An introduction to FEniCS for multidisciplinary design optimization

David Kamensky

Notes for the NASA ULI Software Workshop on 4 February, 2022

1 Introduction

These notes were written to serve as the basis for a live tutorial [1], intended to set graduate engineering students on a path toward learning to use the software FEniCS [2] to perform finite element analysis (FEA) in a multidisciplinary design optimization (MDO) setting. Based on my interactions with many students, both under my direct supervision and through the online FEniCS Q&A forum [3], I believe that the main hurdle for students from science and engineering backgrounds is learning the underlying mathematical framework, not the details of the software's application–programmer interface (API). Many excellent resources already exist to learn the software [4]. These notes are intended to complement those resources, not replace them. Much of the existing FEniCS documentation takes a modern mathematical perspective on FEA for granted. While this perspective is decades-old and familiar to the applied math community, it is still quite alien to most engineers.

These notes begin with a cursory introduction to the mathematical theory of FEA. It is based on a simple model problem, but intended to generalize easily. I then show how to solve the model problem with FEniCS and continue with some interesting variations that show off unique capabilities of FEniCS that are relevant to design optimization. These notes assume familiarity with linear algebra, differential equations, and vector calculus, at a level that is typically covered by undergraduate engineering curricula, along with basic Python programming knowledge and the general computing skills needed to follow existing online instructions for installing FEniCS. The final example will be most compelling to those with some knowledge of design optimization, but that is not a strict prerequisite.

The code examples in these notes are based on version 2019.2.0 of FEniCS, which remains the most widely used version at time of writing. This is colloquially referred to as "old FEniCS" or "legacy FEniCS", since a redesign, FEniCSx, is underway. FEniCSx is not backward compatible and will undoubtedly break all the specific code examples here, but the emphasis of these notes is on the *mindset* required to use FEniCS effectively in an MDO setting, which is largely independent of the API changes between old FEniCS and FEniCSx. The code examples are nonetheless provided in a public Git repository [5], for those interested in following along interactively.

2 What is FEA?

Historically, the ideas and methods of FEA emerged from the subject of structural mechanics, where solid objects are decomposed into finite collections of "elements" that satisfy prescribed kinematic assumptions. Many practitioners in the broader engineering community still view FEA through this lens. However, **the modern understanding of FEA is as a general method for approximating partial differential equations (PDEs)**, with no special relationship to solid or structural mechanics, beyond the aforementioned historical one. Using FEniCS effectively depends crucially on understanding this modern perspective, especially in the setting of MDO, where, by definition, one is interested in multiple disciplines, which might be modeled by a variety of PDEs.

2.1 What are PDEs?

PDEs are a generalization of ordinary differential equations (ODEs). An ODE relates various derivatives of an unknown function u on the domain $(0, L)$, e.g.,

$$\begin{cases} au''(x) + bu'(x) = f(x) \quad \text{for} \quad x \in (0, L) \,, \\ u(0) = u(L) = 0 \,, \end{cases} \tag{1}$$

where the coefficients a and b and the function $f(x)$ are given. A PDE relates *partial* derivatives of an unknown multivariate function u on the domain Ω, e.g.,

$$\begin{cases} -\kappa\left(\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y)\right) = f(x,y) \quad , \quad (x,y) \in \Omega \,, \\ u(x,y) = 0 \qquad\qquad\qquad\qquad\qquad\quad , \quad (x,y) \in \partial\Omega \,, \end{cases} \tag{2}$$

where the coefficient $\kappa > 0$ and the function $f(x, y)$ are given and the notation "$\partial\Omega$" stands for "the boundary of Ω". This particular PDE is often referred to as Poisson's equation, and can be used to model steady heat conduction, diffusion of chemical species, electrical and gravitational potentials, deflections of thin membranes, stresses on cross-sections of torsional members, distributions of pressure in incompressible flows, flow through porous media, and many other things.[1] An undergraduate-level introduction to PDEs can be found in the text of Haberman [7], and the standard graduate-level PDE text is that of Evans [8].

