# 1. FFC: The FEniCS Form Compiler

By Anders Logg, Kristian B. Ølgaard, Marie Rognes and Garth N. Wells

One of the key features of FEniCS is automated code generation for the general and efficient solution of finite element variational problems. This automated code generation relies on a form compiler for offline or just-in-time compilation of code for individual forms. Two different form compilers are available as part of FEniCS. This chapter describes the form compiler FFC. The other form compiler, SyFi, is described in Chapter [alnes-3].

## 1.1 Compilation of variational forms

In simple terms, the solution of finite element variational problems is based on two ingredients: the assembly of linear or nonlinear systems of equations and the solution of those equations. As a result, many finite element codes are similar in their design and implementation. In particular, a central part of most finite element codes is the assembly of sparse matrices from finite element bilinear forms. In Chapter [logg-3], we saw that one may formulate a general algorithm for assembly of sparse tensors from variational forms. However, this algorithm relies on the computation of the element tensor $A_T$ as well as the local-to-global mapping $\iota_T$. Both $A_T$ and $\iota_T$ differ greatly between different finite elements and different variational forms. Special-purpose code is therefore needed. As a consequence, the code for computing $A_T$ and $\iota_T$ must normally be developed by hand for any given application. This is both a tedious and an error-prone task.

The issue of having to develop code for $A_T$ and $\iota_T$ by hand can be resolved by a form compiler. A form compiler generates code for computing $A_T$ and $\iota_T$. This code may then be called by a general purpose routine for assembly of finite element matrices and vectors. In addition to reduced development time, performance may be improved by using code generation since the form compiler can generate efficient code for the computation of $A_T$ by using optimization techniques that are not readily applicable if the code is developed by hand. In Chapters [kirby-8], [kirby-4] and [oelgaard-2], two different approaches to the optimized computation of the element tensor $A_T$ are presented.

From an input describing a finite element variational problem in mathematical notation, the form compiler FFC generates code for the efficient computation of $A_T$ and $\iota_T$, as well as code for computing related quantities. More specifically, FFC takes as input a variational form specified in the UFL form language (described in Chapter [alnes-1]) and generates as output C++ code that conforms to the UFC interface (described in Chapter [alnes-2]). This process is illustrated schematically in Figure 1.1.

Figure 1.1: The form compiler FFC generates C++ code in UFC format from a given finite element variational form in UFL format.

## 1.2 Compiler interfaces

FFC provides three interfaces: a Python interface, a command-line interface, and a just-in-time (JIT) compilation interface. The first two are presented here, while the third is discussed in Section 1.7.

### 1.2.1 Python interface

The Python interface to FFC takes the form of a standard Python module. There are two main entry point functions to the functionality of FFC: `compile_element` and `compile_form`, to compile elements and forms, respectively.

The `compile_element` function expects a finite element or a list of finite elements as its first argument. In addition, a set of optional arguments can be provided:

```
compile_element(elements,
                prefix="Element",
                parameters=default_parameters())
```

The above function generates UFC conforming code for the specified finite element spaces and their corresponding degree-of-freedom maps. The `prefix` argument can be used to control the prefix of the file containing the generated code; the default is "Element". The suffix ".h" will be added automatically. The second optional argument `parameters` should be a Python dictionary with code generation parameters and is described further below.

Similarly, the `compile_form` function expects a form or a list of forms as input along with a set of optional arguments:

```
compile_form(forms,
             object_names={},
             prefix="Form",
             parameters=default_parameters())
```

The above function generates code for each of the given forms and each of the finite elements involved in the definition of the forms and their corresponding degree-of-freedom maps. The arguments `prefix` and `parameters` play the same role as for `compile_element`. The `object_names` dictionary is an optional argument that specifies the names of the coefficients that were used to define the form. This is used by the command-line interface of FFC to allow a user to refer to any coefficients in a form by their names (`f`, `g`, etc).

