l2projection_solution

April 4, 2022

1 A projection into a finite element space

The following requires assembly a finite element matrix (the mass matrix) and a right hand side. We use linear Lagrange shape functions.

So we are looking for the L^2 projection

$$u_h(x) = \sum_k u_k \varphi_k(x)$$

which is the solution of

$$\int_\Omega u_h \varphi_i = \int_\Omega u \varphi_i, \qquad \text{for all } i$$

We assume that on an element E we have

$$\varphi_{g_E(k)}(x) = \hat{\varphi}_k(F_E^{-1}(x))$$

for $k = 0, 1, 2$ and where g_E denotes the local to global dof mapper and F_E is the reference mapping.

So we need to compute

$$M_{kl}^E := \int_{\hat{E}} |DF| \hat{\varphi}_k \hat{\varphi}_l \ , \qquad b_l^E := \int_E u \varphi_l \ ,$$

and distribute these into a global matrix.

[1]:
```
import numpy
import scipy.sparse
import scipy.sparse.linalg
from dune.geometry import quadratureRules, quadratureRule
from dune.grid import cartesianDomain, gridFunction

# We will use a triangular grid for this exercise
from dune.alugrid import aluConformGrid
```

1.1 The shape functions

We use a simple class here to collect all required information about the finite element space, i.e., how to evaluate the shape functions on the reference element (together with their derivatives). We also setup a mapper to attach the degrees of freedom to the entities of the grid.

1.1.1 Task

Write a class LinearLagrangeSpace that provides 3 methods:

```
class LinearLagrangeSpace:
   def __init__(self,view):
       # TODO: Create a mapper for vertex indices
       self.mapper = see yesterdays exercises
       self.localDofs = 3
       self.points = numpy.array( [ [0,0],[1,0],[0,1] ] )
   def evaluateLocal(self, x):
       # TODO: Return a numpy array with the evaluations
       #       of the 3 basis functions in local point x
       # return numpy.array( [] )
       pass
   def gradientLocal(self, x):
       # TODO: Return a numpy array with the evaluations
       #       of the gradients of the 3 basis functions in local point x
       # return numpy.array( dbary )
       pass
```

[2]:
```
class LinearLagrangeSpace:
    def __init__(self,view):
        self.localDofs = 3
        self.view   = view
        self.dim    = view.dimension
        layout      = lambda gt: 1 if gt.dim == 0 else 0
        self.mapper = view.mapper(layout)
        self.points = numpy.array( [ [0,0],[1,0],[0,1] ] )
    def evaluateLocal(self, x):
        bary = 1.-x[0]-x[1], x[0], x[1]
        return numpy.array( bary )
    def gradientLocal(self, x):
        bary  = 1.-x[0]-x[1], x[0], x[1]
        dbary = [[-1.,-1],[1.,0.],[0.,1.]]
        return numpy.array( dbary )
```

1.2 The right hand side and matrix assembly

We need to iterate over the grid, construct the local right hand side and the local system matrix. After finishing the quadrature loop we store the resulting local matrix in a structure provided by the Python package scipy. There are many different storage structures available - we use the so called 'coordinate' (COO) matrix format which requires us construct three vectors, one to store the column indices, one for the row indices, and one for the values. The convention is that entries appearing multiple times are summed up - exactly as we need it. So after computing the local matrix and right hand side vector M^E we store the values M^E_{kl} into the values vector $v_{\text{start}+3l+k} = M^E_{kl}$ and the associated global indices $c_{\text{start}+3l+k} = g_E(k)$ and $r_{\text{start}+3l+k} = g_E(l)$.

1.2.1 Task

Implement a function that assembles the system matrix and the forcing term. For this we will use the LinearLagrangeSpace. Use the attribute mapper from the LinearLagrangeSpace for index mapping. Remember that the gradients of the basis functions obtained from the LinearLagrangeSpace need to be converted to physical space using the jacobianInverseTransposed of the geometry.

```
def assemble(space,force):
   # storage for right hand side
   rhs = numpy.zeros(len(space.mapper))

   # storage for local matrix
   localEntries = space.localDofs
   localMatrix = numpy.zeros([localEntries,localEntries])

   # data structure for global matrix using coordinate (COO) format
   globalEntries = localEntries**2 * space.view.size(0)
   value = numpy.zeros(globalEntries)
   rowIndex, colIndex = numpy.zeros(globalEntries,int), numpy.zeros(globalEntries,int)

   # TODO: implement assembly of matrix and forcing term
   ...

   # convert data structure to compressed row storage (csr)
   matrix = scipy.sparse.coo_matrix((value, (rowIndex, colIndex)),
                     shape=(len(space.mapper),len(space.mapper))).tocsr()
   return rhs,matrix
```

