii < nnodes; ++ii)
    f(ii) = 0.0;

  Eigen::Vector2d vloc;

  for (unsigned int iel = 0; iel < m.nels; ++iel)
    {
      vloc << 0, 0;

      for (unsigned int inode = 0; inode < 2; ++inode)
        {
          for (unsigned int qnode = 0; qnode < qn.size (); ++qnode)
            vloc(inode) +=
              (1.0 -
               fabs (m.nodes[m.elements[iel][inode]] -
                     (m.nodes[m.elements[iel][0]] +
                      m.h * qn[qnode])) / m.h ) *
              (qw[qnode] * m.h *
               f_coeff(m.nodes[m.elements[iel][0]] + m.h * qn[qnode]));

          f(m.elements[iel][inode]) += vloc(inode);
        }
    }

  f(0) = 0;
  f(nnodes - 1) = 0;

  A.coeffRef(0,0) = 1.0e10;
  A.coeffRef(nnodes-1,nnodes-1) = 1.0e10;
  for (unsigned int ii = 1; ii < nnodes; ++ii)
    {
      A.coeffRef(0, ii) = 0.0;
      A.coeffRef(nnodes-1, nnodes-1-ii) = 0.0;
```

Implementation in C++

file fem1d.cpp

```
        A.coeffRef(nnodes-1, nnodes-1-ii) = 0.0;
      }

    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
    A.makeCompressed ();
    solver.analyzePattern(A);
    solver.factorize(A);

    Eigen::VectorXd uh = solver.solve (f);

    for (unsigned int ii = 0; ii < nnodes; ++ii)
      std::cout << m.nodes[ii] << " " << uh(ii, 0) << std:: endl;

    return 0;
};
```