Functional Convergence

API Documentation

December 21, 2009

Contents

1 Package FunctionalConvergence

Usage This package contains several tools to facilitate calculation (solution) verification. The main entry point for the user is the functionalConvergence module. It contains classes for Functional Convergence analysis on time history and field variable data. For scalar data, you can use the ansatz and multiAnsatz modules directly. The other modules serve as libraries for these user interface modules.

Author: Trevor Tippetts

trevorbtippetts_at_gmail.com

1.1 Modules

2 Module FunctionalConvergence.ansatz

Use Ansatz instances to fit an error ansatz to results from multiple meshes.

The most commonly used ansatz uses three parameters, so a good rule of thumb is that you should use at least four meshes with the finest refined to a factor of four relative to the coarsest. With four meshes you can get information on the robustness and quality of fit (see the multiAnsatz module) to test the asymptotic assumption. A factor of four in refinement gives you a minimal spread in the refinement parameter without being too demanding of your computational resources.

Not every quantity will necessarily converge, and quantities you believe should converge might be difficult to prove on realistic simulations. Conserved quantites (energy, momentum, etc.) and/or quanties averaged over extensive times or regions will generally be easier to fit to an ansatz, but your mileage may vary.

2.1 Class Ansatz

```
object ─┐
        FunctionalConvergence.ansatz.Ansatz
```

Known Subclasses: FunctionalConvergence.multiAnsatz.MultiAnsatz

With the number of ys equal to the number of parameters, initial values should be close enough.

```
>>> ys = [4.0201145565477141,4.1143169371042969,4.2449576507624398]
>>> xs = [1.,.5,.25]
>>> a = Ansatz(residualFunc=Ansatz.singleExpExtrapRes,
...            initParams=Ansatz.singleExpExtrapInit)
>>> a.fit(sampleXs=xs,sampleYs=ys)
([4.2023968794997719, 0.1822823229520581, 1.0492892305815129], 0, 'Initial values are close enough.', 2.484449
```

ys that are constant across discretizations return an exponent of 0.

```
>>> ys = [1.0,1.0,1.0]
>>> a.fit(sampleXs=xs,sampleYs=ys)
([1.0, 0.0, 0.0], 0, 'Initial values are close enough.', 0.0)
```

fit error for len(xs) > 3

```
>>> xs = [1/8.,1/4.,1/2.,1.]
>>> ys = [1-(1+e*.01)*x**2 for x,e in zip(xs,[1,-1,1,-1])]
```

Be careful! Sometimes you have to crank down tolerances and it will complain even when the answer looks good.

- Always check the residual (last item returned).
- > 3 meshes and non-constant refinement ratio are particularly ticklish.

for Linear search failed, raise pgtol for Local minima reach, lower pgtol

```
>>> a.fit(sampleXs=xs,sampleYs=ys)
(array([ 1.00169425,  0.99175158,  1.96691864]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 2.7789767689774468e-06)
>>> # ignore failure warning for this test
>>> import warnings; warnings.filterwarnings('ignore')
>>> a.fit(sampleXs=xs,sampleYs=ys,ftol=1e-30,
...       pgtol=1e-30,xtol=1e-30,eta=.01,maxfun=1000)
(array([ 1.00169427,  0.99175158,  1.96691866]), 4, 'Linear search failed', 2.7789767651060615e-06)
>>> warnings.resetwarnings()
```

fit with non-constant refinement ratio

```
>>> xs = (1.,2./3,.5)
>>> ys = [100.-.42*x**2 for x in xs]
>>> a.singleExpExtrapInit(xs,ys)
[99.999996623073088, 0.41999724196876892, 2.0000284218807081]
>>> a.fit(sampleXs=xs,sampleYs=ys,maxfun=200)
(array([ 99.99999688,   0.41999696,   2.0000282 ]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 2.346339438805061e-13)
>>> ys = [100.-.42*e*x**2 for x,e in zip(xs,[1.01,.99,1.01])]
>>> a.singleExpExtrapInit(xs,ys)
[99.98467604143157, 0.40887605808077765, 2.1720747055310641]
>>> a.fit(sampleXs=xs,sampleYs=ys,maxfun=200)
(array([ 99.98467603,   0.40887603,   2.17207471]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 4.5825600971694034e-16
```

with an exponent range, it will use an alternate solution if possible

```
>>> warnings.filterwarnings('ignore')
>>> a.fit((.125,.25,1.),(4.55,5.,6.))
(array([ -9.00741303,  15.00403311,   0.04902193]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 0.00018502633096050865
>>> a.fit((.125,.25,1.),(4.55,5.,6.),exponentRange=(.5,3.))
(array([ 4.69646863,  1.30353136,  1.05125387]), 0, 'Local minima reach (|pg| ~= 0)', 2.123639831442457e-16)
>>> a.fit((.125,.25,1.),(4.55,5.,6.),exponentRange=(.5,1.))
(array([ -9.00741303,  15.00403311,   0.04902193]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 0.00018502633096050865
```

multiple initializations for ¿ 3 meshes

```
>>> xs = [1/8.,1/4.,1/2.,1.]
>>> ys = [1-(1+e*.01)*x**2 for x,e in zip(xs,[1,-1,1,-1])]
>>> ys[2] = ys[2]*1.05
>>> a.fit(sampleXs=xs,sampleYs=ys)
(array([ 0.98841663,  0.9782003 ,  2.24868553]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 7.8184950976487521e-05)
>>> # multiple sets of initial parameters might give a better fit
>>> a.multiInitFit(sampleXs=xs,sampleYs=ys,exponentRange=(.5,3.))
(array([ 0.97648645,  0.9664712 ,  2.3334401 ]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 1.5436006041945928e-07)
>>> # should give same results for 3 meshes as .fit
>>> a.multiInitFit((.125,.25,1.),(4.55,5.,6.),exponentRange=(.5,3.))
(array([ 4.69646863,  1.30353136,  1.05125387]), 0, 'Local minima reach (|pg| ~= 0)', 2.123639831442457e-16)
```

error checking

```
>>> a.fit(sampleXs=(1/8.,1/2.,1.),sampleYs=(.001,.0007,.00035))
(array([  1.13161387e-03,   7.81613870e-04,   8.56714844e-01]), 0, 'Local minima reach (|pg| ~= 0)', 1.19103614
```

From the truchas paper: Some old applications supplied their own initial values. - This can cause problems, so be careful and you probably want to use the builtin init functions

```
>>> dts = [float(dt) for dt in (1,2,5,10,20,30,40,50,60)]
>>> T = [9.643783E+02,9.643741E+02,9.643630E+02,9.643448E+02,9.643030E+02,
...      9.642649E+02,9.642279E+02,9.641911E+02,9.641540E+02]
>>> y = [T[i] for i in (3,4,6)]
>>> h = [dts[i] for i in (3,4,6)]
>>> q = 1.; dtNom = dts[3]
>>> A = abs((y[0]-y[1])/((h[1]/dtNom)**q-(h[0]/dtNom)**q))/y[2]
>>> ye = 1.-A*(h[2]/dtNom)**q
>>> a = Ansatz(residualFunc=Ansatz.singleExpExtrapRes,initParams=[ye,A,q])
>>> # should be 1st order; .286 instead is from poor initParams
>>> a.fit([dt/dtNom for dt in dts[:7]],[x/y[2] for x in T[:7]],fExtrap=ye)
(array([  1.00011621e+00,   4.80001721e-05,   2.85633603e-01]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 4.90369850
>>> a.singleExpExtrapInit([dt/dtNom for dt in dts[:7]],
...                       [x/y[2] for x in T[:7]]) # really good defaults
[1.000159910027262, 3.9722658980350821e-05, 1.0046131086954433]
>>> a2 = Ansatz(residualFunc=Ansatz.singleExpExtrapRes,
...             initParams=Ansatz.singleExpExtrapInit)
>>> a2.fit([dt/dtNom for dt in dts[:7]],[x/y[2] for x in T[:7]])
(array([  1.00015980e+00,   3.99742036e-05,   1.00461311e+00]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 7.10640873

>>> dxs = micaDxs = [1./x for x in (4,3,2,2,2)]; dxNom = micaDxs[-3]
>>> T1 = [790.8434373825589,788.0679925452705,782.1237357637906,782.3876600007129,782.483141975275
>>> T21 = [791.9544905065773,789.2548185980995,783.5936178391257,783.6546814084356,783.78572373190
>>> # some fits that seem like they should be easy require many iterations
>>> a2.fit([dx/dxNom for dx in dxs[:3]],T1[:3])
(array([  8.09624681e+02,   2.74305825e+01,   5.62551696e-01]), 3, 'Max. number of function evaluations reach'
>>> a2.fit([dx/dxNom for dx in dxs[:3]],T1[:3],maxfun=701)
(array([ 797.62054679,   15.49678953,    1.19323851]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 8.6192170700292221e
>>> # be careful if you normalize; it might require tightening tolerances
>>> a2.fit([dx/dxNom for dx in dxs[:3]],T21[:3])
(array([ 798.95427929,   15.36066139,    1.13385707]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 6.7291758502407672e
>>> a2.fit([dx/dxNom for dx in dxs[:3]],[x/T21[2] for x in T21[:3]])
(array([ 1.03459401,  0.0345059 ,  0.53731606]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 1.7515993161923577e-07)
>>> a2.fit([dx/dxNom for dx in dxs[:3]],[x/T21[2] for x in T21[:3]],ftol=1e-13,pgtol=1e-8,maxfun=27
(array([ 1.01959539,  0.01959547,  1.13447634]), 1, 'Converged (|f_n-f_(n-1)| ~= 0)', 1.695847972415786e-13)
```

2.1.1 Methods

__init__(*self*, *residualFunc*=None, *initParams*=None, *jacobian*=None)

residualFunc function that returns a residual to minimize

initParams may be either a sequence of initial values or a callable returning initial values given a sequence of data to fit.

jacobian optional; it might improve computational performance.

Overrides: object.__init__

multiInitFit(*self*, *sampleXs*=`None`, *sampleYs*=`None`, *fExtrap*=`None`, *method*=`'tnc'`, *exponentRange*=`(None, None)`, ****kwargs*)

Performs ansatz fits from multiple initializations.

fit(*self*, *sampleXs*=`None`, *sampleYs*=`None`, *fExtrap*=`None`, *method*=`'tnc'`, *exponentRange*=`(None, None)`, ****kwargs*)

Fits parameter values to the samples, using the instance's residual function and parameter initial values. Extra kwargs get passed to the fitting function.

leastsq(*self*, *params*, *hs*, *es*)

Returns sum of squared residuals, for nonlinear least squares fitting.

singleExpRes(*cls*, *params*, *hs*, *es*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

singleExpLogRes(*cls*, *params*, *hs*, *es*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

This residual uses the log of the ratio, which in theory might be more fair in weighting the fine mesh results. In (limited) practice, it's not clear if either shows a consistent advantage.

singleExpInit(*cls*, *h*, *e*)

Parameter initialization for error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

singleExpExtrapRes(*cls*, *params*, *hs*, *fs*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be field values (extrapolated solution used).

singleExpExtrapInit(*cls*, *hs*, *fs*, *forceGreater*=`False`)

Parameter initialization for error ansatz with a single exponential term in zone size. The y data are assumed to be field values (extrapolated solution used).