As an illustration, we list in Figure 1.2 the specification and compilation of a piecewise continuous quartic finite element (Lagrange element of degree 4) in three dimensions using the FFC Python interface. The two first lines import the UFL and FFC modules respectively. The third line specifies the finite element in the UFL syntax. The last line calls the FFC

```
from ufl import *
from ffc import *
element = FiniteElement("Lagrange", "tetrahedron", 4)
compile_element(element, prefix="P4tet")
```

Figure 1.2: Compiling an element using the FFC Python interface.

```
from ufl import *
from ffc import *

V = FiniteElement("Lagrange", "triangle", 1)
v = TestFunction(V)
u = TrialFunction(V)
f = Coefficient(V)

a = inner(grad(v), grad(u))*dx
L = v*f*dx

compile_form([a, L], prefix="Poisson")
```

Figure 1.3: Compiling a form using the FFC Python interface.

`compile_element` function. The generated code is written to the file "P4tet.h", as specified by the argument `prefix`.

In Figure 1.3, we illustrate the specification and compilation of a variational formulation of Poisson's equation in two dimensions using the Python interface. The last line calls the `compile_form` function. When run, code will be generated for the forms `a` and `L`, and the finite element and degree-of-freedom map associated with the element `V`, and then written to the file "Poisson.h".

### *1.2.2 Command-line interface*

The command-line interface takes a UFL form file or a list of form files as input.

```
>> ffc FormFile.ufl
```

The form file should contain the specification of elements and/or forms in the UFL syntax, and is very similar to the FFC Python interface, as illustrated by the following specification of the same variational problem as in Figure 1.3:

```
V = FiniteElement("Lagrange", "triangle", 1)
v = TestFunction(V)
u = TrialFunction(V)
f = Coefficient(V)

a = inner(grad(v), grad(u))*dx
L = v*f*dx
```

The contents of each form file are wrapped in a Python script and then executed. Such a script is simply a copy of the form file that includes the required imports of FFC and UFL and calls `compile_element` or `compile_form` from the FFC Python interface. The variable

names `a`, `L`, `M`, and `element` are recognized as a bilinear form, a linear form, a functional, and a finite element, respectively.

## 1.3 *Parameters affecting code generation*

The code generated by FFC can be controlled by a number of optional parameters. Through the Python interface, parameters are set in the dictionary `parameters` which is passed to the compile functions. The default values for these may be obtained by calling the function `default_parameters` from the Python interface. Most parameters can also be set on the command-line. All available command-line parameters are listed on the FFC manual page (`man ffc`). We here list some of the parameters which affect the code generation. We list the dictionary key associated with each parameter, and the command-line version in parentheses, if available.

**format (-l)** This parameter controls the output format for the generated code. The default value is "ufc", which indicates that the code is generated according to the UFC specification. Alternatively, the value "dolfin" may be used to generate code according to the UFC format with a small set of additional DOLFIN-specific wrappers.

**representation (-r)** This parameter controls the representation used for the generated element tensor code. There are three possibilities: "auto" (the default), "tensor" and "quadrature". See Section 1.5, and Chapters [kirby-8] and [oelgaard-2] for more details on the different representations. In the case "auto", either the tensor or quadrature representation is selected by FFC. FFC attempts to select the representation which will lead to the most efficient code for the given form.

**split (-f)** This option controls the output of the generated code into a single or multiple files. The default is `False`, in which case the generated code is written to a single file. If set to `True`, separate header (.h) and implementation (`.cpp`) files are generated.

**optimize (-O)** This option controls code optimization features, and the default is `False`. If set to `True`, the code generated for the element tensor is optimized for run-time performance. The optimization strategy used depends on the chosen representation. In general, this will increase the time required for FFC to generate code, but should reduce the run-time for the generated code.

**log_level** This option controls the verbosity level of the compiler. The possible values are, in order of decreasing verbosity: `DEBUG`, `INFO` (default), `ERROR` and `CRITICAL`.

## 1.4 *Compiler design*

FFC breaks compilation into several stages. The output generated at each stage serves as input for the following stage, as illustrated in Figure 1.4. We describe each of these stages below. The individual compiler stages may be accessed through the `ffc.compiler` module. We consider here only the stages involved when compiling forms. For compilation of elements a similar (but simpler) set of stages is used.

