AD libraries + FEniCS/Firedrake

Ivan Yashchuk
Aalto University | Quansight Labs

FEniCS’21 Conference
ivan.yashchuk@aalto.fi

$$\int_{\Omega} \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x - \int_{\Omega} f \cdot v \, \mathrm{d}x = 0$$

```
# Create mesh for the unit square domain
n = 10
mesh = UnitSquareMesh(n, n)

# Define discrete function spaces and functions
V = FunctionSpace(mesh, "CG", 1)
W = FunctionSpace(mesh, "DG", 0)




def fenics_solve(f):

    u = Function(V, name="PDE Solution")
    v = TestFunction(V)
    F = (inner(grad(u), grad(v)) - f * v) * dx
    bcs = [DirichletBC(V, 0.0, "on_boundary")]
    solve(F == 0, u, bcs)
    return u
```

$$\int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x - \int_\Omega f \cdot v \, \mathrm{d}x = 0$$

```python
# Create mesh for the unit square domain
n = 10
mesh = UnitSquareMesh(n, n)

# Define discrete function spaces and functions
V = FunctionSpace(mesh, "CG", 1)
W = FunctionSpace(mesh, "DG", 0)

# Define FEniCS template representation of JAX input
templates = (Function(W),)

@build_jax_fem_eval(templates)
def fenics_solve(f):
    # This function inside should be traceable by fenics_adjoint
    u = Function(V, name="PDE Solution")
    v = TestFunction(V)
    F = (inner(grad(u), grad(v)) - f * v) * dx
    bcs = [DirichletBC(V, 0.0, "on_boundary")]
    solve(F == 0, u, bcs)
    return u

# Calculate the solution jacobian using JAX and adjoint PDE
dudf = jax.jacobian(fenics_solve)(f)
```

Behind the scenes: Tangent and Adjoint PDEs

Symbolic form of the problem is used to derive additional PDEs that are solved for calculating Jacobian-vector and vector-Jacobian products.

Let `F(u, m) = 0` represent the PDE,
`u` represents the solution and `m` represents the parameters.

Jacobian-vector product:

$$\frac{\mathrm{d}u}{\mathrm{d}m}\dot{v} := -\frac{\partial F}{\partial u}^{-1}\frac{\partial F}{\partial m}\dot{v}$$

vector-Jacobian product:

$$\frac{\mathrm{d}u}{\mathrm{d}m}^{*}\bar{v} := -\frac{\partial F}{\partial m}^{*}\left[\frac{\partial F}{\partial u}^{-*}\bar{v}\right]$$

Jacobian-vector product

$$\frac{\mathrm{d}u}{\mathrm{d}m}\dot{v} := -\frac{\partial F}{\partial u}^{-1}\frac{\partial F}{\partial m}\dot{v}$$

```
dFdu = ufl.derivative(F, u)
dFdm = ufl.derivative(F, m, v̇)
dudm_v̇ = fenics.Function(V)
tlm_F = ufl.action(dFdu, dudm_v̇) + dFdm
tlm_F = ufl.replace(tlm_F, {dudm_v̇: fenics.TrialFunction(V)})
fenics.solve(ufl.lhs(tlm_F) == ufl.rhs(tlm_F), dudm_v̇, bcs=hbcs)
```

https://github.com/IvanYashchuk/jax-fenics

Jacobian-transpose-vector product

$$\frac{\mathrm{d}u}{\mathrm{d}m}^* \bar{v} := -\frac{\partial F}{\partial m}^* \left[\frac{\partial F}{\partial u}^{-*} \bar{v}\right]$$

```
dFdu = ufl.derivative(F, u)
adFdu = ufl.adjoint(dFdu)

u_adj = fenics.Function(V)
adj_F = ufl.action(adFdu, u_adj)
adj_F = ufl.replace(adj_F, {u_adj: fenics.TrialFunction(V)})
fenics.solve(adj_F == v̅, u_adj)

dFdm = ufl.derivative(F, fenics_input, fenics.TrialFunction(V))
adFdm = ufl.adjoint(dFdm)
dudm*_v̅ = fenics.assemble(-adFdm * u_adj)
```

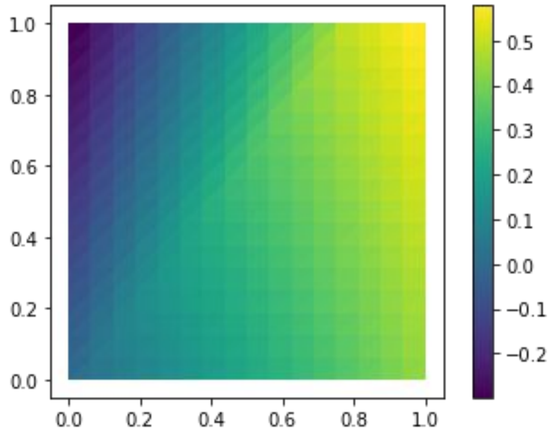
https://github.com/IvanYashchuk/jax-fenics

