FInite element Automatic Tabulator (FIAT) Documentation

Release 2019.2.0.dev0

FEniCS Project

Jul 12, 2019

Contents

FIAT is a Python package for automatic generation of finite element basis functions. It is capable of generating finite element basis functions for a wide range of finite element families on simplices (lines, triangles and tetrahedra), including the Lagrange elements, and the elements of Raviart-Thomas, Brezzi-Douglas-Marini and Nedelec. It is also capable of generating tensor-product elements and a number more exotic elements, such as the Argyris, Hermite and Morley elements.

FIAT is part of the FEniCS Project.

For more information, visit http://www.fenicsproject.org.

CHAPTER 1

Documentation

1.1 Installation

FIAT is normally installed as part of an installation of FEniCS. If you are using FIAT as part of the FEniCS software suite, it is recommended that you follow the installation instructions for FEniCS.

To install FIAT itself, read on below for a list of requirements and installation instructions.

1.1.1 Requirements and dependencies

FIAT requires Python version 2.7 or later and depends on the following Python packages:

- NumPy
- SymPy

These packages will be automatically installed as part of the installation of FIAT, if not already present on your system.

1.1.2 Installation instructions

To install FIAT, download the source code from the FIAT Bitbucket repository, and run the following command:

```
pip install .
```

To install to a specific location, add the `--prefix` flag to the installation command:

```
pip install --prefix=<some directory> .
```

1.2 User manual

Note: This page is work in progress and needs substantial editing.

FIAT (FInite element Automatic Tabulator) is a Python package for defining and evaluating a wide range of different finite element basis functions for numerical partial differential equations. It is intended to make ‘‘difficult” elements such as high-order Brezzi-Douglas-Marini~cite{} elements usable by providing abstractions so that they may be implemented succinctly and hence treated as a black box. FIAT is intended for use at two different levels. For one, it is designed to provide a standard API for finite element bases so that programmers may use whatever elements they need in their code. At a lower level, it provides necessary infrastructure to rapidly deploy new kinds of finite elements without expensive symbolic computation or tedious algebraic manipulation. It is my goal that a large number of people use FIAT without ever knowing it. Thanks to several ongoing projects such as Sundance~cite{}, FFC~cite{}, and PETSc~cite{}, it is becoming possible to to define finite element methods using mathematical notation in some high-level or domain-specific language. The primary shortcoming of these projects is their lack of support for general elements. It is one thing to ‘‘provide hooks” for general elements, but absent a tool such as FIAT, these hooks remain mainly empty. As these projects mature, I hope to expose users of the finite element method to the exotic world of potentially high-degree finite element on unstructured grids using the best elements in H^1, $H(mathrm{div})$, and $H(mathrm{curl})$.

In this brief (and still developing) guide, I will first present the high-level API for users who wish to instantiate a finite element on a reference domain and evaluate its basis functions and derivatives at some quadrature points. Then, I will explain some of the underlying infrastructure so as to demonstrate how to add new elements.

chapter{Using FIAT: A tutorial with Lagrange elements} section{Importing FIAT} FIAT is organized as a package in Python, consisting of several modules. In order to get some of the packages, we use the line begin{verbatim} from FIAT import Lagrange, quadrature, shapes end{verbatim} This loads several modules for the Lagrange elements, quadrature rules, and the simplicial element shapes which FIAT implements. The roles each of these plays will become clear shortly.

section{Important note} Throughout, FIAT defines the reference elements based on the interval $(-1,1)$ rather than the more common $(0,1)$. So, the one-dimensional reference element is $(-1,1)$, the three vertices of the reference triangle are $(-1,-1),(1,-1),(1,-1)$, and the four vertices of the reference tetrahedron are $(-1,-1,-1),(1,-1,-1),(-1,1,-1),(-1,-1,1)$.

section{Instantiating elements} FIAT uses a lightweight object-oriented infrastructure to define finite elements. The verb.Lagrange. module contains a class verb.Lagrange. modeling the Lagrange finite element family. This class is a subclass of some verb.FiniteElement. class contained in another module (verb.polynomial. to be precise). So, having imported the verb.Lagrange. module, we can create the Lagrange element of degree verb.2. on triangles by begin{verbatim} shape = shapes.TRIANGLE degree = 2 U = Lagrange.Lagrange(shape , degree) end{verbatim} Here, verb/shapes.TRIANGLE/ is an integer code indicating the two dimensional simplex. verb.shapes. also defines verb.LINE. and verb.TETRAHEDRON.. Most of the upper-level interface to FIAT is dimensionally abstracted over element shape.

The class verb.FiniteElement. supports three methods, modeled on the abstract definition of Ciarlet. These methods are verb.domain_shape()., verb.function_space()., and verb.dual_basis().. The first of these returns the code for the shape and the second returns the nodes of the finite element (including information related to topological association of nodes with mesh entities, needed for creating degree of freedom orderings).

section{Quadrature rules} FIAT implements arbitrary-order collapsed quadrature, as discussed in Karniadakis and Sherwin~cite{}, for the simplex of dimension one, two, or three. The simplest way to get a quadrature rule is through the function verb.make_quadrature(shape,m)., which takes a shape code and an integer indicating the number of points per direction. For building element matrices using quadratics, we will typically need a second or third order integration rule, so we can get such a rule by begin{verbatim} >>> Q = quadrature.make_quadrature(shape , 2) end{verbatim} This uses two points in each direction on the reference square, then maps them to the reference triangle. We may get a verb/Numeric.array/ of the quadrature weights with the method verb/Q.get_weights()/ and a list of tuples storing the quadrature points with the method verb/Q.get_points()/.

section{Tabulation} FIAT provides functions for tabulating the element basis functions and their derivatives. To get the verb.FunctionSpace. object, we do begin{verbatim} >>> Ufs = U.function_space() end{verbatim} To get the values of each basis function at each of the quadrature points, we use the verb.tabulate(). method begin{verbatim} >>> Ufs.tabulate(Q.get_points()) array([[0.22176167, -0.12319761, -0.11479229, -0.06377178],

> [-0.11479229, -0.06377178, 0.22176167, -0.12319761], [-0.10696938, 0.18696938, -0.10696938, 0.18696938], [0.11074286, 0.19356495, 0.41329796, 0.72239423], [0.41329796, 0.72239423, 0.11074286, 0.19356495], [0.47595918, 0.08404082, 0.47595918, 0.08404082]])

end{verbatim} This returns a two-dimensional verb/Numeric.array/ with rows for each basis function and columns for each input point.

Also, finite element codes require tabulation of the basis functions' derivatives. Each verb/FunctionSpace/ object also provides a method verb/tabulate_jet(i,xs)/ that returns a list of Python dictionaries. The verb.i.th entry of the list is a dictionary storing the values of all verb.i.th order derivatives. Each dictionary maps a multiindex (a tuple of length verb.i.) to the table of the associated partial derivatives of the basis functions at those points. For example, begin{verbatim} >>> Ufs_jet = Ufs.tabulate_jet(1 , Q.get_points()) end{verbatim} tabulates the zeroth and first partial derivatives of the function space at the quadrature points. Then, begin{verbatim} >>> Ufs_jet[0] {(0, 0): array([[0.22176167, -0.12319761, -0.11479229, -0.06377178],

> [-0.11479229, -0.06377178, 0.22176167, -0.12319761], [-0.10696938, 0.18696938, -0.10696938, 0.18696938], [0.11074286, 0.19356495, 0.41329796, 0.72239423], [0.41329796, 0.72239423, 0.11074286, 0.19356495], [0.47595918, 0.08404082, 0.47595918, 0.08404082]])}

end{verbatim} gives us a dictionary mapping the only zeroth-order partial derivative to the values of the basis functions at the quadrature points. More interestingly, we may get the first derivatives in the x- and y- directions with begin{verbatim} >>> Ufs_jet[1][(1,0)] array([[-0.83278049, -0.06003983, 0.14288254, 0.34993778],

> [-0.14288254, -0.34993778, 0.83278049, 0.06003983], [0. , 0. , 0. , 0.], [0.31010205, 1.28989795, 0.31010205, 1.28989795], [-0.31010205, -1.28989795, -0.31010205, -1.28989795], [0.97566304, 0.40997761, -0.97566304, -0.40997761]])

```
>>> Ufs_jet[1][(0,1)]
array([[ -8.32780492e-01,  -6.00398310e-02,   1.42882543e-01,   3.49937780e-01],
       [  7.39494156e-17,   4.29608279e-17,   4.38075188e-17,   7.47961065e-17],
       [ -1.89897949e-01,   7.89897949e-01,  -1.89897949e-01,   7.89897949e-01],
       [  3.57117457e-01,   1.50062220e-01,   1.33278049e+00,   5.60039831e-01],
       [  1.02267844e+00,  -7.29858118e-01,   4.70154051e-02,  -1.13983573e+00],
       [ -3.57117457e-01,  -1.50062220e-01,  -1.33278049e+00,  -5.60039831e-01]])
\end{verbatim}
```

chapter{Lower-level API} Not only does FIAT provide a high-level library interface for users to evaluate existing finite element bases, but it also provides lower-level tools. Here, we survey these tools module-by-module.

section{shapes.py} FIAT currenly only supports simplicial reference elements, but does so in a fairly dimensionally-independent way (up to tetrahedra).

section{jacobi.py} This is a low-level module that tabulates the Jacobi polynomials and their derivatives, and also provides Gauss-Jacobi points. This module will seldom if ever be imported directly by users. For more information, consult the documentation strings and source code.

section{expansions.py} FIAT relies on orthonormal polynomial bases. These are constructed by mapping appropriate Jacobi polynomials from the reference cube to the reference simplex, as described in the reference of Karniadakis and Sherwin~cite{}. The module texttt{expansions.py} implements these orthonormal expansions. This is also a low-level module that will infrequently be used directly, but it forms the backbone for the module texttt{polynomial.py}

section{quadrature.py} FIAT makes heavy use of numerical quadrature, both internally and in the user interface. Internally, many function spaces or degrees of freedom are defined in terms of integral quantities having certain behavior. Keeping with the theme of arbitrary order approximations, FIAT provides arbitrary order quadrature rules on the reference simplices. These are constructed by mapping Gauss-Jacobi rules from the reference cube. While these rules are suboptimal in terms of order of accuracy achieved for a given number of points, they may be generated mechanically in a simpler way than symmetric quadrature rules. In the future, we hope to have the best symmetric existing rules integrated into FIAT.

Unless one is modifying the quadrature rules available, all of the functionality of texttt{quadrature.py} may be accessed through the single function verb.make_quadrature.. This function takes the code for a shape and the number of points in each coordinate direction and returns a quadrature rule. Internally, there is a lightweight class hierarchy rooted at an abstract texttt{QuadratureRule} class, where the quadrature rules for different shapes are actually different classes. However, the dynamic typing of Python relieves the user from these considerations. The interface to an instance consists in the following methods begin{itemize} item verb.get_points()., which returns a list of the quadrature

> points, each stored as a tuple. For dimensional uniformity, one-dimensional quadrature rules are stored as lists of 1-tuples rather than as lists of numbers.

item verb.get_weights()., which returns a texttt{Numeric.array} of quadrature weights.

item verb.integrate(f)., which takes a callable object texttt{f} and returns the (approximate) integral over the domain

item Also, the verb.__call__. method is overloaded so that a quadrature rule may be applied to a callable object. This is syntactic sugar on top of the texttt{integrate} method.

end{itemize}

section{polynomial.py} The texttt{polynomial} module provides the bulk of the classes needed to represent polynomial bases and finite element spaces. The class texttt{PolynomialBase} provides a high-level access to the orthonormal expansion bases; it is typically not instantiated directly in an application, but all other kinds of polynomial bases are constructed as linear combinations of the members of a texttt{PolynomialBase} instance. The module provides classes for scalar and vector-valued polynomial sets, as well as an interface to individual polynomials and finite element spaces.

subsection{texttt{PolynomialBase}}

subsection{texttt{PolynomialSet}} The texttt{PolynomialSet} function is a factory function interface into the hierarchy

1.3 FIAT package

1.3.1 Submodules

1.3.2 FIAT.P0 module

class `FIAT.P0.`**`P0`**(*ref_el*)
  Bases: *`FIAT.finite_element.CiarletElement`*

class `FIAT.P0.`**`P0Dual`**(*ref_el*)
  Bases: *`FIAT.dual_set.DualSet`*

1.3.3 FIAT.argyris module

class `FIAT.argyris.`**`Argyris`**(*ref_el*, *degree*)
  Bases: *`FIAT.finite_element.CiarletElement`*

  The Argyris finite element.

class `FIAT.argyris.`**`ArgyrisDualSet`**(*ref_el*, *degree*)
  Bases: *`FIAT.dual_set.DualSet`*

class FIAT.argyris.**QuinticArgyris**(*ref_el*)
Bases: *FIAT.finite_element.CiarletElement*

The Argyris finite element.

class FIAT.argyris.**QuinticArgyrisDualSet**(*ref_el*)
Bases: *FIAT.dual_set.DualSet*

1.3.4 FIAT.bell module

class FIAT.bell.**Bell**(*ref_el*)
Bases: *FIAT.finite_element.CiarletElement*

The Bell finite element.

class FIAT.bell.**BellDualSet**(*ref_el*)
Bases: *FIAT.dual_set.DualSet*

1.3.5 FIAT.bernstein module

class FIAT.bernstein.**Bernstein**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.FiniteElement*

A finite element with Bernstein polynomials as basis functions.

degree()
The degree of the polynomial space.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

Parameters

- **order** – The maximum order of derivative.
- **points** – An iterable of points.
- **entity** – Optional (dimension, entity number) pair indicating which topological entity of the reference element to tabulate on. If `None`, default cell-wise tabulation is performed.

value_shape()
The value shape of the finite element functions.

class FIAT.bernstein.**BernsteinDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

The dual basis for Bernstein elements.

FIAT.bernstein.**bernstein_Dx**(*points*, *ks*, *order*, *R2B*)
Evaluates Bernstein polynomials or its derivatives according to reference coordinates.

Parameters

- **points** – array of points in BARYCENTRIC COORDINATES
- **ks** – exponents defining the Bernstein polynomial
- **alpha** – derivative order (returns all derivatives of this specified order)
- **R2B** – linear mapping from reference to barycentric coordinates

Returns dictionary mapping from derivative tuples to arrays of Bernstein polynomial values at given points.

FIAT.bernstein.**bernstein_db**(*points*, *ks*, *alpha=None*)
Evaluates Bernstein polynomials or its derivative at barycentric points.

Parameters

- **points** – array of points in barycentric coordinates
- **ks** – exponents defining the Bernstein polynomial
- **alpha** – derivative tuple

Returns array of Bernstein polynomial values at given points.

1.3.6 FIAT.brezzi_douglas_fortin_marini module

class FIAT.brezzi_douglas_fortin_marini.**BDFMDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

FIAT.brezzi_douglas_fortin_marini.**BDFMSpace**(*ref_el*, *order*)

class FIAT.brezzi_douglas_fortin_marini.**BrezziDouglasFortinMarini**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The BDFM element

1.3.7 FIAT.brezzi_douglas_marini module

class FIAT.brezzi_douglas_marini.**BDMDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

class FIAT.brezzi_douglas_marini.**BrezziDouglasMarini**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The BDM element

1.3.8 FIAT.bubble module

class FIAT.bubble.**Bubble**(*ref_el*, *degree*)
Bases: *FIAT.bubble.CodimBubble*

The bubble finite element: the dofs of the Lagrange FE in the interior of the cell

class FIAT.bubble.**CodimBubble**(*ref_el*, *degree*, *codim*)
Bases: *FIAT.restricted.RestrictedElement*

Bubbles of a certain codimension.

class FIAT.bubble.**FacetBubble**(*ref_el*, *degree*)
Bases: *FIAT.bubble.CodimBubble*

The facet bubble finite element: the dofs of the Lagrange FE in the interior of the facets

1.3.9 FIAT.crouzeix_raviart module

class FIAT.crouzeix_raviart.**CrouzeixRaviart**(*cell*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The Crouzeix-Raviart finite element:

K: Triangle/Tetrahedron Polynomial space: P_1 Dual basis: Evaluation at facet midpoints

class FIAT.crouzeix_raviart.**CrouzeixRaviartDualSet**(*cell*, *degree*)
Bases: *FIAT.dual_set.DualSet*

Dual basis for Crouzeix-Raviart element (linears continuous at boundary midpoints).

1.3.10 FIAT.discontinuous module

class FIAT.discontinuous.**DiscontinuousElement**(*element*)
Bases: *FIAT.finite_element.CiarletElement*

A copy of an existing element where all dofs are associated with the cell

degree()
Return the degree of the (embedding) polynomial space.

dmats()
Return dmats: expansion coefficients for basis function derivatives.

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

get_order()
Return the order of the element (may be different from the degree)

get_reference_element()
Return the reference element for the finite element.

mapping()
Return a list of appropriate mappings from the reference element to a physical element for each basis function of the finite element.

num_sub_elements()
Return the number of sub-elements.

space_dimension()
Return the dimension of the finite element space.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

value_shape()
Return the value shape of the finite element functions.

1.3.11 FIAT.discontinuous_lagrange module

FIAT.discontinuous_lagrange.**DiscontinuousLagrange**(*ref_el*, *degree*)

class FIAT.discontinuous_lagrange.**DiscontinuousLagrangeDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

The dual basis for Lagrange elements. This class works for simplices of any dimension. Nodes are point evaluation at equispaced points. This is the discontinuous version where all nodes are topologically associated with the cell itself

class FIAT.discontinuous_lagrange.**HigherOrderDiscontinuousLagrange**(*ref_el*, *degree*)

Bases: *FIAT.finite_element.CiarletElement*

The discontinuous Lagrange finite element. It is what it is.

1.3.12 FIAT.discontinuous_pc module

FIAT.discontinuous_pc.**DPC**(*ref_el*, *degree*)

class FIAT.discontinuous_pc.**DPC0**(*ref_el*)

Bases: *FIAT.finite_element.CiarletElement*

class FIAT.discontinuous_pc.**DPCDualSet**(*ref_el*, *flat_el*, *degree*)

Bases: *FIAT.dual_set.DualSet*

The dual basis for DPC elements. This class works for hypercubes of any dimension. Nodes are point evaluation at equispaced points. This is the discontinuous version where all nodes are topologically associated with the cell itself

class FIAT.discontinuous_pc.**HigherOrderDPC**(*ref_el*, *degree*)

Bases: *FIAT.finite_element.CiarletElement*

The DPC finite element. It is what it is.

1.3.13 FIAT.discontinuous_raviart_thomas module

class FIAT.discontinuous_raviart_thomas.**DRTDualSet**(*ref_el*, *degree*)

Bases: *FIAT.dual_set.DualSet*

Dual basis for Raviart-Thomas elements consisting of point evaluation of normals on facets of codimension 1 and internal moments against polynomials. This is the discontinuous version where all nodes are topologically associated with the cell itself

class FIAT.discontinuous_raviart_thomas.**DiscontinuousRaviartThomas**(*ref_el*, *q*)

Bases: *FIAT.finite_element.CiarletElement*

The discontinuous Raviart-Thomas finite element

1.3.14 FIAT.discontinuous_taylor module

FIAT.discontinuous_taylor.**DiscontinuousTaylor**(*ref_el*, *degree*)

class FIAT.discontinuous_taylor.**DiscontinuousTaylorDualSet**(*ref_el*, *degree*)

Bases: *FIAT.dual_set.DualSet*

The dual basis for Taylor elements. This class works for intervals. Nodes are function and derivative evaluation at the midpoint.

class FIAT.discontinuous_taylor.**HigherOrderDiscontinuousTaylor**(*ref_el*, *degree*)

Bases: *FIAT.finite_element.CiarletElement*

The discontinuous Taylor finite element. Use a Taylor basis for DG.

1.3.15 FIAT.dual_set module

class FIAT.dual_set.**DualSet**(*nodes*, *ref_el*, *entity_ids*)
Bases: object

get_entity_closure_ids()

get_entity_ids()

get_nodes()

get_reference_element()

to_riesz(*poly_set*)

FIAT.dual_set.**make_entity_closure_ids**(*ref_el*, *entity_ids*)

1.3.16 FIAT.enriched module

class FIAT.enriched.**EnrichedElement**(**elements*)
Bases: *FIAT.finite_element.FiniteElement*

Class implementing a finite element that combined the degrees of freedom of two existing finite elements.

This is an implementation which does not care about orthogonality of primal and dual basis.

degree()
Return the degree of the (embedding) polynomial space.

dmats()
Return dmats: expansion coefficients for basis function derivatives.

elements()
Return reference to original subelements

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

value_shape()
Return the value shape of the finite element functions.

1.3.17 FIAT.expansions module

Principal orthogonal expansion functions as defined by Karniadakis and Sherwin. These are parametrized over a reference element so as to allow users to get coordinates that they want.

class FIAT.expansions.**LineExpansionSet**(*ref_el*)
Bases: object

Evaluates the Legendre basis on a line reference element.

get_num_members(*n*)

tabulate (*n*, *pts*)
Returns a numpy array A[i,j] = phi_i(pts[j])

tabulate_derivatives (*n*, *pts*)
Returns a tuple of length one (A,) such that A[i,j] = D phi_i(pts[j]). The tuple is returned for compatibility with the interfaces of the triangle and tetrahedron expansions.

class FIAT.expansions.**TetrahedronExpansionSet** (*ref_el*)
Bases: object

Collapsed orthonormal polynomial expanion on a tetrahedron.

get_num_members (*n*)

tabulate (*n*, *pts*)

tabulate_derivatives (*n*, *pts*)

tabulate_jet (*n*, *pts*, *order=1*)

class FIAT.expansions.**TriangleExpansionSet** (*ref_el*)
Bases: object

Evaluates the orthonormal Dubiner basis on a triangular reference element.

get_num_members (*n*)

tabulate (*n*, *pts*)

tabulate_derivatives (*n*, *pts*)

tabulate_jet (*n*, *pts*, *order=1*)

FIAT.expansions.**get_expansion_set** (*ref_el*)
Returns an ExpansionSet instance appopriate for the given reference element.

FIAT.expansions.**jrc** (*a*, *b*, *n*)

FIAT.expansions.**polynomial_dimension** (*ref_el*, *degree*)
Returns the dimension of the space of polynomials of degree no greater than degree on the reference element.

FIAT.expansions.**xi_tetrahedron** (*eta*)
Maps from [-1,1]^3 to the -1/1 reference tetrahedron.

FIAT.expansions.**xi_triangle** (*eta*)
Maps from [-1,1]^2 to the (-1,1) reference triangle.

1.3.18 FIAT.finite_element module

class FIAT.finite_element.**CiarletElement** (*poly_set*, *dual*, *order*, *formdegree=None*, *mapping='affine'*, *ref_el=None*)
Bases: *FIAT.finite_element.FiniteElement*

Class implementing Ciarlet's abstraction of a finite element being a domain, function space, and set of nodes.

Elements derived from this class are nodal finite elements, with a nodal basis generated from polynomials encoded in a *PolynomialSet*.

degree ()
Return the degree of the (embedding) polynomial space.

dmats ()
Return dmats: expansion coefficients for basis function derivatives.

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

static is_nodal()
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

All implementations/subclasses are nodal including this one.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

Parameters

- **order** – The maximum order of derivative.
- **points** – An iterable of points.
- **entity** – Optional (dimension, entity number) pair indicating which topological entity of the reference element to tabulate on. If `None`, default cell-wise tabulation is performed.

value_shape()
Return the value shape of the finite element functions.

class FIAT.finite_element.**FiniteElement**(*ref_el*, *dual*, *order*, *formdegree=None*, *mapping='affine'*)
Bases: `object`

Class implementing a basic abstraction template for general finite element families. Finite elements which inherit from this class are non-nodal unless they are CiarletElement subclasses.

dual_basis()
Return the dual basis (list of functionals) for the finite element.

entity_closure_dofs()
Return the map of topological entities to degrees of freedom on the closure of those entities for the finite element.

entity_dofs()
Return the map of topological entities to degrees of freedom for the finite element.

get_dual_set()
Return the dual for the finite element.

get_formdegree()
Return the degree of the associated form (FEEC)

get_order()
Return the order of the element (may be different from the degree).

get_reference_element()
Return the reference element for the finite element.

static is_nodal()
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

Subclasses may not necessarily be nodal, unless it is a CiarletElement.

mapping()
Return a list of appropriate mappings from the reference element to a physical element for each basis function of the finite element.

num_sub_elements()
Return the number of sub-elements.

space_dimension()
Return the dimension of the finite element space.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

Parameters

- **order** – The maximum order of derivative.
- **points** – An iterable of points.
- **entity** – Optional (dimension, entity number) pair indicating which topological entity of the reference element to tabulate on. If `None`, default cell-wise tabulation is performed.

FIAT.finite_element.**entity_support_dofs**(*elem*, *entity_dim*)
Return the map of entity id to the degrees of freedom for which the corresponding basis functions take non-zero values

Parameters

- **elem** – FIAT finite element
- **entity_dim** – Dimension of the cell subentity.

1.3.19 FIAT.functional module

class FIAT.functional.**ComponentPointEvaluation**(*ref_el*, *comp*, *shp*, *x*)
Bases: *FIAT.functional.Functional*

Class representing point evaluation of a particular component of a vector function at a particular point x.

tostr()

class FIAT.functional.**FrobeniusIntegralMoment**(*ref_el*, *Q*, *f_at_qpts*)
Bases: *FIAT.functional.Functional*

class FIAT.functional.**Functional**(*ref_el*, *target_shape*, *pt_dict*, *deriv_dict*, *functional_type*)
Bases: `object`

Class implementing an abstract functional. All functionals are discrete in the sense that the are written as a weighted sum of (components of) their argument evaluated at particular points.

evaluate(*f*)
Obsolete and broken functional evaluation.

To evaluate the functional, call it on the target function:

functional(function)

get_point_dict()
Returns the functional information, which is a dictionary mapping each point in the support of the functional to a list of pairs containing the weight and component.

get_reference_element()
Returns the reference element.

get_type_tag()
Returns the type of function (e.g. point evaluation or normal component, which is probably handy for clients of FIAT

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

tostr()

class FIAT.functional.**IntegralMoment**(*ref_el*, *Q*, *f_at_qpts*, *comp=()*, *shp=()*)
Bases: *FIAT.functional.Functional*

An IntegralMoment is a functional

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

class FIAT.functional.**IntegralMomentOfNormalDerivative**(*ref_el*, *facet_no*, *Q*, *f_at_qpts*)
Bases: *FIAT.functional.Functional*

Functional giving normal derivative integrated against some function on a facet.

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

class FIAT.functional.**PointDerivative**(*ref_el*, *x*, *alpha*)
Bases: *FIAT.functional.Functional*

Class representing point partial differentiation of scalar functions at a particular point x.

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

class FIAT.functional.**PointEdgeTangentEvaluation**(*ref_el*, *edge_no*, *pt*)
Bases: *FIAT.functional.Functional*

Implements the evaluation of the tangential component of a vector at a point on a facet of dimension 1.

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

tostr()

class FIAT.functional.**PointEvaluation**(*ref_el*, *x*)
Bases: *FIAT.functional.Functional*

Class representing point evaluation of scalar functions at a particular point x.

tostr()

class FIAT.functional.**PointFaceTangentEvaluation**(*ref_el*, *face_no*, *tno*, *pt*)
Bases: *FIAT.functional.Functional*

Implements the evaluation of a tangential component of a vector at a point on a facet of codimension 1.

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

tostr()

class FIAT.functional.**PointNormalDerivative**(*ref_el*, *facet_no*, *pt*)
Bases: *FIAT.functional.Functional*

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

class FIAT.functional.**PointNormalEvaluation**(*ref_el*, *facet_no*, *pt*)
Bases: *FIAT.functional.Functional*

Implements the evaluation of the normal component of a vector at a point on a facet of codimension 1.

class FIAT.functional.**PointScaledNormalEvaluation**(*ref_el*, *facet_no*, *pt*)
Bases: *FIAT.functional.Functional*

Implements the evaluation of the normal component of a vector at a point on a facet of codimension 1, where the normal is scaled by the volume of that facet.

to_riesz(*poly_set*)
Constructs an array representation of the functional over the base of the given polynomial_set so that f(phi) for any phi in poly_set is given by a dot product.

tostr()

class FIAT.functional.**PointwiseInnerProductEvaluation**(*ref_el*, *v*, *w*, *p*)
Bases: *FIAT.functional.Functional*

This is a functional on symmetric 2-tensor fields. Let u be such a field, p be a point, and v,w be vectors. This implements the evaluation v^T u(p) w.

Clearly v^iu_{ij}w^j = u_{ij}v^iw^j. Thus the value can be computed from the Frobenius inner product of u with wv^T. This gives the correct weights.

FIAT.functional.**index_iterator**(*shp*)
Constructs a generator iterating over all indices in shp in generalized column-major order So if shp = (2,2), then we construct the sequence (0,0),(0,1),(1,0),(1,1)

1.3.20 FIAT.gauss_legendre module

class FIAT.gauss_legendre.**GaussLegendre**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

1D discontinuous element with nodes at the Gauss-Legendre points.

class FIAT.gauss_legendre.**GaussLegendreDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

The dual basis for 1D discontinuous elements with nodes at the Gauss-Legendre points.

1.3.21 FIAT.gauss_lobatto_legendre module

class FIAT.gauss_lobatto_legendre.**GaussLobattoLegendre**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

1D continuous element with nodes at the Gauss-Lobatto points.

class FIAT.gauss_lobatto_legendre.**GaussLobattoLegendreDualSet**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

The dual basis for 1D continuous elements with nodes at the Gauss-Lobatto points.

1.3.22 FIAT.hdiv_trace module

class FIAT.hdiv_trace.**HDivTrace**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.FiniteElement*

Class implementing the trace of hdiv elements. This class is a stand-alone element family that produces a DG-facet field. This element is what's produced after performing the trace operation on an existing H(Div) element.

This element is also known as the discontinuous trace field that arises in several DG formulations.

degree()
Return the degree of the (embedding) polynomial space.

dmats()
Return dmats: expansion coefficients for basis function derivatives.

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

static is_nodal()
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

Subclasses may not necessarily be nodal, unless it is a CiarletElement.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to a given order of basis functions at given points.

Parameters

- **order** – The maximum order of derivative.
- **points** – An iterable of points.
- **entity** – Optional (dimension, entity number) pair indicating which topological entity of the reference element to tabulate on. If None, tabulated values are computed by geometrically approximating which facet the points are on.

Note: Performing illegal tabulations on this element will result in either a tabulation table of *numpy.nan* arrays (*entity=None* case), or insertions of the *TraceError* exception class. This is due to the fact that performing cell-wise tabulations, or asking for any order of derivative evaluations, are not mathematically well-defined.

value_shape()
Return the value shape of the finite element functions.

exception FIAT.hdiv_trace.**TraceError**(*msg*)
Bases: Exception

Exception caused by tabulating a trace element on the interior of a cell, or the gradient of a trace element.

FIAT.hdiv_trace.**barycentric_coordinates**(*points*, *vertices*)
Computes the barycentric coordinates for a set of points relative to a simplex defined by a set of vertices.

Parameters

- **points** – A set of points.
- **vertices** – A set of vertices that define the simplex.

FIAT.hdiv_trace.**construct_dg_element**(*ref_el*, *degree*)
Constructs a discontinuous galerkin element of a given degree on a particular reference cell.

FIAT.hdiv_trace.**extract_unique_facet**(*coordinates*, *tolerance=1e-10*)
Determines whether a set of points (described in its barycentric coordinates) are all on one of the facet subentities, and return the particular facet and whether the search has been successful.

Parameters

- **coordinates** – A set of points described in barycentric coordinates.
- **tolerance** – A fixed tolerance for geometric identifications.

FIAT.hdiv_trace.**map_from_reference_facet**(*point*, *vertices*)
Evaluates the physical coordinate of a point using barycentric coordinates.

Parameters

- **point** – The reference points to be mapped to the facet.
- **vertices** – The vertices defining the physical element.

FIAT.hdiv_trace.**map_to_reference_facet**(*points*, *vertices*, *facet*)
Given a set of points and vertices describing a facet of a simplex in n-dimensional coordinates (where the points lie on the facet), map the points to the reference simplex of dimension (n-1).

Parameters

- **points** – A set of points in n-D.
- **vertices** – A set of vertices describing a facet of a simplex in n-D.
- **facet** – Integer representing the facet number.

1.3.23 FIAT.hdivcurl module

FIAT.hdivcurl.**Hcurl**(*element*)

FIAT.hdivcurl.**Hdiv**(*element*)

1.3.24 FIAT.hellan_herrmann_johnson module

Implementation of the Hellan-Herrmann-Johnson finite elements.

class FIAT.hellan_herrmann_johnson.**HellanHerrmannJohnson**(*cell*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The definition of Hellan-Herrmann-Johnson element. It is defined only in dimension 2. It consists of piecewise polynomial symmetric-matrix-valued functions of degree r or less with normal-normal continuity.

class FIAT.hellan_herrmann_johnson.**HellanHerrmannJohnsonDual**(*cell*, *degree*)
Bases: *FIAT.dual_set.DualSet*

Degrees of freedom for Hellan-Herrmann-Johnson elements.

1.3.25 FIAT.hermite module

class FIAT.hermite.**CubicHermite**(*ref_el*)

Bases: *FIAT.finite_element.CiarletElement*

The cubic Hermite finite element. It is what it is.

class FIAT.hermite.**CubicHermiteDualSet**(*ref_el*)

Bases: *FIAT.dual_set.DualSet*

The dual basis for Lagrange elements. This class works for simplices of any dimension. Nodes are point evaluation at equispaced points.

1.3.26 FIAT.jacobi module

Several functions related to the one-dimensional jacobi polynomials: Evaluation, evaluation of derivatives, plus computation of the roots via Newton's method. These mainly are used in defining the expansion functions over the simplices and in defining quadrature rules over each domain.

FIAT.jacobi.**eval_jacobi**(*a*, *b*, *n*, *x*)

Evaluates the nth jacobi polynomial with weight parameters a,b at a point x. Recurrence relations implemented from the pseudocode given in Karniadakis and Sherwin, Appendix B

FIAT.jacobi.**eval_jacobi_batch**(*a*, *b*, *n*, *xs*)

Evaluates all jacobi polynomials with weights a,b up to degree n. xs is a numpy.array of points. Returns a two-dimensional array of points, where the rows correspond to the Jacobi polynomials and the columns correspond to the points.

FIAT.jacobi.**eval_jacobi_deriv**(*a*, *b*, *n*, *x*)

Evaluates the first derivative of P_{n}^{a,b} at a point x.

FIAT.jacobi.**eval_jacobi_deriv_batch**(*a*, *b*, *n*, *xs*)

Evaluates the first derivatives of all jacobi polynomials with weights a,b up to degree n. xs is a numpy.array of points. Returns a two-dimensional array of points, where the rows correspond to the Jacobi polynomials and the columns correspond to the points.

1.3.27 FIAT.lagrange module

class FIAT.lagrange.**Lagrange**(*ref_el*, *degree*)

Bases: *FIAT.finite_element.CiarletElement*

The Lagrange finite element. It is what it is.

class FIAT.lagrange.**LagrangeDualSet**(*ref_el*, *degree*)

Bases: *FIAT.dual_set.DualSet*

The dual basis for Lagrange elements. This class works for simplices of any dimension. Nodes are point evaluation at equispaced points.

1.3.28 FIAT.mixed module

class FIAT.mixed.**MixedElement**(*elements*, *ref_el=None*)

Bases: *FIAT.finite_element.FiniteElement*

A FIAT-like representation of a mixed element.

Parameters

- **elements** – An iterable of FIAT elements.
- **ref_el** – The reference element (optional).

This object offers tabulation of the concatenated basis function tables along with an entity_dofs dict.

elements()

get_nodal_basis()

is_nodal()
True if primal and dual bases are orthogonal.

mapping()
Return a list of appropriate mappings from the reference element to a physical element for each basis function of the finite element.

num_sub_elements()
Return the number of sub-elements.

tabulate(*order*, *points*, *entity=None*)
Tabulate a mixed element by appropriately splatting together the tabulation of the individual elements.

value_shape()

FIAT.mixed.**concatenate_entity_dofs**(*ref_el*, *elements*)
Combine the entity_dofs from a list of elements into a combined entity_dof containing the information for the concatenated DoFs of all the elements.

1.3.29 FIAT.morley module

class FIAT.morley.**Morley**(*ref_el*)
Bases: *FIAT.finite_element.CiarletElement*

The Morley finite element.

class FIAT.morley.**MorleyDualSet**(*ref_el*)
Bases: *FIAT.dual_set.DualSet*

The dual basis for Lagrange elements. This class works for simplices of any dimension. Nodes are point evaluation at equispaced points.

1.3.30 FIAT.nedelec module

class FIAT.nedelec.**Nedelec**(*ref_el*, *q*)
Bases: *FIAT.finite_element.CiarletElement*

Nedelec finite element

class FIAT.nedelec.**NedelecDual2D**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

Dual basis for first-kind Nedelec in 2d

class FIAT.nedelec.**NedelecDual3D**(*ref_el*, *degree*)
Bases: *FIAT.dual_set.DualSet*

Dual basis for first-kind Nedelec in 3d

FIAT.nedelec.**NedelecSpace2D**(*ref_el*, *k*)
Constructs a basis for the 2d H(curl) space of the first kind which is (P_k)^2 + P_k rot(x)

FIAT.nedelec.**NedelecSpace3D**(*ref_el*, *k*)
Constructs a nodal basis for the 3d first-kind Nedelec space

1.3.31 FIAT.nedelec_second_kind module

class FIAT.nedelec_second_kind.**NedelecSecondKind**(*cell*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The H(curl) Nedelec elements of the second kind on triangles and tetrahedra: the polynomial space described by the full polynomials of degree k, with a suitable set of degrees of freedom to ensure H(curl) conformity.

class FIAT.nedelec_second_kind.**NedelecSecondKindDual**(*cell*, *degree*)
Bases: *FIAT.dual_set.DualSet*

This class represents the dual basis for the Nedelec H(curl) elements of the second kind. The degrees of freedom (L) for the elements of the k'th degree are

d = 2:

vertices: None

edges: L(f) = f (x_i) * t for (k+1) points x_i on each edge

cell: L(f) = int f * g * dx for g in RT_{k-1}

d = 3:

vertices: None

edges: L(f) = f(x_i) * t for (k+1) points x_i on each edge

faces: L(f) = int_F f * g * ds for g in RT_{k-1}(F) for each face F

cell: L(f) = int f * g * dx for g in RT_{k-2}

Higher spatial dimensions are not yet implemented. (For d = 1, these elements coincide with the CG_k elements.)

generate_degrees_of_freedom(*cell*, *degree*)
Generate dofs and geometry-to-dof maps (ids).

1.3.32 FIAT.nodal_enriched module

class FIAT.nodal_enriched.**NodalEnrichedElement**(**elements*)
Bases: *FIAT.finite_element.CiarletElement*

NodalEnriched element is a direct sum of a sequence of finite elements. Dual basis is reorthogonalized to the primal basis for nodality.

The following is equivalent:

- the constructor is well-defined,
- the resulting element is unisolvent and its basis is nodal,
- the supplied elements are unisolvent with nodal basis and their primal bases are mutually linearly independent,
- the supplied elements are unisolvent with nodal basis and their dual bases are mutually linearly independent.

1.3.33 FIAT.orthopoly module

orthopoly.py - A suite of functions for generating orthogonal polynomials and quadrature rules.

Last updated on Wed Jan 1 14:29:25 MST 2014

Modified by David A. Ham (david.ham@imperial.ac.uk), 2016

`FIAT.orthopoly.`**`gauss`**`(`*alpha*, *beta*`)`

Compute the Gauss nodes and weights from the recursion coefficients associated with a set of orthogonal polynomials

Inputs: alpha - recursion coefficients beta - recursion coefficients

Outputs: x - quadrature nodes w - quadrature weights

Adapted from the MATLAB code by Walter Gautschi http://www.cs.purdue.edu/archives/2002/wxg/codes/gauss.m

`FIAT.orthopoly.`**`jacobi`**`(`*N*, *a*, *b*, *x*, *NOPT=1*`)`

JACOBI computes the Jacobi polynomials which are orthogonal on [-1,1] with respect to the weight w(x)=[(1-x)^a]*[(1+x)^b] and evaluate them on the given grid up to P_N(x). Setting NOPT=2 returns the L2-normalized polynomials

`FIAT.orthopoly.`**`jacobiD`**`(`*N*, *a*, *b*, *x*, *NOPT=1*`)`

JACOBID computes the first derivatives of the normalized Jacobi polynomials which are orthogonal on [-1,1] with respect to the weight w(x)=[(1-x)^a]*[(1+x)^b] and evaluate them on the given grid up to P_N(x). Setting NOPT=2 returns the derivatives of the L2-normalized polynomials

`FIAT.orthopoly.`**`lobatto`**`(`*alpha*, *beta*, *xl1*, *xl2*`)`

Compute the Lobatto nodes and weights with the preassigned nodea xl1,xl2

Inputs: alpha - recursion coefficients beta - recursion coefficients xl1 - assigned node location xl2 - assigned node location

Outputs: x - quadrature nodes w - quadrature weights

Based on the section 7 of the paper "Some modified matrix eigenvalue problems" by Gene Golub, SIAM Review Vol 15, No. 2, April 1973, pp.318–334

`FIAT.orthopoly.`**`mm_log`**`(`*N*, *a*`)`

MM_LOG Modified moments for a logarithmic weight function.

The call mm=MM_LOG(n,a) computes the first n modified moments of the logarithmic weight function w(t)=t^a log(1/t) on [0,1] relative to shifted Legendre polynomials.

REFERENCE: Walter Gautschi,''On the preceding paper 'A Legendre polynomial integral' by James L. Blue'', Math. Comp. 33 (1979), 742-743.

Adapted from the MATLAB implementation: https://www.cs.purdue.edu/archives/2002/wxg/codes/mm_log.m

`FIAT.orthopoly.`**`mod_chebyshev`**(*N*, *mom*, *alpham*, *betam*)

Calcuate the recursion coefficients for the orthogonal polynomials which are are orthogonal with respect to a weight function which is represented in terms of its modifed moments which are obtained by integrating the monic polynomials against the weight function.

REFERENCES:

John C. Wheeler, "Modified moments and Gaussian quadratures" Rocky Mountain Journal of Mathematics, Vol. 4, Num. 2 (1974), 287–296

Walter Gautschi, "Orthogonal Polynomials (in Matlab) Journal of Computational and Applied Mathematics, Vol. 178 (2005) 215–234

Adapted from the MATLAB implementation: https://www.cs.purdue.edu/archives/2002/wxg/codes/chebyshev.m

`FIAT.orthopoly.`**`polyval`**(*alpha*, *beta*, *x*)

Evaluate polynomials on x given the recursion coefficients alpha and beta

`FIAT.orthopoly.`**`rec_jaclog`**(*N*, *a*)

Generate the recursion coefficients alpha_k, beta_k

P_{k+1}(x) = (x-alpha_k)*P_{k}(x) - beta_k P_{k-1}(x)

for the monic polynomials which are orthogonal on [0,1] with respect to the weight w(x)=x^a*log(1/x)

Inputs: N - polynomial order a - weight parameter

Outputs: alpha - recursion coefficients beta - recursion coefficients

Adated from the MATLAB code: https://www.cs.purdue.edu/archives/2002/wxg/codes/r_jaclog.m

`FIAT.orthopoly.`**`rec_jacobi`**(*N*, *a*, *b*)

Generate the recursion coefficients alpha_k, beta_k

P_{k+1}(x) = (x-alpha_k)*P_{k}(x) - beta_k P_{k-1}(x)

for the Jacobi polynomials which are orthogonal on [-1,1] with respect to the weight w(x)=[(1-x)^a]*[(1+x)^b]

Inputs: N - polynomial order a - weight parameter b - weight parameter

Outputs: alpha - recursion coefficients beta - recursion coefficients

Adapted from the MATLAB code by Dirk Laurie and Walter Gautschi http://www.cs.purdue.edu/archives/2002/wxg/codes/r_jacobi.m

`FIAT.orthopoly.`**`rec_jacobi01`**(*N*, *a*, *b*)

Generate the recursion coefficients alpha_k, beta_k for the Jacobi polynomials which are orthogonal on [0,1]

See rec_jacobi for the recursion coefficients on [-1,1]

Inputs: N - polynomial order a - weight parameter b - weight parameter

Outputs: alpha - recursion coefficients beta - recursion coefficients

Adapted from the MATLAB implementation: https://www.cs.purdue.edu/archives/2002/wxg/codes/r_jacobi01.m

1.3.34 FIAT.polynomial_set module

class FIAT.polynomial_set.**ONPolynomialSet**(*ref_el*, *degree*, *shape=()*)
Bases: *FIAT.polynomial_set.PolynomialSet*

Constructs an orthonormal basis out of expansion set by having an identity matrix of coefficients. Can be used to specify ON bases for vector- and tensor-valued sets as well.

class FIAT.polynomial_set.**ONSymTensorPolynomialSet**(*ref_el*, *degree*, *size=None*)
Bases: *FIAT.polynomial_set.PolynomialSet*

Constructs an orthonormal basis for symmetric-tensor-valued polynomials on a reference element.

class FIAT.polynomial_set.**PolynomialSet**(*ref_el*, *degree*, *embedded_degree*, *expansion_set*, *coeffs*, *dmats*)
Bases: object

Implements a set of polynomials as linear combinations of an expansion set over a reference element. ref_el: the reference element degree: an order labeling the space embedded degree: the degree of polynomial expansion basis that

must be used to evaluate this space

coeffs: A numpy array containing the coefficients of the expansion basis for each member of the set. Coeffs is ordered by coeffs[i,j,k] where i is the label of the member, k is the label of the expansion function, and j is a (possibly empty) tuple giving the index for a vector- or tensor-valued function.

get_coeffs()

get_degree()

get_dmats()

get_embedded_degree()

get_expansion_set()

get_num_members()

get_reference_element()

get_shape()
Returns the shape of phi(x), where () corresponds to scalar (2,) a vector of length 2, etc

tabulate(*pts*, *jet_order=0*)
Returns the values of the polynomial set.

tabulate_new(*pts*)

take(*items*)
Extracts subset of polynomials given by items.

FIAT.polynomial_set.**form_matrix_product**(*mats*, *alpha*)
forms product over mats[i]**alpha[i]

FIAT.polynomial_set.**mis**(*m*, *n*)
returns all m-tuples of nonnegative integers that sum up to n.

FIAT.polynomial_set.**polynomial_set_union_normalized**(*A*, *B*)
Given polynomial sets A and B, constructs a new polynomial set whose span is the same as that of span(A) union span(B). It may not contain any of the same members of the set, as we construct a span via SVD.

FIAT.polynomial_set.**project** (*f*, *U*, *Q*)
Computes the expansion coefficients of f in terms of the members of a polynomial set U. Numerical integration is performed by quadrature rule Q.

1.3.35 FIAT.quadrature module

class FIAT.quadrature.**CollapsedQuadratureTetrahedronRule** (*ref_el*, *m*)
Bases: *FIAT.quadrature.QuadratureRule*

Implements the collapsed quadrature rules defined in Karniadakis & Sherwin by mapping products of Gauss-Jacobi rules from the cube to the tetrahedron.

class FIAT.quadrature.**CollapsedQuadratureTriangleRule** (*ref_el*, *m*)
Bases: *FIAT.quadrature.QuadratureRule*

Implements the collapsed quadrature rules defined in Karniadakis & Sherwin by mapping products of Gauss-Jacobi rules from the square to the triangle.

class FIAT.quadrature.**GaussJacobiQuadratureLineRule** (*ref_el*, *m*)
Bases: *FIAT.quadrature.QuadratureRule*

Gauss-Jacobi quadature rule determined by Jacobi weights a and b using m roots of m:th order Jacobi polynomial.

class FIAT.quadrature.**GaussLegendreQuadratureLineRule** (*ref_el*, *m*)
Bases: *FIAT.quadrature.QuadratureRule*

Produce the Gauss–Legendre quadrature rules on the interval using the implementation in numpy. This facilitates implementing discontinuous spectral elements.

The quadrature rule uses m points for a degree of precision of 2m-1.

class FIAT.quadrature.**GaussLobattoLegendreQuadratureLineRule** (*ref_el*, *m*)
Bases: *FIAT.quadrature.QuadratureRule*

Implement the Gauss-Lobatto-Legendre quadrature rules on the interval using Greg von Winckel's implementation. This facilitates implementing spectral elements.

The quadrature rule uses m points for a degree of precision of 2m-3.

class FIAT.quadrature.**QuadratureRule** (*ref_el*, *pts*, *wts*)
Bases: object

General class that models integration over a reference element as the weighted sum of a function evaluated at a set of points.

get_points ()

get_weights ()

integrate (*f*)

class FIAT.quadrature.**UFCTetrahedronFaceQuadratureRule** (*face_number*, *degree*)
Bases: *FIAT.quadrature.QuadratureRule*

Highly specialized quadrature rule for the face of a tetrahedron, mapped from a reference triangle, used for higher order Nedelecs

jacobian ()

reference_rule ()

`FIAT.quadrature.compute_gauss_jacobi_points(a, b, m)`
Computes the m roots of P_{m}^{a,b} on [-1,1] by Newton's method. The initial guesses are the Chebyshev points. Algorithm implemented in Python from the pseudocode given by Karniadakis and Sherwin

`FIAT.quadrature.compute_gauss_jacobi_rule(a, b, m)`

`FIAT.quadrature.make_quadrature(ref_el, m)`
Returns the collapsed quadrature rule using m points per direction on the given reference element. In the tensor product case, m is a tuple.

`FIAT.quadrature.make_tensor_product_quadrature(*quad_rules)`
Returns the quadrature rule for a TensorProduct cell, by combining the quadrature rules of the components.

1.3.36 FIAT.quadrature_element module

`class FIAT.quadrature_element.QuadratureElement(ref_el, points)`
Bases: `FIAT.finite_element.FiniteElement`

A set of quadrature points pretending to be a finite element.

`static is_nodal()`
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

Subclasses may not necessarily be nodal, unless it is a CiarletElement.

`tabulate(order, points, entity=None)`
Return the identity matrix of size (num_quad_points, num_quad_points), in a format that monomialintegration and monomialtabulation understands.

`value_shape()`
The QuadratureElement is scalar valued

1.3.37 FIAT.quadrature_schemes module

Quadrature schemes on cells

This module generates quadrature schemes on reference cells that integrate exactly a polynomial of a given degree using a specified scheme.

Scheme options are:

scheme="default"

scheme="canonical" (collapsed Gauss scheme)

Background on the schemes:

Keast rules for tetrahedra: Keast, P. Moderate-degree tetrahedral quadrature formulas, Computer Methods in Applied Mechanics and Engineering 55(3):339-348, 1986. http://dx.doi.org/10.1016/0045-7825(86)90059-9

`FIAT.quadrature_schemes.create_quadrature(ref_el, degree, scheme='default')`
Generate quadrature rule for given reference element that will integrate an polynomial of order 'degree' exactly.

For low-degree (<=6) polynomials on triangles and tetrahedra, this uses hard-coded rules, otherwise it falls back to a collapsed Gauss scheme on simplices. On tensor-product cells, it is a tensor-product quadrature rule of the subcells.

Parameters

- **cell** – The FIAT cell to create the quadrature for.

- **degree** – The degree of polynomial that the rule should integrate exactly.

1.3.38 FIAT.raviart_thomas module

class FIAT.raviart_thomas.**RTDualSet**(*ref_el*, *degree*)

Bases: *FIAT.dual_set.DualSet*

Dual basis for Raviart-Thomas elements consisting of point evaluation of normals on facets of codimension 1 and internal moments against polynomials

FIAT.raviart_thomas.**RTSpace**(*ref_el*, *deg*)

Constructs a basis for the the Raviart-Thomas space (P_k)^d + P_k x

class FIAT.raviart_thomas.**RaviartThomas**(*ref_el*, *q*)

Bases: *FIAT.finite_element.CiarletElement*

The Raviart-Thomas finite element

1.3.39 FIAT.reference_element module

Abstract class and particular implementations of finite element reference simplex geometry/topology.

Provides an abstract base class and particular implementations for the reference simplex geometry and topology. The rest of FIAT is abstracted over this module so that different reference element geometry (e.g. a vertex at (0,0) versus at (-1,-1)) and orderings of entities have a single point of entry.

Currently implemented are UFC and Default Line, Triangle and Tetrahedron.

class FIAT.reference_element.**Cell**(*shape*, *vertices*, *topology*)

Bases: object

Abstract class for a reference cell. Provides accessors for geometry (vertex coordinates) as well as topology (orderings of vertices that make up edges, facecs, etc.

construct_subelement(*dimension*)

Constructs the reference element of a cell subentity specified by subelement dimension.

Parameters **dimension** – *tuple* for tensor product cells, *int* otherwise

get_connectivity()

Returns a dictionary encoding the connectivity of the element.

The dictionary's keys are the spatial dimensions pairs ((1, 0), (2, 0), (2, 1), . . .) and each value is a list with entities of second dimension ordered by local dim0-dim1 numbering.

get_dimension()

Returns the subelement dimension of the cell. For tensor product cells, this a tuple of dimensions for each cell in the product. For all other cells, this is the same as the spatial dimension.

get_entity_transform(*dim*, *entity_i*)

Returns a mapping of point coordinates from the *entity_i*-th subentity of dimension *dim* to the cell.

Parameters

- **dim** – *tuple* for tensor product cells, *int* otherwise
- **entity_i** – entity number (integer)

get_shape()

Returns the code for the element's shape.

get_spatial_dimension()
Returns the spatial dimension in which the element lives.

get_topology()
Returns a dictionary encoding the topology of the element.

The dictionary's keys are the spatial dimensions (0, 1, . . .) and each value is a dictionary mapping.

get_vertices()
Returns an iterable of the element's vertices, each stored as a tuple.

get_vertices_of_subcomplex(*t*)
Returns the tuple of vertex coordinates associated with the labels contained in the iterable t.

class FIAT.reference_element.**DefaultLine**
Bases: *FIAT.reference_element.Simplex*

This is the reference line with vertices (-1.0,) and (1.0,).

get_facet_element()

class FIAT.reference_element.**DefaultTetrahedron**
Bases: *FIAT.reference_element.Simplex*

This is the reference tetrahedron with vertices (-1,-1,-1), (1,-1,-1),(-1,1,-1), and (-1,-1,1).

get_facet_element()

class FIAT.reference_element.**DefaultTriangle**
Bases: *FIAT.reference_element.Simplex*

This is the reference triangle with vertices (-1.0,-1.0), (1.0,-1.0), and (-1.0,1.0).

get_facet_element()

class FIAT.reference_element.**IntrepidTetrahedron**
Bases: *FIAT.reference_element.Simplex*

This is the reference tetrahedron with vertices (0,0,0), (1,0,0),(0,1,0), and (0,0,1) used in the Intrepid project.

get_facet_element()

class FIAT.reference_element.**IntrepidTriangle**
Bases: *FIAT.reference_element.Simplex*

This is the Intrepid triangle with vertices (0,0),(1,0),(0,1)

get_facet_element()

class FIAT.reference_element.**Point**
Bases: *FIAT.reference_element.Simplex*

This is the reference point.

FIAT.reference_element.**ReferenceElement**
alias of *FIAT.reference_element.Simplex*

class FIAT.reference_element.**Simplex**(*shape*, *vertices*, *topology*)
Bases: *FIAT.reference_element.Cell*

Abstract class for a reference simplex.

compute_edge_tangent(*edge_i*)
Computes the nonnormalized tangent to a 1-dimensional facet. returns a single vector.

compute_face_edge_tangents (*dim*, *entity_id*)
Computes all the edge tangents of any k-face with k>=1. The result is a array of binom(dim+1,2) vectors. This agrees with *compute_edge_tangent* when dim=1.

compute_face_tangents (*face_i*)
Computes the two tangents to a face. Only implemented for a tetrahedron.

compute_normal (*facet_i*)
Returns the unit normal vector to facet i of codimension 1.

compute_normalized_edge_tangent (*edge_i*)
Computes the unit tangent vector to a 1-dimensional facet

compute_normalized_tangents (*dim*, *i*)
computes tangents in any dimension based on differences between vertices and the first vertex of the i:th facet of dimension dim. Returns a (possibly empty) list. These tangents are normalized to have unit length.

compute_reference_normal (*facet_dim*, *facet_i*)
Returns the unit normal in infinity norm to facet_i.

compute_scaled_normal (*facet_i*)
Returns the unit normal to facet_i of scaled by the volume of that facet.

compute_tangents (*dim*, *i*)
computes tangents in any dimension based on differences between vertices and the first vertex of the i:th facet of dimension dim. Returns a (possibly empty) list. These tangents are *NOT* normalized to have unit length.

get_dimension ()
Returns the subelement dimension of the cell. Same as the spatial dimension.

get_entity_transform (*dim*, *entity*)
Returns a mapping of point coordinates from the *entity*-th subentity of dimension *dim* to the cell.

Parameters

- **dim** – subentity dimension (integer)
- **entity** – entity number (integer)

make_points (*dim*, *entity_id*, *order*)
Constructs a lattice of points on the entity_id:th facet of dimension dim. Order indicates how many points to include in each direction.

volume ()
Computes the volume of the simplex in the appropriate dimensional measure.

volume_of_subcomplex (*dim*, *facet_no*)

class FIAT.reference_element.**TensorProductCell** (**cells*)
Bases: *FIAT.reference_element.Cell*

A cell that is the product of FIAT cells.

compute_reference_normal (*facet_dim*, *facet_i*)
Returns the unit normal in infinity norm to facet_i of subelement dimension facet_dim.

construct_subelement (*dimension*)
Constructs the reference element of a cell subentity specified by subelement dimension.

Parameters **dimension** – dimension in each "direction" (tuple)

contains_point (*point*, *epsilon=0*)
Checks if reference cell contains given point (with numerical tolerance).

get_dimension()
Returns the subelement dimension of the cell, a tuple of dimensions for each cell in the product.

get_entity_transform(*dim*, *entity_i*)
Returns a mapping of point coordinates from the *entity_i*-th subentity of dimension *dim* to the cell.

Parameters

- **dim** – subelement dimension (tuple)
- **entity_i** – entity number (integer)

volume()
Computes the volume in the appropriate dimensional measure.

class FIAT.reference_element.**UFCHexahedron**
Bases: *FIAT.reference_element.Cell*

This is the reference hexahedron with vertices (0.0, 0.0, 0.0), (0.0, 0.0, 1.0), (0.0, 1.0, 0.0), (0.0, 1.0, 1.0), (1.0, 0.0, 0.0), (1.0, 0.0, 1.0), (1.0, 1.0, 0.0) and (1.0, 1.0, 1.0).

compute_reference_normal(*facet_dim*, *facet_i*)
Returns the unit normal in infinity norm to facet_i.

construct_subelement(*dimension*)
Constructs the reference element of a cell subentity specified by subelement dimension.

Parameters **dimension** – subentity dimension (integer)

contains_point(*point*, *epsilon=0*)
Checks if reference cell contains given point (with numerical tolerance).

get_dimension()
Returns the subelement dimension of the cell. Same as the spatial dimension.

get_entity_transform(*dim*, *entity_i*)
Returns a mapping of point coordinates from the *entity_i*-th subentity of dimension *dim* to the cell.

Parameters

- **dim** – entity dimension (integer)
- **entity_i** – entity number (integer)

volume()
Computes the volume in the appropriate dimensional measure.

class FIAT.reference_element.**UFCInterval**
Bases: *FIAT.reference_element.UFCSimplex*

This is the reference interval with vertices (0.0,) and (1.0,).

class FIAT.reference_element.**UFCQuadrilateral**
Bases: *FIAT.reference_element.Cell*

This is the reference quadrilateral with vertices (0.0, 0.0), (0.0, 1.0), (1.0, 0.0) and (1.0, 1.0).

compute_reference_normal(*facet_dim*, *facet_i*)
Returns the unit normal in infinity norm to facet_i.

construct_subelement(*dimension*)
Constructs the reference element of a cell subentity specified by subelement dimension.

Parameters **dimension** – subentity dimension (integer)

contains_point(*point*, *epsilon=0*)
Checks if reference cell contains given point (with numerical tolerance).

get_dimension()
Returns the subelement dimension of the cell. Same as the spatial dimension.

get_entity_transform(*dim*, *entity_i*)
Returns a mapping of point coordinates from the *entity_i*-th subentity of dimension *dim* to the cell.

Parameters

- **dim** – entity dimension (integer)
- **entity_i** – entity number (integer)

volume()
Computes the volume in the appropriate dimensional measure.

class FIAT.reference_element.**UFCSimplex**(*shape*, *vertices*, *topology*)
Bases: *FIAT.reference_element.Simplex*

construct_subelement(*dimension*)
Constructs the reference element of a cell subentity specified by subelement dimension.

Parameters **dimension** – subentity dimension (integer)

contains_point(*point*, *epsilon=0*)
Checks if reference cell contains given point (with numerical tolerance).

get_facet_element()

class FIAT.reference_element.**UFCTetrahedron**
Bases: *FIAT.reference_element.UFCSimplex*

This is the reference tetrahedron with vertices (0,0,0), (1,0,0),(0,1,0), and (0,0,1).

compute_normal(*i*)
UFC consistent normals.

class FIAT.reference_element.**UFCTriangle**
Bases: *FIAT.reference_element.UFCSimplex*

This is the reference triangle with vertices (0.0,0.0), (1.0,0.0), and (0.0,1.0).

compute_normal(*i*)
UFC consistent normal

FIAT.reference_element.**compute_unflattening_map**(*topology_dict*)
This function returns unflattening map for the given tensor product topology dict.

FIAT.reference_element.**default_simplex**(*spatial_dim*)
Factory function that maps spatial dimension to an instance of the default reference simplex of that dimension.

FIAT.reference_element.**flatten_entities**(*topology_dict*)
This function flattens topology dict of TensorProductCell and entity_dofs dict of TensorProductElement

FIAT.reference_element.**flatten_reference_cube**(*ref_el*)
This function flattens a Tensor Product hypercube to the corresponding UFC hypercube

FIAT.reference_element.**is_hypercube**(*cell*)

FIAT.reference_element.**lattice_iter**(*start*, *finish*, *depth*)
Generator iterating over the depth-dimensional lattice of integers between start and (finish-1). This works on simplices in 1d, 2d, 3d, and beyond

FIAT.reference_element.**linalg_subspace_intersection**(*A*, *B*)
Computes the intersection of the subspaces spanned by the columns of 2-dimensional arrays A,B using the algorithm found in Golub and van Loan (3rd ed) p. 604. A should be in R^{m,p} and B should be in R^{m,q}. Returns an orthonormal basis for the intersection of the spaces, stored in the columns of the result.

FIAT.reference_element.**make_affine_mapping**(*xs*, *ys*)
Constructs (A,b) such that x –> A * x + b is the affine mapping from the simplex defined by xs to the simplex defined by ys.

FIAT.reference_element.**make_lattice**(*verts*, *n*, *interior=0*)
Constructs a lattice of points on the simplex defined by verts. For example, the 1:st order lattice will be just the vertices. The optional argument interior specifies how many points from the boundary to omit. For example, on a line with n = 2, and interior = 0, this function will return the vertices and midpoint, but with interior = 1, it will only return the midpoint.

FIAT.reference_element.**tuple_sum**(*tree*)
This function calculates the sum of elements in a tuple, it is needed to handle nested tuples in TensorProductCell. Example: tuple_sum(((1, 0), 1)) returns 2 If input argument is not the tuple, returns input.

FIAT.reference_element.**ufc_cell**(*cell*)
Handle incoming calls from FFC.

FIAT.reference_element.**ufc_simplex**(*spatial_dim*)
Factory function that maps spatial dimension to an instance of the UFC reference simplex of that dimension.

FIAT.reference_element.**volume**(*verts*)
Constructs the volume of the simplex spanned by verts

1.3.40 FIAT.regge module

Implementation of the generalized Regge finite elements.

class FIAT.regge.**Regge**(*cell*, *degree*)
Bases: *FIAT.finite_element.CiarletElement*

The generalized Regge elements for symmetric-matrix-valued functions. REG(r) in dimension n is the space of polynomial symmetric-matrix-valued functions of degree r or less with tangential-tangential continuity.

class FIAT.regge.**ReggeDual**(*cell*, *degree*)
Bases: *FIAT.dual_set.DualSet*

Degrees of freedom for generalized Regge finite elements.

1.3.41 FIAT.restricted module

class FIAT.restricted.**RestrictedElement**(*element*, *indices=None*, *restriction_domain=None*)
Bases: *FIAT.finite_element.CiarletElement*

Restrict given element to specified list of dofs.

FIAT.restricted.**sorted_by_key**(*mapping*)
Sort dict items by key, allowing different key types.

1.3.42 FIAT.serendipity module

class FIAT.serendipity.**Serendipity**(*ref_el*, *degree*)
Bases: *FIAT.finite_element.FiniteElement*

degree()

dmats()

entity_closure_dofs()
Return the map of topological entities to degrees of freedom on the closure of those entities for the finite element.

entity_dofs()
Return the map of topological entities to degrees of freedom for the finite element.

get_coeffs()

get_dual_set()
Return the dual for the finite element.

get_nodal_basis()

get_num_members(*arg*)

space_dimension()
Return the dimension of the finite element space.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

Parameters

- **order** – The maximum order of derivative.
- **points** – An iterable of points.
- **entity** – Optional (dimension, entity number) pair indicating which topological entity of the reference element to tabulate on. If None, default cell-wise tabulation is performed.

value_shape()

FIAT.serendipity.**e_lambda_0**(*i*, *dim*, *dx*, *dy*, *dz*, *x_mid*, *y_mid*, *z_mid*)

FIAT.serendipity.**f_lambda_0**(*i*, *dim*, *dx*, *dy*, *dz*, *x_mid*, *y_mid*, *z_mid*)

FIAT.serendipity.**i_lambda_0**(*i*, *dx*, *dy*, *dz*, *x_mid*, *y_mid*, *z_mid*)

FIAT.serendipity.**tr**(*n*)

FIAT.serendipity.**v_lambda_0**(*dim*, *dx*, *dy*, *dz*)

1.3.43 FIAT.tensor_product module

class FIAT.tensor_product.**FlattenedDimensions**(*element*)
Bases: *FIAT.finite_element.FiniteElement*

A wrapper class that flattens entity dimensions of a FIAT element defined on a TensorProductCell to one with quadrilateral/hexahedron entities. TensorProductCell has dimension defined as a tuple of factor element dimensions (i, j) in 2D and (i, j, k) in 3D. Flattened dimension is a sum of the tuple elements.

degree()
Return the degree of the (embedding) polynomial space.

dmats()
Return dmats: expansion coefficients for basis function derivatives.

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

is_nodal()
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

Subclasses may not necessarily be nodal, unless it is a CiarletElement.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

value_shape()
Return the value shape of the finite element functions.

class FIAT.tensor_product.**TensorProductElement**(*A*, *B*)
Bases: *FIAT.finite_element.FiniteElement*

Class implementing a finite element that is the tensor product of two existing finite elements.

degree()
Return the degree of the (embedding) polynomial space.

dmats()
Return dmats: expansion coefficients for basis function derivatives.

get_coeffs()
Return the expansion coefficients for the basis of the finite element.

get_nodal_basis()
Return the nodal basis, encoded as a PolynomialSet object, for the finite element.

get_num_members(*arg*)
Return number of members of the expansion set.

is_nodal()
True if primal and dual bases are orthogonal. If false, dual basis is not implemented or is undefined.

Subclasses may not necessarily be nodal, unless it is a CiarletElement.

tabulate(*order*, *points*, *entity=None*)
Return tabulated values of derivatives up to given order of basis functions at given points.

value_shape()
Return the value shape of the finite element functions.

1.3.44 Module contents

FInite element Automatic Tabulator – supports constructing and evaluating arbitrary order Lagrange and many other elements. Simplices in one, two, and three dimensions are supported.

1.4 Release notes

1.4.1 Changes in the next release

Summary of changes

- No changes yet.

Note: Developers should use this page to track and list changes during development. At the time of release, this page should be published (and renamed) to list the most important changes in the new release.

Detailed changes

Note: At the time of release, make a verbatim copy of the ChangeLog here (and remove this note).

1.4.2 Changes in version 2019.1.0

FIAT 2019.1.0 was released on 2019-04-17.

Summary of changes

- Added an implementation of the Bell finite element (K. Bell 1969 doi:10.1002/nme.1620010180), with extra basis functions for transformation theory from Kirby (2018) doi:10.5802/smai-jcm.33.

1.4.3 Changes in version 2018.1.0

FIAT 2018.1.0 was released on 2018-06-14.

Summary of changes

- Remove Python 2 support
- Generalize `Bubble` element to `CodimBubble` to create bubbles on entity of arbitrary codimension; add `FacetBubble`, keep `Bubble` (as bubble on cell)

1.4.4 Changes in version 2017.2.0

FIAT 2017.2.0 was released on 2017-12-05.

Summary of changes

- Add quadrilateral and hexahedron reference cells
- Add quadrilateral and hexahedron elements (with a wrapping class for TensorProductElement)

1.4.5 Changes in version 2017.1.0.post1

FIAT 2017.1.0.post1 was released on 2017-09-12.

Summary of changes

- Change PyPI package name to fenics-fiat.

1.4.6 Changes in version 2017.1.0

FIAT 2017.1.0 was released on 2017-05-09.

Summary of changes

- Extended the discontinuous trace element `HDivTrace` to support tensor product reference cells. Tabulating the trace defined on a tensor product cell relies on the argument `entity` to specify a facet of the cell. The backwards compatibility case `entity=None` does not support tensor product tabulation as a result. Tabulating the trace of triangles or tetrahedron remains unaffected and works as usual with or without an entity argument.

1.4.7 Changes in version 2016.2.0

FIAT 2016.2.0 was released on 2016-11-30.

Summary of changes

- More elegant edge-based degrees of freedom are used for generalized Regge finite elements. This is a internal change and is not visible to other parts of FEniCS.
- The name of the mapping for generalized Regge finite element is changed to "double covariant piola" from "pullback as metric". Geometrically, this mapping is just the pullback of covariant 2-tensor fields in terms of proxy matrix-fields. Because the mapping for 1-forms in FEniCS is currently named "covariant piola", this mapping for symmetric tensor product of 1-forms is thus called "double covariant piola". This change causes multiple internal changes downstream in UFL and FFC. But this change should not be visible to the end-user.
- Added support for the Hellan-Herrmann-Johnson element (symmetric matrix fields with normal-normal continuity in 2D).
- Add method `FiniteElement.is_nodal()` for checking element nodality
- Add `NodalEnrichedElement` which merges dual bases (nodes) of given elements and orthogonalizes basis for nodality
- Restructuring `finite_element.py` with the addition of a non-nodal class `FiniteElement` and a nodal class `CiarletElement`. `FiniteElement` is designed to be used to create elements where, in general, a nodal basis isn't well-defined. `CiarletElement` implements the usual nodal abstraction of a finite element.
- Removing `trace.py` and `trace_hdiv.py` with a new implementation of the trace element of an HDiv-conforming element: `HDivTrace`. It is also mathematically equivalent to the former `DiscontinuousLagrangeTrace`, that is, the DG field defined only on co-dimension 1 entities.
- All nodal finite elements inherit from `CiarletElement`, and the non-nodal `TensorProductElement`, `EnrichedElement` and `HDivTrace` inherit from `FiniteElement`.

Detailed changes

- Enable Travis CI on GitHub
- Add Firedrake quadrilateral cell

- Add tensor product cell
- Add facet -> cell coordinate transformation
- Add Bubble element
- Add discontinuous Taylor element
- Add broken element and H(div) trace element
- Add element restrictions onto mesh entities
- Add tensor product elements (for tensor product cells)
- Add H(div) and H(curl) element-modifiers for TPEs
- Add enriched element, i.e. sum of elements (e.g. for building Mini)
- Add multidimensional taylor elements
- Add Gauss Lobatto Legendre elements
- Finding non-vanishing DoFs on a facets
- Add tensor product quadrature rule
- Make regression tests working again after few years
- Prune modules having only __main__ code including transform_morley, transform_hermite (ff86250820e2b18f7a0df471c97afa87207e9a7d)
- Remove newdubiner module (b3b120d40748961fdd0727a4e6c62450198d9647, reference removed by cb65a84ac639977b7be04962cc1351481ca66124)
- Switch from homebrew factorial/gamma to math module (wraps C std lib)

1.4.8 Changes in version 2016.1.0

FIAT 2016.1.0 was released on 2016-06-23.

- Minor fixes

1.4.9 Changes in version 1.6.0

FIAT 1.6.0 was released on 2015-07-28.

- Support DG on facets through the element `Discontinuous Lagrange Trace`

[FIXME: These links don't belong here, should go under API reference somehow.]

- genindex
- modindex

Python Module Index

f

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X