Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 308 (2016) 151–181
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

PetIGA: A framework for high-performance isogeometric analysis

L. Dalcin[a,b,c,d,*], N. Collier[e], P. Vignal[f], A.M.A. Côrtes[b], V.M. Calo[b]

[a] *Extreme Computing Research Center (ECRC), King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia*
[b] *Center for Numerical Porous Media (NumPor) Computer, Electrical and Mathematical Sciences & Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia*
[c] *Centro de Investigación de Métodos Computacionales (CIMEC), Santa Fe, Argentina*
[d] *Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Santa Fe, Argentina*
[e] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[f] *Center for Numerical Porous Media (NumPor), Materials Science & Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia*

Received 26 July 2015; received in revised form 6 May 2016; accepted 8 May 2016
Available online 25 May 2016

Abstract

We present PetIGA, a code framework to approximate the solution of partial differential equations using isogeometric analysis. PetIGA can be used to assemble matrices and vectors which come from a Galerkin weak form, discretized with Non-Uniform Rational B-spline basis functions. We base our framework on PETSc, a high-performance library for the scalable solution of partial differential equations, which simplifies the development of large-scale scientific codes, provides a rich environment for prototyping, and separates parallelism from algorithm choice. We describe the implementation of PetIGA, and exemplify its use by solving a model nonlinear problem. To illustrate the robustness and flexibility of PetIGA, we solve some challenging nonlinear partial differential equations that include problems in both solid and fluid mechanics. We show strong scaling results on up to 4096 cores, which confirm the suitability of PetIGA for large scale simulations.

Keywords: Isogeometric analysis; High-performance computing; Finite element method; Open-source software

1. Introduction

Isogeometric analysis, a finite element method originally proposed in 2005 [1,2], was originally motivated by the desire to find a technique for solving partial differential equations which would simplify or remove the problem of converting geometric descriptions for discretizations in the engineering design process. Once a design is born inside a Computer Aided Design (CAD) program, converting the CAD representation to an analysis-suitable form usually is a bottleneck of the engineering analysis process. Isogeometric methods aim to use CAD representations directly

* Correspondence to: 4700 King Abdullah University of Science and Technology al-Khawarizmi Bldg (Bldg 1), Office # 0109 Thuwal 23955-6900, Kingdom of Saudi Arabia.
E-mail addresses: dalcinl@gmail.com (L. Dalcin), nathaniel.collier@gmail.com (N. Collier), philippe.vignal@kaust.edu.sa (P. Vignal), adrimacortes@gmail.com (A.M.A. Côrtes), vmcalo@gmail.com (V.M. Calo).

http://dx.doi.org/10.1016/j.cma.2016.05.011
0045-7825/© 2016 Elsevier B.V. All rights reserved.

by using the Non-Uniform Rational B-spline (NURBS) basis, circumventing the need to generate an intermediate geometrical description. The term *isogeometric* reflects that as the finite element space is refined, the geometrical representation can be preserved exactly. NURBS technologies have been used in CAD for decades due to their properties, particularly the smoothness and ability to represent conic sections. The key insight of isogeometric analysis is to use the geometrical map of the NURBS representation as a basis for the push forward used in analysis. This allows isogeometric modeling to advance where predecessors have found limitations [3–6].

In addition to the geometrical benefits, the basis is also well-suited to solving higher-order partial differential equations, such as the ones related to phase-field problems [7–9] or large deformation shell formulations [10–12]. Classical finite element spaces use basis functions which are C^0 continuous across element boundaries, making them unsuitable for higher-order problems using a primal Galerkin formulation. The NURBS-based spaces may be constructed to possess arbitrary degrees of inter-element continuity for any spatial dimension. These higher-order continuous basis functions have been numerically [13–16] and theoretically [17,18] observed to possess superior approximability per degree of freedom when compared to their C^0 counterparts. However, when used to discretize a Galerkin weak form, the higher-order continuous basis functions have also been shown to result in linear systems which are more expensive to solve with multifrontal direct solvers [19,20] and iterative solvers [21]. These results motivate the development of efficient, scalable software frameworks which can mitigate the increase of cost.

This paper describes a scalable implementation of tensor-product, NURBS-based isogeometric analysis. Despite the fact that writing every piece of code from scratch is still a common practice in research groups, we believe that reusable software should become the norm in the scientific community. Otherwise, years of accumulated expertise in the field are disregarded [22,23], which is inconsistent with the open and incremental nature of scientific discovery. However, the choice to depend on existent software should not be made lightly. Software libraries require maintenance and development to adapt to changing software and hardware requirements. This means that they require both personnel and funding to continue to exist. Furthermore, interfacing with other libraries can require development work on the library side. Without willing developers to support and assist third parties in using their libraries, the process can become cumbersome. We believe the benefits outweigh the risks, yet these are factors to consider when one plans to reuse the work of others.

PETSc [24–26], the *Portable Extensible Toolkit for Scientific Computation*, is a collection of algorithms and data structures for the solution of scientific problems, particularly those modeled by partial differential equations. PETSc is applicable to a wide range of problem sizes, including extreme large-scale simulations, where high-performance parallel computation is a must. PETSc uses the message-passing interface (MPI) model for communication, but provides high-level interfaces with collective semantics so that typical users rarely have to make message-passing calls directly.

PETSc provides a rich environment for modeling scientific problems as well as for rapid algorithm design and prototyping. The library enables easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility. PETSc is object-oriented in style, with components that may be changed via a command-line interface at runtime. These components include:

- Index sets to describe permutations, indexing, renumbering, and communication patterns;
- Matrices and vectors that provide basic linear algebra abstractions;
- Krylov subspace methods and preconditioners that include extreme-scale multigrid [27] and state of the art domain decomposition methods [28] with an emphasis on isogeometric analysis [29];
- Nonlinear solvers and time stepping algorithms; and
- Distributed arrays for parallelizing structured grid-based problems.

PETSc is also designed to be highly modular, enabling the interoperability with specialized parallel libraries like Hypre [30], Trilinos/ML [31], MUMPS [32], and others through a unified interface. Other scientific packages geared towards solving partial differential equations use components from PETSc (for example deal.II [33], FEniCS [34], libMesh [35], and PETSc-FEM [36]).

In this paper we describe an approach to reuse PETSc algorithms and data structures to obtain a high-performance framework designed for isogeometric analysis. We implement parallel vector and matrix assembly using PETSc data structures and interface into PETSc's wide range of solvers. We call our framework PetIGA. It is freely available [37] and under active development.

PetIGA contributes to a growing collection of software enabling research in isogeometric analysis. *GeoPDEs* [38] is an easy to use isogeometric toolkit written in Octave (and fully compatible with MATLAB). *igatools* [39] and

G+Smo [40] are recently developed libraries for isogeometric analysis, which use advanced C++ metaprograming techniques. *SfePy* [41] is a Python framework for solving systems of coupled partial differential equations with the finite element method and includes capabilities for isogeometric analysis [42]. *MFEM* [43] is a lightweight and scalable C++ library for high-order finite element methods with adaptive mesh refinement and also includes capabilities for isogeometric analysis.

The remainder of this paper is organized as follows. In Section 2, we detail the implementation as well as the features of the framework. In Section 3, we tackle a model problem for nonlinear applications, and go through all the steps to solve it using our framework. We showcase applications in Section 4 and discuss performance results in Section 5. Finally, we provide some concluding remarks in Section 6.

2. Implementation

PetIGA follows closely the design philosophies that have modeled PETSc over the last 20 years. Our framework is implemented in the C programming language with a few computational kernels written in Fortran. PetIGA presents to users an object oriented interface that promotes encapsulation through the use of opaque data structures. Most of the PetIGA interface revolves around the management of `IGA` contexts. These contexts are just handles (or pointers) to opaque structures which refer to internal implementation details as well as user-defined data and routines specific to the problem to be solved. Users manipulate `IGA` contexts through various routines in charge of defining options and parameters, performing computations, and creating subordinate solver contexts from PETSc.

The full application programming interface (API) of PetIGA is too large to be presented here. We refer to Section 3 and Appendix B for a taste of the API through an application example. This section describes many algorithmic choices we made that guided the development of our framework. We also discuss technical aspects which are crucial to achieve high-performance in distributed-memory parallel platforms.

2.1. *B-spline basis functions*

Let $\Xi = \{\xi_0, \ldots, \xi_m\}$ be a non-decreasing sequence of real numbers, i.e., $\xi_i \le \xi_{i+1}, i = 0, \ldots, m-1$. The ξ_i are called *knots* and Ξ is the *knot vector*. By using the Cox–de Boor recursion formula [44,45], the ith B-spline basis function of p-degree, $i = 0, \ldots, m-p-1$, denoted $B_{i,p}(\xi)$, is defined as

$$B_{i,0}(\xi) = \begin{cases} 1 & \text{if } \xi_i \le \xi < \xi_{i+1}, \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

$$B_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} B_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} B_{i+1,p-1}(\xi). \tag{2}$$

Writing $B_{i,p}$ instead of $B_{i,p}(\xi)$ for brevity, the first derivative of a basis function is given by

$$B'_{i,p}(\xi) = \frac{p}{\xi_{i+p} - \xi_i} B_{i,p-1} + \frac{p}{\xi_{i+p+1} - \xi_{i+1}} B_{i+1,p-1}. \tag{3}$$

Repeated differentiation of the previous expression produces the general formula for the kth derivative $B^{(k)}_{i,p}(\xi)$ of $B_{i,p}(\xi)$

$$B^{(k)}_{i,p}(\xi) = p \left(\frac{B^{(k-1)}_{i,p-1}}{\xi_{i+p} - \xi_i} + \frac{B^{(k-1)}_{i+1,p-1}}{\xi_{i+p+1} - \xi_{i+1}} \right). \tag{4}$$

Even though this is the standard way of expressing the basis functions, we compute their values and derivatives using the more computationally efficient algorithms detailed in [46].

2.2. *Tensor product basis functions*

By using a tensor product structure, a one-dimensional basis can be extended to multi-dimensions. Here we write a three-dimensional extension of the one-dimensional B-spline basis. Let

$$\Xi = \{\xi_0, \ldots, \xi_{n+p+1}\}, \qquad \mathrm{H} = \{\eta_0, \ldots, \eta_{m+q+1}\}, \qquad \mathrm{Z} = \{\zeta_0, \ldots, \zeta_{o+r+1}\}$$

be knot vectors which define three sets of one-dimensional B-spline basis functions

$$\{B_{i,p}(\xi), i = 0, \dots, n\}, \qquad \{B_{j,q}(\eta), j = 0, \dots, m\}, \qquad \{B_{k,r}(\zeta), k = 0, \dots, o\}$$

of degree p, q, and r, respectively. The three-dimensional B-spline basis functions are defined as

$$M_{ijk}(\xi, \eta, \zeta) = B_{i,p}(\xi) B_{j,q}(\eta) B_{k,r}(\zeta). \tag{5}$$

For the sake of notational convenience and dimension independence, we denote multi-dimensional B-spline basis functions as $M_A(\boldsymbol{\xi})$ in the following. For the particular case of three dimensions, $\boldsymbol{\xi}$ denotes the parametric coordinate (ξ, η, ζ) and A is the global basis function index, i.e., $A = i + j(n+1) + k(n+1)(m+1)$.