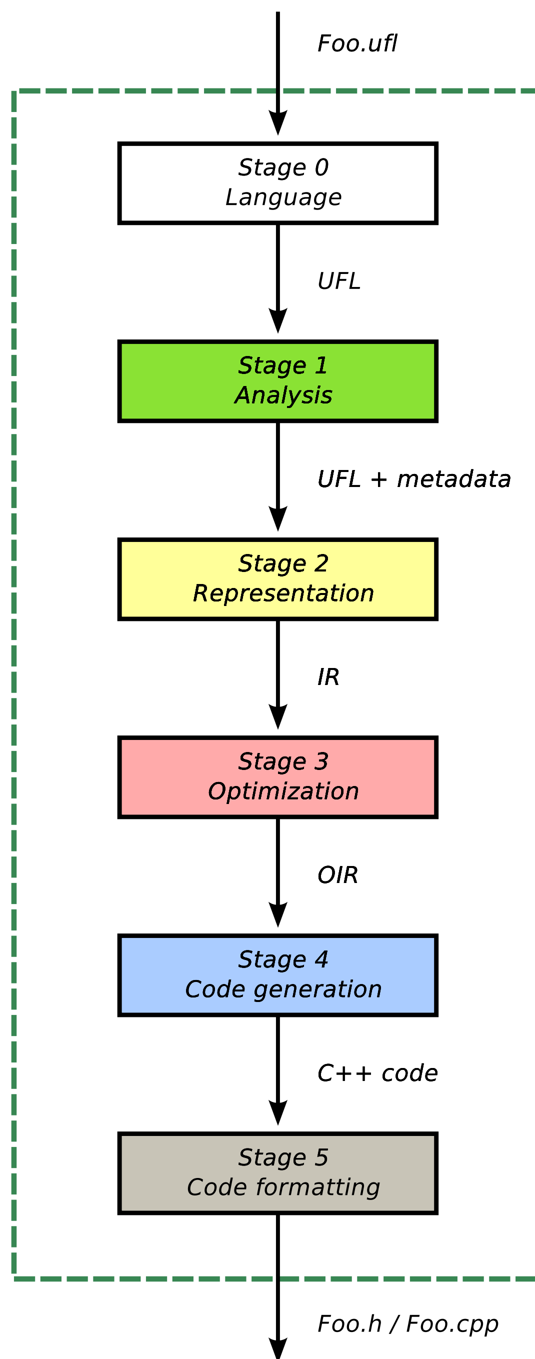


Figure 1.4: Form compilation broken into six sequential stages: Language, Analysis, Representation, Optimization, Code generation and Code Formatting. Each stage generates output based on input from the previous stage.

**Compiler stage 0: Language (parsing)** In this stage, the user-specified form is interpreted and stored as a UFL abstract syntax tree (AST). The actual parsing is handled by Python and the transformation to a UFL form object is implemented by operator overloading in UFL.

Input: Python code or `.ufl` file
Output: UFL form

**Compiler stage 1: Analysis** This stage preprocesses the UFL form and extracts form meta data (`FormData`), such as which elements were used to define the form, the number of coefficients and the cell type (intervals, triangles, or tetrahedra). This stage also involves selecting a suitable representation for the form if that has not been specified by the user (see Section 1.5 below).

Input: UFL form
Output: preprocessed UFL form and form meta data

**Compiler stage 2: Code representation** This stage examines the input and generates all data needed for code generation. This includes generation of finite element basis functions, extraction of data for mapping of degrees of freedom, and possible precomputation of integrals. Most of the complexity of compilation is handled in this stage.

The intermediate representation is stored as a dictionary, mapping names of UFC functions to the data needed for generation of the corresponding code. In simple cases, like `ufc::form::rank`, this data may be a simple number like '2'. In other cases, like `ufc::cell_tensor::tabulate_tensor`, the data may be a complex data structure that depends on the choice of form representation.

