# DUNE PDELab Tutorial 09

## Using Code Generation to Create Local Operators

PDELab Team

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 205, D-69120 Heidelberg

August 31, 2021

# Contents

# Introduction

## Introduction

- This tutorial gives a short introduction to using `dune-codegen`[1]
- `dune-codegen` uses code generation to solve PDEs. This is done by describing the PDE in a domain-specific language (DSL) and generating C++ code for the local integration kernels
- We use UFL[2] as DSL
- The generated code can be used in `dune-pdelab`
- This makes it easier to use PDELab for your application

[1] https://gitlab.dune-project.org/extensions/dune-codegen

[2] https://bitbucket.org/fenics-project/ufl

# Introduction

We will look at a quick example to get some idea how this looks like.

# Hello World: Poisson Problem

- Strong formulation:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= g && \text{on } \partial\Omega, \end{aligned}$$

- Discrete weak formulation: Find $u_h \in U_h$ with

$$r_h^{Poisson}(u_h, v_h) = \int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f \, v_h \, dx = 0 \qquad \forall v_h \in V_h$$

- Parameter functions:

$$\begin{aligned} f(x) &= -2d \\ g(x) &= \|x\|_2^2 \end{aligned}$$

# UFL file for Poisson Problem

Discrete weak formulation: Find $u_h \in U_h$ with

$$r_h^{Poisson}(u_h, v_h) = \int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f \, v_h \, dx = 0 \qquad \forall v_h \in V_h$$

```
cell = triangle
V = FiniteElement("CG", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)

dim = 2
x = SpatialCoordinate(cell)
g = x[0]*x[0]+x[1]*x[1]
f = -2*dim

r = (inner(grad(u), grad(v)) - f*v)*dx

# dune-codegen specific
exact_solution = g
interpolate_expression = g
is_dirichlet = 1
```

# Goals of this Talk

**Goals of this talk**

- Explain how to write down PDEs in UFL
- Show how `dune-codegen` modifies/extends UFL
- Show how it is integrated into the build system

**Before this we will**

- Give a short overview over the workflow
- Talk about differences to other code generation approaches

# Resources

This tutorial is partially based on

- “Code Generation for High Performance PDE Solvers on Modern Architectures” by Dominic Kempf
- “Unified Form Language: A domain-specific language for weak formulations of partial differential equations” M. S. Alnaes, A. Logg, K. B. Ølgaard, M. E. Rognes and G. N. Wells
- UFL documentation `https://fenics.readthedocs.io/projects/ufl/en/latest/index.html`

# Contents

# The Big Picture

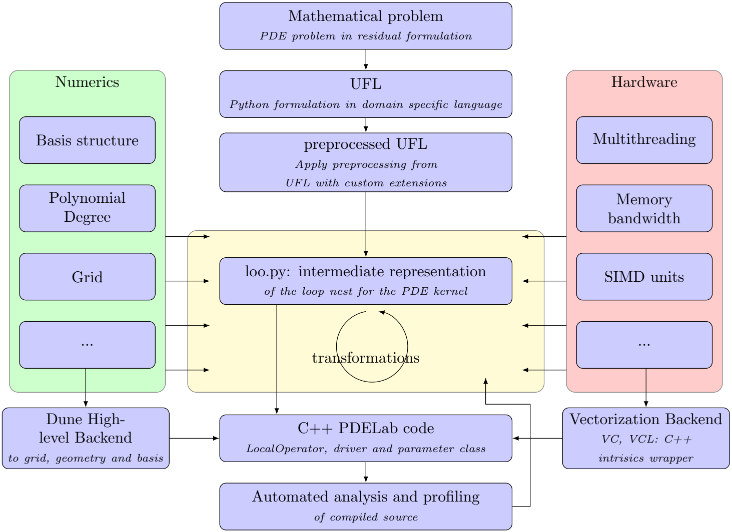
- Research goals of `dune-codegen`:
  - Generate high performance code
  - Do performance optimizations on intermediate representation (in our case Loopy[3]), e.g. SIMD vectorization
- Difference to other code generation approaches:
  - Do not generate everything. Instead generate code against a well defined interface of a C++ framework.
  - The workflow is CMake and C++ driven and not controlled through Python.
  - Main focus on generating high performance code

[3]https://mathema.tician.de/software/loopy/

# Typical Workflow

- Have a dune module that depends on `dune-codegen`
- Write a UFL file describing the PDE
- Add a target in CMake (see build system part)
- Go to the build directory and type make. CMake will generate the C++ code and compile the executable.