R(*self*, *f*, *m*, *c*)

Computes the discriminant, R, for a set of 3 meshes.

paramsFromq(*cls*, *q*, *ys*, *hs*, *sfm*, *scm*)

Computes the prefactor, A, and the extrapolated solution, ye, from the exponent, q.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

2.1.2 Properties

Name	Description
Inherited from object	
__class__	

3 Module FunctionalConvergence.extractor

The purpose of this module and the Extractor parent class is to provide a roughly uniform interface for extracting finite element simulation results from a variety of file formats.

3.1 Variables

Name	Description
vtkTest	**Value:** `'# vtk DataFile Version 3.0\nvtk output\nASCII\nDATASET U...`
vtkTest2	**Value:** `'# vtk DataFile Version 3.0\nvtk output\nASCII\nDATASET U...`
vtkTest3	**Value:** `'# vtk DataFile Version 3.0\nvtk output\nASCII\nDATASET U...`
vtkTest3_0	**Value:** `'# vtk DataFile Version 3.0\nvtk output\nASCII\nDATASET U...`

3.2 Class Extractor

```
object ─┐
        │
    FunctionalConvergence.extractor.Extractor
```

Known Subclasses: FunctionalConvergence.extractor.Abaqus, FunctionalConvergence.extractor.Pickle, FunctionalConvergence.extractor.PyTable

This is a parent class and should not be used directly; see the subclasses for a usable interface.

3.2.1 Methods

association(*self*, *fieldName*)

Fields with values at nodes need to be handled differently from fields with values at elements. This method identifies the association given a field name, based on the subclass attributes _nodeFieldNames and _elementFieldNames.

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

3.2.2 Properties

Name	Description
Inherited from object	
__class__	

3.3 Class Abaqus

```
object ─┐
        │
FunctionalConvergence.extractor.Extractor ─┐
                                           │
                                   FunctionalConvergence.extractor.Abaqus
```

Abaqus odb file results extractor. Time histories and fields use the names in the abaqus odb file. Uses the odbaccess module, so it must be run with abaqus python. Save the results you need under abaqus python; then you can read the pickle in any other python.

Use this command to test this class:

```
time /usr/local/abaqus/Commands/abq675 python ./pyTest.py extractor.Abaqus
```

```
>>> a = Abaqus('../1Dwave/1Dwave.1.odb')
>>> a.scalarField('U',0)[0]
0.038882236927747726
>>> a.timeHistory('ETOTAL')[0][-1], a.timeHistory('ETOTAL')[1][-1]
(9.9999999747524271e-07, -0.077574290335178375)
>>> a.nodes[0]
array([-1.0, -0.5, 0.5], 'd')
>>> a.elements[0]
(1, 2, 3, 4, 5, 6, 7, 8)
>>> a.save('1Dwave.1.pickle',(('U',0),('ETOTAL',)))
```

If you specify an invalid part or give no time history region when there are multiple ones in the odb, it will raise an exception and list the ones that are available.

```
>>> try: Abaqus('../1Dwave/1Dwave.1.odb',part='asdf')
... except KeyError,e: print e
"abaqus file ../1Dwave/1Dwave.1.odb does not have part asdf. available part names:\n             ['PART-1-1']"
>>> #import pdb; pdb.set_trace()
```

3.3.1 Methods

__init__(*self*, *filenameIn*, *part*=`'PART-1-1'`, *step*=`'Step-1'`, *historyRegion*=`None`)

Currently this can only handle one abaqus part and step at a time.
Overrides: object.__init__

---

timeHistory(*self*, *historyName*, *historyComponent*=`0`, *historyRegion*=`None`)

Extract a time history from the abaqus odb.

historyComponent meaningless; it is only here to keep the same signature as other classes' timeHistory method.

---

association(*self*, *fieldName*)

Unlike other Extractors, this one can get association from the odb. Overrides: FunctionalConvergence.extractor.Extractor.association

---

scalarField(*self*, *fieldName*, *fieldComponent*, *frameNumber*=`-1`)

Extract a scalar field from the abaqus odb.

The name might be misleading, since you can pick one component from a vector or tensor field.

---

nodeLocations(*self*)

List of Node coordinates.

elementConnectivity(*self*)

List of ordered nodes for each element.

exportPyTables(*self*, *filename*=`None`, *results*=`None`)

Not yet implemented.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.extractor.Extractor(Section 3.2)

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

3.3.2 Properties

Name	Description
Inherited from object	
__class__	

3.4 Class Pickle

object ┐

FunctionalConvergence.extractor.Extractor ┐

FunctionalConvergence.extractor.Pickle

Known Subclasses: FunctionalConvergence.extractor.ParadynPickle, FunctionalConvergence.extractor.Vtk

Other types of extractors (eg., abaqus) can save their data in this format to avoid future library dependencies.

```
>>> import StringIO
>>> p = Pickle('1Dwave.1.pickle')
>>> p.nodes[0]
[-1.0, -0.5, 0.5]
>>> p.elements[0]
(1, 2, 3, 4, 5, 6, 7, 8)
>>> p.scalarField('U',0)[0]
0.038882236927747726
>>> p.timeHistory('ETOTAL')[0][-1], p.timeHistory('ETOTAL')[1][-1]
(9.9999999747524271e-07, -0.077574290335178375)
>>> p.export(filename='1Dwave.1',fields=(('U',0),),type='vtk')
>>> s = StringIO.StringIO()
```

can write itdf files for embedding as a 3d object in pdf files

```
>>> #p.idtf(s,fields=('test',zip(*p.nodes)[0]),nColors=8); s.seek(0); s.read()
>>> p.idtf(file('displacement.1.idtf','w'),fields=('U',0),nColors=256)
```

3.4.1 Methods

__init__(*self*, *fileIn*=`None`)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

scalarField(*self*, *fieldName*, *fieldComponent*, *frameNumber*=`None`)

Fields and time histories are stored in the same way by the Pickle Extractor. Therefore, frameNumber can also be the history region. But both methods are provided for convience and intelligibility.

timeHistory(*self*, *historyName*, *historyComponent*=`0`, *historyRegion*=`None`)

historyRegion the abaqus history region, if specified. Can also use `*` as a very simple quasi-wildcard

export(*self*, *filename*, *fields*=`None`, *type*=`'vtk'`)

Save file for visualizing scalar node-based fields. Currently only vtk format is supported.

Has a pretty ugly interface right now -- you have been warned.

idtf(*self*, *fileIn*=`None`, *filename*=`None`, *fields*=`None`, ***kwargs*)

Export a scalar field as an IDTF file. This file can then be converted to a U3D file with the IDTFConverter.exe program, and the U3D can be embedded in a PDF document.

kwargs passed through to idtf.IDTF

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.extractor.Extractor(Section 3.2)

association(*self*, *fieldName*)

Fields with values at nodes need to be handled differently from fields with values at elements. This method identifies the association given a field name, based on the subclass attributes _nodeFieldNames and _elementFieldNames.

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

3.4.2 Properties

Name	Description
Inherited from object __class__	

3.5 Class ParadynPickle

```
object ─┐
FunctionalConvergence.extractor.Extractor ─┐
          FunctionalConvergence.extractor.Pickle ─┐
                                                  FunctionalConvergence.extractor.ParadynPickl
```

Adapted to use time history data from paradyn, generated and pickled by miles buechler.

3.5.1 Methods

__init__(*self*, *filename*=`None`, *velocities*=`()`)

---

velocities *Deprecated. Use the methods in functionalConvergence.TimeHistoryFunctionalConvergence instead.* Iterable of strings identifying which velocities you would like numerically differentiated to obtain accelerations.

Overrides: object.__init__

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.extractor.Pickle(Section 3.4)

export(*self*, *filename*, *fields*=`None`, *type*=`'vtk'`)

Save file for visualizing scalar node-based fields. Currently only vtk format is supported.

Has a pretty ugly interface right now -- you have been warned.

idtf(*self*, *fileIn*=`None`, *filename*=`None`, *fields*=`None`, ***kwargs*)

Export a scalar field as an IDTF file. This file can then be converted to a U3D file with the IDTFConverter.exe program, and the U3D can be embedded in a PDF document.

kwargs passed through to idtf.IDTF

scalarField(*self*, *fieldName*, *fieldComponent*, *frameNumber*=`None`)

Fields and time histories are stored in the same way by the Pickle Extractor. Therefore, frameNumber can also be the history region. But both methods are provided for convience and intelligibility.

timeHistory(*self*, *historyName*, *historyComponent*=`0`, *historyRegion*=`None`)

historyRegion the abaqus history region, if specified. Can also use `*` as a very simple quasi-wildcard

Inherited from FunctionalConvergence.extractor.Extractor(Section 3.2)

association(*self*, *fieldName*)

Fields with values at nodes need to be handled differently from fields with values at elements. This method identifies the association given a field name, based on the subclass attributes _nodeFieldNames and _elementFieldNames.

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

3.5.2 Properties

Name	Description
Inherited from object	
__class__	

3.6 Class PyTable

```
object ─┐
        │
FunctionalConvergence.extractor.Extractor ─┐
                                           │
                                  FunctionalConvergence.extractor.PyTable
```

Someday i might use pytables to make a smaller, faster database.

3.6.1 Methods

__init__(*self*, *filenameIn*=`None`)

Not yet implemented. Overrides: object.__init__

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.extractor.Extractor(Section 3.2)

association(*self*, *fieldName*)

Fields with values at nodes need to be handled differently from fields with values at elements. This method identifies the association given a field name, based on the subclass attributes _nodeFieldNames and _elementFieldNames.

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

3.6.2 Properties

Name	Description
Inherited from object	
__class__	

3.7 Class Vtk

```
object ─┐
FunctionalConvergence.extractor.Extractor ─┐
          FunctionalConvergence.extractor.Pickle ─┐
                                                  FunctionalConvergence.extractor.Vtk
```

Reads vtk files output by VisIt, the visualization code written at LLNL. The motivation behind writing this was to read field data from paradyn, which has no other convenient way to be read.

```
>>> import StringIO
>>> s = StringIO.StringIO(vtkTest3); s0 = StringIO.StringIO(vtkTest3_0)
>>> v = Vtk(s0,s)
>>> v.nodes
[[-3.0, 0.0, 0.5], [-3.0, 0.0, 0.375], [-3.1666699999999999, 0.0, 0.5], [-3.1666699999999999, 0.0, 0.375], [-3.3333
>>> v.elements
([9, 13, 14, 10, 11, 15, 16, 12], [13, 17, 18, 14, 15, 19, 20, 16], [17, 21, 22, 18, 19, 23, 24, 20], [21, 25, 26,
>>> v._results
{('derived/prin_stress/1', 0): [-6.4112900000000002, -4.2261600000000001, -4.2126000000000001, -4.2126000000000001,
>>> v.association('derived/prin_stress/1')
'element'
```

can save to a pickle

```
>>> p = StringIO.StringIO()
>>> v.save(p,(('derived/prin_stress/1',0),),0)
>>> p.seek(0)
>>> po = Pickle(p)
>>> (po.nodes==v.nodes, po.elements==v.elements, po._results==v._results)
(True, True, True)
```

3.7.1 Methods

__init__(*self*, *fileIn0*, *fileIn*)

The vtk file does not include initial node coordinates, so you need to give two files.

fileIn0 vtk file from initial (undeformed) time

fileIn vtk file from time of interest

Overrides: object.__init__

uniqueInit(*self*, *fileIn0*, *fileIn*)

init needed for profiling code to work.

fileIn0 has the time = 0 node locations. fileIn has the field data at the desired time.

half the time is spent on 'ns =' line, about a third on 'els =' line. surprisingly, only 5% on 'field =' line. big chunk goes to sum(), most of the rest to split(). might be faster if i used real regexps and avoided the sum.

