spuq Documentation

Release 0.1alpha

spuq Development Team

October 07, 2011

CONTENTS

Contents:

CHAPTER

ONE

TUTORIAL

1.1 First steps

Blah blah blah

CHAPTER

TWO

CONCEPTS

2.1 Bases and operators

Bases and operators form the two closely related central concepts of spuq. This if for several reasons:

- Refinement of a basis: gives you a new basis and operators that restrict or project vectors from one basis to the other
- Operators on restricted base can be formed by composition of the operator with restriction or projection operator
- Many complicated bases as tensor product bases of simpler bases
- Forming of subbases, which is more or less coarsening
- Subbases of tensor product bases indexed by multiindex sets can give PCE bases
- Function bases: can be evaluated, can be orthogonalised via their Gram matrix, for TP bases via the tensor product of those matrices
- Simple construction: non-orthogonal basis with Gram matrix to orthogonal basis

Operators:

- Compositions can be formed efficiently
- Note that *efficiently* here means: if the vectors that the operators act on are large so that the significant portion of the runtime is spent in some matrix-vector operation

Advantages:

- a vector is not just a collection of numbers, you always know where what the numbers *mean*, since you always have the basis with it
- with the basis alongside you can compute norms in the real space instead of just the Euclidean
- you can restrict and project
- you cannot easily mess up since the operators check that the vectors come from their domain

2.1.1 Some mathematical notation

We have

$$H^1(\mathcal{D}; L_2(\Omega)) \simeq H^1(\mathcal{D}) \otimes L_2(\Omega) \tag{2.1}$$

where $H^1(\mathcal{D})$ is the Sobolev space ...

2.1.2 An example

Some demonstration of operator stuff:

```
A = FullLinearOperator( 1 + rand(3, 5) )
B = FullLinearOperator( 1 + rand(7, 3) )
print A.domain_dim(), A.codomain_dim()
print B.domain_dim(), B.codomain_dim()

x = FullVector( rand( 5,1 ) )
print x
```

Operators can be multiplied:

```
C = B * A
print C.domain_dim(), C.codomain_dim()
```

operator composition can be performed in a number of ways:

```
print B(A(x))
print (B * A)(x)
print B * A * x
print B * (A * x)
print (B * A) * x
```

similar as above, only as matrices:

```
print (B*A).as_matrix() * x.as_vector()
print B.as_matrix() * (A.as_matrix() * x.as_vector())
```

you can transpose (composed) operators:

```
AT=A.transpose()
BT=B.transpose()
CT=C.transpose()

y = FullVector( rand( CT.domain_dim(),1 ) )
print CT*y
print AT*(BT*y)
```

can add and subtract operators:

```
print (B * (A+A))*x
print C*x+C*x
print (C-C)*x
print C*x-C*x
```

you can pre- and post-multiply vectors with scalars:

```
print 3*x-x*3
```

you can multiply operators with scalars or vectors with scalars:

```
print (3*C)*x
print (C*3)*x
print 3*(C*x)
```

Note: This example was copied verbatim from test-zone/operators/test-operator-algebra.py

CHAPTER

THREE

CONCEPT FOR THE IMPLEMENTATION OF THE RESIDUAL BASED ERROR ESTIMATOR

In this document all necessary formulas for the implementation of the residual-based a posteriori error estimator for the SGFEM method proposed by C. Gittelson and Ch. Schwab shall be collected.

Basic guidelines for this document

- all necessary formulas shall be collected, such that a complete workflow could be implemented from it
- math formulas should all be supplemented by code sketches
- the prose need not be nice, better complete

3.1 Model setting

Elliptic boundary value problem

$$-\nabla(a \cdot \nabla u) = f \text{ in } D \tag{3.1}$$

Homogeneous Dirichlet boundary conditions $u = 0$ on D

D is a Lipschitz domain. For the application we will restrict this to a rectangular domain, maybe simply $[0, 1]^2$, ok? Definition in FEniCS will look like:

```
# create mesh and define function space
mesh = UnitSquare(1, 1)
V = FunctionSpace(mesh, 'CG', 1)
```

3.1.1 Coefficient

The coefficient has the form

$$a(y, x) = \bar{a}(x) + \sum_{m=1}^{M} y_m a_m(x)$$

Note: How do we define the functions $a_m(x)$ on the FunctionSpace object V in FEniCS how to we embed this into a spuq Basis object?