Input: preprocessed UFL form and form meta data
Output: intermediate representation (IR)

**Compiler stage 3: Optimization** This stage examines the intermediate representation and performs optimizations. Such optimization may involve FErari based optimizations as discussed in Chapter [kirby-3] or symbolic optimization as discussed in Chapter [oelgaard-2]. Data stored in the intermediate representation dictionary is then replaced by new data that encode an optimized version of the function in question.

Input: intermediate representation (IR)
Output: optimized intermediate representation (OIR)

**Compiler stage 4: Code generation** This stage examines the optimized intermediate representation and generates the actual C++ code for the body of each UFC function. The code is stored as a dictionary, mapping names of UFC functions to strings containing the C++ code. As an example, the data generated for `ufc::form::rank` may be the string "`return 2;`".

We emphasize the importance of separating stages 2, 3 and 4. This allows stages 2 and 3 to focus on algorithmic aspects related to finite elements and variational forms, while stage 4 is concerned only with generating C++ code from a set of instructions prepared in earlier compilation stages.

Input: optimized intermediate representation (OIR)
Output: C++ code

**Compiler stage 5: Code formatting** This stage examines the generated C++ code and formats it according to the UFC format, generating as output one or more `.h`/`.cpp` files conforming to the UFC specification. This is where the actual writing of C++ code takes place. This stage relies on templates for UFC code available as part of the UFC module `ufc_utils`.

Input: C++ code
Output: C++ code files

## 1.5 Form representation

Two different approaches to code generation are implemented in FFC. One based on traditional quadrature and another on a special tensor representation. We address these representations here briefly and refer readers to Chapter [oelgaard-2] for details of the quadrature representation and to Chapter [kirby-8] for details of the tensor representation.

### 1.5.1 Quadrature representation

The quadrature representation in FFC is selected using the option `-r quadrature`. As the name suggests, the method to evaluate the local element tensor $A_T$ involves a loop over integration points and adding the contribution from each point to $A_T$. To generate code for quadrature, FFC calls FIAT during code generation to tabulate finite element basis functions and their derivatives at a suitable set of quadrature points on the reference element. It then goes on to generate code for computing a weighted average of the integrand defined by the UFL AST at these quadrature points.

### 1.5.2 Tensor representation

When FFC is called with the `-r tensor` option, it attempts to extract a monomial representation of the given UFL form, that is, rewrite the given form as a sum of products of basis functions and their derivatives. Such a representation is not always possible, in particular if the form is expressed using operators other than addition, multiplication and linear differential operators. If unsuccessful, FFC falls back to using quadrature representation.

If the transformation is successful, FFC computes the tensor representation $A_T = A_0 : G_T$, as described in Chapter [kirby-8], by calling FIAT to compute the reference tensor $A_0$. Code is then generated for computing the element tensor. Each entry of the element tensor is obtained by computing an inner product between the geometry tensor $G_K$ and a particular slice of the reference tensor. It should be noted that the entries of the reference tensor are known during code generation, so these numbers enter directly into the generated code.

### 1.5.3 Automatic selection of representation

If the user does not specify which representation to use, FFC will try to automatically select the "best" representation which in this case is defined as the representation with the best run-time performance. As described in Chapter [oelgaard-2], the run-time performance depends on many factors and it might not be possible to give a precise *a priori* answer as to which representation will be best for a particular variational form. In general, the more complex the form (in terms of the number of derivatives and the number of function products),

the more likely quadrature is to be preferable. See [Ølgaard and Wells(2010)] for a detailed discussion on form complexity and comparisons between tensor and quadrature representations. In [Ølgaard and Wells(2010)] it was suggested that the selection should be based on an estimate of the operation count to compute the element tensor $A_T$. However, it turns out to be difficult to obtain an estimate which is accurate enough for this purpose. Therefore, the following crude strategy to select the representation has been implemented. First, FFC will try to generate the tensor representation and in case it fails, quadrature representation will be selected. If the tensor representation is generated successfully, each monomial is investigated and if the number of coefficients plus derivatives is greater than three, then quadrature representation is selected for the given variational form.