[3]:
```
def assemble(space,force):
    # storage for right hand side
    rhs = numpy.zeros(len(space.mapper))

    # storage for local matrix
    localEntries = space.localDofs
    localMatrix = numpy.zeros([localEntries,localEntries])

    # data structure for global matrix using coordinate (COO) format
    globalEntries = localEntries**2 * space.view.size(0)
    value = numpy.zeros(globalEntries)
    rowIndex, colIndex = numpy.zeros(globalEntries,int), numpy.
 ↪zeros(globalEntries,int)

    # iterate over grid and assemble right hand side and system matrix
    start = 0
    for e in view.elements:
        geo = e.geometry
        indices = space.mapper(e)
        localMatrix.fill(0)
        for p in quadratureRule(e.type, 4):
            x = p.position
            w = p.weight * geo.integrationElement(x)
            # evaluate the basis function at the quadrature point
            phiVals = space.evaluateLocal(x)
            # assemble the right hand side
            rhs[indices] += w * force(e,x) * phiVals[:]
            # matrix values
            for i in range(localEntries):
                for j in range(localEntries):
                    localMatrix[i,j] += phiVals[i]*phiVals[j] * w
        # store indices and local matrix for COO format
        indices = space.mapper(e)
        for i in range(localEntries):
            for j in range(localEntries):
                entry = start+i*localEntries+j
                value[entry]    = localMatrix[i,j]
                rowIndex[entry] = indices[i]
                colIndex[entry] = indices[j]
        start += localEntries**2

    # convert data structure to compressed row storage (csr)
    matrix = scipy.sparse.coo_matrix((value, (rowIndex, colIndex)),
                         shape=(len(space.mapper),len(space.mapper))).tocsr()
    return rhs,matrix
```

1.3 The main part of the code

Construct the grid and a grid function for the right hand side, compute the projection and plot on a sequence of global grid refinements:

First construct the grid

[4]:
```
domain = cartesianDomain([0, 0], [1, 1], [10, 10])
view   = aluConformGrid(domain)
```

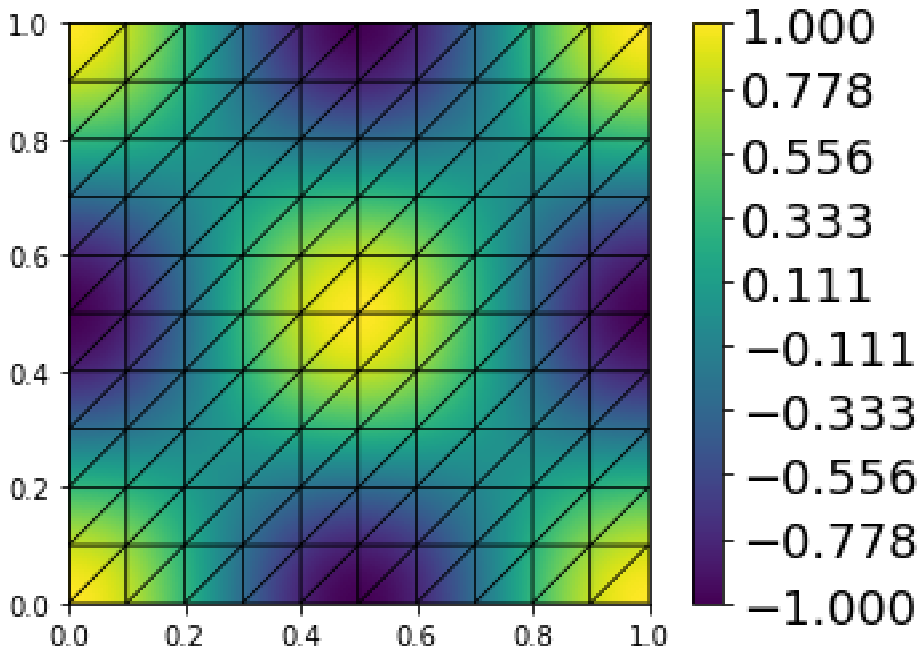
```
Created parallel ALUGrid<2,2,simplex,conforming> from input stream.


GridParameterBlock: Parameter 'refinementedge' not specified, defaulting to
'ARBITRARY'.
WARNING (ignored): Could not open file 'alugrid.cfg', using default values 0 <
[balance] < 1.2, partitioning method 'ALUGRID_SpaceFillingCurve(9)'.

You are using DUNE-ALUGrid, please don't forget to cite the paper:
Alkaemper, Dedner, Kloefkorn, Nolte. The DUNE-ALUGrid Module, 2016.
```

then the grid function to project

```
[5]: @gridFunction(view)
     def u(p):
         x,y = p
         return numpy.cos(2*numpy.pi*x)*numpy.cos(2*numpy.pi*y)
     u.plot(level=3)
```