What kind of functions shall we take for the a_m first? Piecewise? Trigonometric? What about $\bar{a}$? Constant?

Specification of the random variables y_m. Need to be defined on $[-1, 1]$ with $\#supp(y_m) = \infty$. Take simply uniform $U(-1, 1)$? Needs to be symmetric?

3.1.2 Operator

Continuous operator

$$\bar{A} = -\nabla \bar{a}(x) \cdot \nabla$$

$$\bar{\mathcal{A}} = \mathrm{Id} \otimes \bar{A}$$

$$A_m = -\nabla a_m(x) \cdot \nabla$$

$$K_m = r(y) \mapsto r(y) y_m$$

$$\mathcal{A}_m = K_m \otimes A_m$$

$$\mathcal{A} = \bar{\mathcal{A}} + \sum_{m=1}^{\infty} \mathcal{A}_m \tag{3.2}$$

Discrete operator

For each μ we have some V_μ

solution has the form

$$w(y, x) = \sum_{\mu \in \Lambda} w_\mu(x) P_\mu(y)$$

ordered basis $B_\mu = \{b_{\mu,i}\}$ of V_μ

$$w(y, x) = \sum_{\mu \in \Lambda} \sum_{i=1}^{\#B_\mu} w_{\mu,i} b_{\mu,i}(x) P_\mu(y)$$

for each vector $w_\mu = [\ldots w_{\mu,i} \ldots]^T$ of length $\#B_\mu$ we discretise equation (3.2).

Evaluation of the discrete operator A_m:

```
A = MultiOperator( a, rvs )
P = PDE()
def MultiOperator.apply( w ):
    beta = rvs.get_orthogonal_poly().monic_coeffs
    v = MultiVector()
    delta = w.active_set()
    for mu in delta:
        A0 = P.assemble( a[0], w[mu].basis )
        v[mu] = A0 * w[mu]
        for m in xrange(1,100):
            Am = P.assemble( a[m], w[mu].basis )
            mu1 = mu.add( (m,1) )
            if mu1 in Delta:
                v[mu] += Am * beta(m, mu[m] + 1) * w[mu1].basis.project(w[mu].basis.mesh,INTERPOLATE
            mu2 = mu.add( (m,-1) )
            if mu2 in Delta:
                v[mu] += Am * beta(m, mu[m]) * w[mu2].basis.project(w[mu].basis.mesh,INTERPOLATE)
    return v
```

3.2 Algorithms

3.2.1 Solve algorithm

Solve algorithm:

```
def solve( eps, w0, eta0, chi ):
  w=w0; eta=eta0;
  for i in xrange(1,):
    [w,zeta]=pcg( w, chi*xi )
    (eta,eta_S)=error_estimator( w, zeta )
    if eta<=eps:
      return w
    w=refine(w,eta_S)
```

Identification of variables:

- `eps` = ϵ, threshold for the total estimated error
- `w0` = w_N^0, initial solution, is a collection of multiindices with associated vectors that include the basis used for this multiindex; the parameter $\mathcal{V}^{1 or 0}$ is included in `w0`
- `xi0` = ξ^0 error bound of the initial solution (?), estimate $\xi^0 := (1-\gamma)^{-1/2}\|f\|_{V^*}$ (see note 3)
- `chi` = χ parameter that determines the accuracy of the solver; between 0 and 1 (exclusive)

Note: maybe we can pass ζ^0 instead of ξ^0 and compute ξ^0 using the error estimator, i.e. swapping lines 2 and 3 of the algorithm

Note: why does $\mathcal{V}$ have a different index than w in the paper; should be the same

Note: we rename ξ to η; further the error estimator returns also the local error, not only the global one

3.2.2 Error estimator

Definitions

The residual error estimator follows from a partial integration of the residual

$$\langle r_\mu(w_N), v\rangle = \int_D f\delta_{\mu 0} - \sigma_\mu(w_N)\cdot\nabla v\,dx, \quad v\in H^1_0(\Omega),$$

for some given approximation $w_N \in \mathcal{V}_N$.