- probably better to have nodes as an array, anyway

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.extractor.Pickle(Section 3.4)

export(*self*, *filename*, *fields*=`None`, *type*=`'vtk'`)

Save file for visualizing scalar node-based fields. Currently only vtk format is supported.

Has a pretty ugly interface right now -- you have been warned.

idtf(*self*, *fileIn*=`None`, *filename*=`None`, *fields*=`None`, ***kwargs*)

Export a scalar field as an IDTF file. This file can then be converted to a U3D file with the IDTFConverter.exe program, and the U3D can be embedded in a PDF document.

kwargs passed through to idtf.IDTF

scalarField(*self*, *fieldName*, *fieldComponent*, *frameNumber*=`None`)

Fields and time histories are stored in the same way by the Pickle Extractor. Therefore, frameNumber can also be the history region. But both methods are provided for convience and intelligibility.

timeHistory(*self*, *historyName*, *historyComponent*=`0`, *historyRegion*=`None`)

historyRegion the abaqus history region, if specified. Can also use `*` as a very simple quasi-wildcard

Inherited from FunctionalConvergence.extractor.Extractor(Section 3.2)

association(*self*, *fieldName*)

Fields with values at nodes need to be handled differently from fields with values at elements. This method identifies the association given a field name, based on the subclass attributes _nodeFieldNames and _elementFieldNames.

save(*self*, *fileOut*=`None`, *results*=`None`, *protocol*=`None`)

Save a copy of some results to a pickle. This method is useful when you want to extract abaqus results or if you just want a smaller file with a subset of the results.

3.7.2 Properties

Name	Description
Inherited from object	
__class__	

4 Module FunctionalConvergence.functionalConvergence

This module is intended to be the main entry point for users. For functional convergence analysis on time history data, see the TimeHistoryFunctionalConvergence class. For functional convergence analysis on field (x,y,z) data, see the FieldFunctionalConvergence class.

The ansatz module documentation also has some general tips for success.

Each class in this package gives some example usage code, but here are some additional examples for running analyses with TimeHistoryFunctionalConvergence and FieldFunctionalConvergence and plotting the results. (Unfortunately, code lines do not wrap in the final documentation, so see the source as the best reference.)

```
def fieldPlot(self,field,errorsName,convergentName,computedName,componentName):
    import pylab
    fig = field.multiAnsatz(exponentRange=(.5,3.),colorByComponent=True,relative=Tru
    pylab.title('Relative Component Errors')
    pylab.xlabel('Relative Discretization')
    fig.savefig(errorsName)

    fig = pylab.figure(figsize=(8,2)); pylab.grid(True)
    pylab.title('Convergent Error Fraction')
    pylab.semilogx(field.deltaXs,
                   field.convergentErrorFraction((.5,3.)),'-o')
    #pylab.semilogx(field.deltaXs,
    #               1-field.nonConvergentErrorFraction((.5,3.)),'-o')
    a = list(pylab.axis()); a[2:] = (0.,1.); pylab.axis(a)
    pylab.xlabel('Relative Discretization')
    #pylab.legend(('conv','1-nonconv'),loc='best')
    fig.savefig(convergentName)

    #use coarsest mesh for now; change to feOutputs[-1] and field.components()[-1].T
    field.feOutputs[0].idtf(fileIn=file(componentName.replace('.u3d','.idtf'),'w'),f
              nColors=256)

    #import pdb;pdb.set_trace()
    #field.feOutputs[0].idtf(fileIn=file(computedName.replace('.u3d','.idtf'),'w'),f
    field.feOutputs[0].idtf(fileIn=file(computedName.replace('.u3d','.idtf'),'w'),fi
    print 'idtfs created. run IDTFConverter.exe to convert to u3d.'


def timeHistoryPlots(self,th,errorsName,convergentName,computedName,componentName,th
    import pylab
    import numpy
```

```
# for drafts, leave about 1000 points
if self.quality == 'draft':
    stride = int(len(th.interpolatedRefTimes[-1])/1000)+1
else: stride = 1
# for some bizzare reason, if there are spaces in the file name latex
#  will put part of the file name into the text
# default figure dimensions are 8x6
fig = th.multiAnsatz(exponentRange=(.5,3.),colorByComponent=True,relative=True)
pylab.title('Relative Component Errors')
pylab.xlabel('Relative Discretization')
#pylab.ylabel('Relative Error')
fig.savefig(errorsName)

fig = pylab.figure(figsize=(8,2)); pylab.grid(True)
pylab.title('Convergent Error Fraction')
pylab.semilogx(th.deltaXs,th.convergentErrorFraction((.5,3.)),'-o')
# i think having a second metric can confuse the issue right now...
#pylab.semilogx(th.deltaXs,
#                1-th.nonConvergentErrorFraction((.5,3.)),'-o')
a = list(pylab.axis()); a[2:] = (0.,1.); pylab.axis(a)
pylab.xlabel('Relative Discretization')
#pylab.legend(('conv','1-nonconv'),loc='best')
fig.savefig(convergentName)

fig = pylab.figure(figsize=(8,4)); pylab.grid(True)
ax = pylab.subplot(111)
#upper,lower = th.uncertainties((.5,3.),stddevs=1.)
upper,lower = th.uncertainties((.5,3.),stddevs=1.)
t = pylab.array(th.interpolatedRefTimes[-1])
tb,te = [t.searchsorted(x) for x in th.tBounds]
#pylab.plot(t[tb:te:stride]*1000.,upper[tb:te:stride],label='upper')
#pylab.plot(t[tb:te:stride]*1000.,lower[tb:te:stride],label='lower')
import matplotlib
verts = numpy.concatenate((numpy.array((t[tb:te:stride]*1000.,yScale*upper[tb:te
                           numpy.fliplr(numpy.array((t[tb:te:stride]*1000.,yScal
poly = matplotlib.patches.Polygon(verts,facecolor='0.8',edgecolor='0.8')
ax.add_patch(poly)
pylab.title('Time Histories')
#for fe,c,i in zip(th.feOutputs,self.computedTHcolors,range(100)):
t = th.interpolatedRefTimes[-1]
t = pylab.array(t)
tb,te = [t.searchsorted(x) for x in th.tBounds]
for co,c,i in zip(th.interpolatedRefValues[-1],self.computedTHcolors,range(100))
    #t,co = fe.timeHistory(*th._variableID)
```

```
            #t = pylab.array(t)
            #tb,te = [t.searchsorted(x) for x in th.tBounds]
            pylab.plot(t[tb:te:stride]*1000.,yScale*co[tb:te:stride],color=c,label=thLab
        #import pdb;pdb.set_trace()
        pylab.legend(loc='best')
        pylab.xlabel('Time (ms)')
        pylab.ylabel(ylabel)
        fig.savefig(computedName)

        fig = pylab.figure(figsize=(8,4)); pylab.grid(True)
        pylab.title('Components')
        t = pylab.array(th.interpolatedRefTimes[-1])
        tb,te = [t.searchsorted(x) for x in th.tBounds]
        #for co,c,i in zip(th.scaledComponents()[-1].T,self.cs,range(100)):
        for co,c,i in zip(th.components()[-1].T,self.cs,range(100)):
            pylab.plot(t[tb:te:stride]*1000.,co[tb:te:stride],color=c,label=str(i))
        #pylab.legend(loc='best') # color coding makes the legend less useful
        pylab.xlabel('Time (ms)')
        #pylab.ylabel(ylabel)
        fig.savefig(componentName)
        #import pdb;pdb.set_trace()

        return


    def timeHistoryPeaksPlots(self,th,components,thLabels,ylabel,computedName,peaksName,
        import pylab
        import numpy
        # find maxs
        t = th.interpolatedRefTimes[-1]
        if filter:
            #import pdb;pdb.set_trace()
            #t,cos = th.filter(t,th.interpolatedRefValues[-1].T)
            t,cos = th.filter(t,components[1].T)
            cos = cos.T
        #else: cos = th.interpolatedRefValues[-1]
        else:
            cos = components[1]
        iMin = t.searchsorted(tMin)
        iMaxs = [iMin+i for i in cos[:,iMin:].argmax(axis=1)]
        tMaxs = [t[i] for i in iMaxs]
        coMaxs = [yScale*co[i] for co,i in zip(cos,iMaxs)]
        tBounds = [min(tMaxs),max(tMaxs)]
        span = tBounds[1]-tBounds[0]
```

```
if span < .001: pad = .0005
else: pad = .1*span
tBounds = [tBounds[0]-pad,tBounds[1]+pad]
tBounds[0] = max(tBounds[0],th.tBounds[0])
tBounds[1] = min(tBounds[1],th.tBounds[1])
tb,te = [t.searchsorted(x) for x in tBounds]
tMaxs = [1000.*tm for tm in tMaxs]
#import pdb;pdb.set_trace()

# for drafts, leave about 1000 points
if self.quality == 'draft':
    stride = int(len(th.interpolatedRefTimes[-1])/1000)+1
else: stride = 1
# default figure dimensions are 8x6
figDim = (15,4)

fig = pylab.figure(figsize=figDim); pylab.grid(True)
ax = pylab.subplot(111)
#upper,lower = th.uncertainties((.5,3.),stddevs=1.)
upper,lower = th.uncertainties((.5,3.),stddevs=1.,
                               derivative=derivative,filter=True)
                               #derivative=derivative,filter=filter)
#pylab.plot(t[tb:te:stride]*1000.,upper[tb:te:stride],label='upper')
#pylab.plot(t[tb:te:stride]*1000.,lower[tb:te:stride],label='lower')
import matplotlib
verts = numpy.concatenate((numpy.array((t[tb:te:stride]*1000.,yScale*upper[tb:te
                           numpy.fliplr(numpy.array((t[tb:te:stride]*1000.,yScal
poly = matplotlib.patches.Polygon(verts,facecolor='0.8',edgecolor='0.8')
ax.add_patch(poly)
pylab.title('Time Histories')
#for fe,c,i in zip(th.feOutputs,self.computedTHcolors,range(100)):
#t = th.interpolatedRefTimes[-1]
#t = components[0]
##t = t
#tb,te = [t.searchsorted(x) for x in th.tBounds]
for co,c,i in zip(cos,self.computedTHcolors,range(100)):
    #t,co = fe.timeHistory(*th._variableID)
    #t = pylab.array(t)
    #tb,te = [t.searchsorted(x) for x in th.tBounds]
    pylab.plot(t[tb:te:stride]*1000.,yScale*co[tb:te:stride],color=c,label=thLab
pylab.legend(loc='best')
pylab.xlabel('Time (ms)')
pylab.ylabel(ylabel)
pylab.plot(tMaxs,coMaxs,'D',color='orange',markersize=12)
```

```
        fig.savefig(computedName)


        #fig = pylab.figure(figsize=(8,4)); pylab.grid(True)
        fig = pylab.figure(figsize=figDim); pylab.grid(True)
        pylab.title('Peak Values')
        pylab.semilogx(th.deltaXs,coMaxs,'-D',color='orange')
        pylab.xlabel('Relative Discretization')
        pylab.ylabel(ylabel)
        fig.savefig(peaksName)

        #pylab.show()


    def timeHistoryDiffPlots(self,th,components,thLabels,ylabel,computedName,componentNa
        '''filter controls filtering time histories; differentiated components are alway
        import pylab
        import numpy
        # for drafts, leave about 1000 points
        if self.quality == 'draft':
            stride = int(len(th.interpolatedRefTimes[-1])/1000)+1
        else: stride = 1
        # default figure dimensions are 8x6
        figDim = (15,4)

        fig = pylab.figure(figsize=figDim); pylab.grid(True)
        ax = pylab.subplot(111)
        t = th.interpolatedRefTimes[-1]
        tb,te = [t.searchsorted(x) for x in th.tBounds]
        #upper,lower = th.uncertainties((.5,3.),stddevs=1.)
        upper,lower = th.uncertainties((.5,3.),stddevs=1.,
                                       derivative=derivative,filter=True)
                                       #derivative=derivative,filter=filter)
        #pylab.plot(t[tb:te:stride]*1000.,upper[tb:te:stride],label='upper')
        #pylab.plot(t[tb:te:stride]*1000.,lower[tb:te:stride],label='lower')
        import matplotlib
        verts = numpy.concatenate((numpy.array((t[tb:te:stride]*1000.,yScale*upper[tb:te
                                   numpy.fliplr(numpy.array((t[tb:te:stride]*1000.,yScal
        poly = matplotlib.patches.Polygon(verts,facecolor='0.8',edgecolor='0.8')
        pylab.title('Time Histories')
        #for fe,c,i in zip(th.feOutputs,self.computedTHcolors,range(100)):
        #t = th.interpolatedRefTimes[-1]
        t = components[0]
        t = t
```

```
tb,te = [t.searchsorted(x) for x in th.tBounds]
if filter:
    #import pdb;pdb.set_trace()
    #t,cos = th.filter(t,th.interpolatedRefValues[-1].T)
    t,cos = th.filter(t,components[1].T)
    cos = cos.T
#else: cos = th.interpolatedRefValues[-1]
else: cos = components[1]
for co,c,i in zip(cos,self.computedTHcolors,range(100)):
    #t,co = fe.timeHistory(*th._variableID)
    #t = pylab.array(t)
    #tb,te = [t.searchsorted(x) for x in th.tBounds]
    pylab.plot(t[tb:te:stride]*1000.,yScale*co[tb:te:stride],color=c,label=thLab
a = pylab.axis()
ax.add_patch(poly)
loc = 'best'
if tb == 0:
    pylab.axis(a) # don't let upper and lower change the axis
    loc = 'upper left'
pylab.legend(loc=loc)
pylab.xlabel('Time (ms)')
pylab.ylabel(ylabel)
fig.savefig(computedName)



fig = pylab.figure(figsize=figDim); pylab.grid(True)
pylab.title('Components')
t = pylab.array(th.interpolatedRefTimes[-1])
tb,te = [t.searchsorted(x) for x in th.tBounds]
#for co,c,i in zip(th.scaledComponents()[-1].T,self.cs,range(100)):
#for co,c,i in zip(th.diffComponents(derivative=derivative,filter=filter)[-1].T,
for co,c,i in zip(th.diffComponents(derivative=derivative,filter=True)[-1].T,sel
    pylab.plot(t[tb:te:stride]*1000.,co[tb:te:stride],color=c,label=str(i))
#pylab.legend(loc='best') # color coding makes the legend less useful
pylab.xlabel('Time (ms)')
#pylab.ylabel(ylabel)
fig.savefig(componentName)
#import pdb;pdb.set_trace()
```