PDEs like (2) arise naturally from conservation laws in physics. Using vector calculus notation we can rewrite the left-hand side of (2) as

$$\nabla \cdot \mathbf{q} \,, \quad \text{where} \quad \mathbf{q} = -\kappa \nabla u \,, \tag{3}$$

i.e., the divergence of a flux vector, $\mathbf{q}$, indicating the flow of quantity u. In Poisson's equation, the flow is "downhill"; if u were thermal energy, it would express that heat flows from hot to cold. Applying divergence theorem, we see that the flow through the boundary balances the rate of production by the distributed source f:

$$\int_{\partial\Omega} \mathbf{q} \cdot \mathbf{n} \, d\partial\Omega = \int_{\Omega} f \, d\Omega \,, \tag{4}$$

where $\mathbf{n}$ is the unit outward normal to $\partial\Omega$. Many physically-relevant PDEs follow from conservation laws by similar reasoning with different flux definitions, and much of what follows can be generalized. However, we shall focus on (2) as a model problem in these notes.

For both ODEs and PDEs, it can be useful to think of them in terms of a differential operator. E.g., (2) can be written

$$Au = f \,, \quad \text{where} \quad A = -\kappa\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial x^2}\right) = \nabla \cdot (-\kappa \nabla(\cdot)) = -\kappa \nabla^2 \,. \tag{5}$$

The operator A in this case is linear, i.e.,

$$A(\alpha u + \beta v) = \alpha A u + \beta A v \tag{6}$$

for functions u and v and scalars α and β. Thus, we can make an analogy to solving a linear system of equations

$$\mathbf{A}\mathbf{u} = \mathbf{f} \,, \tag{7}$$

for the unknown vector $\mathbf{u} \in \mathbb{R}^N$, given the $N \times N$ matrix $\mathbf{A}$ and vector $\mathbf{f} \in \mathbb{R}^N$. We know how to solve this by inverting the matrix $\mathbf{A}$ with Gaussian elimination, as implemented by MATLAB's backslash operator or linear solver routines in other software libraries.

To solve the PDE (5), one would ideally "invert" the differential operator A, but there is no general algorithm to do so! Let us look back to the more tractable linear system (7) for inspiration. An equivalent way to write this is to "test" whether it's satisfied along every direction in $\mathbb{R}^N$: Find $\mathbf{u} \in \mathbb{R}^N$ such that, for all $\mathbf{v} \in \mathbb{R}^N$,

$$(\mathbf{A}\mathbf{u}, \mathbf{v}) = (\mathbf{f}, \mathbf{v}) \,, \quad \text{where} \quad (\mathbf{a}, \mathbf{b}) = \mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b} \text{ is the dot product.} \tag{8}$$

If this is true, we can clearly set $\mathbf{v}$ to the i^{th} standard basis vector $\mathbf{e}_i$, to verify that the i^{th} equation in the system is satisfied. (Conversely, if it holds for every basis vector, it holds for every test vector $\mathbf{v}$, since both sides are linear in $\mathbf{v}$.)

[1] An interesting informal discussion on why this partial differential operator is so ubiquitous can be found on the MathOverflow forum [6], including thoughts from some prominent mathematicians.

This leads us to consider testing whether a PDE is satisfied when "dotted with" arbitrary functions: Find a function u such that, for all test functions v,

$$(Au, v) = (f, v) \,. \tag{9}$$

When considering functions instead of vectors, let

$$(f, g) = \int_\Omega fg \, d\Omega \,, \tag{10}$$

which is analogous to the dot product; with some loss of rigour, we can view it as adding up the products of the corresponding "components" of two functions, which are indexed by points in Ω rather than integers.