2.3. *NURBS basis functions*

Given the B-spline basis functions $M_A(\boldsymbol{\xi})$, we can define the corresponding NURBS [46] basis functions as

$$N_A(\boldsymbol{\xi}) = \frac{w_A M_A(\boldsymbol{\xi})}{w(\boldsymbol{\xi})} \tag{6}$$

where w_A are the projective weights and the weighting function appearing in the denominator is

$$w(\boldsymbol{\xi}) = \sum_B w_B M_B(\boldsymbol{\xi}). \tag{7}$$

The derivation of NURBS basis function derivatives in parametric and spatial coordinates can be found in Appendix A.

2.4. *Periodic boundary conditions*

Due to the prevalent use of open knot vectors by the IGA community, applications which require the enforcement of periodic boundary conditions typically do so by building a system of constraints on the coefficients. For example, in [47], the authors detail constraint equations which enforce C^1 periodicity. However, we prefer to build periodicity into the function space due to its simplicity and generality. We do this by *unclamping* the knot vector as in the parlance of [46]. Let $\Xi = \{\xi_0, \dots, \xi_m\}$ be an open (or *clamped*) knot vector which encodes a set of p-degree B-spline basis functions $\{B_{i,p}(\xi), i = 0, \dots, n\}$, where $m = n + p + 1$. Unclamping the knot vector for a desired continuity C^k at the boundary, $0 \le k \le p - 1$, amounts to redefining $k + 1$ knot values at the left and right ends.

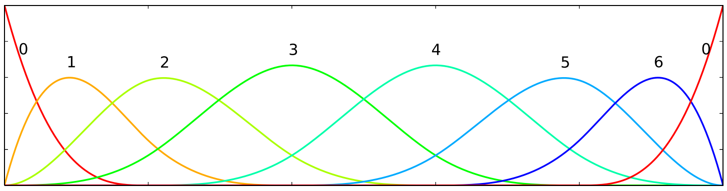
$$\xi_{k-i} = \xi_p - \xi_{n+1} + \xi_{n-i}, \qquad 0 \le i \le k, \tag{8}$$

$$\xi_{m-k+i} = \xi_{n+1} - \xi_p + \xi_{p+i+1}, \qquad 0 \le i \le k. \tag{9}$$

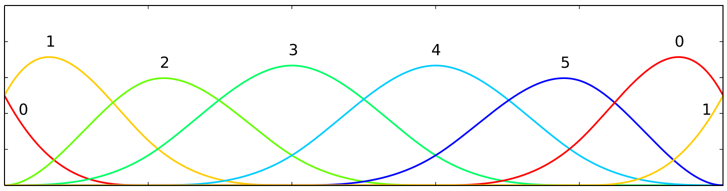
In Fig. 1 we present a C^2 cubic B-spline space with varying orders of continuity across the periodic boundary. Each of these knot vectors was obtained by unclamping the open knot vector $\Xi = \{0, 0, 0, 0, 0.2, 0.4, 0.6, 0.8, 1, 1, 1, 1\}$ using Eqs. (8) and (9). Each unique basis function is labeled with its global number and colored distinctly such that basis functions which pass the periodic interface may be identified.

Eqs. (8) and (9) are sufficient when utilizing the basis in the parametric domain. In cases where the parametric domain is mapped, the original control points of the mapping which correspond to the open knot vector also need to be unclamped. In [46, p. 577], algorithm A12.1 performs this operation but is limited to C^{p-1} unclamping. We present a generalization in Algorithm 1 for C^k unclamping where `U` is the array of knots and `Pw` is the array of control points in homogeneous coordinates. In Fig. 2, we present the effect of our algorithm when applied to a quarter circular arc.

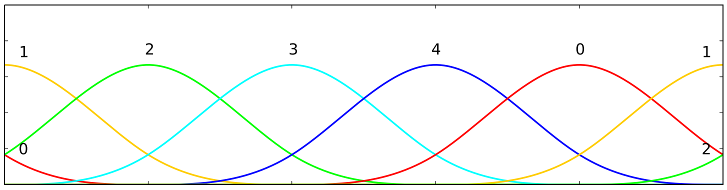
After unclamping a knot vector, imposing periodic boundary conditions no longer requires the use of constraint equations. Periodicity can now be embedded into the function space by eliminating redundant basis functions in one of the domain boundaries. A practical way of dealing with this approach in a computer code is to properly map indices of basis functions that were eliminated to the corresponding ones that remain. In the non-periodic case, the number of basis functions is $n + 1$. Imposing periodic boundary conditions with continuity order k reduces the number of basis



(a) C^0 periodicity, $\Xi = \{-0.2, 0, 0, 0, 0.2, 0.4, 0.6, 0.8, 1, 1, 1, 1.2\}$.



(b) C^1 periodicity, $\Xi = \{-0.4, -0.2, 0, 0, 0.2, 0.4, 0.6, 0.8, 1, 1, 1.2, 1.4\}$.



(c) C^2 periodicity, $\Xi = \{-0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6\}$.

Fig. 1. Three cases of periodicity for a C^2 cubic B-spline space. In each case, the open knot vector was unclamped using Eqs. (8) and (9). Unique basis functions are labeled by their global numbering and colored distinctly. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

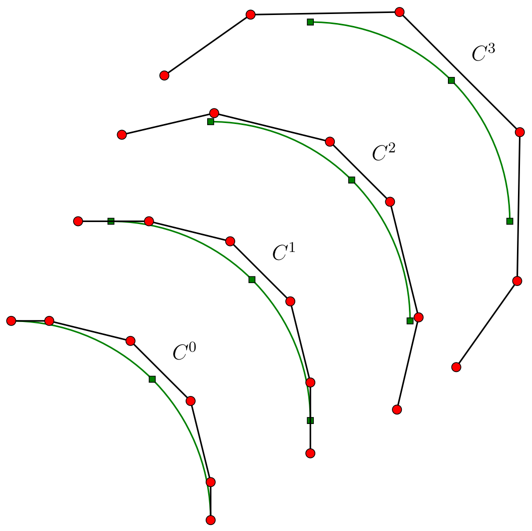


Fig. 2. A quarter circular arc of degree four repeatedly unclamped by Algorithm 1.

functions to $n - k$. Any basis function index i outside the interval $[0, n - k - 1]$ can be wrapped-around using the mapping $i \mapsto \text{mod}\,(i, n - k)$, where

$$\text{mod}\,(a, b) = a - b \left\lfloor \frac{a}{b} \right\rfloor \tag{10}$$

and the symbol $\lfloor \cdot \rfloor$ denotes the floor function, i.e., $\lfloor x \rfloor$ is the largest integer not greater than x.

Algorithm 1 Pseudocode for unclamping curves.

```
UnclampCurve(n,p,k,U,Pw) {
  /* Input:
      n - index of last basis function
      p - polynomial degree
      k - continuity order
      U, Pw - knot vector and control points
    Output:
      U, Pw (modified in-place) */
  m = n + p + 1; /* index of last knot */
  for (i=0; i<=k; i++) /* Unclamp at left end */
    U[k-i] = U[p] - U[n+1] + U[n-i];
  for (i=p-k-1; i<=p-2; i++)
    for (j=i; j>=0; j--) {
      alpha = (U[p]-U[p+j-i-1])/(U[p+j+1]-U[p+j-i-1]);
      Pw[j] = (Pw[j]-alpha*Pw[j+1])/(1-alpha);
    }
  for (i=0; i<=k; i++) /* Unclamp at right end */
    U[m-k+i] = U[n+1] - U[p] + U[p+i+1];
  for (i=p-k-1; i<=p-2; i++)
    for (j=i; j>=0; j--) {
      alpha = (U[n+1]-U[n-j])/(U[n-j+i+2]-U[n-j]);
      Pw[n-j] = (Pw[n-j]-(1-alpha)*Pw[n-j-1])/alpha;
    }
}
```

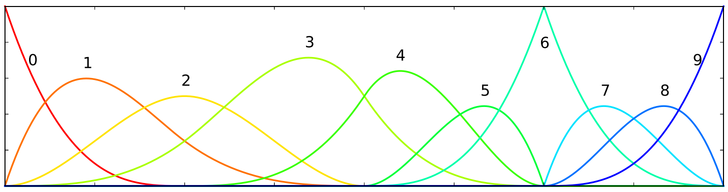
2.5. *Adjacency graph*

For numerical methods employing basis functions with local support, it is important to compute the adjacency graph of interacting degrees of freedom. This graph contains information required for the proper preallocation of sparse matrices implemented in compressed sparse row (CSR) or column (CSC) formats. This preallocation is crucial for efficient matrix assembly. Furthermore, prior knowledge of the sparse matrix nonzero pattern enables the use of specialized differentiation techniques, such as the approximation of Jacobian matrices by colored finite differences, see Section 2.8.

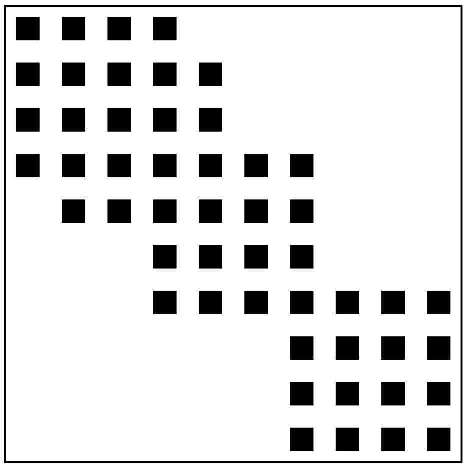
Given an array of knots U encoding a set of B-spline basis functions of p-degree, Algorithm 2 computes the left-most (ℓ) and right-most (r) indices of the basis functions interacting with the ith basis function. That is, the support of the basis function $N_{i,p}$ has non-empty intersection with the support of $N_{j,p}$, for $\ell \le j \le r$. In one dimension, all basis functions with indices in the set $\{\ell, \ell+1, \ldots, r-1, r\}$ are adjacent to the ith basis function. For dimensions higher than one, the adjacency graph is computed from the index sets $\{\ell_d, \ell_d+1, \ldots, r_d-1, r_d\}$ for each d-dimension using the tensor-product structure and numbering in lexicographical order. When imposing periodic boundary conditions with continuity order k, Algorithm 2 may return indices outside the interval $[0, n-k-1]$. Again, the index mapping given in (10) has to be used to wrap-around index values.

Algorithm 2 Pseudocode for computing the adjacency graph.

```
BasisStencil(i,p,U,l,r) {
  /* Input:
      i - index of basis
      p - polynomial degree
      U - knot vector
     Output: l,r */
      l - index of left-most overlapping basis function
      r - index of right-most overlapping basis function */
  j = i;
  while (U[j] == U[j+1]) j++;
  l = j - p;
  j = i + p + 1;
  while (U[j] == U[j-1]) j--;
  r = j - 1;
}
```



(a) A sample representative cubic B-spline basis consisting of four elements where the inter-element continuity progressively decreases according to the knot vector {0, 0, 0, 0, 2, 4, 4, 6, 6, 6, 8, 8, 8, 8}.



(b) Nonzero structure where the function space represented above is used as test and trial functions in a Galerkin finite element method. Nonzero entries are indicated by a square.

Fig. 3. Generation of the nonzero structure of matrices using Algorithm 2.