4.1 Class FunctionalConvergence

```
object ─┐
        │
        FunctionalConvergence.functionalConvergence.FunctionalConvergence
```

Known Subclasses: FunctionalConvergence.functionalConvergence.FieldFunctionalConvergence, FunctionalConvergence.functionalConvergence.TimeHistoryFunctionalConvergence

This is a parent class and should not be used directly; see the subclasses for a usable interface.

4.1.1 Methods

multiAnsatz(*self*, *residualFunc*=`None`, *initParams*=`None`, *exponentRange*=`None`, *colorByComponent*=`False`, *relative*=`False`, ***kwargs*)

interface to the capabilities of the multiAnsatz module; returns a plot of the multiple fit results

extrapolated(*self*, *exponentRange*)

solution extrapolated from the convergent modes

convergentProjections(*self*, *exponentRange*)

not currently used

convergent part of each computed solution. the difference between this and the computed solution is the residual, for bootstrap residual resampling

residualResample(*self*, *iterations*=`1`)

not currently used

resample distribution

ansatz(*self*, *deltaXs*=`None`, *gcSlice*=`None`)

interface to the capabilities of the ansatz module. returns a list of fits from multiple initializations

generalizedCoordinates(*self*)

generalized coordinates, η, defined as $\eta = V\sigma$ where the data matrix, Y, is decomposed with a SVD.

$$Y = \left[\begin{array}{ccc} y_1(\Delta x_c) & y_1(\Delta x_m) & y_1(\Delta x_f) \\ \vdots & \vdots & \vdots \end{array} \right] = U\sigma V^T \tag{1}$$

nonConvergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components

convergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components

nonConvergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components divided by the L_2 norm of the estimated total error.

$convergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

convergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components divided by the L_2 norm of the estimated total error

$extnormalconvergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

components(*self*)

note that these are not orthogonal; they are not the left singular vectors of the data matrix. these components represent the fields which, when scaled by the generalized coordinates, sum to the computed solution fields. diag(sqrt(weights))*these will produce orthogonal modes; ie, left singular vectors of diag(sqrt(weights))*(the data matrix). this method was called modes previously, but the name was changed to avoid confusion with orthogonal modes.

scaledComponents(*self*)

components, scaled by the squared singular values. these are sometimes useful for visualization since you can see relative magnitude as well as morphology.

orthogonalModes(*self*)

not currently used (because FieldFunctionalConvergence redefines it).

left singular vectors of diag(sqrt(weights))*(the data matrix). needs some refactoring to work on time histories because the inner product approximation to integration is not diagonal. nothing else depends on it, so it is not critical.

saveCache(*self*, *fileOut*=`None`)

saves the in-memory cache of integral matrices to a pickle file, to be loaded on a future execution of the same or similar script. since the vast majority of computation time is usually in doing the integrals, this caching system greatly speeds up analyses when you need to rerun a script. an integral matrix only has rows and columns equal in number to the number of meshes, so the pickle files tend to be quite small.

loadCache(*self*, *fileIn*=`None`)

warning: clobbers current cache

loads a previously saved cache of integral matrices from a pickle file.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

4.1.2 Properties

Name	Description
Inherited from object	
__class__	

4.2 Class TimeHistoryFunctionalConvergence

```
object ─┐
FunctionalConvergence.functionalConvergence.FunctionalConvergence ─┐
                                                                   FunctionalConvergence.func
```

Usage examples and correctness tests

This class implements the Functional Convergence algorithms for time history data.

```
>>> import extractor
>>> import numpy
>>> feIndices = (1,2,4,)
>>> try:
...     fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...            for i in feIndices]
... except IOError:
...     import os
...     r = os.system('/usr/local/abaqus/Commands/abq675 python functionalConvergence.py')
...     fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...            for i in feIndices]
>>> fes[0]._results[('test',0)] = ((0.,1.,3.,5.,7.),(0.,1.,3.,5.,7.))
>>> fes[1]._results[('test',0)] = ((0.,2.,4.,4.5,7.),(0.,4.,8.,9.,14.))
>>> f = TimeHistoryFunctionalConvergence(fes[:2],(1.,.5))
>>> x = f._crossInterpolate(((0.,1.,3.,5.,7.),(0.,2.,4.,4.5,7.)),
...                         ((0.,1.,3.,5.,7.),(0.,4.,8.,9.,14.)))
>>> x #doctest: +NORMALIZE_WHITESPACE
(array([ 0. ,  1. ,  2. ,  3. ,  4. ,  4.5,  5. ,  7. ]),
 (array([ 0. ,  1. ,  2. ,  3. ,  4. ,  4.5,  5. ,  7. ]),
  array([  0.,   2.,   4.,   6.,   8.,   9.,  10.,  14.])))
```

```
>>> f._crossInterpolate(((0.,1.,3.,5.,7.),(0.,2.,4.,4.5,7.)),
...                     ((0.,1.,3.,5.,7.),(0.,4.,8.,9.,14.)),
...                     tBounds=(.5,1.5)) #doctest: +NORMALIZE_WHITESPACE
(array([ 0.5,  1. ,  1.5]), (array([ 0.5,  1. ,  1.5]),
                            array([ 1.,  2.,  3.])))
>>> f._interpolate((4.2,),x[0],x[1][1])
array([ 8.4])
>>> f._integrateProduct(*x) == 2*7**3/3.
True
>>> f.updateIntegrals('test',0)
>>> (f.integrals == [[7**3/21.,2*7**3/21.],[2*7**3/21.,4*7**3/21.]]).all()
True
>>> f = TimeHistoryFunctionalConvergence(fes,(1.,.5,.25))
>>> f.updateIntegrals('ETOTAL'); f.integrals#doctest: +NORMALIZE_WHITESPACE
array([[ 0.01074333,  0.00421427,  0.00143863],
       [ 0.00421427,  0.00196711,  0.00065865],
       [ 0.00143863,  0.00065865,  0.00025392]])
>>> f.generalizedCoordinates()
array([[ -1.93384364e-05,  -1.72395918e-03,   5.19411104e-03],
       [ -6.71550896e-03,   1.49649657e-02,   4.94196644e-03],
       [ -1.03432260e-01,  -4.17155848e-02,  -1.42307669e-02]])
>>> #returned tuple is (extrapolated, prefactor, exponent)
>>> f.ansatz() #doctest: +NORMALIZE_WHITESPACE
[([-0.00035632510105344296, 0.00033698666462904934, -2.0209188665783593], 0, 'Initial values are close enough.
```

1st column mode is dominated by fine mesh, but it is not convergent. ordinarily, convergence would be questionable because the two convergent modes are dominated by the coarse and medium meshes. in this case, the time history is converging to zero, so it makes sense.

```
>>> f._toComponentMatrix()
array([[  -0.64566486,  -22.88301843,   -8.18232531],
       [ -57.55893753,   50.99294597,   -3.30003895],
       [ 173.41913723,   16.839693  ,   -1.12576835]])
>>> # .components()[mesh][component][time]
>>> # the time steps are roughly (not exactly) halved
>>> [mesh.T[-1][:5] for mesh in f.components()] #doctest: +NORMALIZE_WHITESPACE
[array([ 0.        ,  0.19438734,  0.39626199,  0.59566605,  0.78229577])]
```