We encounter a technical difficulty in considering "all functions", though, because (f, g) may not be defined for arbitrary f and g, so we must set some restrictions on our test function v. In light of our analogy to linear algebra, it would be convenient if we could place the same restrictions on u and v, so that the operator A is like a square matrix. Considering the particular operator from (5), we can use the product rule and divergence theorem to get

$$\begin{aligned}
(Au, v) &= \int_\Omega (\nabla \cdot \mathbf{q}) v \, d\Omega && (11)\\
&= \int_\Omega \nabla \cdot (v\mathbf{q}) - \mathbf{q} \cdot \nabla v \, d\Omega && (12)\\
&= \int_{\partial\Omega} (v\mathbf{q}) \cdot \mathbf{n} \, d\partial\Omega - \int_\Omega \mathbf{q} \cdot \nabla v \, d\Omega && (13)\\
&= -\int_\Omega \mathbf{q} \cdot \nabla v \, d\Omega && (14)\\
&= \int_\Omega \kappa \nabla u \cdot \nabla v \, d\Omega \,, && (15)
\end{aligned}$$

where the boundary term is zero from assuming that v satisfies the same zero boundary condition as u on $\partial\Omega$. This operation of "moving the derivative and flipping the sign" is often referred to as **integration by parts** in the PDE literature. From (15), we can conclude that the restrictions we need to place on u and v are that they satisfy the boundary condition and have square integrable gradients, i.e., that they are in a **function space** V satisfying these restrictions:

$$u, v \in V := \left\{ \text{Functions } f \text{ such that } \int_\Omega |\nabla f|^2 \, d\Omega < \infty \text{ and } f = 0 \text{ on } \partial\Omega \right\} . \tag{16}$$

Thus, we arrive at the **weak form** of Poisson's equation: Find $u \in V$ such that, for all $v \in V$,

$$\int_\Omega \kappa \nabla u \cdot \nabla v \, d\Omega = \int_\Omega f v \, d\Omega \,. \tag{17}$$

It is "weaker" in the sense that we're only seeking a solution in V, which does not enforce the condition that second derivatives of u exist. Thus, it is not immediately apparent that it will satisfy the original PDE, since (2) requires the stronger condition that u has second derivatives defined at every point in the domain. Under many conditions, the weak solution will in fact satisfy the strong form (2), but the weak problem remains well-posed in more extreme situations, e.g., if f is a concentrated source, inducing a discontinuity in the flux.

Remark 1. Weak forms of PDEs are not unique. We could just have well introduced a second operator B, and considered $(Au, Bv) = (f, Bv)$ $\forall v$. In the linear algebra analogy, this would be like introducing a matrix $\mathbf{B}$ and considering $(\mathbf{Au}, \mathbf{Bv}) = (\mathbf{f}, \mathbf{Bv})$ $\forall \mathbf{v}$. If $\mathbf{B}$ is invertible, this is clearly still equivalent to $\mathbf{Au} = \mathbf{f}$. However, in the PDE setting, the choice of B affects what restrictions must be placed on the space V.

Remark 2. The extension of linear algebra from finite-dimensional vector spaces to infinite-dimensional function spaces is a branch of mathematics called **functional analysis** [9], which underpins much of modern PDE theory, but we shall settle for a loose formal analogy in these notes.

2.2 From weak PDEs to FEA

The design of FEniCS is based on the insight that, given a weak form of a PDE, the remaining steps of implementing the finite element method can be completely automated. We shall proceed with the model problem (2) as an example, but, to emphasize the general applicability of what follows, we abbreviate it: Find $u \in V$ such that, for all $v \in V$,

$$a(u, v) = L(v) \,, \tag{18}$$

where the **bilinear form** a and **linear form** L are

$$a(u, v) := \int_\Omega \kappa \nabla u \cdot \nabla v \, d\Omega \quad \text{and} \quad L(v) := \int_\Omega f v \, d\Omega \,. \tag{19}$$

Any linear PDE can be rewritten in this form, with different definitions of a and L. FEniCS is essentially a system for automating the remainder of this section, given symbolic definitions of a and L.

In summary, FEA replaces V with a finite-dimensional subset V^h, which consists of functions that are piecewise polynomial over some mesh of elements whose diameters are proportional to a length scale h. I.e., we now want to solve: Find $u^h \in V^h$ such that, for all $v^h \in V^h$,

$$a\left(u^h, v^h\right) = L\left(v^h\right) \,. \tag{20}$$

Ideally, the solution u^h should converge to u as elements get smaller, i.e., $u^h \to u$ as $h \to 0$. This can be proven mathematically under suitable assumptions, but we shall ignore convergence analysis for now, in favor of the operational "how-to" aspects of FEA.