## 1.6 Optimization

The optimization stage of the FFC compiler is concerned with the run-time efficiency of the generated code for computing the local finite element tensor. Optimization is available for both tensor and quadrature representations, and they both operate on the intermediate representation generated in stage two. The output in both cases is a new set of instructions (an optimized intermediate representation) for the code generation stage. The goal of the optimizations is to reduce the number of operations needed to compute the element tensor $A_T$.

Due to the dissimilar nature of the quadrature and tensor representations, the optimizations applied to the two representations are different. To optimize the tensor representation, FFC relies on the Python module FErari (see Chapter [kirby-3]) to perform the optimizations. Optimization strategies for the quadrature representation are implemented as part of the FFC module itself and are described in Chapter [oelgaard-2]. For both representations, the optimizations come at the expense of an increased generation time for FFC and for very complicated variational forms, hardware limitations can make the compilation impossible.

Optimizations are switched on by using the command-line option `-O` or through the Python interface by setting the parameter `optimize` equal to `True`. For the quadrature representation, there exist four optimization strategy options, and these can be selected through the command-line interface by giving the additional options `-f eliminate_zeros`, `-f simplify_expressions`, `-f precompute_ip_const` and `-f precompute_basis_const`, and through the Python interface by setting these parameters equal to `True` in the options dictionary. The option `-f eliminate_zeros` can be combined with any of the other three options. Only one of the optimizations `-f simplify_expressions`, `-f precompute_ip_const` and `-f precompute_basis_const` can be switched on at one time, and if two are given `-f simplify_expressions` takes precedence over `-f precompute_ip_const` which in turn takes precedence over `-f precompute_basis_const`. If no specific optimization options are given, that is, only `-O` is specified, the default is to switch on the optimizations `-f eliminate_zeros` and `-f simplify_expressions`.

## 1.7 Just-in-time compilation

FFC can also be used as a just-in-time (JIT) compiler. In a scripted environment, UFL objects can be passed to FFC, and FFC will return Python modules. Calling the JIT compiler involves calling the `jit` function available as part of the FFC Python module:

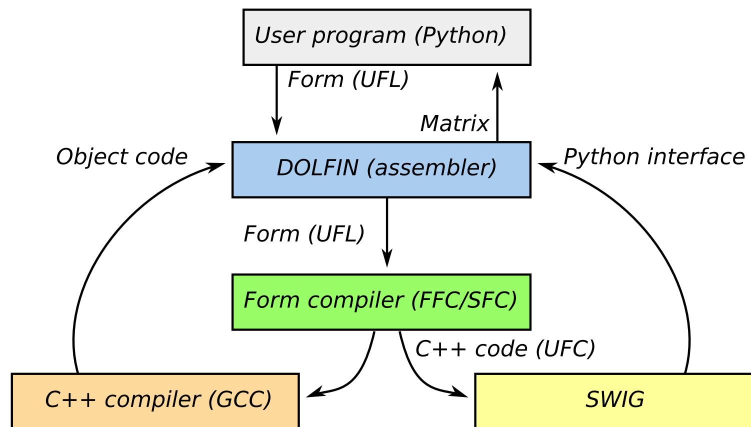


Figure 1.5: JIT compilation of variational forms coordinated by DOLFIN, and relying on UFL, FFC, UFL, SWIG, and GCC.

```
(compiled_object, compiled_module, form_data) \
    = jit(ufl_object, parameters=None, common_cell=None)
```

where `ufl_object` is either a UFL form or finite element object, `parameters` is an optional dictionary containing form compiler settings and `common_cell` is an optional argument that may be used to specify the cell (interval, triangle or tetrahedron) for a form where the cell is not specified as part of the form[1] The `jit` function returns a tuple, where `compiled_form` is a Python module which wraps either `ufc::form` or `ufc::finite_element` (depending on the type of UFL object passed to the form compiler), `compiled_module` is a Python module the wraps the all the generated UFC code (this includes finite elements, degree of freedom maps, etc) and `form_data` is a UFL object that contains details of the form which has been computed during the code generation, such as the number of coefficient functions in a form.