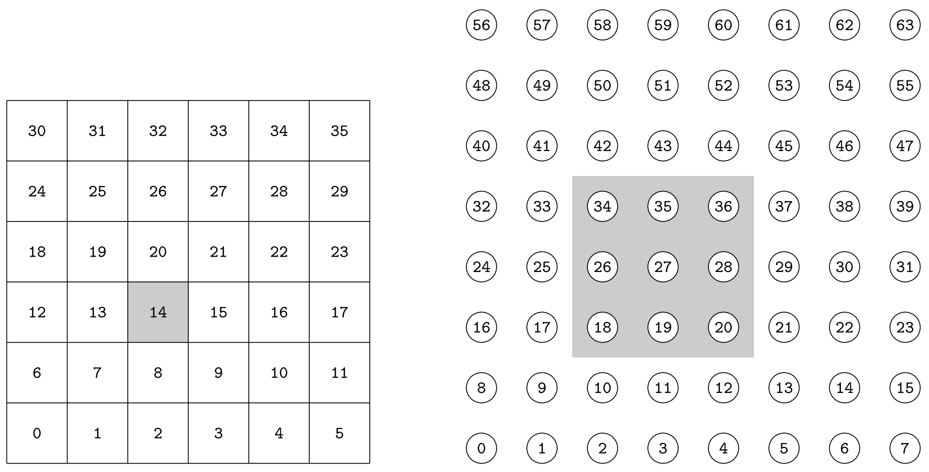
Fig. 3(a) presents a set of cubic basis functions which varies in inter-element continuity. Each basis function is labeled by a global number and colored distinctly. Fig. 3(b) shows the corresponding nonzero pattern for the sparse matrix obtained by applying Algorithm 2 to each basis function.

2.6. *Partitioning*

In this section we describe the parallel partitioning and inter-process communication PetIGA uses. The ideas and techniques presented here closely resemble the ones used in PETSc to handle structured problems (through the `DMDA` component) as well as unstructured problems (through the `DMPlex` component), see [25]. However, PetIGA implements its own data structures tailored to the specifics of isogeometric analysis, particularly the use of higher-continuous, tensor-product polynomial spaces with possibly arbitrary continuity orders across discrete domains. In the following, *element* refers to a non-empty knot span (as defined in [2]) and *node* refers to an abstract entity that can encompass control points and weights of a NURBS geometry, control variables of either a scalar or vector field, unknown coefficients, and/or residual equation values.

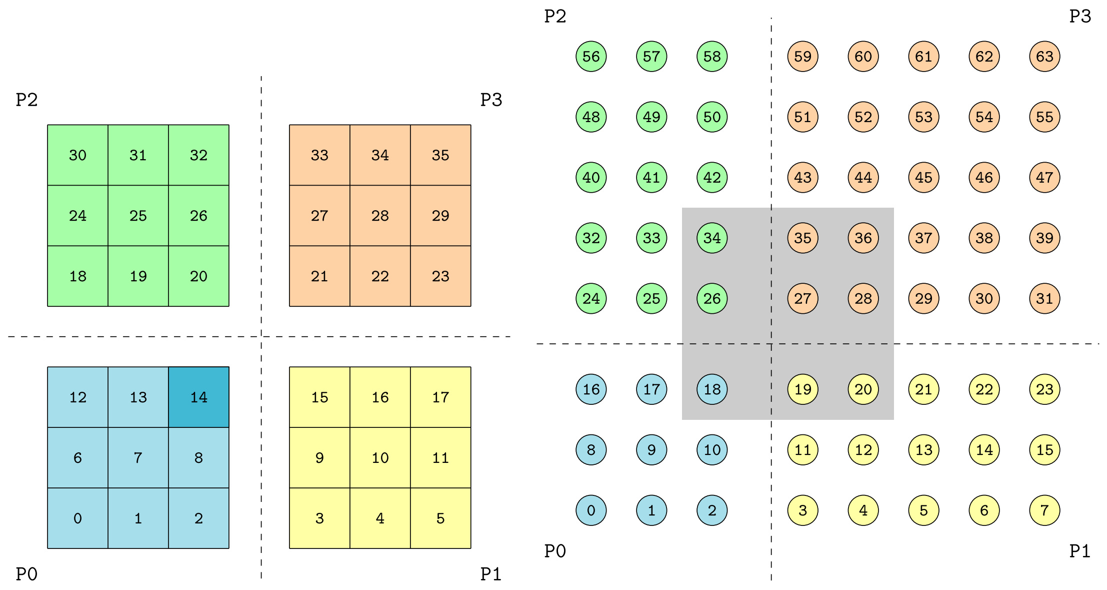
Fig. 4 depicts elements and nodes corresponding to a two-dimensional quadratic C^1 discrete space with 6×6 elements and 8×8 nodes. Both elements and nodes are labeled with an index which starts at zero and follows a natural numbering. In Fig. 4(b), a subset of nodes is drawn inside a shaded region. These nodes correspond to basis functions with support on the element highlighted in Fig. 4(a).

Given a 2×2 grid of processes labeled from zero to three, Fig. 5 depicts a partition of the sets of elements and nodes. Fig. 5(a) highlights a single element (drawn with a darker color) while Fig. 5(b) highlights its corresponding subset of nodes (drawn inside a shaded region). The highlighted element is assigned to process zero whereas the nodes inside



(a) Element grid. (b) Node grid.

Fig. 4. A two-dimensional quadratic C^1 space with 6×6 elements and 8×8 nodes. The shaded region in Fig. 4(b) contains the nodes which correspond to basis functions with support on the element highlighted in Fig. 4(a).

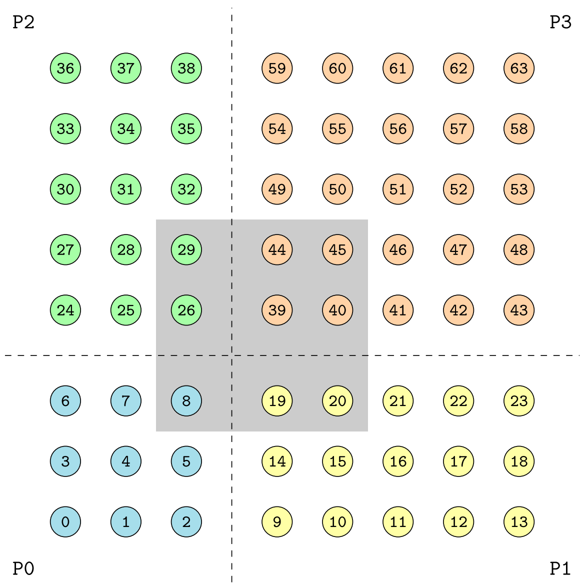


(a) Element partition. (b) Node partition and natural numbering.

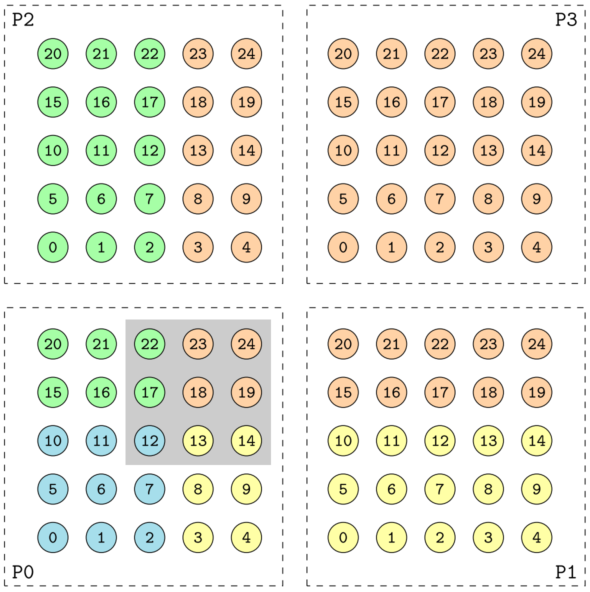
Fig. 5. The element and node grids in Fig. 4 are partitioned and distributed over a set of four processes {P0, P1, P2, P3} arranged in a 2×2 process grid. Elements and nodes are labeled according to the natural numbering, and assigned a distinct color that depends on which process they belong to. The shaded region in Fig. 5(b) contains the nodes related to the element highlighted in Fig. 5(a). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

the shaded region are assigned to different processes. Up to this stage, elements and nodes are still labeled according to the natural numbering defined in Fig. 4.

Fig. 6(a) defines a new node numbering. This global numbering is block-contiguous and follows the process numbering. The *natural* numbering in Fig. 5(b) and the *global* numbering in Fig. 6(a) define a bijective *natural* $\mapsto$ *global* mapping. The global numbering enhances locality, thus improving overall parallel performance. However, this numbering depends on the size and layout of the process grid. As the natural numbering does not depend on the partitioning, it is better suited for tasks involving data persistence, such as pre-/post-processing and checkpoint/restart. PetIGA employs the natural numbering when datafiles have to be read or written to disk, and uses the *natural* $\mapsto$ *global* mapping on the fly to convert to/from the global numbering.



(a) Global node grid and global numbering.

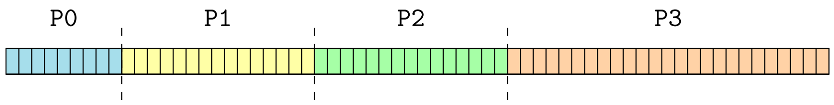


(b) Local node grids and local numberings.

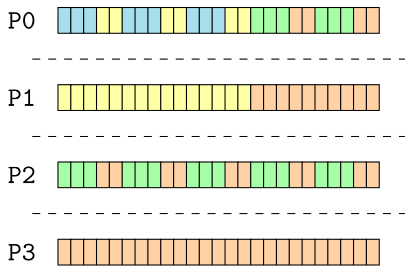
Fig. 6. Global and local node grids. Global nodes are colored according to the process they belong to, and labeled following a block-contiguous global numbering. Nodes in the local grids are labeled following a local numbering. The local grids in processes P0, P1, and P2 contain both strictly-local and ghost nodes, while the local grid in process P3 only contains strictly-local nodes. The shaded region in Fig. 6(a) is the same as in Fig. 5(b). The nodes corresponding to the element highlighted in Fig. 5(a) are now fully contained within the local grid of process P0 as shown in Fig. 6(b). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The subset of nodes drawn inside the shaded region in Fig. 6(a) indicates that computations performed in the highlighted element in Fig. 5(a) would require communication with neighboring processes. As most distributed-memory parallel codes using the message-passing paradigm and dealing with grid-based methods, PetIGA efficiently handles inter-process communication by introducing augmented *local grids* at each process. Conceptually, a local grid is a sequential entity that belongs to a single process, while the global grid is a single entity that is distributed across processes. Within a local grid, two disjoint subsets of nodes can be distinguished: the *strictly-local nodes* which replicate the in-process subset of nodes of the global grid, and the *ghost nodes* (also known as *halo nodes*) which replicate global nodes assigned to neighboring processes. Fig. 6(b) depicts the local grids of each process corresponding to the global grid in Fig. 6(a). Nodes in the local grids are labeled with an index that starts at zero and defines the *local* numbering. Strictly-local nodes and ghost nodes are distinguished by assigning them a color that matches the process the corresponding global node is assigned to. For example, in Fig. 6(b), in the local grid of process zero, blue nodes are strictly local nodes while the rest are ghost nodes. Data transfer between the global and local grids is managed through injective *local* $\mapsto$ *global* mappings defined within each process. Data associated to strictly-local nodes is handled with in-process memory transfers, while data associated to ghost nodes has to be communicated back and forth with neighboring processes. The communication costs increase linearly with the number of ghost nodes, which in turn depend on the continuity order at inter-process interfaces: spaces of continuity C^k lead to a number of ghost nodes proportional to $k + 1$. However, quality parallel PDE solvers are usually implemented as to overlap halo exchange communication with local computation. This optimization helps alleviate the additional communication overhead associated to fat interfaces.