For our model problem, consider piecewise linear and continuous functions on some triangulation of Ω. A basis function can be associated with each non-boundary vertex as depicted in Figure 1. The gradients of such functions will be constant on each element, and thus clearly satisfy the requirements of the space V defined in (16). If we assume

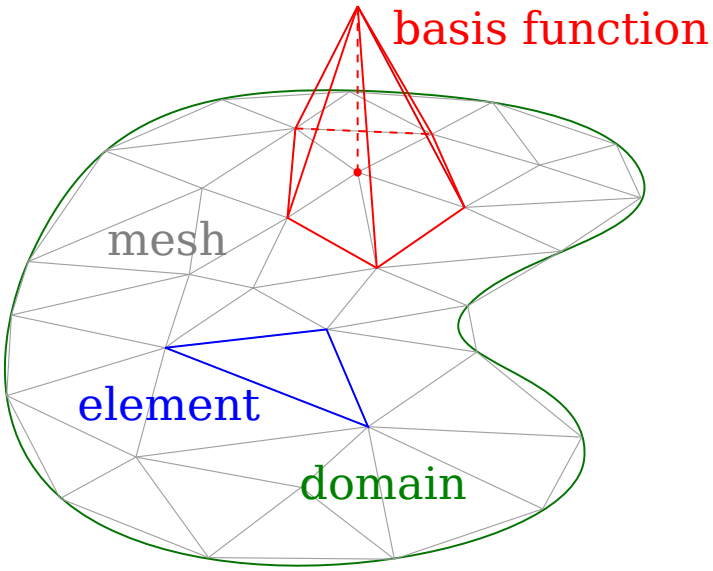

Figure 1: A single basis function for a space of piecewise linear continuous functions defined on a mesh of triangle elements.

there are N such basis functions, denoted $\{\phi_1, \cdots, \phi_N\}$, and notice that it is sufficient to satisfy (20) for v^h equal to each basis function, then we have come full circle and arrived at a linear algebra problem:

$$\mathbf{A}\mathbf{u} = \mathbf{f} \,, \tag{21}$$

where $\mathbf{u}$ is the vector of unknown coefficients $\mathbf{u} = [u_1, \cdots, u_N]^T$ for the discrete solution

$$u^h(x, y) = u_1\phi_1(x, y) + \cdots + u_N\phi_N(x, y) \,, \tag{22}$$

the matrix $\mathbf{A}$ has components

$$A_{ij} = a(\phi_j, \phi_i) \,, \tag{23}$$

and the vector $\mathbf{f}$ has components

$$f_i = L(\phi_i) \,. \tag{24}$$

Due to the historical connection with structural mechanics, $\mathbf{A}$ is often called the "stiffness matrix", while $\mathbf{f}$ is often called the "load vector".

Remark 3. Continuing the train of thought from Remark 1, note that different choices of B will lead to different forms a and L, and thus different approximate solutions u^h. The operator B can be tuned for different problems, depending on the features of the problem and what aspects of the solution are considered important. Methods with $B = 1$ (as above) are often referred to as the "Bubnov–Galerkin"—or just "Galerkin"—methods, while other formulations are called "Petrov–Galerkin" methods. One often sees the misconception that FEA is inherently unsuitable for PDEs where the Bubnov–Galerkin method performs poorly. However, this is analogous to claiming that finite differences are unsuitable for fluid mechanics, because the specific choice of a centered difference stencil is unstable for advection.[2] FEniCS is an excellent tool for experimenting with different choices of a and L, to develop optimal solvers for different disciplines.

3 Automated FEA with FEniCS

FEniCS provides a Python-based domain-specific language, called the Unified Form Language (UFL) [11], for defining forms a and L. It includes powerful computer algebra functionality, to partially automate even the process of writing down the weak problem. UFL descriptions of forms are then compiled [12] automatically into C++ routines that plug into the high-performance solver DOLFIN [13]. Typically, users interact with DOLFIN through its Python API, such that operations in UFL and DOLFIN blend seamlessly together in a single script.