[array([0. , 0.8191986 , 1.47996525, 1.42355322, 0.94876351]), array([0. , 0.40392554, 0.79916985, 1.15317098, 1.42583566]), array([0. , 0.19438734, 0.39626199, 0.59566605, 0.78229577])]

Here is an example of plotting some results. (They are commented so as not to run as doctests.)

```
#    >>> import pylab as p
#    >>> for fe in fes: p.plot(*fe.timeHistory('ETOTAL'))
#    >>> len(f.extrapolated((.5,3.))), len(f.interpolatedRefTimes)
#    >>> for t,e in zip(f.interpolatedRefTimes,f.extrapolated((.5,3.))): p.plot(
#    >>>  #for t,e in zip(f.interpolatedRefTimes,f.extrapolated((.5,3.))): print
#    >>> p.grid()
#    >>> p.legend(('0','1','2','e0','e1','e2',))
#    >>> p.figure()
#    >>>  #import pdb;pdb.set_trace()
#    >>>  #hs = [fe.timeHistory('ETOTAL') for fe in f.feOutputs]
#     #>>>  #interpss = [[f._crossInterpolate((h0[0],h1[0]),(h0[1],h1[1])) for h1
#     #...              for h0 in hs]
#    >>>  #[[(p.plot(interp[0],interp[1][0],'o'),p.plot(interp[0],interp[1][1],'
#    >>> ms = 'o+x'
#    >>> for i in range(len(feIndices)):
#    ...   p.plot(*fes[i].timeHistory('ETOTAL'))
#    ...   for m,t,x in zip(ms,f.interpolatedRefTimes,f.interpolatedRefValues):
#    ...     p.plot(t,x[i],m)
#    ...   p.grid()
#    ...   p.legend(('0','1','2'))
#    ...   p.figure()
#    >>>  #p.figure()
#    >>> for c in f.components()[0].T: p.plot(f.interpolatedRefTimes[0],c)
#    >>> p.grid()
#    >>> p.legend(('0','1','2'))
#    >>> p.figure()
#    >>> for c in f.components()[1].T: p.plot(f.interpolatedRefTimes[1],c)
#    >>> p.grid()
#    >>> p.legend(('0','1','2'))
#    >>> p.figure()
#    >>> for c in f.components()[-1].T: p.plot(f.interpolatedRefTimes[-1],c)
#    >>> p.grid()
#    >>> p.legend(('0','1','2'))
#    >>> p.show()
```

```
>>> f.nonConvergentError((.5,3.))
array([  1.93384364e-05,   1.72395918e-03,   5.19411104e-03])
>>> f.convergentError((.5,3.))
array([ 0.11225253,  0.05002423,  0.02229395])
```

these two metrics of relative nonconvergent error are not equal because the squared L_2 errors sum to 1. (these are L_2 quantities, not squared.) it is not clear to me which is more useful; the second will always be smaller

```
>>> f.nonConvergentErrorFraction((.5,3.))
```

```
array([  1.72276175e-04,   3.44420360e-02,   2.26905982e-01])
>>> 1-f.convergentErrorFraction((.5,3.))
array([  1.48395403e-08,   5.93302926e-04,   2.60833325e-02])
```

```
>>> [mesh[:5] for mesh in f.extrapolated((.5,3.))] #doctest: +NORMALIZE_WHITESPACE
[array([ 0.        , -0.00335795, -0.00756562, -0.01157266, -0.01413355])]
```

can use multiAnsatz to test asymptotic assumption

```
>>> import multiAnsatz; from ansatz import Ansatz; feIndices = (1,2,4,8)
>>> import StringIO, hashlib, warnings; s = StringIO.StringIO()
>>> fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...        for i in feIndices]
>>> f1 = TimeHistoryFunctionalConvergence(fes,(1.,.5,.25,.125))
>>> f1.updateIntegrals('ETOTAL',0)
>>> warnings.filterwarnings('ignore')
>>> fig = f1.multiAnsatz(exponentRange=(.5,3.),pgtol=1e-8)
>>> warnings.resetwarnings()
>>> #fig.savefig(s); s.seek(0); hashlib.sha1(s.read()).hexdigest()
'dd70eef892b78d72d28b2eda9f144e17613b5fba'
>>> #import pylab; pylab.show()
```

uncertainty estimator

needs gcs, ansatzes, residuals: since it needs gcs, it should be at the FunctionalConvergence level or higher. we need to generate a distribution for extrapolated value of each gc assumed independent of each other. many statistics can be computed as stats on sum of indep rand vars.

```
>>> f1.gcPDF(gcIndex=0,exponentRange=(.5,3.)) #doctest: +NORMALIZE_WHITESPACE
array([[ 2.95641391e-05, 4.63192399e-04, -2.01779161e-03, 2.03533391e-03],
       [ 2.50000000e-01, 2.50000000e-01, 2.50000000e-01, 2.50000000e-01]])
>>> #f1.ansatz()
>>> gcs,ps = f1.gcPDF(gcIndex=3,exponentRange=(.5,3.))
>>> sum(gcs)/len(gcs)
0.006067577269747851
>>> f1.gcPDFmeans(exponentRange=(.5,3.))
[0.00012757471010546513, 0.0019109360632137782, 0.0069101831039161399, 0.006067577269747851]
>>> [numpy.sqrt(x) for x in f1.gcPDFvariances(exponentRange=(.5,3.))]
[0.0014460625074991997, 0.0030485599987097366, 0.0026219662557701532, 0.00033103368227088481]
>>> [u[:5] for u in f1.uncertainties((.5,3.))] #doctest: +NORMALIZE_WHITESPACE
[array([ 0.        , -0.00048529, -0.00059388, -0.00161884, -0.00115717]),
 array([ 0.        , -0.00671924, -0.01554435, -0.02023308, -0.02326962])]
>>> [u[:3] for u in f1.uncertainties((.5,3.),derivative=1)] #doctest: +NORMALIZE_WHITESPACE
[array([ -99370.13265344,  -60803.0533754 , -104530.24809945]),
 array([-1375859.74702827, -1591461.63650739, -1395098.399241  ])]
```

```
>>> #import matplotlib;matplotlib.use('WxAgg');import pylab; pylab.histogram(gcs);pylab.show()
```

results caching

```
>>> f1._cache[(('ETOTAL', 0, None), (None, None))]['integrals'] #doctest: +NORMALIZE_WHITESPACE
array([[ 1.07433308e-02, 4.21427319e-03, 1.43863211e-03, 3.11237569e-04],
       [ 4.21427319e-03, 1.96711225e-03, 6.58646687e-04, 1.46007212e-04],
       [ 1.43863211e-03, 6.58646687e-04, 2.53916548e-04, 7.25825229e-05],
       [ 3.11237569e-04, 1.46007212e-04, 7.25825229e-05, 4.26378823e-05]])
>>> f1.saveCache('/tmp/testCache.pickle')
>>> del(f1._cache);del(f1.feOutputs); f1.loadCache('/tmp/testCache.pickle')
>>> f1._cache[(('ETOTAL', 0, None), (None, None))]['integrals'] #doctest: +NORMALIZE_WHITESPACE
array([[ 1.07433308e-02, 4.21427319e-03, 1.43863211e-03, 3.11237569e-04],
       [ 4.21427319e-03, 1.96711225e-03, 6.58646687e-04, 1.46007212e-04],
       [ 1.43863211e-03, 6.58646687e-04, 2.53916548e-04, 7.25825229e-05],
       [ 3.11237569e-04, 1.46007212e-04, 7.25825229e-05, 4.26378823e-05]])
```

error checking

```
>>> gcs = f.generalizedCoordinates()
>>> (abs(numpy.dot(gcs.T,gcs)-f.integrals)<1e-17).all()
True
>>> (abs(numpy.dot(gcs,gcs.T)-numpy.diag(numpy.linalg.eigvalsh(f.integrals)))<1e-17
True
```

```
    #>>> (abs(sum([numpy.dot(om.T, om) for om in f.orthogonalModes()])      #
    #True
```

```
>>> f._interpolate((1.5,2.5,3.),(0.,1.,2.,3.),(0.,1.,2.,1.))
array([ 1.5,  1.5,  1. ])
>>> ts = [x/10. for x in range(20)]; xs = [x**2 for x in ts]
>>> f._interpolate((.5,1.5),ts,xs)
array([ 0.25,  2.25])

>>> fakeT = numpy.array((0.,.5,1.,1.25,2.5),ndmin=2)
>>> #import pdb;pdb.set_trace()
>>> f.interpolatedRefTimes = fakeT
>>> def fakeComponents():
...     return numpy.array(numpy.array((3.25*fakeT.T,4.2*fakeT.T),ndmin=2),ndmin=3).T
>>> f.components = fakeComponents; f.diffComponents(derivative=1)[0].T
array([[ 3.25,  3.25,  3.25,  3.25,  3.25],
       [ 4.2 ,  4.2 ,  4.2 ,  4.2 ,  4.2 ]])
>>> f.diffComponents(derivative=2)[0].T
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> fakeT = numpy.array(numpy.arange(.3,.4,.01),ndmin=2)
>>> f.interpolatedRefTimes = fakeT
>>> def fakeComponents():
...     return numpy.array(numpy.array((fakeT.T**2,4.2*fakeT.T**2),ndmin=2),ndmin=3).T
>>> f.components = fakeComponents; f.diffComponents(derivative=2)[0].T
array([[ 1. ,  1.5,  2. ,  2. ,  2. ,  2. ,  2. ,  2. ,  2. ,  1.5,  1. ],
       [ 4.2,  6.3,  8.4,  8.4,  8.4,  8.4,  8.4,  8.4,  8.4,  6.3,  4.2]])
```

3rd order butterworth low-pass filter

```
>>> dt = 1e-7; t = numpy.arange(0.,1e5*dt,dt)
>>> tt = numpy.array(t,ndmin=2).T
>>> ys = numpy.array(numpy.array((numpy.ones_like(t),
...                               numpy.sin(2*numpy.pi*5e3*t),
...                               numpy.sin(2*numpy.pi*10e3*t),
...                               numpy.sin(2*numpy.pi*20e3*t),
...                               numpy.cos(2*numpy.pi*t*.5/dt)),
...                              ndmin=2),ndmin=3).T
>>> tf,yfs = f.filter(t,ys)
>>> numpy.sqrt(numpy.sum(yfs**2,axis=0)/len(yfs)).T
array([[  9.98141470e-01,   7.00745395e-01,   4.99134258e-01,
          8.86999284e-02,   7.23602445e-05]])
>>> # results are garbage, but data are in the right format
>>> #import pdb;pdb.set_trace()#(1,11,2)
>>> f.diffComponents(derivative=2,filter=False)[0].T
array([[ 1. ,  1.5,  2. ,  2. ,  2. ,  2. ,  2. ,  2. ,  2. ,  1.5,  1. ],
       [ 4.2,  6.3,  8.4,  8.4,  8.4,  8.4,  8.4,  8.4,  8.4,  6.3,  4.2]])
```