The flux σ_μ for $\mu\in\Lambda$ is defined by

$$\sigma_\mu(w_N) := \bar{a}\nabla w_{N,\mu} + \sum_{m=1}^{\infty} a_m \nabla(\beta^m_{\mu_m+1}\Pi_\mu^{\mu+\epsilon_m} w_{N,\mu+\epsilon_m} + \beta^m_{\mu_m}\Pi_\mu^{\mu-\epsilon_m} w_{N,\mu-\epsilon_m}).$$

We have to evaluate the volume and edge contributions in elements $T\in\mathcal{T}_\mu$ and on edges $S\in\mathcal{S}_\mu$ of the error estimator,

$$\begin{aligned}\eta_{\mu,T}(w_N) &:= h_T\|\bar{a}^{-1/2}(f\delta_{\mu 0} + \nabla\cdot\sigma_\mu(w_N))\|_{L^2(T)}\\ \eta_{\mu,S}(w_N) &:= h_S^{1/2}\|\bar{a}^{-1/2}[\sigma_\mu(w_N)]_S\|_{L^2(S)}\end{aligned}$$

These sum up to the total error estimator

$$\eta_\mu(w_N) := \left(\sum_{T\in\mathcal{T}_\mu}\eta_{\mu,T}(w_N)^2 + \sum_{S\in\mathcal{S}_\mu}\eta_{\mu,S}(w_N)^2\right)^{1/2}.$$

Note that for conforming piecewise affine approximations (i.e. continuous linear elements) the divergence of σ_μ simplifies to

$$\nabla\cdot\sigma_\mu(w_N) = \nabla\bar{a}\cdot\nabla w_{N,\mu} + \sum_{m=1}^{\infty}\nabla a_m\cdot\nabla(\beta^m_{\mu_m+1}\Pi_\mu^{\mu+\epsilon_m} w_{N,\mu+\epsilon_m} + \beta^m_{\mu_m}\Pi_\mu^{\mu-\epsilon_m} w_{N,\mu-\epsilon_m}).$$

Algorithm for the evaluation of σ_μ:

```
w = MultiVector()
m = MultiIndex( (...) )
T0 = IntialMesh()
w[m] = FenicsVector(T0)
# sigma_mu
# a = (Function, Function, Function, ... )
newDelta = extend(Delta)
for mu in newDelta:
    sigma_x = a[0]( w[mu].mesh.nodes ) * w[mu].dx()
    for m in xrange(1,100):
        mu1 = mu.add( (m,1) )
        if mu1 in Delta:
            sigma_x += a[m]( w[mu].mesh.nodes ) * beta(m, mu[m]+1) *\
                        w[mu1].project( w[mu].mesh ).dx()
        mu2 = mu.add( (m,-1) )
        if mu2 in Delta:
            sigma_x += a[m]( w[mu].mesh.nodes ) * beta(m, mu[m]) *\
                        w[mu2].project( w[mu].mesh ).dx()
```

The function `error_estimator`:

```
def error_estimator( w, zeta, c_eta, c_Q ):
```

Projection $\Pi_\mu^\nu : V_\nu \to V_\mu$ for some $\mu, \nu \in \Lambda$ can be an arbitrary map such as the L^2-projection, the $\mathcal{A}$-orthogonal projection or nodal interpolation.

3.2.3 Refinement

The marking/refinement procedure is three-fold:

1. (for active indices $\mu \in \Lambda$) evaluation of the residual error estimator $\hat{\eta}_{\mu,S}(w_N)$ and edge marking of respective FEM meshes $\mathcal{T}_\mu$
2. (for active indices and their *neighbourhood*) estimation of the projection errors and marking of respective FEM meshes
3. activation of new indices based on the projection estimation of 2.