To demonstrate, consider again our model discretized weak problem (20). In FEniCS, we can solve this on the rectangle $\Omega = [1, 2] \times [0, \pi/2]$ and plot the results with a simple Python script:

```
from fenics import *
import math
M = 16; N = 32
mesh = RectangleMesh(Point(1,0),
                     Point(2,math.pi/2),
                     M,N)
x = SpatialCoordinate(mesh)
kappa = Constant(1)
f = sin(2*x[1])
V = FunctionSpace(mesh,"CG",1)
u = TrialFunction(V)
v = TestFunction(V)
a = kappa*dot(grad(u),grad(v))*dx
L = f*v*dx
bc = DirichletBC(V,Constant(0),"on_boundary")
u = Function(V)
solve(a==L,u,bc)
from matplotlib import pyplot
plot(u)
pyplot.show()
u.rename("u","u")
File("u.pvd") << u
```

While impressively concise, many other libraries and turn-key software solutions provide similar convenience for the Poisson equation. What value does FEniCS add? This is impossible to convey fully without introducing more

[2] In fact, the method outlined here for our model problem is exactly equivalent to using central differences on a uniform structured mesh. Basic finite volume methods can also be recovered as special cases of the discontinuous Galerkin [10] family of finite element methods.

complicated PDE systems, but the impulse to cover more complicated problems is incompatible with the constraint of brevity imposed on the present document, especially when assuming no prior experience with those problems. As such, I shall settle for some interesting transformations of our model problem that are suggestive of extrapolations to multiphysics and/or optimization scenarios.

First, readers with more mathematical background may already know that (17) can be interpreted as energy minimization. It is equivalent to saying that the directional derivative of the functional

$$E(w) := \frac{1}{2}a(w,w) - L(w) \tag{25}$$

in every direction v is zero at u, i.e., that u is a stationary point of E:

$$\frac{d}{d\epsilon}E(u+\epsilon v)\bigg|_{\epsilon=0} = a(u,v) - L(v) = 0 \quad \forall v \in V\,. \tag{26}$$

Many practical problems, like hyperlastic solid mechanics, derive from quite complicated energy functionals, which must traditionally be differentiated by hand (with great effort and many opportunities for error) to formulate numerical methods. However, UFL includes automatic symbolic differentiation, such that we can solve the problem as follows:

```
u = Function(V)
E = (0.5*kappa*dot(grad(u),grad(u)) - f*u)*dx
v = TestFunction(V)
R = derivative(E,u,v)
solve(R==0,u,bcs=[bc,])
```

The form `R` is a *symbolic* representation of the derivative of `E` in the direction of an arbitrary `TestFunction`. A residual *vector* (corresponding to $\mathbf{R} := \mathbf{A}\mathbf{u} - \mathbf{f}$ in our notation from before) can be obtained by plugging in each basis function for `V` as the test function `v`. The DOLFIN function to do this is `assemble()`, but, in the above script, that step is hidden behind the `solve()` function, rather than performed explicitly. Although, in this case, `R` is linear in `u`, the above workflow extends directly to nonlinear problems, like hyperelasticity, which may be defined as stationary points of more complicated energy functionals. The `solve` function defaults to using a Newton iteration; it takes a second symbolic derivative of `R` to obtain a Jacobian form, which can in turn be `assembled` into a matrix. (For a linear problem, it converges in a single iteration.)

Remark 4. In engineering pedagogy, energy minimization is frequently taken as a point of departure for introducing FEA in solid mechanics. This has led to widespread misconceptions that the existence of an energy functional is necessary to formulate FEA, that it is sufficient to get good results, and/or that FEA is inapplicable or deficient for problems that are not derived from energy minimization.