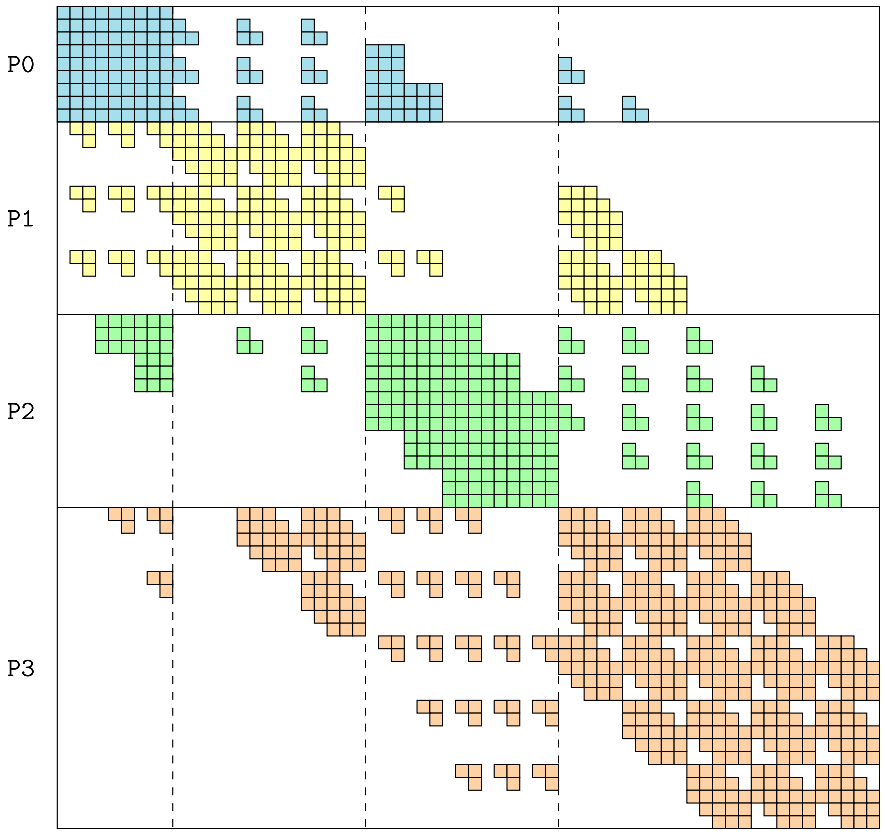
Lastly, Fig. 7 shows the layout of vectors and sparse matrices. The global vector in Fig. 7(a) is the concrete data structure used to store data associated with nodes from the global grid of Fig. 6(a). Global vectors are distributed data structures built out of local one-dimensional arrays of floating point values. Similarly, the local vectors in Fig. 7(b) are the concrete counterparts of the local grids presented in Fig. 6(b). These are nonetheless sequential data structures which are independent of each other. Global sparse matrices are distributed by rows across processes, as shown in Fig. 7(c). Thanks to the locality induced by the block-contiguous global numbering, most of the entries within a process belong to the denser diagonal blocks. This in turn improves overall performance of global matrix–vector products.



(a) Parallel layout of a global vector distributed across processes.



(b) Sequential layout of local vectors within each process.



(c) Parallel layout of a global sparse matrix distributed by rows across processes.

Fig. 7. Layout of vectors and sparse matrices.

2.7. *Assembly*

Through the pseudocode in Algorithm 3, we describe how a residual vector coming from the discretization of a nonlinear partial differential equation is assembled in parallel. We use arrow symbols to reflect movement of memory, either within or across processes, and the equality symbol to signify when computation occurs. Given a global vector $\mathbf{U}$ which constitutes the current state of the solution to the nonlinear problem, we obtain the *local vector* $\mathbf{U}_\ell$ by

Algorithm 3 Pseudocode for global residual vector assembly.

```
function FormResidualVector($\mathbf{U}$)
    $\mathbf{U}_\ell \longleftarrow \mathbf{U}$
    for each element $e$ in subpatch do
        $\mathbf{U}_e \longleftarrow \mathbf{U}_\ell$
        $\mathbf{G}_e \longleftarrow$ Geometry($e$)
        for each quadrature point $q$ in element $e$ do
            $\mathbf{x}_q, w_q \longleftarrow$ Quadrature($q$)
            $J_q, \{N_q\} =$ ShapeFuns($\mathbf{x}_q, \mathbf{G}_e$)
            $\mathbf{F}_q =$ Integrand($\mathbf{x}_q, \{N_q\}, \mathbf{U}_e$)
            $J_q w_q \mathbf{F}_q \xrightarrow{+} \mathbf{F}_e$
        end for
        $\mathbf{F}_e \xrightarrow{+} \mathbf{F}_{\text{cache}}$
    end for
    $\mathbf{F}_{\text{cache}} \xrightarrow{+} \mathbf{F}$
    return $\mathbf{F}$
end function
```

gathering off-process values through inter-process communication and packing them together with process-owned values. The steps which follow are standard practice in finite element codes. We loop over all elements e within a subpatch and gather solution and geometry coefficients $\mathbf{U}_e$ and $\mathbf{G}_e$, respectively. Then, we loop over quadrature points q within the element and compute the set of nonzero basis functions and their spatial derivatives (which we collectively denote as $\{N_q\}$) as well as the determinant of the Jacobian the geometrical mapping J_q. The main difference between applications lies in the evaluation of the integrand at a quadrature point. This problem-specific evaluation routine has to be provided by the user. We then accumulate the quadrature point contributions $\mathbf{F}_q$ into the element residual vector $\mathbf{F}_e$, taking into account the quadrature weight w_q and Jacobian determinant J_q. Each element residual $\mathbf{F}_e$ is then assembled into the global residual vector $\mathbf{F}$. In this step, PETSc automatically handles off-processor contributions by storing them in a cache which we designate as $\mathbf{F}_{\text{cache}}$. Finally, after the completion of the element loop, the cached contributions are communicated and assembled into the global vector $\mathbf{F}$. Matrices are assembled in a similar fashion. Again, the user only has to provide the problem-specific evaluation routine computing the integrand.

This approach to abstraction hides from the user the details of sparse matrix and vector assembly as well as parallelism. In turn, application codes are shorter and easier to read as they only contain code relevant to the physics of the modeled problem.

2.8. Numerical differentiation

In the process of setting up new codes to solve nonlinear problems, one of the most time-consuming tasks involves the analytical derivation and subsequent coding of Jacobians. While exploring new ideas, models, and/or formulations to a problem, the expression of the nonlinear residual may change considerably, triggering the need to update the definition of the Jacobian. Moreover, the residual may contain complicated expressions (e.g., non-trivial material constitutive relations) whose derivatives are hard to obtain. Numerical differentiation can alleviate these burdens by providing a reliable way to approximate the Jacobian at the expense of additional computational time.

PETSc has built-in facilities within its nonlinear solver implementations to approximate Jacobians through numerical differentiation using two different techniques.

- *Matrix-free Newton–Krylov*. In this approach, a sparse matrix is never assembled. Instead, matrix–vector products involving the Jacobian are approximated within the Krylov linear solver using a first-order difference formula [48]. Even though this method can considerably reduce the memory requirements as well as the computational time, the lack of an actual matrix prevents using black-box preconditioning techniques. Unless the user can provide a (matrix-free) problem-specific preconditioner, this methodology is only suitable for nonlinear problems where the Jacobian is well conditioned. Otherwise, linear solvers may fail to converge or require large iteration counts and thus, residual evaluations, leading to a drastic increase in computational time. Matrix-free techniques can be enabled at runtime through the command line option `-snes_mf`.

- *Colored finite differences*. Unlike the previous approach, in this case a sparse matrix is assembled. A first-order difference formula is also used to form columns of the Jacobian matrix. Instead of computing columns individually, the method exploits sparsity by finding a partition of the set of columns using graph coloring techniques. In this way, many columns (more specifically, those having the same color) can be computed at once with a single global residual evaluation. We refer the reader to [49] for a thorough review on the topic. This black-box technique is sensible to use as long as the number of colors is small, which is the case in many discretization methods, and particularly linear finite elements. For higher-order finite element methods such as isogeometric analysis, the number of colors increases with the support of the basis functions. Colored finite differences techniques can be enabled at runtime through the command line option `-snes_fd_color`.

Being based on global residual evaluations, the approach PETSc takes is black-box and applicable to many discretization methods as it does not require explicit grid information. However, in the context of finite element methods, it is possible to exploit locality to develop an equivalent but more efficient implementation. Rather than computing the global Jacobian by evaluating the global residual vector, evaluations can instead be performed locally at the quadrature point level.

In the following, we recall the notation in Section 2.7 and details of Algorithm 3. In the local numerical differentiation approach available in our framework, the kth column of the Jacobian matrix $\partial \mathbf{F}_q/\partial \mathbf{U}_e$ is computed at the quadrature point q within an element e using the following first-order finite difference approximation

$$\frac{\partial \mathbf{F}_q}{\partial \mathbf{U}_e}\hat{\mathbf{e}}_k \approx \frac{1}{\delta}\Big(\mathbf{F}_q(\mathbf{U}_e + \delta \hat{\mathbf{e}}_k) - \mathbf{F}_q(\mathbf{U}_e)\Big) \tag{11}$$

where $\hat{\mathbf{e}}_k$ denotes a vector with a 1 in the kth entry and zeros elsewhere, while δ is the difference step. Following the standard approach taken in [50], the difference step is computed as

$$\delta = \eta\sqrt{1 + \|\mathbf{U}_e\|} \tag{12}$$

where η should be set to the square root of the relative error in the evaluations of $\mathbf{F}_q$. As a default, this relative error is set to $\eta = \sqrt{\epsilon}$, where ϵ is the floating point machine epsilon (for double precision floating point numbers, $\epsilon \approx 2.22 \times 10^{-16}$).

In Appendix C, we present benchmark results comparing the numerical differentiation techniques mentioned in this section when applied to the three-dimensional Bratu equation. Even though explicitly coding the Jacobian is always the fastest way of computing this matrix, the local approach discussed in the previous paragraph seems to be a wiser choice when approximating it numerically.

We consider numerical differentiation to be an extremely useful feature to have at hand while prototyping and debugging. Ultimately, users should weigh computational overhead against development effort to decide whether numerical differentiation helps them achieve their particular goals.

2.9. *Additional features*

Our framework uses standard tensor product Gauss–Legendre quadrature. Tensor product Gauss–Lobatto quadrature rules are also available. Research on quadrature rules tailored to isogeometric analysis started in [51]. Support for optimal, B-spline-specific, quadrature rules is ongoing [52–54]. PetIGA also supports tensor product basis functions defined as Lagrange interpolants on either Newton–Cotes points (as in traditional finite elements) or Gauss–Lobatto points (as in spectral finite elements [55]). Although these methods are not the primary focus of our library, their availability can be useful to researchers willing to explore isogeometric analysis as an alternative to these well-established technologies. Even though PetIGA is geared towards handling Galerkin-type finite element formulations, the framework also supports isogeometric collocation methods [56].