FEM residuals

We employ an edge-based Dörfler marking strategy for all edges $S \in \mathcal{S}_\mu$ with the edge indicator

$$\hat{\eta}_{\mu,S} := \left(\eta_{\mu,S}(w_N)^2 + \frac{1}{d+1} \sum_{T:\ S \in \mathcal{S}_\mu \cap \partial T} \eta_{\mu,T}(w_N)^2 \right)^{1/2}$$

such that, for some parameter $0 < \vartheta_\eta < 1$, a set

$$\hat{\mathcal{S}}_\mu \subset \bigcup_{\mu \in \Lambda} \{\mu\} \times \mathcal{S}_\mu$$

of small cardinality is obtained for which holds

$$\sum_{(\mu,S) \in \hat{\mathcal{S}}_\mu} \hat{\eta}_{\mu,S}^2 \geq \vartheta_\eta^2 \sum_{\mu \in \Lambda} \eta_\mu(w_N)^2.$$

Let $\mathcal{T}_N := \bigcup_{\mu \in \Lambda} \{\mu\} \times \mathcal{T}_\mu$ encode the set of all elements of all meshes paired with the respective multiindex μ, i.e. for all element $T \in \mathcal{T}_\mu$ for any $\mu \in \Lambda$ there is a tuple $(\mu, T) \in \mathcal{T}_N$.

Let $\mathcal{T}_\eta \subset \mathcal{T}_N$ be the subset of elements which have at least one edge in $\hat{\mathcal{S}}_\mu$ and mark these elements for refinement.

Projection errors

TODO

Activation of new indices

TODO

3.2.4 PCG

This should be implemented as a standard preconditioned conjugate gradient solver, where the special treatment necessary for the specific structure of w_N is hidden in a generalised vector class (`FEMVector`) that takes care of that.

Meaning of the variables

- ρ = `r` residual
- s = `s` preconditioned residual
- v = `v` search direction
- w = `w` solution
- ζ is the enery norm (w.r.t. $\bar{\mathcal{A}}$) of the preconditioned residual s, i.e. $\|s\|^2_{\bar{\mathcal{A}}}$

Algorithm:

```
def pcg( A, A_bar, w0, eps ):
  # use forgetful_vector for vectors
  w[0] = w0
  r[0] = f - apply(A, w[0])
  v[0] = solve(A_bar, r[0])
  zeta[0] = r[0].inner(s[0])
  for i in count(1):
    if zeta[i-1] <= eps**2:
      return (w[i-1], zeta[i-1])
    z[i-1] = apply(A, v[i-1])
    alpha[i-1] = z[i-1].inner(v[i-1])
    w[i] = w[i-1] + zeta[i-1] / alpha[i-1] * v[i-1]
    r[i] = r[i-1] - zeta[i-1] / alpha[i-1] * z[i-1]
    s[i] = solve(A_bar, r[i])
    zeta[i] = r[i].inner(s[i])
    v[i] = s[i] - zeta[i] / zeta[i-1] * v[i-1]
```

3.3 Data structures

3.3.1 Vectors

Sketch for the generalised vector class for `w` which we call `MultiVector`:

```
class MultiVector(object):
  #map multiindex to Vector (=coefficients + basis)
  def __init__(self):
    self.mi2vec = dict()

  def extend( self, mi, vec ):
    self.mi2vec[mi] = vec

  def active_indices( self ):
    return self.mi2vec.keys()

  def get_vector( self, mi ):
    return self.mi2vec[mi]

  def __add__(self, other):
    assert self.active_indices() == other.active_indices()
    newvec = FooVector()
    for mi in self.active_indices():
      newvec.extend( mi, self.get_vector(mi)+other.get_vector(mi))
    return newvec

  def __mul__():
    pass

  def __sub__():
    pass
```

The `MultiVector` needs a set of *normal* vectors which represent a solution on a single FEM mesh:

```
class FEMVector(FullVector):
  INTERPOLATE = "interpolate"

  def __init__(self, coeff, basis ):
    assert isinstance( basis, FEMBasis )
    self.FullVector.__init__(coeff, basis)

  def project(self, basis, type=FEMVector.INTERPOLATE):
    assert isinstance( basis, FEMBasis )
    newcoeff = FEMBasis.project( self.coeff, self.basis, basis, type )
    return FEMVector( newcoeff, basis )
```

The `FEMVector``s need a basis which should be fixed to a ``FEMBasis` and derivatives (which could be a FEniCS or dolfin basis or whatever FEM software is underlying it):

```
class FEMBasis(FunctionBasis):
  def __init__(self, mesh):
    self.mesh = mesh

  def refine(self, faces):
    (newmesh, prolongate, restrict)=self.mesh.refine( faces )
    newbasis = FEMBasis( newmesh )
    prolop = Operator( prolongate, self, newbasis )
    restop = Operator( restrict, newbasis, self )
    return (newbasis, prolop, restop)

  @override
  def evaluate(self, x):
    # pass to dolfin
    pass

  @classmethod
  def project( coeff, oldbasis, newbasis, type ):
    # let dolfin do the transfer accoring to type
    pass
```