Because UFL is embedded within Python, users can extend it to include new operations. To demonstrate this, let us reinterpret the rectangular domain of the preceding examples as a rectangle in circular polar coordinates, corresponding to a section of a circular annulus in physical space. Thus, the horizontal direction on our mesh is radius r, the vertical direction is angle θ, and the x and y coordinates within our domain are

$$x = r\cos\theta \quad , \quad y = r\sin\theta\,, \tag{27}$$

or, in a vectorial notation,

$$\boldsymbol{\xi} = \begin{pmatrix} r \\ \theta \end{pmatrix} \quad , \quad \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\theta \\ r\sin\theta \end{pmatrix}. \tag{28}$$

We can translate the mapping from $\boldsymbol{\xi}$ to $\mathbf{x}$ into UFL directly:

```
xi = SpatialCoordinate(mesh)
r = xi[0]
theta = xi[1]
x = as_vector([r*cos(theta),
               r*sin(theta)])
```

The UFL `grad()` function is unaware of our reinterpretation of the mesh coordinates, and corresponds to $\partial/\partial\boldsymbol{\xi}$, i.e.,

$$[\nabla_{\text{UFL}} u] = \left[\frac{\partial u}{\partial \boldsymbol{\xi}}\right] = \begin{bmatrix} \partial u/\partial r \\ \partial u/\partial \theta \end{bmatrix} . \tag{29}$$

However, this does not correspond to the "∇" in our physical PDE, which is $\partial/\partial\mathbf{x}$. To resolve this difference, we can use the multivariate chain rule:

$$\nabla u = \frac{\partial u}{\partial \mathbf{x}} = \frac{\partial u}{\partial \boldsymbol{\xi}} \cdot \frac{\partial \boldsymbol{\xi}}{\partial \mathbf{x}} = \frac{\partial u}{\partial \boldsymbol{\xi}} \cdot \left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}\right)^{-1} = \nabla_{\text{UFL}} u \cdot (\nabla_{\text{UFL}} \mathbf{x})^{-1} . \tag{30}$$

We can define this as an ordinary Python function:

```
grad_xi = grad
dx_dxi = grad_xi(x)
def grad_x(f):
    df_dxi = grad_xi(f)
    dxi_dx = inv(dx_dxi)
    return dot(df_dxi, dxi_dx)
```

Similarly, the default integration measure `dx` in UFL corresponds simply to $dr\,d\theta$ (or, $d\boldsymbol{\xi}$), whereas our physical PDE is integrated with respect to $dx\,dy$ (or, $d\mathbf{x}$). However, recalling basic multivariate calculus, we can simply insert the Jacobian determinant of the mapping $\mathbf{x}(\boldsymbol{\xi})$, e.g.,

$$\int_\Omega f\, d\mathbf{x} = \int_{\mathbf{x}^{-1}(\Omega)} f\, \det\left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}\right) d\boldsymbol{\xi} , \tag{31}$$

or, in UFL (continuing with definitions from above),

```
dxi = dx
int_f_dx = f*det(dx_dxi)*dxi
```

What we've done here extends far beyond just polar coordinates, though. Beyond the definition of `xi`, the rest of the code applies to *any* deformation of the domain. In particular, assume that some vector-valued function $\hat{\mathbf{u}}(\boldsymbol{\xi})$ deforms each node of our mesh, with linear interpolation over elements:

```
V_vec = VectorFunctionSpace(mesh,"CG",1)
u_hat = Function(V_vec)
x = xi + u_hat
```

The vector of degrees of freedom associated with `u_hat` can be filled in with the solution to a variational problem or nodal displacements from some other source *after* using it to symbolically define forms, e.g.,

```
E = (0.5*kappa*dot(grad_x(u),grad_x(u)) - f*u)*det(dx_dxi)*dxi
R = derivative(E,u)
```

For demonstration purposes, consider filling `u_hat` in with a projection of some analytical deformation given in symbolic form:

```
A = Constant(1e-2)
u_hat_sym = as_vector([A*sin(2*pi*xi[0])*sin(2*pi*xi[1]),
                       A*sin(2*pi*xi[0])*sin(2*pi*xi[1])])
u_hat.assign(project(u_hat_sym,V_vec))
```

We can combine the results in Paraview with an "Append Attributes" filter, then warp by the vector field `u_hat` to visually confirm that solutions for different values of the deformation amplitude `A` approximate the same solution $u(\mathbf{x})$.