Creation of initial geometries is a nontrivial task. Representation of a volume using NURBS is cumbersome and mostly developed manually, and bridging Computer Aided Design and Computer Aided Engineering is ongoing. To ease the handling of NURBS geometries we developed igakit [57], a Python package providing low-level interfaces to Fortran 90 routines in charge of manipulating knot vectors and control point data, as well as a high-level interface

```
from igakit.cad import *

C0 = circle(radius=1)
C1 = circle(radius=2)
annulus = ruled(C0, C1)

pipe = extrude(annulus,
               displ=3,
               axis=2,
               ).reverse(2)

elbow = revolve(annulus,
                point=(3,0,0),
                axis=(0,-1,0),
                angle=Pi/2)

bentpipe = join(pipe, elbow,
                axis=2)
```

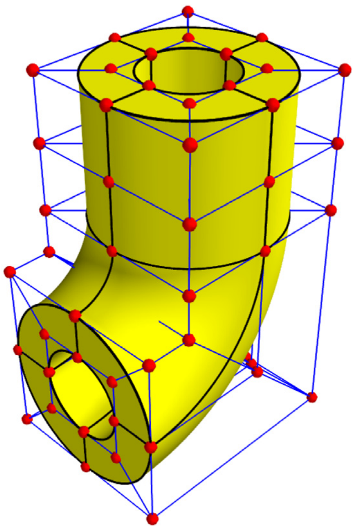


Fig. 8. Generating a bent pipe (see [2], section 2.4) with igakit. The definition of this NURBS volume geometry entails four successive steps. First, an annulus is generated as a ruled surface between two concentric circles of different radii. Then, a pipe is generated by extruding the annulus along the appropriate direction. Next, an elbow is generated by revolving the base annulus 90° around the appropriate axis. Finally, the pipe and the elbow are joined along one of the parametric directions.

to simplify the definition of these geometries by hand. Details on using igakit to create geometries such as the one presented in Fig. 8 can be found in [58].

3. Example

In this section we solve the Bratu equation as a model application to highlight some of the useful features users have access to when using our framework.

3.1. The Bratu equation

The Bratu equation is a nonlinear second-order boundary value problem [59]. The strong form of the equation can be stated as: find $u : \bar{\Omega} \to \mathbb{R}$ such that

$$-\Delta u = \lambda \exp(u), \quad \boldsymbol{x} \in \Omega, \tag{13}$$

$$u = 0, \quad \boldsymbol{x} \in \partial\Omega, \tag{14}$$

where $\bar{\Omega}$ is the unit square in $[0, 1]^2$, $\partial\Omega$ denotes the domain boundary, Δ represents the Laplace operator, $u \equiv u(\boldsymbol{x})$ is a scalar field defined in Ω and λ is a positive constant. No solution exists when λ goes above $\lambda_{max} = 6.80812$ as the equation possesses a bifurcation point. This equation models the steady-state of a nonlinear reaction and heat conduction problem, and results from a simplification of the solid-fuel ignition model.

Within the framework of Galerkin finite elements, let $\mathcal{V}$ denote the trial and weighting function spaces, where $\mathcal{V}$ belongs to $\mathcal{H}_0^1$, i.e., the Sobolev space of square-integrable functions with square-integrable first derivatives and zero value on $\partial\Omega$. The weak form is obtained by multiplying Eq. (13) by a test function w and integrating by parts. The variational problem can then be defined as that of finding $u \in \mathcal{V}$ such that for all $w \in \mathcal{V}$

$$(\nabla w, \nabla u)_\Omega - (w, \lambda \exp(u))_\Omega = 0, \tag{15}$$

where $(\cdot, \cdot)_\Omega$ denotes the $\mathcal{L}^2$ inner product on domain Ω. The finite-dimensional problem can then be formulated as: find $u^h \in \mathcal{V}^h$, where $\mathcal{V}^h \subset \mathcal{V}$, such that for all $w^h \in \mathcal{V}^h$

$$\left(\nabla w^h, \nabla u^h\right)_\Omega - \left(w^h, \lambda \exp\left(u^h\right)\right)_\Omega = 0, \tag{16}$$

where w^h, u^h and their respective gradients are defined as the linear combinations

$$w^h = \sum_A W_A N_A(\boldsymbol{x}), \qquad u^h = \sum_A U_A N_A(\boldsymbol{x}), \tag{17}$$

$$\nabla w^h = \sum_A W_A \nabla N_A(\boldsymbol{x}), \qquad \nabla u^h = \sum_A U_A \nabla N_A(\boldsymbol{x}), \tag{18}$$

where N_A are the basis functions and W_A, U_A are the control variables. Denoting $\mathbf{U} = \{U_A\}$ the vector of coefficients, we define the residual vector as $\mathbf{R}(\mathbf{U}) = \{R_A\}$, where R_A is obtained from (16)–(18) as

$$R_A = \left(\nabla N_A, \nabla u^h\right)_\Omega - \left(N_A, \lambda \exp\left(u^h\right)\right)_\Omega . \tag{19}$$

As the problem is nonlinear, solving it with Newton's method requires the specification of the Jacobian $\mathbf{J} = \partial\mathbf{R}/\partial\mathbf{U}$. In this particular problem, its entries are defined as

$$J_{AB} = (\nabla N_A, \nabla N_B)_\Omega - \left(N_A, \lambda \exp\left(u^h\right) N_B\right)_\Omega . \tag{20}$$

3.2. Implementation

To code a program within the framework, the user first needs to include the PetIGA C header file

```c
#include <petiga.h>
```

For the sake of simplicity, two macros are defined to set the dimensionality of the problem to two and implement the dot product of two-vectors

```c
#define dim 2
#define dot(a,b) (a[0]*b[0]+a[1]*b[1])
```

To change the dimensionality of the problem to one or three, these two lines should be modified accordingly. Then, a C structure is used to handle the problem-specific parameter λ. This approach is preferred to the alternative of using global variables.

```c
typedef struct {
  double lambda;
} Params;
```

The residual routine implementing Eq. (19) reads

```c
int Residual(IGAPoint p,const double U[],double R[],void *ctx) {
  int     a,nen       = p->nen;
  double  (*N0)       = (typeof(N0)) p->shape[0];
  double  (*N1)[dim]  = (typeof(N1)) p->shape[1];
  double  u,grad_u[dim],lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  IGAPointFormGrad (p,U,grad_u);
  for (a=0; a<nen; a++)
    R[a] = dot(N1[a],grad_u) - lambda*exp(u)*N0[a];
  return 0;
}
```

Recalling Section 2.7, the residual routine constitutes the integrand to be evaluated at each quadrature point (line 9 of Algorithm 3) to compute local contributions to the global residual vector. The `Residual` routine has the following arguments

- an input pointer `p`, of type `IGAPoint`, used as a quadrature point context holding discretization data required to perform the residual evaluation,
- an input floating point array `U`, which contains the local control variables gathered from the global vector $\mathbf{U}$, i.e., the coefficients corresponding to basis functions whose support contains the quadrature point, recall Eqs. (17) and (18),
- an output floating point array `R`, where the routine is expected to return local contributions to be assembled in the global residual vector $\mathbf{R}$, recall Eq. (19),
- an input opaque pointer `ctx`, used to pass problem-specific information to the residual routine.

The code proceeds to declare the following local variables

- two integers `a` and `nen`, the first to be used as a loop index, while the second is initialized to the number of local basis functions,
- two pointers `N0` and `N1`, initialized from data within the `IGAPoint`, used in the following for convenient access to the values of local basis functions (0th derivatives) and their first derivatives,
- two floating point variables `u` and `grad_u` to store the values of u^h and ∇u^h at the quadrature point
- a floating point variable `lambda` to store the value of λ, initialized from the `Params` structure through the opaque pointer `ctx`.

Next, the routines `IGAPointFormValue` and `IGAPointFormGrad` are invoked to compute the values of u^h and ∇u^h at the quadrature point, following Eqs. (17) and (18). Finally, local residual contributions are computed in a loop following Eq. (19) by using the previously defined `dot` macro as well as the `exp` routine from the standard library of the C programming language. The quadrature weights and Jacobian determinant of the geometry mapping are handled internally, which slightly simplifies the coding.

The Jacobian routine implementing Eq. (20) reads

```
int Jacobian(IGAPoint p,const double U[],double J[],void *ctx) {
  int    a,b,nen     = p->nen;
  double (*N0)       = (typeof(N0)) p->shape[0];
  double (*N1)[dim] = (typeof(N1)) p->shape[1];
  double u,lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  for (a=0; a<nen; a++)
    for (b=0; b<nen; b++)
      J[a*nen+b] = dot(N1[a],N1[b]) - lambda*exp(u)*N0[a]*N0[b];
  return 0;
}
```

The `Jacobian` routine is almost the same as the previous `Residual` routine. The main differences are the output floating point array `J` where the routine is expected to return local contributions to be assembled in the global Jacobian matrix **J**, and the double-loop computing these contributions following Eq. (20).

The code of the main program routine then begins

```
int main(int argc, char *argv[]) {
  PetscInitialize(&argc,&argv,NULL,NULL);
```

The first statement invokes the `PetscInitialize` routine, which internally handles the initialization of the PETSc library. This routine is fed with the command line arguments to the program. PETSc uses these arguments to build a database of options. These options are queried later.

The code proceeds to create and initialize an `iga` context of type `IGA`. The `IGA` type is a key component in PetIGA. This core data structure contains all the information related to discretization and parallel communication.

```
  IGA iga;
  IGACreate(PETSC_COMM_WORLD,&iga);
  IGASetDim(iga,dim);
  IGASetDof(iga,1);
  IGASetFromOptions(iga);
  IGASetUp(iga);
```

To properly setup the `iga` context, the user only needs to hardwire in code a couple of problem-specific parameters, and ask the framework to handle the rest at runtime through the options database

- the routine `IGASetDim` specifies the number of space dimensions, in this particular example it is set to two,
- the routine `IGASetDof` specifies the number of components in the solution, in this particular example it is set to one as the code is dealing with a scalar problem,
- the routine `IGASetFromOptions` queries the options database to let users change different properties of the discretization such as number of elements, polynomial degree and regularity of the approximation space, type of basis functions to use, quadrature rules, number of quadrature points, among others. When values are not specified by the user, PetIGA selects default values such as 16 elements per direction, unit domain, uniform refinement,

quadratic C^1 spaces, and Gauss–Legendre quadrature. Even though other ways to initialize an IGA context are available, this is the simplest one. When more complex discretizations and/or geometries are required, the IGA context can be initialized by loading binary datafiles,

- finally, the routine `IGASetUp` prepares the data structure to be used in what follows. This step handles the parallel partitioning of the domain and prepares internal data structures that manage parallel communication.

Boundary conditions are handled in the code snippet that follows

```
  int direction,side;
  for (direction=0; direction<dim; direction++) {
    for (side=0; side<2; side++) {
      int field = 0; double value = 0.0;
      IGASetBoundaryValue(iga,direction,side,field,value);
    }
  }
```

The routine `IGASetBoundaryValue` is used to set the Dirichlet boundary conditions of this problem, as specified in Eq. (14), and takes as arguments

- the `IGA` context in which the boundary condition is to be set,
- the parametric direction, which takes values `0` or `1` to specify either the first or second parametric directions, respectively,
- the boundary side, which takes values `0` or `1` to specify either the left or right side along the parametric direction, respectively,
- the component index, which is `0` for scalar problems,
- finally, the value to be enforced at the boundary.

Although not highlighted in this example, PetIGA allows users to specify more general and possibly nonlinear boundary conditions. These boundary conditions should be handled in the user-defined residual and Jacobian routines.