On some platforms, this test fails with clearly bogus results. Not sure if it is some kind of architecture or scipy/numpy version problem.

```
>>> f.filterFreq = 1.; f.diffComponents(derivative=2,filter=True)[0].T
array([[ 0.1401103 ,  0.2543905 ,  0.45469833,  0.62197579,  0.77983212,
         0.92853643,  1.06835823,  1.19956727,  1.32243343,  1.0507016 ,
         0.71958868],
       [ 0.58846326,  1.06844011,  1.90973299,  2.61229832,  3.27529489,
         3.89985301,  4.48710457,  5.03818255,  5.5542204 ,  4.41294671,
         3.02227245]])
>>> #import matplotlib;matplotlib.use('WxAgg');import pylab; fig=pylab.figure();pylab.plot(tf,yf);pylab.plot
>>> #fig.savefig('tmp.png')
>>> #pylab.show()
```

.cacheInterpRefs sets .interpolatedRefValues and .interpolatedRefTimes without changing .integrals. it uses ._cache, if available, or picks it from .feOutputs otherwise. this is useful for setting derivative time histories to show with components and gcs derived from the integrated time histories. call .cacheInterpRefs right after .updateIntegrals and .filter(f.interpolatedRefTimes[-1],f.interpolatedRefValues[-1]) to plot.

```
>>> del(f.interpolatedRefValues); del(f.interpolatedRefTimes)
>>> irts,irvs = f.cacheInterpRefs(historyName='ETOTAL')
>>> irvs[-1][-1][:3], irts[-1][:3]
(array([ 0.        , -0.01367905, -0.03400893]), array([  0.00000000e+00,   9.87662663e-09,   1.97532533e-08])
```

4.2.1 Methods

__init__(*self*, *feOutputs*, *deltaXs*, *ansatzKwargs*={})

feOutputs iterable (eg., list or tuple) of extractor.Extractor instances

deltaXs iterable of relative discretizations. For example, if the elements are halved from coarse to medium and from medium to fine, deltaXs=(1.,.5,.25)

ansatzKwargs optional dict of args to pass through to the ansatz fit calls

Overrides: object.__init__

filter(*cls*, *t*, *ys*, *filterFreq*=`None`)

3rd order Butterworth low-pass filter. It is usually very important to filter before taking time derivatives, especially for explicit dynamics simulations. Experimental data are usually filtered this way, anyway.

diffComponents(*self*, *derivative*=1, *filter*=`False`)

n-order derivative of the components

gcPDF(*self*, *gcIndex*=`None`, *exponentRange*=`None`)

Probability density function for a generalized coordinate.

gcIndex generalized coordinate number to use

exponentRange tuple of lower and upper bounds on the exponent for convergence

No reason fields can't use this, too, so it could be moved up to the FunctionalConvergence class. But it's easiest to see here.

gcPDFmeans(*self*, *exponentRange*=`None`)

Means of the generalized coordinate probability density functions.

uncertainties(*self*, *exponentRange*=`None`, *derivative*=`0`, *stddevs*=`1.0`, *filter*=`False`)

Simple uncertainty estimator, defined as mean +/- 1 standard deviation.

gcPDFvariances(*self*, *exponentRange*=`None`)

Variances of the generalized coordinate probability density functions.

updateIntegrals(*self*, *historyName*, *historyComponent*=`0`, *tBounds*=`(None, None)`, *historyRegion*=`None`)

Computes and internally stores the matrix of inner product integrals. This method must be called before requesting any other results. Be aware that, unless these integrals have already been cached, this method will likely be the most computationally expensive part of your script.

cacheInterpRefs(*self*, *historyName*, *historyComponent*=`0`, *tBounds*=`(None, None)`, *historyRegion*=`None`)

returns cached interps, so they can be plotted while self.integrals and self.interpolatedRef* correspond to a derivative

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.functionalConvergence.FunctionalConvergence(Sect 4.1)

ansatz(*self*, *deltaXs*=`None`, *gcSlice*=`None`)

interface to the capabilities of the ansatz module. returns a list of fits from multiple initializations

components(*self*)

note that these are not orthogonal; they are not the left singular vectors of the data matrix. these components represent the fields which, when scaled by the generalized coordinates, sum to the computed solution fields. diag(sqrt(weights))*these will produce orthogonal modes; ie, left singular vectors of diag(sqrt(weights))*(the data matrix). this method was called modes previously, but the name was changed to avoid confusion with orthogonal modes.

convergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components

convergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components divided by the L_2 norm of the estimated total error

$extnormalconvergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

convergentProjections(*self*, *exponentRange*)

not currently used

convergent part of each computed solution. the difference between this and the computed solution is the residual, for bootstrap residual resampling

extrapolated(*self*, *exponentRange*)

solution extrapolated from the convergent modes

generalizedCoordinates(*self*)

generalized coordinates, η, defined as $\eta = V\sigma$ where the data matrix, Y, is decomposed with a SVD.

$$Y = \begin{bmatrix} y_1(\Delta x_c) & y_1(\Delta x_m) & y_1(\Delta x_f) \\ \vdots & \vdots & \vdots \end{bmatrix} = U\sigma V^T \tag{2}$$

loadCache(*self*, *fileIn*=`None`)

warning: clobbers current cache

loads a previously saved cache of integral matrices from a pickle file.

multiAnsatz(*self*, *residualFunc*=`None`, *initParams*=`None`, *exponentRange*=`None`, *colorByComponent*=`False`, *relative*=`False`, ***kwargs*)

interface to the capabilities of the multiAnsatz module; returns a plot of the multiple fit results

nonConvergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components

nonConvergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components divided by the L_2 norm of the estimated total error.

$convergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

orthogonalModes(*self*)

not currently used (because FieldFunctionalConvergence redefines it).

left singular vectors of diag(sqrt(weights))*(the data matrix). needs some refactoring to work on time histories because the inner product approximation to integration is not diagonal. nothing else depends on it, so it is not critical.

residualResample(*self*, *iterations*=`1`)

not currently used

resample distribution

saveCache(*self*, *fileOut*=`None`)

saves the in-memory cache of integral matrices to a pickle file, to be loaded on a future execution of the same or similar script. since the vast majority of computation time is usually in doing the integrals, this caching system greatly speeds up analyses when you need to rerun a script. an integral matrix only has rows and columns equal in number to the number of meshes, so the pickle files tend to be quite small.

scaledComponents(*self*)

components, scaled by the squared singular values. these are sometimes useful for visualization since you can see relative magnitude as well as morphology.

4.2.2 Properties

Name	Description
Inherited from object	

continued on next page

Name	Description
__class__	

4.3 Class FieldFunctionalConvergence

```
object ─┐
        │
FunctionalConvergence.functionalConvergence.FunctionalConvergence ─┐
                                                                   │
                                                                   FunctionalConvergence.func
```

This class implements the Functional Convergence algorithms for fields.

Limited experience with this method suggests that it is difficult to get good field convergence information in a dynamic simulation, especially at later times. This is because small differences in wave propagation speed cause phase errors that increase over time. Field Functional Convergence might therefore be more useful on quasi-static problems or dynamic problems in which higher frequency waves are heavily damped.

```
>>> import extractor
>>> import sampler
>>> import numpy
>>> #import pdb;pdb.set_trace()
>>> feIndices = (1,2,4,)
>>> try:
...     fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...             for i in feIndices]
... except IOError:
...     import os
...     r = os.system('/usr/local/abaqus/Commands/abq675 python functionalConvergence.py')
...     fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...             for i in feIndices]
>>> f = FieldFunctionalConvergence(sampler.NodeSampler,fes,.01,(1.,.5,.25))
>>> f.updateIntegrals('U',0); f.integrals
array([[  8.76353136e-05,   8.78002796e-05,   8.81555428e-05],
       [  8.78002796e-05,   8.91446247e-05,   8.96970537e-05],
       [  8.81555428e-05,   8.96970537e-05,   9.05765084e-05]])
>>> f.generalizedCoordinates()
array([[  6.37828161e-05,  -2.63081849e-04,   1.98696809e-04],
       [ -8.01389474e-04,   2.28554129e-04,   5.59864655e-04],
       [ -9.32679046e-03,  -9.43520936e-03,  -9.49860935e-03]])
>>> #returned tuple is (extrapolated, prefactor, exponent)
>>> f.ansatz() #doctest: +NORMALIZE_WHITESPACE
```

```
[([-7.1690985961687931e-05, 0.00013547380208226855, -0.49850807029597499],
  0, 'Initial values are close enough.', 5.8774717541114375e-39),
 ([0.00071698098462327146, 0.0015183704590040301, 1.6363093958798205],
  0, 'Initial values are close enough.', 1.1754943508222875e-38),
 ([-0.0094836500878483516, 0.00015685963099555886, 1.6951816259616503],
  0, 'Initial values are close enough.', 5.6832683783298355e-37)]

>>> #1st column is a junk mode(?), 2nd is a correction, 3rd is an average
>>> f._toComponentMatrix()
array([[  565.6474261 ,  -795.09982812,   -35.03207051],
       [-2333.09816061,   226.76033857,   -35.43929941],
       [ 1762.11000668,   555.47059853,   -35.67743416]])
>>> # .components()[mesh][component][node]
>>> [mesh.T[-1][:5] for mesh in f.components()] #doctest: +NORMALIZE_WHITESPACE
[array([-4.22615831, -4.16192439, -4.16192439, -4.22615831, -4.22615831]),
 array([-4.22615831, -4.20392549, -4.20392549, -4.22615831, -4.22615831]),
 array([-4.22615831, -4.21260479, -4.21260479, -4.22615831, -4.22615831])]
>>> f.exportComponents() #spits out vtk files showing component shapes
>>> f.idtfComponents()
>>> f.nonConvergentError((.5,3.))
array([  6.37828161e-05,   2.63081849e-04,   1.98696809e-04])
>>> f.convergentError((.5,3.))
array([ 0.00152645,  0.00049082,  0.00015783])
```

array([0.00154066, 0.00051174, 0.00018071])

these two metrics of relative nonconvergent error are not equal because the squared L_2 errors sum to 1 (these are L_2 quantities, not squared) - not clear to me which is more useful; the second will always be smaller

```
>>> f.nonConvergentErrorFraction((.5,3.))
array([ 0.0417486 ,  0.47241801,  0.78303748])
>>> 1-f.convergentErrorFraction((.5,3.))
array([ 0.00087185,  0.11862538,  0.37802548])
```

residual resample provides bootstrap confidence on convergence parameters

```
>>> [mesh[:5] for mesh in f.extrapolated((.5,3.))] #doctest: +NORMALIZE_WHITESPACE
[array([ 0.04064333,  0.04144828,  0.04144828,  0.04064333,  0.04064333]),
 array([ 0.04064333,  0.04118255,  0.04118255,  0.04064333,  0.04064333]),
 array([ 0.04064333,  0.04086264,  0.04086264,  0.04064333,  0.04064333])]

>>> #f.residualResample(iterations=10)
```

can use multiAnsatz to test asymptotic assumption

```
>>> import multiAnsatz; from ansatz import Ansatz; feIndices = (1,2,4,8)
>>> import StringIO, hashlib, warnings; s = StringIO.StringIO()
>>> fes = [extractor.Pickle('1Dwave.'+str(i)+'.pickle')
...        for i in feIndices]
>>> f1 = FieldFunctionalConvergence(sampler.NodeSampler,fes,.01,
...                                 (1.,.5,.25,.125))
>>> f1.updateIntegrals('U',0)
>>> warnings.filterwarnings('ignore')
>>> fig = f1.multiAnsatz(exponentRange=(.5,3.),pgtol=1e-8)
>>> warnings.resetwarnings()
>>> fig.savefig(s); s.seek(0); hashlib.sha1(s.read()).hexdigest()
'6514ccb136a4d323915ff5c8eb5c2108c92fc16e'
>>> #import pylab; pylab.show()
```