# Form Compiler Approach



# Contents

# UFL: Poisson

Discrete weak formulation: Find $u_h \in U_h$ with

$$r_h^{Poisson}(u_h, v_h) = \int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f \, v_h \, dx = 0 \qquad \forall v_h \in V_h$$

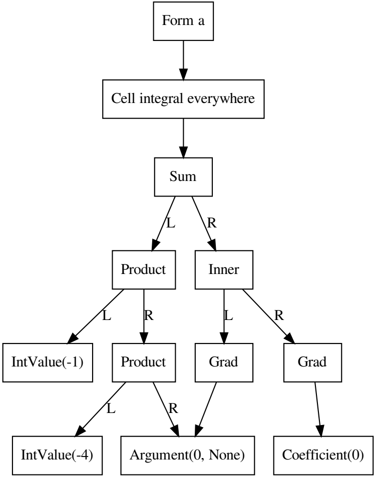
UFL file:

```
cell = triangle
V = FiniteElement("CG", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)

dim = 2
x = SpatialCoordinate(cell)
g = x[0]*x[0]+x[1]*x[1]
f = -2*dim

r = (inner(grad(u), grad(v)) - f*v)*dx

# dune-codegen specific
exact_solution = g
interpolate_expression = g
is_dirichlet = 1
```

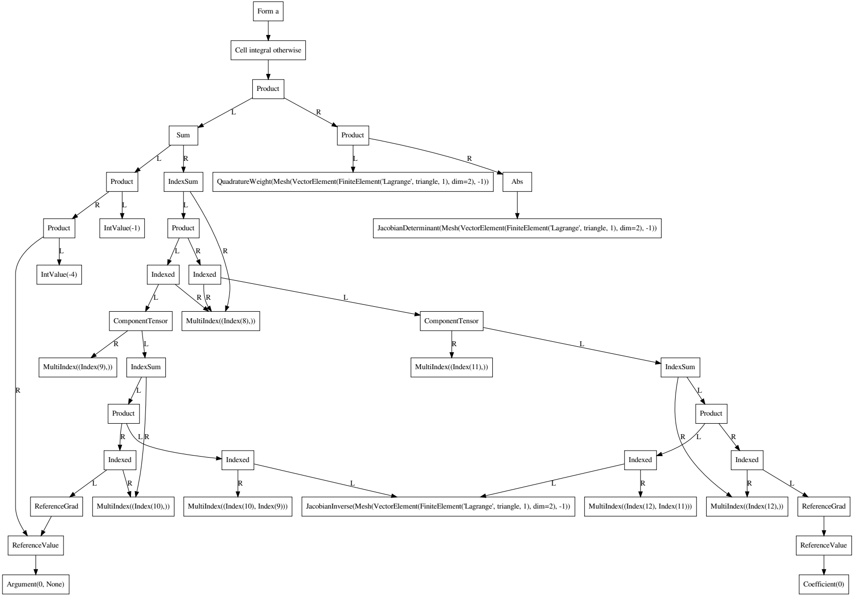
# UFL: About

- Domain specific language for describing weak formulations of PDE discretizations
- Notation stays close to methematical formulation
- Embedded in Python
- Only desribes cell/facet local computations. There is no notion of a grid or a description of an element loop
- The form is described by an abstract syntax trees (AST)
- UFL can apply transformation on the AST e.g.:
  - Calculation of the Jacobian of the residual
  - Geometry lowering

# UFL: AST


Form a
Cell integral everywhere
Sum
L
R
Product
Inner
L
R
L
R
IntValue(-1)
Product
Grad
Grad
L
R
IntValue(-4)
Argument(0, None)
Coefficient(0)