The final step to configure the `IGA` context requires the specification of the user-defined residual and Jacobian callbacks, as shown in the following lines of code

```
  Params params;
  params.lambda = IGAGetOptReal(NULL,"-lambda",6.80);
  IGASetFormFunction(iga,Residual,&params);
  IGASetFormJacobian(iga,Jacobian,&params);
```

A local variable `params` of type `Params` is declared, the `lambda` member is initialized from a user-specified command line option, or from a hardwired default value otherwise. The calls to `IGASetFormFunction` and `IGASetFormJacobian` register within the `IGA` context the user-defined residual and Jacobian routines. They also store the pointer to the `Params` instance which is used in `Residual` and `Jacobian` to access the λ parameter, as explained previously.

The code then proceeds to solve the nonlinear problem

```
  SNES snes;
  IGACreateSNES(iga,&snes);
  SNESSetFromOptions(snes);

  Vec U;
  IGACreateVec(iga,&U);
  SNESSolve(snes,NULL,U);
```

The routine `IGACreateSNES` creates a nonlinear solver context and performs additional initialization such as associating the global vector to form the residual and the global matrix in which to form the Jacobian within the solver context. The routine `SNESSetFromOptions` enables users to further configure the nonlinear solver through command line options. Finally, a global vector is created to store the solution coefficients and the routine `SNESSolve` is invoked to solve the problem.

A call to the routine `VecViewFromOptions` creates a viewer context which can be used to visualize the solution vector in real-time, or store the solution as a VTK datafile [60,61]

```
  VecViewFromOptions(U,NULL,"-output");
```

Lastly, all PETSc and PetIGA contexts created through the code are destroyed to free resources and the routine `PetscFinalize` is invoked to properly shutdown the framework

```
  VecDestroy(&U);
  SNESDestroy(&snes);
  IGADestroy(&iga);
  PetscFinalize();
  return 0;
}
```

4. Applications

To illustrate the flexibility of our software, we showcase applications which highlight strengths of isogeometric analysis. The problems come from standard engineering domains, such as solid and fluid mechanics, as well as from less traditional areas, such as phase-field modeling. We have chosen to solve nonlinear, partial differential equations in order to demonstrate our code framework on a challenging subset of problems. In each problem, a nonlinear residual functional is obtained from the weak form of the partial differential equation. Many problem-specific details we omit here and refer the reader to the referenced software package [37] where they can find full details in the form of demonstration programs.

4.1. Steady-state hyper-elasticity

The first of our examples is a hyper-elastic material model in the context of large deformation, applied to a cylindrical tube [62]. The strong form can be expressed as: find the displacement $\mathbf{U} : \overline{\Omega} \mapsto \mathbb{R}^3$ such that

$$\begin{aligned} \nabla \cdot \mathbf{P} &= \mathbf{0} \quad \text{in } \Omega, \\ \mathbf{U} &= \mathbf{G} \quad \text{on } \Gamma_D, \\ \mathbf{P} \cdot \mathbf{N} &= \mathbf{0} \quad \text{on } \Gamma_N, \end{aligned}$$

where $\mathbf{P}$ is the first Piola–Kirchhoff stress tensor, $\mathbf{G}$ is the prescribed displacement on the Dirichlet boundary Γ_D, $\mathbf{N}$ is the outward normal of the Neumann boundary Γ_N in the reference configuration (see [63,64] for more details). We use a Neo-Hookean material model, which relates the second Piola–Kirchhoff stress tensor $\mathbf{S}$ to the right Cauchy–Green strain tensor $\mathbf{C}$ by the relationship

$$\mathbf{S} = \frac{\lambda}{2}\left(J^2 - 1\right)\mathbf{C}^{-1} + \mu\left(\mathbf{I} - \mathbf{C}^{-1}\right),$$

where $\mathbf{C} = \mathbf{F}^{\mathrm{T}}\mathbf{F}$, $\mathbf{F}$ is the deformation gradient, $J = \det(\mathbf{F})$, and λ, μ are the Lamé constants from linear elasticity. The first Piola–Kirchhoff stress tensor is defined by $\mathbf{P} = \mathbf{FS}$ and the symmetry condition $\mathbf{P}^{\mathrm{T}}\mathbf{F} = \mathbf{F}^{\mathrm{T}}\mathbf{P}$.

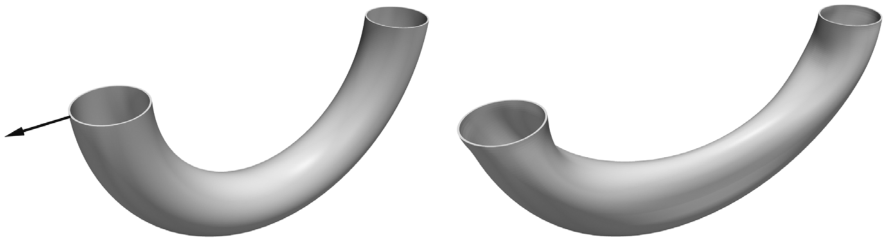
We solve the linearized weak form of these equations using Newton's method in an updated-Lagrangian approach. In Fig. 9(a) we depict the reference geometry, a circular tube discretized with a mesh of $16 \times 64 \times 4$ quadratic B-spline functions. The right side of the tube is fixed and the left side is displaced to the left over 15 load steps. In this case we configured PETSc to interface with MUMPS and solved the system using this parallel direct solver. The deformed configuration is shown in Fig. 9(b).

4.2. Time-dependent problems

The following three examples discretize time-dependent, nonlinear problems. We first detail our solution strategy for these problems. For simplicity, we consider a scalar problem. We seek to find $\dot{u}, u \in \mathcal{U}$ such that

$$\mathcal{R}(w; \dot{u}, u) = 0 \quad \forall w \in \mathcal{W},$$

where $\mathcal{R}$ is a time-dependent, nonlinear residual functional.



(a) Undeformed tube. (b) Deformed tube.

Fig. 9. Deformation of an aluminum tube, modeled using a Neo-Hookean material model. The arrow represents the direction and magnitude of the displacement imposed as a boundary condition on the left end while the right end is kept fixed.

We first use a semi-discrete approach by discretizing $\dot{u}, u$ in space with finite-dimensional subspaces $\mathcal{U}^h \subset \mathcal{U}$, leaving the problem continuous in time. The span of the set of basis functions $\{N_B(\boldsymbol{x})\}_{B=0\ldots n}$, $\boldsymbol{x} \in \Omega$ define the subspace $\mathcal{U}^h$. Similarly, we choose $\mathcal{W}^h \subset \mathcal{W}$ where span $(\{N_A(\boldsymbol{x})\}_{A=0\ldots n-1})$ defines $\mathcal{W}^h$. Given two sequences of time-dependent coefficients $\{\dot{U}_B(t)\}$, $\{U_B(t)\}$, discrete functions $\dot{u}^h, u^h$ are defined as

$$\dot{u}^h(\boldsymbol{x}, t) = \sum_B \dot{U}_B(t) N_B(\boldsymbol{x}),$$
$$u^h(\boldsymbol{x}, t) = \sum_B U_B(t) N_B(\boldsymbol{x}).$$

Denoting $\dot{\mathbf{U}} = \{\dot{U}_B(t)\}$, $\mathbf{U} = \{U_B(t)\}$, we can write the residual vector,

$$\mathbf{R}\left(\dot{\mathbf{U}}, \mathbf{U}\right) = \{R_A\},$$

where

$$R_A = \mathcal{R}\left(N_A; \dot{u}^h, u^h\right).$$

Next, we discretize in time using the generalized-α method for first order systems [65]. Given $\dot{\mathbf{U}}_n, \mathbf{U}_n$, we seek $\dot{\mathbf{U}}_{n+1}, \mathbf{U}_{n+1}$ such that

$$\mathbf{R}\left(\dot{\mathbf{U}}_{n+\alpha_m}, \mathbf{U}_{n+\alpha_f}\right) = 0 \tag{21}$$
$$\dot{\mathbf{U}}_{n+\alpha_m} = \dot{\mathbf{U}}_n + \alpha_m\left(\dot{\mathbf{U}}_{n+1} - \dot{\mathbf{U}}_n\right) \tag{22}$$
$$\mathbf{U}_{n+\alpha_f} = \mathbf{U}_n + \alpha_f\left(\mathbf{U}_{n+1} - \mathbf{U}_n\right) \tag{23}$$
$$\mathbf{U}_{n+1} = \mathbf{U}_n + \Delta t\left(\gamma\dot{\mathbf{U}}_{n+1} + (1-\gamma)\dot{\mathbf{U}}_n\right) \tag{24}$$

where $\Delta t = t_{n+1} - t_n$ is the time step, and $\alpha_f, \alpha_m, \gamma$ are parameters which define the method. The generalized-α method was designed to filter (damp) high-frequency modes of the solution which are under-approximated. As opposed to linear problems, low and high frequency modes interact in nonlinear problems. Spurious high frequency modes lead to contamination of the resolved modes of the problem. The method parameters $\alpha_f, \alpha_m, \gamma$ can be chosen using the spectral radius $\rho_\infty \in [0, 1]$ of the amplification matrix as $\Delta t \to \infty$ by

$$\alpha_m = \frac{1}{2}\left(\frac{3-\rho_\infty}{1+\rho_\infty}\right)$$
$$\alpha_f = \frac{1}{1+\rho_\infty}$$
$$\gamma = \frac{1}{2} + \alpha_m - \alpha_f$$

which leads to a second order, unconditionally stable method. The value of ρ_∞ uniquely defines the method and can be chosen to filter a desired amount of high frequency modes.

In our experience, when approaching time-dependent, nonlinear problems, the generalized-α method is effective. The impact of the interaction of low and high frequency modes is problem dependent and not something necessarily understood in advance. In the case where no filtering is needed ($\rho_\infty = 1$), the generalized-α method reduces to the implicit midpoint rule. Due to the popularity of the generalized-α method particularly among researchers in the isogeometric community, we have added it to PETSc's time stepping algorithms and is available independently of the PetIGA framework. In the examples that follow, we solve Eq. (21) iteratively using Newton's method, which requires the computation of the Jacobian of the residual vector at each iteration.

4.2.1. *Cahn–Hilliard equation*

The Cahn–Hilliard equation governs the evolution of a binary mixture undergoing the process of phase separation. Let $\Omega \in \mathbb{R}^d$ be an open set, where $d = 2, 3$. The boundary of Ω with unit outward normal $\boldsymbol{n}$ is denoted Γ and is composed of two complementary parts Γ_g and Γ_h. Denoting c the concentration of one of the components of the mixture, the problem can be stated in strong form as: find $c : \overline{\Omega} \times (0, T) \mapsto \mathbb{R}$ such that

$$\begin{aligned}
\frac{\partial c}{\partial t} - \nabla \cdot (M_c \nabla (\mu_c - \lambda \Delta c)) = 0 \quad & \text{in} \quad \Omega \times (0, T),\\
c = g \quad & \text{on} \quad \Gamma_g \times (0, T),\\
M_c \nabla (\mu_c - \lambda \Delta c) \cdot \boldsymbol{n} = h \quad & \text{on} \quad \Gamma_h \times (0, T),\\
M_c \lambda \nabla c \cdot \boldsymbol{n} = 0 \quad & \text{on} \quad \Gamma \times (0, T),\\
c(\boldsymbol{x}, 0) = c_0(\boldsymbol{x}) \quad & \text{in} \quad \overline{\Omega},
\end{aligned}$$

where c_0 is the initial concentration, M_c is the mobility, μ_c represents the chemical potential of a binary mixture in the absence of phase interfaces, and λ is a positive constant such that $\sqrt{\lambda}$ represents a length scale of the problem related to the interface thickness between the two phases. The mobility and chemical potential are nonlinear functions of the concentration

$$\begin{aligned}
M_c &= Dc(1 - c),\\
\mu_c &= \frac{1}{2\theta} \log \frac{c}{1 - c} + 1 - 2c,
\end{aligned}$$

in which D is a positive constant with dimensions of diffusivity and θ is a dimensionless number which represents the ratio between critical and absolute temperatures.