Of course, one could argue that we've still gained nothing, since we could have simply deformed our mesh by directly modifying the nodal coordinates, then solved the Poisson equation using some other existing solver. However, retaining the symbolic form of the deformation and recalling UFL's differentiation capability from before, we can easily perform an operation of interest to shape optimization. We can differentiate the problem with respect to $\hat{\mathbf{u}}$:

```
dR_du_hat = derivative(R, u_hat)
```

The variable `dR_du_hat` is a *symbolic* representation of the directional derivative with respect to `u_hat`, in the direction of an arbitrary `TrialFunction` from the space `V_vec` (which is left anonymous by omitting the optional third argument to `derivative()`). We can "assemble" this to evaluate it for every basis function of the space `V_vec`, which results in a matrix of the partial derivatives of the assembled residual vector with respect to each degree of freedom of the space `V_vec` (i.e., nodal displacements):

```
dR_du_hat_mat = assemble(dR_du_hat)
```

This is the sense in which **symbolic differentiation of weak forms is equivalent to partial differentiation of their discrete residuals**, i.e., "optimize-then-discretize" and "discretize-then-otpimize" are equivalent when working within a variational FEA framework. The famous dolfin-adjoint library [14] is based on this insight, as is more recent work using FEniCS in an MDO framework for topology optimization [15]. In the context of MDO, and invoking the "component" abstraction of the OpenMDAO library [16], the nodal displacements might be the output of some other component that defines them in terms of a reduced set of geometric design variables, either through an explicit formula or by solving an implicit problem (e.g., an elliptic PDE to smoothly interpolate between given boundary deformations).

4 Conclusion

In summary, I have introduced the concept of casting PDEs in weak form, from which finite element discretizations follow *automatically*, as realized by FEniCS. This is arguably the most natural way to discretize PDEs for the purpose of MDO, because it is universaly applicable to any discipline modeled by a PDE system and it dissolves the distinction between "optimize-then-discretize" and "discretize-then-optimize", providing simultaneous access to the advantages of both approaches. I demonstrated how to leverage the latter property here, in the context of shape derivatives. For reasons of brevity and accessibility to general engineering audiences, we needed to sacrifice some mathematical rigour and limit ourselves to a single PDE. UC San Diego students interested in greater mathematical depth and more diverse physical applications are welcome to take or audit my graduate class, *Finite Element Analysis for Coupled Problems*. Materials for the course are also available to the general public [17].

References

[1] [ULI Workshop 2022] FEniCS. `https://www.youtube.com/watch?v=wou9jlplraw`, Accessed February 2022.

[2] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[3] FEniCS Discourse Group. `https://fenicsproject.discourse.group`, Accessed February 2022.

[4] FEniCS Documentation. `https://fenicsproject.org/documentation/`, Accessed February 2022.

[5] FEniCS examples for the 2022 ULI software workshop. `https://github.com/david-kamensky/uli-workshop-examples-2022`, Accessed February 2022.

[6] Why is the Laplacian ubiquitous? `https://mathoverflow.net/questions/54986/why-is-the-laplacian-ubiquitous`, Accessed February 2022.

[7] R. Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*. Pearson Education, 2012.

[8] L. C. Evans. *Partial Differential Equations (Graduate Studies in Mathematics, Vol. 19)*. American Mathematical Society, Providence, Rhode Island, 2002.

[9] E. Kreyszig. *Introductory Functional Analysis with Applications*. Wiley Classics Library. Wiley, 1989.

[10] B. Cockburn, G. E. Karniadakis, and C.-W. Shu. *Discontinuous Galerkin Methods: Theory, Computation and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[11] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, March 2014.

[12] R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006.

[13] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):20:1–20:28, April 2010.

[14] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, 2013.

[15] J. Yan, R. Xiang, D. Kamensky, M. T. Tolley, and J. T. Hwang. Topology optimization with automated derivative computation for multidisciplinary design problems. *Structural and Multidisciplinary Optimization*, 2022. Accepted.

[16] J. S. Gray, J. T. Hwang, J. R. R. A. Martins, K. T. Moore, and B. A. Naylor. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019.

[17] Finite Element Analysis for Coupled Problems. `https://david-kamensky.eng.ucsd.edu/teaching/mae-207-fea-for-coupled-problems`, Accessed February 2022.