Composition with other JAX programs

Let's use jax.stax to set up network initialization and evaluation functions

```
# Define R^2 -> R^1 function
net_init, net_apply = jax.experimental.stax.serial(Dense(2), Relu, Dense(10), Relu, Dense(1))

nn_predictions = net_apply(net_params, W.tabulate_dof_coordinates())
f_nn = numpy_to_fenics(nn_predictions, fenics.Function(W))
```



https://nbviewer.jupyter.org/github/IvanYashchuk/jax-fenics-adjoint/blob/master/notebooks/poisson-intro.ipynb

In [ ]:
```python
def eval_nn(net_params):
    f_nn = np.ravel(net_apply(net_params, W.tabulate_dof_coordinates()))
    u = fenics_solve(f_nn)
    norm_u = np.linalg.norm(u)
    return norm_u
```

In [ ]:
```python
%%time
jax.grad(eval_nn)(net_params)
```

```
CPU times: user 340 ms, sys: 7.82 ms, total: 348 ms
Wall time: 337 ms
```

Out[ ]:
```
[(DeviceArray([[ 0.11934833, -0.08526856],
               [ 0.11315436, -0.16367367]], dtype=float32),
  DeviceArray([ 0.22669935, -0.25705874], dtype=float32)),
 (),
 (DeviceArray([[ 0.0000000e+00,  0.0000000e+00,  3.4377411e-01,
                -3.8510424e-01,  4.1671190e-02,  0.0000000e+00,
                 2.8023928e-01, -2.3091170e-06, -4.5939538e-01,
                 0.0000000e+00],
               [ 0.0000000e+00,  0.0000000e+00,  1.2621611e-02,
                -1.4139040e-02,  1.5299511e-03,  0.0000000e+00,
                 1.0288941e-02, -3.4431054e-08, -1.6866626e-02,
                 0.0000000e+00]], dtype=float32),
  DeviceArray([-0.0000000e+00, -0.0000000e+00,  3.9731458e-01,
               -4.4508159e-01,  4.8161194e-02,  0.0000000e+00,
                3.2388464e-01, -3.3104476e-05, -5.3094304e-01,
                0.0000000e+00], dtype=float32)),
 (),
 (DeviceArray([[0.00000000e+00],
               [0.00000000e+00],
               [2.40548089e-01],
               [1.17404766e-01],
               [3.21369737e-01],
               [0.00000000e+00],
               [3.51112783e-01],
               [1.75102848e-08],
               [1.22634955e-01],
               [0.00000000e+00]], dtype=float32),
  DeviceArray([0.6358373], dtype=float32))]
```

“Physics-Informed” Neural Networks (PINN)

Poisson problem,

$$\text{Find} \quad u \in V \quad \text{such that} \quad a(u,v) = L(v) \quad \text{for all } v \in V$$

with

$$a(u,v) = \int_\Omega \nabla v \cdot \nabla u \, dx, \qquad L(v) = \int_\Omega v \, f \, dx$$

Equivalent minimization problem:

$$J(v) = \frac{1}{2} a(v,v) - L(v)$$

$$J(u) \le J(v) \quad \forall v \in V$$

Usual FEM: $v(x) = \sum_j c_j \psi_j(x)$ Taking `c = nn(x; coefficients)`

and solving the minimization problem for neural network coefficients we get PINN.

“Physics-Informed” Neural Networks (PINN)

```
def eval_nn(net_params)
"""
Given the neural network parameters assemble
the energy integral using FEniCS and return the value
ℝ^m → ℝ, m = size(net_params)
"""
    v_nn = net_apply(net_params, W.tabulate_dof_coordinates())
    value = fenics_assemble_energy(v_nn)
    return value

jax.value_and_grad(eval_nn)(net_params) # (ℝ, ℝ^m)
jax.hessian(eval_nn)(net_params) # ℝ^m × ℝ^m
```

dolfin-adjoint is not enough but pyadjoint is

pyadjoint/dolfin-adjoint is an automatic differentiation for FEniCS and Firedrake
+ an interface to selected optimization libraries (SciPy, IPOpt, Moola, PyROL)

The goal is to embed PDE solvers inside other programs for

- composition with other differentiable programs (for example neural networks)
- probabilistic parameter estimation
- interface to optimization and sampling libraries outside of dolfin-adjoint

This work is about serialisation layer using NumPy arrays and API that simplifies embedding of FEniCS/Firedrake in AD libraries

Finite Element Chain Rules (inspired by ChainRules.jl)