We solve the Cahn–Hilliard equation in dimensionless form and with periodic boundary conditions as detailed in [7]. We show snapshots of the isocontours of the solution in three dimensions at different times during the simulation in Fig. 10.

4.2.2. *Navier–Stokes–Korteweg equations*

The Navier–Stokes–Korteweg equations are a phase-field model for water and water vapor two-phase flows. Let $\Omega \in \mathbb{R}^d$ be an open set, where $d = 2, 3$. The boundary of Ω with unit outward normal $\boldsymbol{n}$ is denoted Γ. The problem can be stated in strong form as: find the density $\rho : \overline{\Omega} \times (0, T) \mapsto (0, b)$ and the velocity $\boldsymbol{u} : \overline{\Omega} \times (0, T) \mapsto \mathbb{R}^d$ such that

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \quad & \text{in} \quad \Omega \times (0, T),\\
\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{I}) - \nabla \cdot \boldsymbol{\tau} - \nabla \cdot \boldsymbol{\zeta} = \rho \boldsymbol{f} \quad & \text{in} \quad \Omega \times (0, T),\\
\boldsymbol{u} = \boldsymbol{0} \quad & \text{on} \quad \Gamma \times (0, T),\\
\nabla \rho \cdot \boldsymbol{n} = 0 \quad & \text{on} \quad \Gamma \times (0, T),\\
\rho(\boldsymbol{x}, 0) = \rho_0(\boldsymbol{x}) \quad & \text{in} \quad \overline{\Omega},\\
\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}) \quad & \text{in} \quad \overline{\Omega},
\end{aligned}$$

where $\boldsymbol{u}_0$ and ρ_0 are initial values of the density and velocity, respectively. The function $\boldsymbol{f}$ represents the body force per unit mass. The viscous stress tensor is given as

$$\boldsymbol{\tau} = \bar{\mu} \left(\nabla \boldsymbol{u} + \nabla^T \boldsymbol{u} \right) + \bar{\lambda} \nabla \cdot \boldsymbol{u} \boldsymbol{I}$$

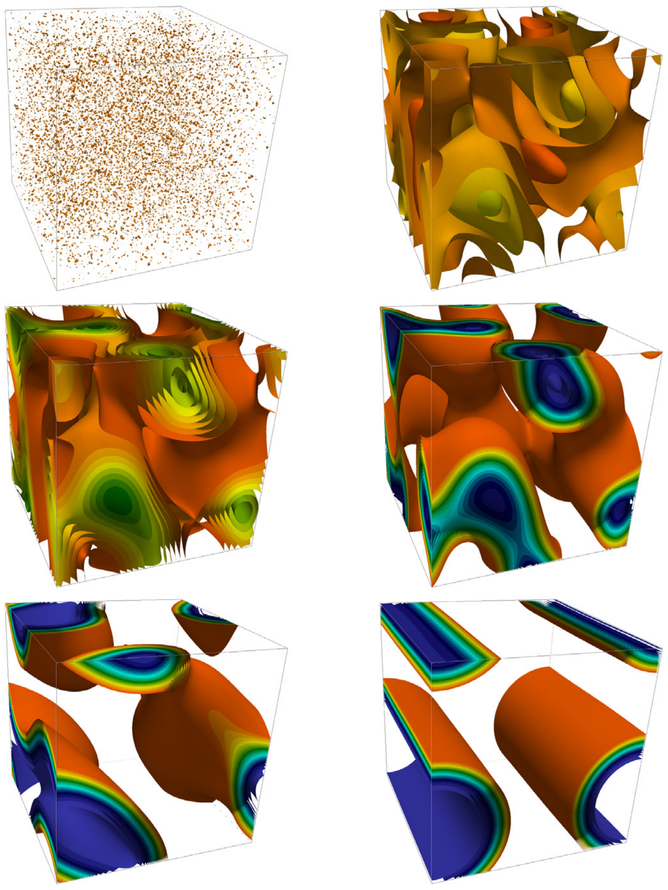


Fig. 10. Transient solution to the Cahn–Hilliard problem in three dimensions, subject to a random initial condition and periodic boundary conditions. The weak form is discretized in space by a mesh of 256^3 elements of C^1 quadratic B-splines.

where $\bar{\mu}$ and $\bar{\lambda}$ are the viscosity coefficients and $\boldsymbol{I}$ is the identity tensor. The Korteweg tensor is defined as

$$\boldsymbol{\zeta} = \lambda\left(\rho\Delta\rho + \frac{1}{2}\|\nabla\rho\|^2\right)\boldsymbol{I} - \lambda\nabla\rho\otimes\nabla\rho$$

where λ is the capillarity coefficient. The pressure is given by the van der Waals equation

$$p = Rb\frac{\rho\theta}{b-\rho} - a\rho^2,$$

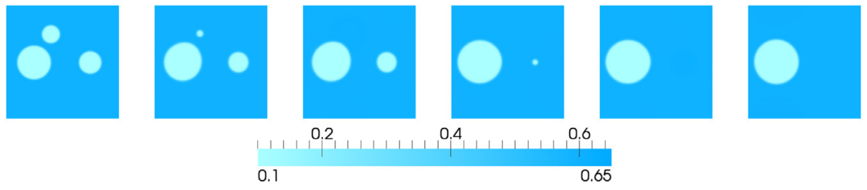
where a, b are constants, R is the ideal gas constant, and θ is the temperature, which for the isothermal model is assumed to be constant.

We solve the three-bubble test problem as detailed in [8] and show here the density and magnitude of the velocity for a two-dimensional solution in Fig. 11 and a three-dimensional solution in Fig. 12.

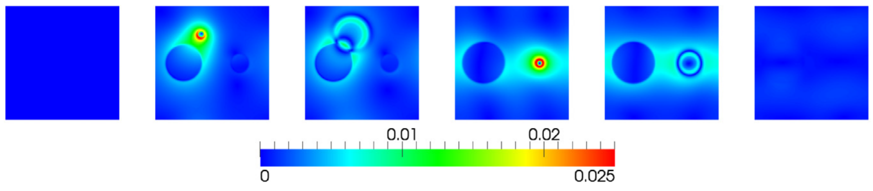
4.2.3. Navier–Stokes equations

We solve the incompressible Navier–Stokes equations stabilized with the variational multiscale method as formulated in [66]. Let $\Omega \in \mathbb{R}^d$ be an open set, where $d = 2, 3$. The boundary of Ω with unit outward normal $\boldsymbol{n}$ is denoted Γ. The problem can be stated in strong form as: find the velocity $\boldsymbol{u} : \overline{\Omega} \times (0, T) \mapsto \mathbb{R}^d$ and the pressure (divided by the constant density) $p : \overline{\Omega} \times (0, T) \mapsto (0, b)$ such that

$$\begin{aligned} \frac{\partial\boldsymbol{u}}{\partial t} + \nabla\cdot(\boldsymbol{u}\otimes\boldsymbol{u}) + \nabla p = \nu\Delta\boldsymbol{u} + \boldsymbol{f} \quad &\text{in} \quad \Omega\times(0,T), \\ \nabla\cdot\boldsymbol{u} = 0 \quad &\text{in} \quad \Omega\times(0,T), \end{aligned}$$

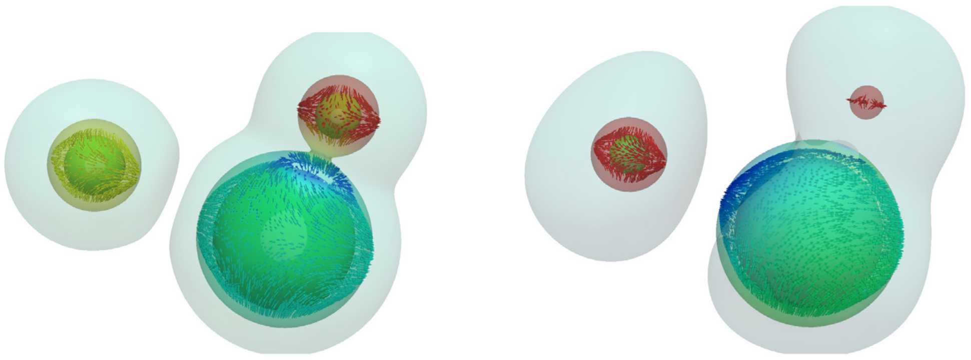


(a) Density.



(b) Velocity (magnitude).

Fig. 11. Time evolution (left to right) of the density and the magnitude of the velocity for the isothermal NSK equations on the three bubble test case problem.



(a) Near the beginning of the simulation. (b) Prior to the collapse of a bubble.

Fig. 12. Three-dimensional version of the three bubble problem for the Navier–Stokes–Korteweg equation. Isocontour surfaces reflect density values $\rho = \{0.15, 0.55\}$ revealing the location of the three bubbles. Velocity vectors are shown on each isocontour and are colored by the velocity magnitude. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

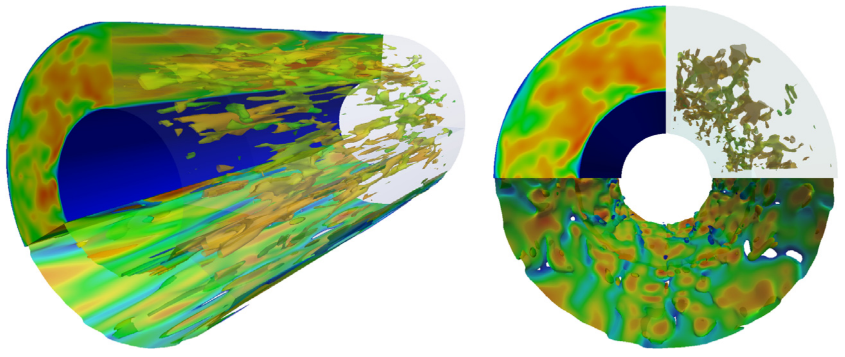
$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{0} && \text{on} \quad \Gamma \times (0, T), \\ \boldsymbol{u}(\boldsymbol{x}, 0) &= \boldsymbol{u}_0(\boldsymbol{x}) && \text{in} \quad \overline{\Omega}, \end{aligned}$$

where $\boldsymbol{u}_o$ is the initial velocity, $\boldsymbol{f}$ represents the body force per unit volume, and ν is the kinematic viscosity.

We solve a turbulent flow in a concentric pipe as presented in [67] the results of which we plot in Fig. 13. The domain is periodic in the streamwise direction. No-slip boundary conditions are set at the inner and outer cylinder surfaces and the initial condition is set using a laminar flow profile. The simulation is forced using a pressure gradient in the form of a body force in the streamwise direction.

