Interpreted programming in FEniCS

Johan Jansson

jjan@csc.kth.se

CSC (old NADA)

KTH

Contents

- Motivate higher-level programming languages (abstraction)
- Overview of PyDOLFIN (FEniCS/DOLFIN Python interface)

Science

- Formulate Model = Formulate Equation (Modeling)
- Solve Equation (Computation)

Model = Differential Equations (DE)

$$\dot{u} = f(u, \nabla u), \qquad \text{in } \Omega \times (0, T]$$

$+$ initial and boundary conditions

$$\dot{u} = \frac{\partial u}{\partial t}$$

$$\nabla u = (\frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \frac{\partial u}{\partial x_3})$$

Basic Models = DE

Navier: Solid Mechanics

$$\ddot{u} - \nabla \cdot \sigma = f, \quad \sigma = \mu e(u)$$

Navier-Stokes (Fluid Mechanics)

$$\dot{v} + v \cdot \nabla v - \nu \Delta v + \nabla p = f, \quad \nabla \cdot v = 0$$

Maxwell (Electromagnetism)

$$\nabla \times H = J + \dot{D}, \quad \nabla \times E = -\dot{B}, \quad \nabla \cdot D = \rho, \quad \nabla \cdot B = 0$$

Schrödinger (Quantum Mechanics)

$$i\dot{\psi} - \frac{h^2}{2m} \triangle \psi + V\psi = 0$$

A few more

FEniCS

Automation of Computational Mathematical Modeling (ACMM)

Automation of:

(a) Discretization of differential equations

(b) Solution of discrete systems

(c) Error control of discrete solutions

(a)-(c): Galerkin's method (FEM) + Duality

FEniCS interface

1. Input DE
2. ???
3. Interpret solution of DE

2 includes:

- Manipulation/generation of DE, discrete systems.
- Primitives for solving discrete systems.
- ...

Aim to remove dividing line between manual computation/expression manipulation on paper and computer programming.

Perspective on computer programming

- All programming languages (in practical use) are Turing complete.
- A Turing machine describes all of mathematics (definition).

Choosing a language is thus only a question of efficiency or administration: having to do as little manual work as possible.

Automation = Maximal Laziness

Knuth:

> "Premature optimization is the root of all evil"

- Keep a high abstraction level. Do not optimize.
- Typically: 90% of the time is spent in 10% of the source code. Do not optimize 90% of the source code.
- Resist urge to be clever.

Higher-level language

Abstraction progression:

Assembler operate on numbers

C/Fortran operate on arrays

Python/... operate on functions (equations)

Vision: Implementation of algorithms on form/equation level (stabilization, error control).
Higher-level languages commonly interpreted.

Interpreted language

Compiled Translate source code into lower level code before execution. Static typing.

Interpreted Translate source code into lower level code during execution. Allows dynamic typing, dynamic creation of new types.

Allows introducing new definitions/abstractions while running the program, analogy to pen & paper development.

“Pocket-calculator” interface (Matlab/Octave, UNIX shell).

FEniCS interface example

```
class MyFunction(Function):
    def eval(self, point, i):
        return sin(point[1]) + 1.0

K = FiniteElement("Lagrange", "triangle", 1)

mesh = UnitSquare(20, 20)

f = MyFunction()

Pf = project(f, K, mesh)

plot(Pf)
```

Interface example

```
def projection(K):
    # Construct projection form in FFC representation
    v = TestFunction(K)
    U = TrialFunction(K)

    g = Function(K)

    a = dot(v, U) * dx
    L = dot(v, f) * dx

    return [a, L]
```

```
def project(f, K, mesh):
    # Assemble discrete system
    M = Matrix()
    b = Vector()
    assemble(a, L, M, b, f, mesh)

    # Solve discrete system
    x = Vector()
    solve(M, x, b)

    # Define a function from computed degrees of freedom
    Pf = Function(x, mesh, K)
```

Time-dependent PDE

Automatic time-discretization by MG:

$$\dot{u} = f(t,u) \quad \text{in } \Omega \times (0,T]$$

$$\int_\Omega \dot{u}v = \int_\Omega f(t,u)v \quad \text{in } \Omega \times (0,T], \forall v \in V$$

$$M\dot{\xi} = b(t,\xi) \quad \text{in } (0,T]$$

DOLFIN can automatically construct M and $b(t,\xi)$ from a description of $\int_\Omega \dot{u}v$ and $\int_\Omega f(t,u)v$ in the **FFC** form language.

Exists in interface as: `TimeDependentPDE`.

Elasticity example: form

```
K1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
K2 = FiniteElement("Vector Lagrange", "tetrahedron", 1)

Kmix = element1 + element2

(w_0, w_1) = TestFunctions(Kmix)
(Udot_0, Udot_1) = TrialFunctions(Kmix)
(U_0, U_1) = Functions(Kmix)

f = Function(K2)

# Dimension of domain
d = element1.shapedim()

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

sigma = mult(10.0, epsilon(U_0))

a = (dot(Udot_0, w_0) + dot(Udot_1, w_1)) * dx
L = (dot(U_1, w_0) - dot(sigma, epsilon(w_1)) + dot(f, w_1)) * dx
```

Elasticity example: PDE

```
class ElasticityPDE(TimeDependentPDE):
    def __init__(self, mesh, f, bc, T):

        forms = import_formfile("Elasticity.form")

        # Initialize variables...

    def fu(self, x, dotx, t):

        # Assemble right-hand side

        FEM_assemble(self.L(), dotx, self.mesh())
        FEM_applyBC(dotx, self.mesh(), self.a().trial(), self.bc())

        dotx.div(m)
```

Ko

All DE solved using same interface, just specify the DE: $f(u)$.

Ko: Large deformation elasto-visco-plasto with contact.

Ko Stair 1

Ko Stair 2

Ko Frontal

Contact implemented as mass-spring model in C++, transparently used in Python, TimeDependentPDE interface.

SWIG: automatic interface generation

Principle similar to FEniCS: automatically computes a mapping from a C/C++ interface to Python (+other languages). Can use interface language for tailoring.

Consequence: can use compiled language to define low-level algorithms and data structures, can use an interpreted language for structure and further abstraction.

“JIT”

Use SWIG/compiler to transparently generate efficient implementation of low-level algorithms: form evaluation, coefficient evaluation.

```
K = FiniteElement("Lagrange", "triangle", 1)
f = Function(K)

a = dot(grad(v), grad(U))*dx
L = v*f*dx

# Import compiled forms
forms = import_form([a, L, None], ‘‘Poisson’’)

a = forms.PoissonBilinearForm()
L = forms.PoissonLinearForm(f)
```

```
coeffs = import_header(‘‘Coefficients.h’’)

# Import compiled coefficient
f = coeffs.MySource()
```

Instant.

Performance

- Performance overhead of using Python negligible
- Utilize pre-computation (forms) and selective compilation (coefficients) to achieve performance.
- General principle: $\Omega(n)$ algorithms should be implemented in a compiled language.

Weaknesses

- Multiple abstractions for same concept (triplicate of Finite Element in FIAT, FFC, DOLFIN). Name collisions.
- Python definitions cannot easily be used in C++ (need SWIG in reverse mode). Possible solution: embed Python interpreter.
- Linear algebra interface not very complete in Python.

Future

Simple observation

State of the art in mathematics: Perelman submits proof of Poincaré. Proof is manually checked, takes months.

Automation/computers have existed for over 50 years, still have not penetrated very deep into science, except essentially as adding machines.