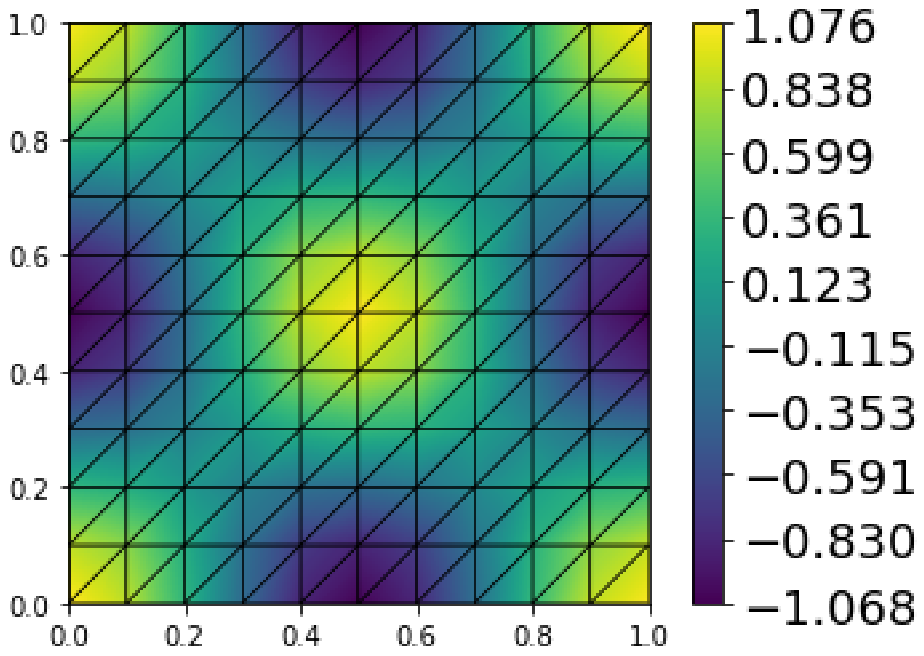


and then do the projection on a series of globally refined grids

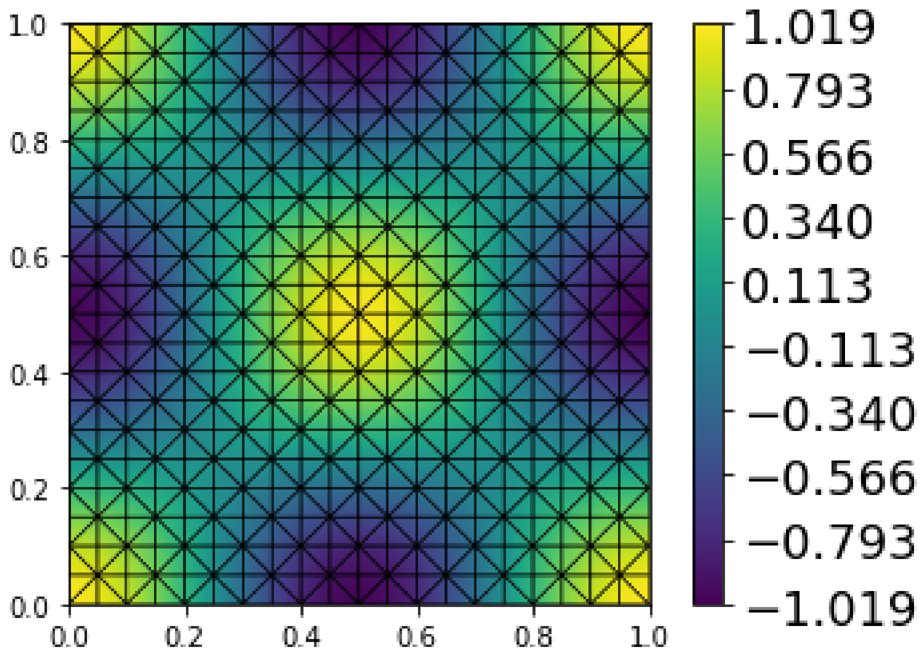
```
[6]: for ref in range(3):
         space  = LinearLagrangeSpace(view)
         print("number of elements:",view.size(0),"number of dofs:",len(space.
      ↪mapper))

         rhs,matrix = assemble(space, u)
         dofs = scipy.sparse.linalg.spsolve(matrix,rhs)
         @gridFunction(view)
         def uh(e,x):
             indices = space.mapper(e)
             phiVals = space.evaluateLocal(x)
             localDofs = dofs[indices]
             return numpy.dot(localDofs, phiVals)
         uh.plot(level=1)
         view.hierarchicalGrid.globalRefine(2)
```

