Developing a compiler to understand finite element data

Teodoro Fields Collin, John Reppy, Gordon Kindlmann, **Charisee Chiw**[*]
Dept of Computer Science, University of Chicago.

We have developed a domain-specific compiler to visualize data for the FEniCS community. In another submission we demonstrate the results of this work; visualizations of finite element data. That submission will demonstrate images that have been created and present new informed visualization techniques that can been created using the language. In this abstract submission we answer the question of "how". We describe the development of our language and how we can support the type of high-level analysis necessary to visualize finite element data.

In order to enable interesting visualizations of FEM field, it was necessary to create a compiler that understood FEM fields. Our Diderot compiler is designed to represent FEM fields internally, evaluate these fields at points, and supports a rich level of tensor operators on them. By pushing domain knowledge inside our compiler we are able to take derivatives which supports a richer class of visualization algorithms. Lastly, by developing a domain-specific compiler we can find common computations involved with field evaluation that were obscured by our previous method.

Implementation The work of solving a PDE is complicated and accomplished by other established tools and languages The Diderot language is not designed to solve PDEs but to enable visualizations of the PDE solutions. Visualization features, such as crest lines or surface curvature, can involve applying multiple derivatives to the solution. To be able to correctly sample and differentiate the FE field the compiler needs to understand the underlying data structure.

The PDE result or FE field is treated as an abstract field in the Diderot syntax. In the program the the user can apply computations to the field and probe it at a position. The compiler generates code that will evaluate, differentiate, and apply computations to the FE field Internally, the compiler needs to get certain information from FE field, such as basis functions, and coordinates. By understanding the FE field domain we can enable domain-specific optimization, prioritize when to gather certain information, and create smart domain-specific constructors.

Interface discussion Designing the right interface between two separate systems provides many potential challenges. One has to balance the ability to leverage what each system does best and the need for an accessible interface. If we are applying computations on large datasets it can be crucial to generate efficient code and consider how to control memory management. If the user base is wide and diverse it may be more important to have an intuitive user-friendly interface. Lastly, there may be aspects of language interoperability that need to be addressed to ensure correctness along with efficiency.

Sensible interface design is a common issue for FEM systems, as they generally also interface with other systems. For example, Dolfin [6] interfaces with PETSc [2], mesh generators, and ParaView [1].

We considered multiple approaches to connect Diderot to FEM. Our first approach to connect Diderot and FEM relied on an outside source to evaluate a *FE* field [5]. FEniCS provided call-back functions to do point-wise evaluation of *FE* field for a handful of computations on fields. This initial approach could correctly sample a field over space, but it was limited. There was no way to apply the second or third derivative to a field, values that are needed to compute visualization features such as surface curvature or crest lines. As a result, the sophisticated visualization programs Diderot was designed to support could not be supported on *FE* field.

There are various other approaches we considered. For one we could have transformed all the solutions to a particular type of polynomial, similar to how a higher-order solution is projected down to a linear solution. However, with this approach we are changing the data, losing information about the solution, and visualizing a convoluted version of the solution. *Diderot* probes the original solution and allows a fine-grain innate view of the solution and it's properties. Another approach would be to resample the data as a grid. By dumping the data to a field we lose the mesh geometry and won't be able to compute derivatives. This type of obscurity can be a limitation for future work. For instance, if we want to develop syntax to exploit mesh geometry.

Diderot is able to translate symbolic high level math into machine code. To take advantage of what Diderot does well it is necessary to make the field structure visible to Diderot. We made a design choice

[*]Corresponding Author, cchiw@uchicago.edu

to provide Diderot sufficient knowledge about the mesh geometry and preserve the original *FE* field. Our approach allows us to see the solution as it is. *Diderot* allows us to take field derivatives and apply optimizations that were otherwise obscured.

In the future, we could develop a framework to enable the easy programmability between *Diderot* and other tools. By doing so, we can make visualization programs more accessible to non-visualization experts and replace the work of rewriting and reworking code. One method would be to offer baked-in recipes for a collection of programs. This can be accomplished by providing a library, each with a method that links to a Diderot program that can be initiated with *FE* field data. Unfortunately, as shown in previous work [5] it can be easy to initiate sophisticated visualization programs incorrectly. Another method is to allow the user to specify a visualization algorithm and the geometric feature with a textual description.

Domain-specific optimization The Diderot compiler performs a number of previously discussed optimizations. The source compiler uses dead-code elimination and value numbering to remove redundant computations [4]. To address compilation issues the source compiler uses various EIN-based methods to reduce the size of the IR [3]. By learning the finite element domain, the *Diderot* compiler can apply domain-specific optimizations that were otherwise impossible or unlikely.

If we use the *Diderot* syntax to describe the *FE* field then the compiler can be knowledgeable about the mesh and function space used. The compiler can refer to a stashed data file or an internal representation of the data for the necessary values. The relevant information can be gathered at compile time rather than execution time.

By creating abstract operators to represent the expanded probe then the redundant computations can be detected and eliminated. For example, it is typical for a program to check if a position is inside a field before probing it. An inside test and probe field term creates some of the same domain-specific operators. To implement both we find the right reference cell and translate the position. By splitting the field operators into steps we are able to catch the redundant computations.

Similarly our informative operators can reduce the size of the generated code. The compiler creates functions in the generated code that are dependent on the mesh or the function space or a specific solution. For example, if a program uses multiple fields defined by the same mesh, but different solution data then the mesh-dependent functions are only generated once. While there is still optimizations and reductions left to do in the generated code we believe this is a step in the right direction.

References

[1] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. In *Visualization Handbook*, pages 717–731. Academic Press, Inc., Orlando, FL, USA, 2005.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2017.

[3] C. Chiw, G. L. Kindlman, and J. Reppy. EIN: An intermediate representation for compiling tensor calculus. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing*, July 2016.

[4] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 111–120, June 2012. (alphabetical author order).

[5] C. Chiw, G. L. Kindlmann, and J. H. Reppy. An exploration to visualize finite element data with a DSL. *CoRR*, abs/1706.05718, 2017.

[6] DOLFIN. *DOLFIN*. Available from https://bitbucket.org/fenics-project/dolfin.