FEniCS passing | Firedrake passing | codecov 77%

A serialisation layer using NumPy arrays
+ API that simplifies embedding of FEniCS/Firedrake in AD libraries

```
def evaluate_primal(
    firedrake_function: Callable[..., BackendVariable],
    firedrake_templates: Collection[BackendVariable],
    *numpy_inputs: np.array,
) -> Tuple[np.array, BackendVariable, Collection[BackendVariable], pyadjoint.Tape]

def evaluate_pushforward(
    firedrake_output: BackendVariable,
    firedrake_inputs: Collection[BackendVariable],
    tape: pyadjoint.Tape,
    Δnumpy_inputs: Collection[np.array],
) -> Collection[np.array]

def evaluate_pullback(
    firedrake_output: BackendVariable,
    firedrake_inputs: Collection[BackendVariable],
    tape: pyadjoint.Tape,
    Δnumpy_output: np.array,
) -> Collection[np.array]
```

https://github.com/IvanYashchuk/fecr

FECR

PyMC3

```python
from fenics_pymc3 import create_fenics_theano_op
# Define FEniCS template representation of Theano/NumPy input
# that is we promise that our arguments are of the following types
# the choice is between Constant and Function
templates = (fenics_adjoint.Constant(0.0), fenics_adjoint.Constant(0.0))

@create_fenics_theano_op(templates)
def solve_elasticity(E, ρ_g):
    ...
    return solution

import pymc3 as pm
import theano.tensor as tt

loads = [[1.], [2.5], [5.]]
measurements = [0.11338, 0.28346, 0.56693]

with pm.Model() as model:

    E = pm.Normal("E", mu=1.1e5, sigma=0.3e5, shape=(1,))

    maximum_deflections = []
    for i in range(len(measurements)):
        ρ_g = loads[i]
        predicted_displacement = solve_elasticity(E, ρ_g)
        maximum_deflection = tt.max(predicted_displacement)
        maximum_deflections.append(maximum_deflection)
    maximum_deflections = tt.stack(maximum_deflections)

    d = pm.Normal("d", mu=maximum_deflections, sd=1e-3, observed=measurements)
```

```python
map_estimate = pm.find_MAP(model=model)
print(f"MAP estimate of E is {map_estimate['E']}")
# 100% [22/22 00:04<00:00 logp = 6.6135,
# ||grad|| = 1.1144e-05]
# MAP estimate of E is [125000.40465515]

with model:
    trace = pm.sample(100, chains=1, cores=1, tune=100)
pm.summary(trace)
# Sequential sampling (1 chains in 1 job)
# NUTS: [E]
#               mean       sd       hdi_3%     hdi_97%
# E[0]  132148.824  285.805  131637.637  132484.79
```

https://ivanyashchuk.github.io/fenics_pymccon2020/

Julia | Turing.jl

```julia
function solve_firedrake(kappa0, kappa1)
    ...
    firedrake.solve(F == 0, u, bcs = bcs)
    return u
end

@register_fem_function(
  zygote_solve_firedrake, templates, solve_firedrake)

@model function fit_diffusion_coef(data)
   σ ~ InverseGamma(3, 0.5)
   kappa0 ~ truncated(Normal(1.0, 0.5), 1e-5, 2)
   kappa1 ~ truncated(Normal(0.7, 0.5), 1e-5, 2)
   predicted_solution = zygote_solve_firedrake([kappa0], [kappa1])
   data ~ MvNormal(predicted_solution, σ)
end
model = fit_diffusion_coef(noisy_solution)
chain = sample(model, NUTS(.65), 1000)
```

Summary Statistics

parameters	**mean**	**std**	**naive_se**	**mcse**	**ess**	**rhat**
Symbol	Float64	Float64	Float64	Float64	Float64	Float64
kappa0	1.2497	0.3789	0.0120	0.0357	130.3485	1.0001
kappa1	0.5443	0.1711	0.0054	0.0155	139.7087	1.0001
σ	0.0143	0.0019	0.0001	0.0001	178.8158	1.0043

Quantiles

parameters	**2.5%**	**25.0%**	**50.0%**	**75.0%**	**97.5%**
Symbol	Float64	Float64	Float64	Float64	Float64
kappa0	0.5183	0.9850	1.2590	1.5483	1.9140
kappa1	0.2295	0.4291	0.5424	0.6698	0.8579
σ	0.0111	0.0130	0.0142	0.0154	0.0188

https://github.com/IvanYashchuk/PyFenicsAD.jl

Summary | AD + FEniCS/Firedrake

What?

Automatic forward and reverse differentiation of FEniCS/Firedrake composable with JAX | PyMC3 | Julia

Why?

Reuse existing well established libraries instead of reinventing the wheels in "differentiable physics" fashion

Composability with other libraries of host AD:

Including PDEs in probabilistic modelling using PyMC3 | Turing.jl | NumPyro (JAX)

Interfacing with optimization and sampling libraries

What's next?

Arbitrary higher-order derivatives for JAX and Julia

Distributed array interface

Compatibility with JAX's JIT compilation

How to get started?

Step 1: Install Firedrake or FEniCS
For embedding in other AD libraries:
https://github.com/IvanYashchuk/fecr

For JAX interface:
https://github.com/IvanYashchuk/jax-fenics-adjoint

For PyMC3 interface:
https://github.com/IvanYashchuk/fenics-pymc3

For Julia interface:
https://github.com/IvanYashchuk/PyFenicsAD.jl