When the JIT compiler is called, internally FFC generates UFC code for the given form or finite element, compiles the generated code using a C++ compiler, and then wraps the result as a Python module using SWIG and Instant (see Chapter [wilbers]). The returned objects are ready to be used from Python. The generated and wrapped code is cached by the JIT compiler, so if the JIT compiler is called twice for the same form or finite element, the cached version is used. The cache directory is created by Instant, and can be cleaned by running the command `instant-clean`. The interactions of various components in the JIT process are illustrated in Figure 1.5.

The Python interface of DOLFIN makes extensive use of JIT. It makes it possible to combine the performance features of generated C++ code with the ease of a scripted interface.

## 1.8 *Historical notes*

FFC was first released in 2004 as a research code capable of generating C++ code for simple variational forms, see [Kirby and Logg(2006), Kirby and Logg(2007)]. Ever since its first release, FFC has relied on FIAT as a backend for computing finite element basis functions. In

[1] This is used by DOLFIN to allow simple specification of expressions like `f = Expression("sin(x[0])")`.

```
class Poisson : public PDE
{
public:

  Poisson(Function& source) : PDE(3)
  {
    add(f, source);
  }

  real lhs(const ShapeFunction& u, const ShapeFunction& v)
  {
    return (grad(u), grad(v))*dx;
  }

  real rhs(const ShapeFunction& v)
  {
    return f*v*dx;
  }

private:

  ElementFunction f;

};
```

Figure 1.6: Implementation of Poisson's equation in DOLFIN 0.5.2 using C++ operator overloading. Note the use of `operator,` for inner product.

2005, the DOLFIN assembler was redesigned to rely on code generated by FFC at compile-time for evaluation of the element tensor. Earlier versions of DOLFIN were based on a run-time system for evaluation of variational forms in C++ via operator overloading, see Tables 1.6–1.8.

Important milestones in the development of FFC include support for mixed elements (2005), FErari-based optimizations (2006), JIT compilation (2007), discontinuous Galerkin methods (2007), see [Ølgaard et al.(2008)Ølgaard, Logg, and Wells], $H(\mathrm{div})/H(\mathrm{curl})$ elements (2007–2008), see [Rognes et al.(2008)Rognes, Kirby, and Logg], code generation based on quadrature (2007), see [Ølgaard and Wells(2010)], the introduction of the UFC interface (2007), and optimized quadrature code generation (2008). In 2009, the FFC form language was replaced by the new UFL form language.

```
name = "Poisson"
element = FiniteElement("Lagrange", "triangle", 1)

v = BasisFunction(element)
u = BasisFunction(element)
f = Function(element)

a = v.dx(i)*u.dx(i)*dx
L = v*f*dx
```

Figure 1.7: Implementation of Poisson's equation in DOLFIN 0.5.3 using the new FFC form language. Note that the `grad` operator was missing in FFC at this time. It was also at this time that the test and trial functions changed places...

```
element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)

a = inner(grad(v), grad(u))*dx
L = v*f*dx
```

Figure 1.8: Implementation of Poisson's equation in DOLFIN 0.9.7 using the new UFL form language which was introduced in FFC 0.6.2.

# References

[Kirby and Logg(2006)] R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006. ISSN 0098-3500.

[Kirby and Logg(2007)] R. C. Kirby and A. Logg. Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software*, 33(3), 2007. ISSN 0098-3500.

[Ølgaard and Wells(2010)] K. B. Ølgaard and G. N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37(1):8:1–8:23, 2010. URL `http://dx.doi.org/10.1145/1644001.1644009`.

[Ølgaard et al.(2008)Ølgaard, Logg, and Wells] K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous galerkin methods. *SIAM Journal on Scientific Computing*, 31(2):849–864, 2008. doi: 10.1137/070710032. URL `http://dx.doi.org/10.1137/070710032`.

[Rognes et al.(2008)Rognes, Kirby, and Logg] M. Rognes, R. C. Kirby, and A. Logg. Efficient assembly of $h(\mathrm{div})$ and $h(\mathrm{curl})$ conforming finite elements. *SIAM J. Sci. Comput.*, 2008. submitted by Marie 2008-10-24, resubmitted by Marie 2009-05-22.