# UFL: AST - Preprocessed



# UFL: `FiniteElement`

```
V = FiniteElement(family, cell, degree)
```

- family: String representing a finite element family
  - `'CG'` Continuous Lagrange finite element
  - `'DG'` Discontinuous Galerkin Lagrange finite element
- Possible Cells:

| Dimension | Simplex Cell | Cube Cell |
| --- | --- | --- |
| 0 | `vertex` | `vertex` |
| 1 | `interval` | `interval` |
| 2 | `triangle` | `quadrilateral` |
| 3 | `tetrahedron` | `hexahedron` |

- Instead you can also write `Cell('triangle')`
- degree: Polynomial degree

# UFL: Form

- UFL expresses forms

$$\begin{aligned} a : W_1 \times \cdots \times W_m \times V_1 \times \cdots \times V_n &\to \mathbb{R} \\ (w_1, \ldots, w_m, v_1, \ldots, v_n) &\mapsto a(w_1, \ldots, w_m; v_1, \ldots, v_n) \end{aligned}$$

- Linear in the arguments $v_1, \ldots, v_n$
- Possibly nonlinear in coefficient functions $w_1, \ldots, w_m$
- PDELab uses a residual formulation: Find $u \in U$ with

$$r(u, v) = 0 \qquad \forall v \in V$$

- $r$ is linear in $v$ but might be nonlinear in $u$

# UFL: `TrialFunction` and `TestFunction`

```
u = TrialFunction(V)
v = TestFunction(V)
```

- `TrialFunction` and `TestFunction` represent finite element functions.
- Take `FiniteElement` as argument
- Note: In contrast to other UFL based frameworks our forms need not be linear in the `TrialFunction`

# UFL: Defining Expressions

```
dim = 2
x = SpatialCoordinate(cell)
g = x[0]*x[0]+x[1]*x[1]
f = -2*dim

r = (inner(grad(u), grad(v)) - f*v)*dx
```

- `SpatialCoordinate`: Global coordinate
- `grad(u)`: Gradient of $u$
- `inner(A,B)`: Inner product

$$A : B = \sum_{i_0} \cdots \sum_{i_{n-1}} A_{i_0 \cdots i_{n-1}} B_{i_0 \cdots i_{n-1}}$$

- `dx`: Multiplication with `dx` indicates a volume integral

# UFL: `dune-codegen` Specific

```
# dune-codegen specific
exact_solution = g
interpolate_expression = g
is_dirichlet = 1
```

- Main goal of `dune-codegen` is to generate the local integration kernel
- For testing and solving simple problem an automated driver can be generated. For the correct handling of the boundary condition we need to add some information to the UFL file
- `exact_solution`: Can be set for writing tests if solution is known
- `is_dirichlet`: Expression that may depend on $x$ and returns 1 if this is a dirichlet boundary condition. This is used only for driver generation.
- `interpolate_expression`: This is used as Dirichlet boundary value
- `dune-codegen` imports everything from the UFL module via `from UFL import *`

# Contents

# UFL: Towards More Complex Forms

In the following we show some important features of UFL. This is by no means complete, see the official documentation for further details `https://fenics.readthedocs.io/projects/ufl/en/latest/index.html`.

# UFL: Math Expressions

- Math functions, e.g. `*`, `/`, `+`, `-`, `abs`, `exp`, `ln`, `sqrt`, trigonometric functions, ...
- Comparison operator: `eq`, `ne`, `le`, `ge`, `lt` and `gt`
- Conditionals:

$$\texttt{conditional(cond, A, B)} = \begin{cases} \texttt{A} & \texttt{cond is True} \\ \texttt{B} & \texttt{cond is False} \end{cases}$$

- Vector-, matrix- and tensor-valued objects can be created through `as_vector`, `as_matrix` and `as_tensor`

```
a = as_matrix([[1.0, 2.0],[3.0, 4.0]])
```

- See the official documentation for tensor algebra operations

# UFL: Geometric Quantities

- `SpatialCoordinate(cell)`: Global coordinate
- `FacetNormal(cell)`: Unit outer normal vector
- `CellVolume(cell)` and `FacetArea(cell)`