The FEMBasis needs a mesh class for refinement and transfer of solutions from one mesh to another. This mesh shall have derived class that encapsulat specific Mesh classes (that come e.g. from Dolfin)

```
# in spuq.fem?
class FEMMesh( object ):
  def refine( self, faces ):
    return NotImplemented

# in spuq.adaptors.fenics
class FenicsMesh( FEMMesh ):
  def __init__(self):
    from dolfin import Mesh
    self.fenics_mesh = Mesh()

  def refine( self, faces ):
    new_fenics_mesh = self.fenics_mesh.refine(faces)
    prolongate = lambda x: fenics.project( x, fenics_mesh,
                                           new_fenics_mesh )
    restrict = lambda x: fenics.project( x, new_fenics_mesh,
                                         fenics_mesh )
    return (Mesh( new_fenics_mesh ), prolongate, restrict)
```

Refinement:

```
b0 = FEMBasis( FEniCSMesh() )
coeffs = whatever()
v0 = FEMVector( coeffs, b0 )
faces = marking_strategy( foo )
(b1, prol, rest) = b0.refine( faces )
v1 = prol( v0 )
assert v1.get_basis() == b1
assert v1.__class__ == v2.__class__
```

3.4 Questions

- What kind of requirements are there for the projectors Π_μ^ν?

CHAPTER

FOUR

WEBLINKS FOR DEVELOPMENT

This file contains links that aid in the development of spuq. This should not be a general link collection but should only contain links that are often needed to look up stuff with out asking google again and again for the same stuff. Links that are included should be:

4.1 Coding references

4.1.1 Programming resources

Python

Libraries

- numpy reference: http://docs.scipy.org/doc/numpy/reference/
- numpy user guide: http://docs.scipy.org/doc/numpy/user/
- scipy reference: http://docs.scipy.org/doc/scipy/reference/
- numpy/scipy cookbook: http://www.scipy.org/Cookbook
- Traits user manual: http://github.enthought.com/traits/traits_user_manual/index.html

4.1.2 Version control

- git resources from numpy: http://docs.scipy.org/doc/numpy/dev/gitwash/git_resources.html#git-resources
- git cheat sheet: http://help.github.com/git-cheat-sheets/
- the git book: http://book.git-scm.com/index.html

4.1.3 Writing documentation

Sphinx

- Sphinx: http://sphinx.pocoo.org/contents.html
- autodoc: http://sphinx.pocoo.org/ext/autodoc.html

reStructuredText

- Quickref: http://docutils.sourceforge.net/docs/user/rst/quickref.html
- Wikipedia: http://en.wikipedia.org/wiki/ReStructuredText
- Wikipedia.de: http://de.wikipedia.org/wiki/ReStructuredText
- Emacs mode: http://docutils.sourceforge.net/docs/user/emacs.html

4.2 Scientific References

4.2.1 Quadrature

4.2.2 Orthogonal polynomials

CHAPTER

FIVE

PACKAGE SPUQ.LINALG

The basic linear algebra classes of spuq, abstracting the concepts of bases, vectors, and operators.

5.1 Basis

class `spuq.linalg.basis.`**`Basis`**

Bases: `object`

Abstract base class for basis objects

`dim`

The dimension of this basis.

exception `spuq.linalg.basis.`**`BasisMismatchError`**

Bases: `exceptions.ValueError`

class `spuq.linalg.basis.`**`CanonicalBasis`**`(`*dim*`)`

Bases: `spuq.linalg.basis.Basis`

`dim`

class `spuq.linalg.basis.`**`FunctionBasis`**

Bases: `spuq.linalg.basis.Basis`

`gramian`

The Gramian as a LinearOperator (not necessarily a matrix)

`spuq.linalg.basis.`**`check_basis`**`(`*basis1*, *basis2*, *descr1='basis1'*, *descr2='basis2'*`)`

Throw if the bases do not match

5.2 Vector

5.3 Operator

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX