February 11, 2017

Visit http://www.fenics.org/ for the latest version of this manual.
Send comments and suggestions to <package unspecified>-dev@fenics.org.

# Contents

# Chapter 1

# Introduction

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 [] 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 [], FFC [], and PETSc [], 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 2

# Installation

FIAT uses the standard Python `distutils` tools. From the top directory, one executes `python setup.py install`. This will put FIAT into the `site-packages` directory. Superuser permission (such as `su` or `sudo`) may be required to write to this directory. For more configuration options, one may type `python setup.py --help` or consult the online Python documentation at `http://docs.python.org/inst/inst.html`

FIAT requires the commonly used `Numeric` package.

# Chapter 3

# Using FIAT: A tutorial with Lagrange elements

## 3.1 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

```
from FIAT import Lagrange, quadrature, shapes
```

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.

## 3.2 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 $(-1, -1, -1), (1, -1, -1), (-1, 1, -1), (-1, -1, 1)$.

## 3.3 Instantiating elements

FIAT uses a lightweight object-oriented infrastructure to define finite elements. The `Lagrange` module contains a class `Lagrange` modeling the Lagrange finite element family. This class is a subclass of some `FiniteElement` class contained in another module (`polynomial` to be precise). So, having imported the `Lagrange` module, we can create the Lagrange element of degree `2` on triangles by

```
shape = shapes.TRIANGLE
degree = 2
U = Lagrange.Lagrange( shape , degree )
```

Here, `shapes.TRIANGLE` is an integer code indicating the two dimensional simplex. `shapes` also defines `LINE` and `TETRAHEDRON`. Most of the upper-level interface to FIAT is dimensionally abstracted over element shape.

The class `FiniteElement` supports three methods, modeled on the abstract definition of Ciarlet. These methods are `domain_shape()`, `function_space()`, and `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).

## 3.4 Quadrature rules

FIAT implements arbitrary-order collapsed quadrature, as discussed in Karniadakis and Sherwin [], for the simplex of dimension one, two, or three. The simplest way to get a quadrature rule is through the function `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

```
>>> Q = quadrature.make_quadrature( shape , 2 )
```

This uses two points in each direction on the reference square, then maps them to the reference triangle. We may get a `Numeric.array` of the quadrature weights with the method `Q.get_weights()` and a list of tuples storing the quadrature points with the method `Q.get_points()`.

## 3.5 Tabulation

FIAT provides functions for tabulating the element basis functions and their derivatives. To get the `FunctionSpace` object, we do

```
>>> Ufs = U.function_space()
```

To get the values of each basis function at each of the quadrature points, we use the `tabulate()` method

```
>>> 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]])
```

This returns a two-dimensional `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 `FunctionSpace` object also provides a method `tabulate_jet(i,xs)` that returns a list of Python dictionaries. The `i`th entry of the list is a dictionary storing the values of all `i`th order derivatives. Each dictionary maps a multiindex (a tuple of length `i`) to the table of the associated partial derivatives of the basis functions at those points. For example,

```
>>> Ufs_jet = Ufs.tabulate_jet( 1 , Q.get_points() )
```

tabulates the zeroth and first partial derivatives of the function space at the quadrature points. Then,

```
>>> 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]])}
```

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

```
>>> 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]])
```

# Chapter 4

# 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.

## 4.1 shapes.py

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

## 4.2 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.

## 4.3 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 []. The module `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 `polynomial.py`

## 4.4 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 `quadrature.py` may be accessed through the single function `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 `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

- `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.
- `get_weights()`, which returns a `Numeric.array` of quadrature weights.
- `integrate(f)`, which takes a callable object `f` and returns the (approximate) integral over the domain
- Also, the `__call__` method is overloaded so that a quadrature rule may be applied to a callable object. This is syntactic sugar on top of the `integrate` method.

## 4.5 polynomial.py

The `polynomial` module provides the bulk of the classes needed to represent polynomial bases and finite element spaces. The class `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 `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.

### 4.5.1 `PolynomialBase`

### 4.5.2 `PolynomialSet`

The `PolynomialSet` function is a factory function interface into the hierarchy

# Chapter 5

# Wish list and open problems

While FIAT is highly functional as a tool for tabulating basis functions at quadrature points, there are a lot of interesting things to do. In case anybody wants to help out, I have chosen to describe some of these issues here.

## 5.1 Stable/fast VDM inversion

## 5.2 Symmetric quadrature rules

## 5.3 Declarative top-level language

## 5.4 Integration with SMART-type tools