# UFL: Integral Measures

- Multiplication with a measure describes an integral object over a local cell or facet
- `dx`: Integral over cell
- `ds`: Integral over boundary facet
- `dS`: Integral over interior facet
- Measures can be restricted to a subdomain. See the example about mixed Dirichlet and Neumann conditions on the next slides

# Example: Mixed Boundary Conditions

Strong formulation:

$$
\begin{aligned}
-\Delta u + q(u) &= f && \text{in } \Omega, \\
u &= g && \text{on } \Gamma_D \subset \partial\Omega, \\
-\nabla u \cdot \nu &= j && \text{on } \Gamma_N \subset \partial\Omega
\end{aligned}
$$

Weak discrete formulation: Find $u_h \in U_h$ with

$$
\begin{aligned}
r_h^{NLP}(u_h, v_h) = \int_\Omega \nabla u_h \cdot \nabla v_h \, dx + \int_\Omega q(u)\, v \, dx \\
- \int_\Omega f \, v_h \, dx + \int_{\Gamma_N} j \, v \, ds = 0 \qquad \forall v_h \in V_h
\end{aligned}
$$

Parameter functions:

$$
\begin{aligned}
f(x) &= -2d \\
g(x) &= \|x\|_2^2 \\
j(x) &= -\begin{pmatrix} 2x_0 \\ 2x_1 \end{pmatrix} \cdot \nu
\end{aligned}
$$

# Example: Mixed Boundary Conditions

```
cell = triangle
degree = 1
V = FiniteElement("CG", cell, degree)
u = TrialFunction(V)
v = TestFunction(V)

x = SpatialCoordinate(cell)
dim = 2
eta = 2
g = x[0]*x[0]+x[1]*x[1]
f = -2*dim+eta*g*g

def q(u):
    return eta*u*u

# Decide where to apply which boundary
# 0: Neumann
# 1: Dirichlet
bctype = conditional(Or(x[0]<1e-8, x[0]>1.-1e-8), 0, 1)
sgn = conditional(x[0] > 0.5, 1., -1.)
j = -2.*sgn*x[0]

# Define the boundary measure that knows where we are...
ds = ds(subdomain_data=bctype)

r = inner(grad(u), grad(v))*dx \
  + q(u)*v*dx \
  - f*v*dx \
  + j*v*ds(0)

exact_solution = g
is_dirichlet = bctype
interpolate_expression = g
```

# UFL: DG Operators

UFL provides operators for implementation of Discontinuous Galerkin (DG) methods. These methods are discontinuous at interior facets. This means you have two values for the 'inside' cell and the 'outside' cell.

- `avg(u)`: Average between those values $\frac{1}{2}(u|_{T^+} + u|_{T^-})$
- `jump(u)`: Difference between the values $u|_{T^+} - u|_{T^-}$
- Restriction: Expression can be restricted to the inside or the outside cell by typing `u('+')` or `u('-')`[4]

[4] In the literature '-' usually denotes the inside and '+' the outside cell.

# UFL: FiniteElement

**VectorElement**

V= VectorElement(family, cell , degree [, size ])

- Combination of a basic element for a vector field
- `familty`, lstinlinecell, `degree` like `FiniteElement` above
- `size`: Optional, default equal to dimension

**TensorElement**

V = TensorElement(family, cell , degree [, shape, symmetry])

- Like `VectorElement` but for `shape` given as tuple
- Symmetry can be expressed as Python dictionary `symmetry={(0,1): (1,0)}`

**MixedElement**

V = MixedElement(element1, element2 [,...])