Results are automatically cached for computational convenience when the .integrals method is called. This cache can be saved and loaded so you can save time when you know you will rerun your script.

```
>>> f1._cache[(('U', 0, None),)]['integrals'] #doctest: +NORMALIZE_WHITESPACE
array([[ 8.73535512e-05, 8.73485092e-05, 8.74426878e-05, 8.75648291e-05],
       [ 8.73485092e-05, 8.85645820e-05, 8.89274490e-05, 8.90975653e-05],
       [ 8.74426878e-05, 8.89274490e-05, 8.97068420e-05, 8.99493683e-05],
       [ 8.75648291e-05, 8.90975653e-05, 8.99493683e-05, 9.02866655e-05]])
>>> f1.saveCache('/tmp/testCache.pickle')
>>> del(f1._cache);del(f1.samplers); f1.loadCache('/tmp/testCache.pickle')
>>> f1._cache[(('U', 0, None),)]['integrals'] #doctest: +NORMALIZE_WHITESPACE
array([[ 8.73535512e-05, 8.73485092e-05, 8.74426878e-05, 8.75648291e-05],
       [ 8.73485092e-05, 8.85645820e-05, 8.89274490e-05, 8.90975653e-05],
       [ 8.74426878e-05, 8.89274490e-05, 8.97068420e-05, 8.99493683e-05],
       [ 8.75648291e-05, 8.90975653e-05, 8.99493683e-05, 9.02866655e-05]])
```

The following doctests are for error checking, to prevent previous bugs from reappearing.

```
>>> gcs = f.generalizedCoordinates()
>>> (abs(numpy.dot(gcs.T,gcs)-f.integrals)<1e-19).all()
True
>>> (abs(numpy.dot(gcs,gcs.T)-numpy.diag(numpy.linalg.eigvalsh(f.integrals)))<1e-19
True
>>> (abs(sum([numpy.dot(om.T, om) for om in f.orthogonalModes()])            -numpy
True
```

4.3.1 Methods

__init__(*self*, *sampler*, *feOutputs*, *meshTolerance*, *deltaXs*, *ansatzKwargs*={})

sampler a sampler instance (see the sampler module)

feOutputs iterable (eg., list or tuple) of extractor.Extractor instances

meshTolerance mesh tolerance parameter passed through to mapper; should be the maximum distance from an exterior element surface that a point should be considered in or on the mesh

deltaXs iterable of relative discretizations. For example, if the elements are halved from coarse to medium and from medium to fine, deltaXs=(1.,.5,.25)

ansatzKwargs optional dict of args to pass through to the ansatz fit calls

Overrides: object.__init__

updateIntegrals(*self*, *fieldName*=`None`, *fieldComponent*=`None`, *frameNumber*=`None`)

Computes and internally stores the matrix of inner product integrals. This method must be called before requesting any other results. Be aware that, unless these integrals have already been cached, this method will likely be the most computationally expensive part of your script.

orthogonalModes(*self*)

left singular vectors of diag(sqrt(weights))*(the data matrix) Overrides: FunctionalConvergence.functionalConvergence.FunctionalConvergence.orthogonalModes

exportComponents(*self*)

Exports the components to a vtk file, for visualization.

idtfComponents(*self*)

Exports the components to idtf files, for conversion to u3d files to be embedded in a pdf document.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.functionalConvergence.FunctionalConvergence(Sect 4.1)

ansatz(*self*, *deltaXs*=`None`, *gcSlice*=`None`)

interface to the capabilities of the ansatz module. returns a list of fits from multiple initializations

components(*self*)

note that these are not orthogonal; they are not the left singular vectors of the data matrix. these components represent the fields which, when scaled by the generalized coordinates, sum to the computed solution fields. diag(sqrt(weights))*these will produce orthogonal modes; ie, left singular vectors of diag(sqrt(weights))*(the data matrix). this method was called modes previously, but the name was changed to avoid confusion with orthogonal modes.

convergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components

convergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by convergent components divided by the L_2 norm of the estimated total error

$extnormalconvergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

convergentProjections(*self*, *exponentRange*)

not currently used

convergent part of each computed solution. the difference between this and the computed solution is the residual, for bootstrap residual resampling

extrapolated(*self*, *exponentRange*)

solution extrapolated from the convergent modes

generalizedCoordinates(*self*)

generalized coordinates, η, defined as $\eta = V\sigma$ where the data matrix, Y, is decomposed with a SVD.

$$Y = \begin{bmatrix} y_1(\Delta x_c) & y_1(\Delta x_m) & y_1(\Delta x_f) \\ \vdots & \vdots & \vdots \end{bmatrix} = U\sigma V^T \tag{3}$$

loadCache(*self*, *fileIn*=`None`)

warning: clobbers current cache

loads a previously saved cache of integral matrices from a pickle file.

multiAnsatz(*self*, *residualFunc*=`None`, *initParams*=`None`, *exponentRange*=`None`, *colorByComponent*=`False`, *relative*=`False`, ***kwargs*)

interface to the capabilities of the multiAnsatz module; returns a plot of the multiple fit results

nonConvergentError(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components

nonConvergentErrorFraction(*self*, *exponentRange*)

L_2 norm of the estimated error that is represented by nonconvergent components divided by the L_2 norm of the estimated total error.

$convergenterrorfraction^2 + extnormalconvergenterrorfraction^2 = 1$

residualResample(*self*, *iterations*=`1`)

not currently used

resample distribution

saveCache(*self*, *fileOut*=`None`)

saves the in-memory cache of integral matrices to a pickle file, to be loaded on a future execution of the same or similar script. since the vast majority of computation time is usually in doing the integrals, this caching system greatly speeds up analyses when you need to rerun a script. an integral matrix only has rows and columns equal in number to the number of meshes, so the pickle files tend to be quite small.

scaledComponents(*self*)

components, scaled by the squared singular values. these are sometimes useful for visualization since you can see relative magnitude as well as morphology.

4.3.2 Properties

Name	Description
Inherited from object __class__	

5 Module FunctionalConvergence.mapperWrapper

The mapperWrapper module wraps the mapper command-line tool. It is included with the FunctionalConvergence package to compute approximate integrals over mesh domains, though it could be used for any other mesh result interpolation.

5.1 Class Mapper

```
object ─┐
        │
   FunctionalConvergence.mapperWrapper.Mapper
```

Mapper provides a python interface to the mapper command-line tool written by R. (Bob) Stevens. Mapper interpolates a field on a mesh to a set of arbitrary points inside that mesh.

```
>>> import extractor
>>> #a = extractor.Abaqus('../1Dwave/1Dwave.1.odb')
>>> a = extractor.Pickle('1Dwave.1.pickle')
>>> m = Mapper(a,.1)
>>> m.map(((-4.5,.1,.1),(-4.25,-.2,.3)),'U',0)
[0.05574084, 0.053360629999999999]
>>> #m.setup()
```

interpolating node-associated field

```
>>> m.writeInput('U',0)
>>> file(m.workingDir+'/t.mesh').readlines()[:3]
['*MESH TOLERANCE, TOL=0.1\n', '*NODE\n', '\t1\t-1\t-0.5\t0.5\n']
>>> file(m.workingDir+'/t.mesh').readlines()[147]
'\t1\t1\t1\t2\t3\t4\t5\t6\t7\t8\n'
>>> file(m.workingDir+'/t.results').readlines()[:2]
['*NODAL RESULTS, TYPE=SCALER\n', '\t0.0388822\n']
>>> m.writeInterp(((4.5,.1,.1),(4.75,-.2,.3)))
>>> file(m.workingDir+'/t.ips').readlines()[:2]
['\t1\t-1\t4.5\t0.1\t0.1\n', '\t2\t-1\t4.75\t-0.2\t0.3\n']
>>> m.execute().split('\n')[-2]
'The lowest confidence was 1.0000 at interpolation point #1'
>>> file(m.workingDir+'/t.out').readlines()[0]
'       1   1.232028e-10\n'
>>> m.readOutput()
[1.2320280000000001e-10, 5.5480669999999998e-11]
```

interpolating element-associated field

```
>>> a._results[('mockElementField',1)] = [x**2 for x in range(len(a.elements))]
>>> a._elementFieldNames = ('mockElementField',)
>>> m.writeInput('mockElementField',1)
>>> file(m.workingDir+'/t.results').readlines()[:2]
['*ELEMENT RESULTS, TYPE=SCALER\n', '\t0\n']
>>> m.execute().split('\n')[-2]
'The lowest confidence was 1.0000 at interpolation point #1'
>>> file(m.workingDir+'/t.out').readlines()[0]
'       1   2.135616e+03\n'
>>> m.readOutput()
[2135.616, 2037.9690000000001]
>>> #import pdb; pdb.set_trace()
```

5.1.1 Methods

__init__(*self*, *feOutput*, *meshTolerance*, *fieldName*=`None`, *fieldComponent*=`None`, *frameNumber*=`None`)

feOutput is an extractor.Extractor instance that contains the mesh and results from which to interpolate.

meshTolerance is a tolerance parameter for distance from a point to an exterior surface. (See Bob's mapper docs for details.) The precise value is probably not critical for the Functional Convergence application.

The other parameters are used to identify the field in **feOutput**. After creating a Mapper instance, call the map method. Overrides: object.__init__

map(*self*, *interpPoints*, *fieldName*=`None`, *fieldComponent*=`None`, *frameNumber*=`None`)

User entry point; returns the interpolated results.

writeInterp(*self*, *interpPoints*)

Writes the file identifying interpolation points for mapper to read.

writeInput(*self*, *fieldName*, *fieldComponent*, *frameNumber*=`None`, *baseName*=`'t'`)

Writes all mapper input files except interpolation points.

readOutput(*self*)

Reads the interpolated results output file from mapper.

execute(*self*)

Calls the mapper executable.

setup(*self*)

Creates a uniquely named temp directory.

__del__(*self*)

The temp working directory is deleted when the Mapper instance is deallocated.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

5.1.2 Properties

Name	Description
Inherited from object	
__class__	

6 Module FunctionalConvergence.multiAnsatz

The multiAnsatz module provides the user with a convenient way to perform multiple ansatz fits to sampled combinations of meshes with a single method call. Methods are also provided for convenience in plotting results of these multiple fits.