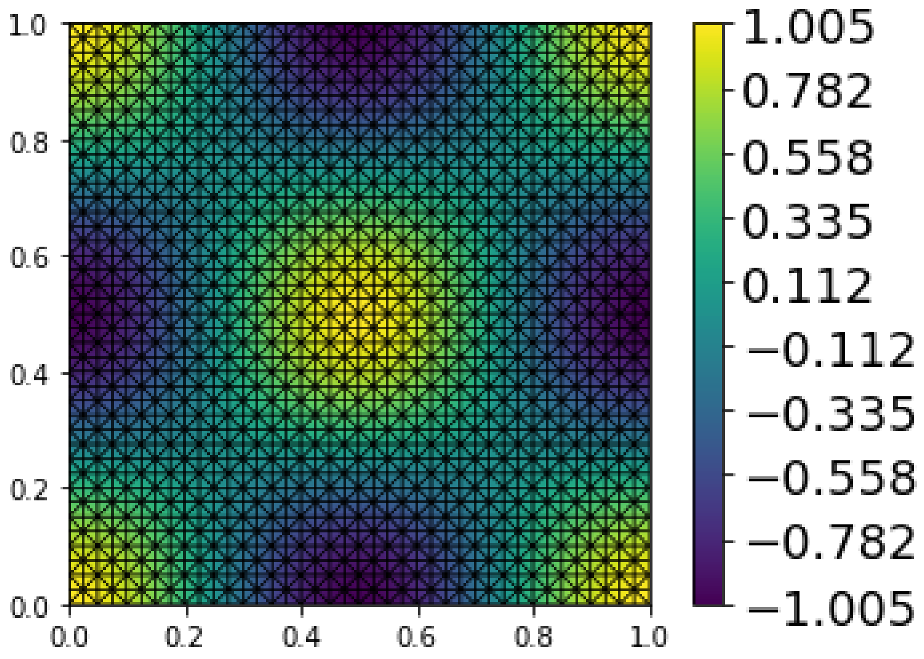
```
number of elements: 200 number of dofs: 121
```



```
number of elements: 800 number of dofs: 441
```



```
number of elements: 3200 number of dofs: 1681
```



2 Some tasks:

1. Compute some error of the projection, i.e., in maximum difference between uh and u at the center of each element, or some L^2 type error over the domain, e.g.,

$$\sqrt{\int_\Omega |u - u_h|^2}$$

What are the EOCs?

Experimental order of convergence: (EOC)

This is a simple proceedure to test if a numerical scheme works:

Assume for example that one can prove that the error e_h on a grid with grid spacing h satisfies

$$e_h \approx Ch^p$$

then

$$\log \frac{e_h}{e_H} \approx p \log \frac{h}{H}$$

which can be used to get a good idea about the convergence rate p using the errors computed on two different levels of a hierarchical grid.

2. Implement an interpolation (e.g. as a method on the `LinearLagrangeSpace` class) and compare the errors of the interpolation with the errors/EOCs you computed for the projection.

3. Add a class with quadratic finite elements and look at the errors/EOCs.
4. Have a look at the errors/EOCs in the derivatives:

$$\sqrt{\int_\Omega |\nabla u - \nabla u_h|^2}$$

   Recall how the local basis functions are defined and use the chain rule, the required method on the element's geometry is `jacobianInverseTransposed(x)`...

5. Add a stiffness matrix and solve the Neuman problem:

$$\begin{aligned} -\Delta u + u &= f, && \text{in } \Omega, \\ \nabla u \cdot n &= 0, && \text{on } \partial\Omega, \end{aligned}$$

   where f is for example given by

```
@gridFunction(view)
def forcing(p):
    return u(p)*(2*(2*numpy.pi)**2+1)
```

Solution: this is implemented in `laplaceNeumann.py`

6. change the PDE, i.e., include varying coefficients...