- Arbitrary combination of finite elements
- Can also be created like this `V = element1*element2`

# UFL: Trialfunctions and Testfunctions

- You can get the test- and trialfunctions of these spaces using the `split` command

```
FE_V = VectorElement('CG', triangle, 2)
FE_P = FiniteElement('CG', triangle, 1)
TH = FE_V * FE_P
u, p = split(TrialFunction(TH))
v, q = split(TestFunction(TH))
```

- There is also an abbreviation (don't miss the additional s)

```
u, p = TrialFunctions(TH)
v, q = TestFunctions(TH)
```

# Example: Wave Equation as First Order System

Strong formulation as first order system:

$$
\begin{aligned}
\partial_t u_1 - c^2 \Delta u_0 &= 0 && \text{in } \Omega \times \Sigma, \\
\partial_t u_0 - u_1 &= 0 && \text{in } \Omega \times \Sigma, \\
u_0 &= 0 && \text{on } \partial\Omega, \\
u_1 &= 0 && \text{on } \partial\Omega, \\
u_0 &= q && \text{at } t = 0, \\
u_1 &= w && \text{at } t = 0.
\end{aligned}
$$

Weak discrete formulation: Find $(u_0(t), u_1(t)) \in U_0 \times U_1$ s.t.

$$
\begin{aligned}
d_t (u_1, v_0)_{0,\Omega} + c^2 (\nabla u_0, \nabla v_0)_{0,\Omega} &= 0 \quad \forall v_0 \in U_0 \\
d_t (u_0, v_1)_{0,\Omega} - (u_1, v_1)_{0,\Omega} &= 0 \quad \forall v_1 \in U_1
\end{aligned}
$$

Parameters: Speed of sound $c = 1$

# Example: Wave Equation as First Order System

$$
\begin{aligned}
d_t(u_1, v_0)_{0,\Omega} + c^2(\nabla u_0, \nabla v_0)_{0,\Omega} &= 0 \quad \forall v_0 \in U_0 \\
d_t(u_0, v_1)_{0,\Omega} - (u_1, v_1)_{0,\Omega} &= 0 \quad \forall v_1 \in U_1
\end{aligned}
$$

```
cell = quadrilateral

V = VectorElement("CG", cell, 1)
u0, u1 = TrialFunctions(V)
v0, v1 = TestFunctions(V)

c = 1.0

mass = inner(u1, v0) * dx \
     + inner(u0, v1) * dx

r = c**2 * inner(grad(u0), grad(v0)) * dx \
  - inner(u1, v1) * dx
```

# UFL: Derivatives

- `grad(u)`: Gradient of u
- `div(u)`: Divergence of u
- `curl(u)`: Curl of u (only for finite element functions with three components)
- `u.dx(d)`: D'th partial derivative $\frac{\partial u}{\partial x_d}$
- UFL can also compute derivatives of forms or expressions wrt to `Variables` or `Coefficients` (Note: In `dune-codegen` the `TrialFunction` is a `Coefficient`)

```
u = Coefficient(element)
w = sin(u**2)
w = variable(w)
F = w**2

# Derivative of expression F
dF_w = diff(F, w)
dF_u = diff(F, u)
```

# UFL: `dune-codegen` Specific

- As mentioned before `dune-codegen` uses the residual formulation. The provided residual form may be nonlinear in the trial function.[5]
- Your UFL file may contain multiple forms. `dune-codegen` will generate local operators for all forms listed in the ini file, eg

```
[formcompiler]
operators = mass, poisson
```

- See the build system part of this tutorial for more options!

[5] In our case the trialfunction is a Coefficient and not an Argument.

# UFL: `dune-codegen` Specific

- For testing automated drivers can be generated. We use the following convention for instationary problems: If there are exactly two forms and one is called `mass` we assume that the problem is instationary and generate a suitable driver.[6]
- Instationary problems can have time dependent parameters but UFL has no notion of time. In `dune-codegen` you can get a variable representing the time by

  ```
  t = get_time(cell)
  ```

[6] Keep in mind that `dune-codegen` was developed to generate local operator. The driver generation was mainly done for testing.

# Example: Heatequation

```
cell = quadrilateral

x = SpatialCoordinate(cell)
time = get_time(cell)

g = cos(2*pi*time)*cos(pi*x[0])**2*cos(pi*x[1])**2

V = FiniteElement("CG", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)

mass = (u*v)*dx
poisson = inner(grad(u), grad(v))*dx

interpolate_expression = g
is_dirichlet = 1
```

# Contents

# CMake: `dune_add_generated_executable`

- We need to generate C++ code and compile it
- Add a code generation target to your `CMakeLists.txt`

```
dune_add_generated_executable(
    UFLFILE uflfile
    INIFILE inifile
    TARGET target
    [SOURCE source]
)
```

- `UFLFILE`: UFL file describing the PDE
- `INIFILE`: Ini file with code generation option under `[formcompiler]` section
- `TARGET`: Name of the executable
- `SOURCE`: C++ file used for building the target. This is optional, if omitted a minimal driver willl be generated

# CMake: `dune_add_generated_executable`

- Automated driver generation is mainly developed for automated software tests
- For complicated applications handwritten drivers will be necessary. This requires control over the file- and classname of the generated local operator.
- Can be done in the ini file

```
[formcompiler]
operators = r
...

[formcompiler.r]
filename = r_operator.hh
classname = ROperator
```

# Ini File: Global Options

- `operators`: Comma separated list of form names for which want to generate operators [**`default`** `r`]
- `explicit_time_stepping`: Use explicit time stepping (in instationary case) [`0/1`, **`default`** `0`]

# Ini File: Form Options

- Options for a form called `r` need to be put into the `[formcompiler.r]` section
- `filename`: Name of the generated local operator file `[str, optional]`
- `classname`: Name of the local operator class `[str, optional]`
- `numerical_jacobian`: Use numerical differentiation for assembling the Jacobian of the residual `[0/1, default 0]`
- `quadrature_order`: Order of quadrature `[int>0,],optional, guessed by UFL if omitted)`
- `geometry_mixins`: Information about grid properties that can lead to simplified gemometry evaluations `[generic/axiparallel/equidistant]`

# Ini File: Options for Generated Driver

## Grid generation

- Grid generation options are at the top under no section
- Quadrilateral grid

```
cells = 32 32
extension = 1. 1.
```

- Simplex grid

```
lowerleft = 0.0 0.0
upperright = 1.0 1.0
elements = 32 32
elementType = simplical
```

- Gmsh grid

```
gmshFile = cylinder2dmesh1.msh
```

# Ini File: Options for Generated Driver

**Name of vtk output**

- Under section `[wrapper.vtkcompare]`
- `name`: Basename (without ending) of vtk output

**Parameters for Instationary problems**

- Need to be put into the `[instat]` section
- `T`: End of time intervall
- `dt`: Time step size
- `output_every_nth`: Write visualization output for every nth time step

# CMake: Example Heatequation

```
cell = quadrilateral

x = SpatialCoordinate(cell)
time = get_time(cell)

V = FiniteElement("CG", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)

mass = (u*v)*dx
poisson = inner(grad(u), grad(v))*dx

# This example uses a hand written driver so these ar not needed!
# g = cos(2*pi*time)*cos(pi*x[0])**2*cos(pi*x[1])**2
# interpolate_expression = g
# is_dirichlet = 1
```

## CMakeLists.txt

```
dune_add_generated_executable(TARGET heatequation
                              UFLFILE heatequation.ufl
                              INIFILE heatequation.ini
                              SOURCE heatequation_driver.cc
                              )

dune_symlink_to_source_files(FILES heatequation.ini)
```

# CMake: Example Heatequation

## heatequation.ini

```
cells = 32 32
extension = 1. 1.

[wrapper.vtkcompare]
name = heatequation

[instat]
T = 1
dt = 0.01
output_every_nth = 5

[formcompiler]
operators = mass, poisson
explicit_time_stepping = 0

[formcompiler.mass]
filename = heatequation_mass_operator.hh
classname = MassOperator
geometry_mixins = equidistant

[formcompiler.poisson]
filename = heatequation_poisson_operator.hh
classname = PoissonOperator
geometry_mixins = equidistant
```

# Examples

In the folder `tutorial09/src` you can find several examples:

- Poisson equation from tutorial00
- Nonlinear Poisson equation with mixed boundary from tutorial01
- Heat equation from tutorial03
- Wave equation from tutorial04

In the exercises you will additionally find examples for:

- Navier Stokes equation modeling the flow around a cylinder from tutorial08
- Discontinuous Galerkin discretization of the Poisson equation