6.1 Class MultiAnsatz

```
object ─┐
        │
FunctionalConvergence.ansatz.Ansatz ─┐
                                     │
                                     FunctionalConvergence.multiAnsatz.MultiAnsatz
```

You can create an iterator or generate plots directly from a MultiAnsatz instance.

```
>>> import StringIO, hashlib
>>> xs = [1/8.,1/4.,1/2.,1.]
>>> ys = ([1-(1+e*.01)*x**2 for x,e in zip(xs,[1,-1,1,-1])],
...       (.0013,.0025,.01455,.045),(.000085,.00004,.0000695,.000035))
>>> m = MultiAnsatz(residualFunc=Ansatz.singleExpExtrapRes,
...                    initParams=Ansatz.singleExpExtrapInit,
...                    sampleXs=xs,sampleYs=ys,exponentRange=(.5,3.),
...                    pgtol=1e-8)
>>> m.multifit().next() #doctest: +NORMALIZE_WHITESPACE
[((0.125, 0.25, 0.5),
  (0.98421875000000003, 0.93812499999999999, 0.74750000000000005),
  ([0.99891891891891893, 1.0397663765460363, 2.0480942882010433],
   0, 'Initial values are close enough.', 5.2511683639933228e-33)),
((0.125, 0.25, 1.0),
 (0.98421875000000003, 0.93812499999999999, 0.010000000000000009),
 (array([ 0.97655567,  0.96655567,  2.32626143]),
  1, 'Converged (|f_n-f_(n-1)| ~= 0)', 4.8594366773280982e-19)),
((0.125, 0.5, 1.0),
 (0.98421875000000003, 0.74750000000000005, 0.010000000000000009),
 (array([ 0.97199327,  0.96199327,  2.0993546 ]),
  1, 'Converged (|f_n-f_(n-1)| ~= 0)', 1.3232558498629955e-18)),
((0.25, 0.5, 1.0),
 (0.93812499999999999, 0.74750000000000005, 0.010000000000000009),
 ([1.0045714285714284, 0.99457142857142844, 1.9519057117989558],
  0, 'Initial values are close enough.', 7.7037197775489434e-34)),
((0.125, 0.25, 0.5, 1.0),
 (0.98421875000000003, 0.93812499999999999, 0.74750000000000005, 0.010000000000000009),
 (array([ 1.00169425,  0.99175158,  1.96691864]),
  1, 'Converged (|f_n-f_(n-1)| ~= 0)', 2.7789767689774468e-06))]
>>> f = m.plot()
>>> s = StringIO.StringIO()
>>> f.savefig(s); s.seek(0); hashlib.sha1(s.read()).hexdigest()
'fcd1db6745eb92beca5af97021ac7bb3e7844826'
```

can choose to keep all combination fits of the same component one color

```
>>> f = m.plot(colorByComponent=True)
>>> s.seek(0); f.savefig(s); s.seek(0); hashlib.sha1(s.read()).hexdigest()
'3626cf48a0fefa9ba374418e9fccee788d55b5f4'
```

can also make a separate figure for each component

```
>>> fs = m.plot(subfigures=True)
>>> s.seek(0); ns = [f.savefig(s) for f in fs]; s.seek(0)
>>> hashlib.sha1(s.read()).hexdigest()
'f004072b3023d076e90d295f5ad3525da62c35f7'
>>> #import pylab; pylab.show()
```

error checking

```
>>> [x for x in m._combinations('asdf',2)]
[('a', 's'), ('a', 'd'), ('a', 'f'), ('s', 'd'), ('s', 'f'), ('d', 'f')]
```

6.1.1 Methods

__init__(*self*, *residualFunc*=`None`, *initParams*=`None`, *jacobian*=`None`, *sampleXs*=`None`, *sampleYs*=`None`, *fExtrap*=`None`, *method*=`'tnc'`, *exponentRange*=`(None, None)`, ***kwargs*)

residualFunc function that returns a residual to minimize

initParams may be either a sequence of initial values or a callable returning initial values given a sequence of data to fit.

jacobian optional; it might improve computational performance.

Overrides: object.__init__ extit(inherited documentation)

plot(*self*, *colorByComponent*=`False`, *subfigures*=`False`, *relative*=`False`, ***kwargs*)

All quantities are relative, normalized by L2 norm of extrapolated solution.

multifit(*self*, ***kwargs*)

Generator of ansatz fits.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.ansatz.Ansatz(Section 2.1)

R(*self*, *f*, *m*, *c*)

Computes the discriminant, R, for a set of 3 meshes.

fit(*self*, *sampleXs*=`None`, *sampleYs*=`None`, *fExtrap*=`None`, *method*=`'tnc'`, *exponentRange*=`(None, None)`, ***kwargs*)

Fits parameter values to the samples, using the instance's residual function and parameter initial values. Extra kwargs get passed to the fitting function.

leastsq(*self*, *params*, *hs*, *es*)

Returns sum of squared residuals, for nonlinear least squares fitting.

multiInitFit(*self*, *sampleXs*=`None`, *sampleYs*=`None`, *fExtrap*=`None`, *method*=`'tnc'`, *exponentRange*=`(None, None)`, ***kwargs*)

Performs ansatz fits from multiple initializations.

paramsFromq(*cls*, *q*, *ys*, *hs*, *sfm*, *scm*)

Computes the prefactor, A, and the extrapolated solution, ye, from the exponent, q.

singleExpExtrapInit(*cls*, *hs*, *fs*, *forceGreater*=`False`)

Parameter initialization for error ansatz with a single exponential term in zone size. The y data are assumed to be field values (extrapolated solution used).

singleExpExtrapRes(*cls*, *params*, *hs*, *fs*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be field values (extrapolated solution used).

singleExpInit(*cls*, *h*, *e*)

Parameter initialization for error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

singleExpLogRes(*cls*, *params*, *hs*, *es*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

This residual uses the log of the ratio, which in theory might be more fair in weighting the fine mesh results. In (limited) practice, it's not clear if either shows a consistent advantage.

singleExpRes(*cls*, *params*, *hs*, *es*)

Error ansatz with a single exponential term in zone size. The y data are assumed to be errors (exact solution used).

6.1.2 Properties

Name	Description
Inherited from object	
__class__	

7 Module FunctionalConvergence.pyTest

7.1 Functions

monkeypatchDoctest()

monkeypatchTrace()

7.2 Variables

Name	Description
debug	**Value:** `False`
coverage	**Value:** `False`

8 Module FunctionalConvergence.sampler

The sampler module provides classes for implementing sampling algorithms for approximate integration over a mesh.

Usage Notes The python shipped with abaqus seems to be lacking in its unicode support, which causes problems with pretty-printing greek letters and other special chars. Use this command to ascii-ize the output:

```
/usr/local/abaqus/Commands/abq675 python ~/local/bin/pyTest.py sampler | cat
```

Or use an extractor.Pickle file with this command:

```
time ./pyTest.py sampler
```

Start an isympy shell with

```
python ../../third-party/sympy-0.6.4/bin/isympy
```

8.1 Class NodeSampler

```
object ─┐
        FunctionalConvergence.sampler.NodeSampler
```

Known Subclasses: FunctionalConvergence.sampler.MonteCarloSampler

samples at nodes only. iterates over elements, assigning an equal fraction of the element's volume to each node in the element.

- each node's weight will depend on how many elements it is in and the volumes of those elements.

to generate the pickle fe output needed by this doctest, test extractor.py.

subclasses of NodeSampler just need to implement .points and .weights.

```
>>> import extractor
>>> fe = extractor.Pickle('1Dwave.1.pickle')
>>> n = NodeSampler(fe)
```

for the node sampler, the points are just the nodes

```
>>> n.points() == n.fe.nodes
True
```

error checking

```
>>> n.volumes(((0.,0.,0.),(1.,0.,0.),(1.,1.,0.),(0.,1.,0.),
...             (0.,0.,.5),(1.,0.,.5),(1.,1.,.5),(0.,1.,.5)),
...            ((1,2,3,4,5,6,7,8),))
array([ 0.5])
>>> sum(n.weights) == n.volume #weight for any sampler should sum to volume
True
>>> str(NodeSampler._V12)[:70] #first instance caches symbolic stuff
'Poly((z[4] + z[5] - z[2] - z[3])*x[0]*y[1] + (z[1] - z[3])*x[0]*y[2] +'
```

8.1.1 Methods

__init__(*self*, *feOutput*)

feOutput extractor.Extractor (or subclass) instance

Overrides: object.__init__

points(*self*)

returns sample points.

volumes(*self*, *nodes*=`None`, *elements*=`None`)

returns an array of volumes of all the elements.

It's much faster to do this once for all elements than each one separately inside a higher-level loop.

nodeWeights(*self*, *nodes*=`None`, *elements*=`None`)

returns weights that should be multiplied by each node field value to approximate the integral.

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

8.1.2 Properties

Name	Description
Inherited from object	
__class__	

8.2 Class MonteCarloSampler

```
object ─┐
        │
FunctionalConvergence.sampler.NodeSampler ─┐
                                           │
                                           FunctionalConvergence.sampler.MonteCarloSamp
```

The MonteCarloSampler is not quite ready for prime time, but the essential elements are here if you need it and want to polish it off. Very subjectively with limited testing, it seems that the NodeSampler class does a good enough job for a lot less computation. I suppose that if you have high enough gradients on the mesh that NodeSampler does not work, you should probably suspect any results from your simulations, anyway.

This test integral is the x coord of element centroid.

```
>>> import extractor
>>> fe = extractor.Pickle('1Dwave.1.pickle')
>>> #fe2 = extractor.Pickle('1Dwave.2.pickle')
>>> nodeXs = zip(*fe.nodes)[0]
>>> elementXs = [sum([nodeXs[n-1] for n in e])/8. for e in fe.elements]
>>> fe._results[('testField',0)] = elementXs
>>> mc = MonteCarloSampler(fe)
>>> #fe.scalarField('testField',0)
>>> #mc.
```

```
>>> fe.elements = fe.elements[:2]
>>> fe.nodes = fe.nodes[:12]
>>> mc2 = MonteCarloSampler(fe)
```

#¿¿¿ mc2._polySolve(('et*et*et-4.25*et*et+6*et','-nu*nu*et+nu','nu*ze-et'), #... ('3*et*et-8.5*et+6','-2*nu*et+1','nu'), #... ((1.25,.64,.03),)*len(fe.elements)) #array([[0.25, 0.8 , 0.35], # [0.25, 0.8 , 0.35]])

```
>>> mc2.points(randomSeed=0)
array([[-1.36587584, -0.14240532,  0.19861831],
       [-2.02992209, -0.2881726 ,  0.17705294]])
```

```
>>> #import pdb;pdb.set_trace()
```

8.2.1 Methods

__init__(*self*, *feOutput*)

feOutput extractor.Extractor (or subclass) instance

Overrides: object.__init__ extit(inherited documentation)

points(*self*, *nodes*=`None`, *elements*=`None`, *randomSeed*=`None`)

randomSeed should be set for repeatable results (*eg.*, for testing); otherwise, ignore.

Overrides: FunctionalConvergence.sampler.NodeSampler.points

Inherited from object

__delattr__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __str__()

Inherited from FunctionalConvergence.sampler.NodeSampler(Section 8.1)

nodeWeights(*self*, *nodes*=`None`, *elements*=`None`)

returns weights that should be multiplied by each node field value to approximate the integral.

volumes(*self*, *nodes*=`None`, *elements*=`None`)

returns an array of volumes of all the elements.

It's much faster to do this once for all elements than each one separately inside a higher-level loop.

8.2.2 Properties

Name	Description
Inherited from object __class__	

Index