5. Performance

In this section we present scaling results of our code framework, applied to the incompressible Navier–Stokes equations described in Section 4. Scaling of time-dependent, nonlinear problems can be difficult to quantify as solver component performance changes with problem size and decomposition. The nonlinearity increases slightly with the problem size, which can lead to extra Newton iterations for larger problems. The condition number of the linear system increases considerably with the size of the problem, requiring more linear iterations for convergence. At the same time, domain-decomposition-based preconditioners become weaker as more processors are employed, further increasing the number of iterations required by the linear solver.



(a) Perspective view. (b) Front view.

Fig. 13. Turbulent flow through concentric cylinders as in [67]. The top-left quadrant is a pseudocolor plot of the streamwise velocity. The top-right quadrant shows isocontours of the vorticity magnitude for smaller values. The bottom quadrants show isocontours of the vorticity magnitude for larger values. Both sets of contours are colored by the streamwise velocity. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 1
Parallel efficiency on a single node of Stampede.

Mesh	Number of cores						
	1	2	4	6	8	12	16
$n_{el} = 32^3$	100%	98%	95%	90%	90%	89%	91%
$n_{el} = 64^3$	100%	94%	92%	90%	90%	86%	89%

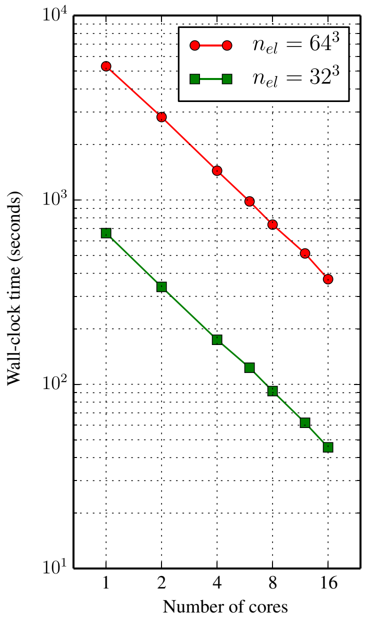
With the goal of measuring the performance of the code while disregarding the algorithmic issues mentioned above, we perform tests with the following assumptions and constraints.

- The problem is the flow between two plates problem with no-slip boundary conditions at the plate walls.
- The initial condition is set to the steady-state laminar flow profile.
- We discretize the domain using C^1 quadratic B-splines and employ full Gauss–Legendre quadrature with three quadrature points per direction on each element.
- We limit the test problem to ten time steps.
- We enforce the use of two nonlinear Newton iterations per time step.
- We enforce the use of 30 GMRES iterations per Newton iteration.
- The preconditioner is block Jacobi with one block per process. and ILU(0) (incomplete LU factorization with zero fill-in) in each block.

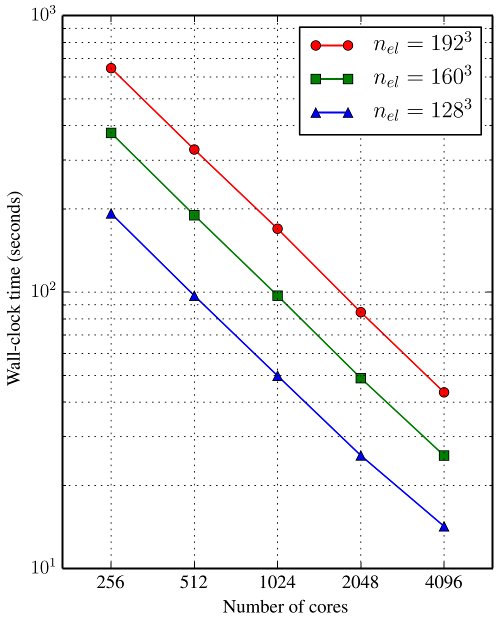
We ran these tests on Stampede [68], a distributed memory supercomputing system featuring compute nodes with 2×8-core Intel Xeon E5-2680 (Sandy Bridge) processors, 32 GB of memory per node, and a InfiniBand Mellanox interconnect. In Fig. 14(a) and Table 1, we show strong scaling and efficiency of the Navier–Stokes code on a single node using different discretization sizes indicated by the number of elements used, n_{el}. In Fig. 14(b) and Table 2, we show strong scaling and efficiency on multiple compute nodes. In these multiple node runs, the parallel efficiency surpasses 80%, and is above 90% in all but one of the cases. These values are achieved for local problem sizes as small as 512 elements per core.

6. Conclusions

In this paper we present PetIGA, a scalable implementation of isogeometric analysis for linear/nonlinear and static/transient problems. By being built on top of PETSc, the framework gives users a robust and versatile platform to solve partial differential equations. We show that the framework scales well on up to 4096 cores on the Navier–Stokes problem, and is therefore well suited for large scale applications. Even though primarily conceived for distributed-memory computing environments, PetIGA is also able to extract excellent performance on nowadays shared memory multicore laptop and desktop computers. Our software is already being used by members of the community to tackle



(a) Single node runs.



(b) Multiple node runs.

Fig. 14. Strong scaling on Stampede. Wall-clock times correspond to runs of the Navier–Stokes problem described in Section 4. Each run spans ten time steps with a fixed number of Newton and GMRES iterations.

Table 2
Parallel efficiency on multiple nodes of Stampede.

Mesh	Number of cores				
	256	512	1024	2048	4096
$n_{el} = 128^3$	100%	99%	97%	94%	85%
$n_{el} = 160^3$	100%	99%	97%	96%	92%
$n_{el} = 192^3$	100%	98%	95%	95%	93%

challenging problems related to parallel multifrontal direct solvers [69], fast multipole-based preconditioners for elliptic equations [70], multilevel Monte Carlo algorithms geared towards the approximation of stochastic models [71], fluid–structure interaction [72], and finite strain gradient elasticity [73].

Up to now, we focused on making single patch geometries run as efficiently as possible. Single patch geometries are certainly enough for a wide range of academic applications. However, support for multipatch geometries is a must to tackle more complex simulations. Still, extending our framework to support multiple patches while keeping all of its user-friendly features and parallel distributed memory capabilities is a nontrivial task that will be addressed in future work. A multi-field extension of PetIGA is being developed on top of the current implementation to handle discrete differential forms. Promising results have already been obtained using divergence- and integral-conforming spaces to solve the Navier–Stokes–Cahn–Hilliard equation [74,75].

Acknowledgments

We would like to acknowledge the open source software packages that made this work possible: `PETSc` [24], `NumPy` [76], `matplotlib` [77], `IPython` [78]. We would like to thank Lina María Bernal Martinez, Gabriel Andres Espinosa Barrios, Federico Fuentes-Caycedo, Juan Camilo Mahecha Zambrano for their work on the hyper-elasticity implementation as a final project to the *Non-linear Finite Element* class taught by V.M. Calo and N. Collier for the Mechanical Engineering Department at Universidad de Los Andes in Bogotá, Colombia in July 2012. We would like to thank Adel Sarmiento Rodriguez for the visualization work on Figs. 12 and 13.

This work is part of the European Union’s Horizon 2020 research and innovation programme of the Marie Skłodowska-Curie grant agreement No. 644602. This work was also supported by the Center for Numerical Porous Media at King Abdullah University of Science and Technology and Agencia Nacional de Promoción Científica y Tecnológica grants PICT 0938–13, PICT 2660–14 and PICT-E 0191–14.

Appendix A. Higher-order NURBS derivatives

Omitting the explicit dependence on $\boldsymbol{\xi}$ for notational convenience, derivatives of the NURBS weighting function (7) with respect to the parametric coordinates are simply

$$w_{,\alpha} = \frac{\partial w}{\partial \xi_\alpha} = \sum_B w_B M_{B,\alpha} \tag{A.1}$$

$$w_{,\alpha\beta} = \frac{\partial^2 w}{\partial \xi_\alpha \partial \xi_\beta} = \sum_B w_B M_{B,\alpha\beta} \tag{A.2}$$

$$w_{,\alpha\beta\gamma} = \frac{\partial^3 w}{\partial \xi_\alpha \partial \xi_\beta \partial \xi_\gamma} = \sum_B w_B M_{B,\alpha\beta\gamma}. \tag{A.3}$$

Using the chain rule, the first, second, and third derivatives with respect to ξ_α of the rational basis function N_A defined in (6) can be expressed as

$$N_{A,\alpha} = \frac{w_A M_{A,\alpha} - N_A w_{,\alpha}}{w} \tag{A.4}$$

$$N_{A,\alpha\beta} = \frac{w_A M_{A,\alpha\beta} - N_A w_{,\alpha\beta}}{w} - \frac{N_{A,\beta} w_{,\alpha} + N_{A,\alpha} w_{,\beta}}{w} \tag{A.5}$$

$$\begin{aligned} N_{A,\alpha\beta\gamma} = {} & \frac{w_A M_{A,\alpha\beta\gamma} - N_A w_{,\alpha\beta\gamma}}{w} - \frac{N_{A,\alpha} w_{,\beta\gamma} + N_{A,\beta} w_{,\alpha\gamma} + N_{A,\gamma} w_{,\alpha\beta}}{w} \\ & - \frac{N_{A,\beta\gamma} w_{,\alpha} + N_{A,\alpha\gamma} w_{,\beta} + N_{A,\alpha\beta} w_{,\gamma}}{w}. \end{aligned} \tag{A.6}$$

When using the isoparametric concept [79], we need to express derivatives in terms of spatial coordinates, not their parametric counterparts. Higher-order derivatives of the geometrical mapping and basis functions are required if one is to solve a higher-order partial differential equations on a mapped geometry. As these expressions are not standard in the literature, we provide them in the following for the sake of completeness.

Under the isoparametric concept, the geometric mapping is defined as

$$\boldsymbol{x}(\boldsymbol{\xi}) = \sum_A \boldsymbol{x}_A N_A(\boldsymbol{\xi}), \tag{A.7}$$

where N_A was defined in (6) and $\boldsymbol{x}_A$ are the control point locations in physical space. Thus, the parametric derivatives of this mapping are

$$\boldsymbol{x}_{,\alpha} = \frac{\partial \boldsymbol{x}(\boldsymbol{\xi})}{\partial \xi_\alpha} = \sum_A \boldsymbol{x}_A N_{A,\alpha}, \tag{A.8}$$

$$\boldsymbol{x}_{,\alpha\beta} = \frac{\partial \boldsymbol{x}(\boldsymbol{\xi})}{\partial \xi_\alpha \partial \xi_\beta} = \sum_A \boldsymbol{x}_A N_{A,\alpha\beta}, \tag{A.9}$$

$$\boldsymbol{x}_{,\alpha\beta\gamma} = \frac{\partial \boldsymbol{x}(\boldsymbol{\xi})}{\partial \xi_\alpha \partial \xi_\beta \partial \xi_\gamma} = \sum_A \boldsymbol{x}_A N_{A,\alpha\beta\gamma}. \tag{A.10}$$

For simplicity, we write them in index notation as $x_{i,\alpha}$, $x_{i,\alpha\beta}$, and $x_{i,\alpha\beta\gamma}$. Denoting the inverse mapping as $\boldsymbol{\xi}(\boldsymbol{x})$, the first spatial derivatives can be computed by matrix inversion through the identity

$$\delta_{ij} = x_{i,\epsilon}\, \xi_{\epsilon,j}, \tag{A.11}$$

where δ_{ij} is the Kronecker delta. By repeated differentiation, second and third spatial derivatives of the inverse mapping follow:

$$\xi_{\alpha,ij} = x_{m,\epsilon\mu}\,\xi_{\alpha,m}\,\xi_{\epsilon,i}\,\xi_{\mu,j} \tag{A.12}$$

$$\xi_{\alpha,ijk} = -x_{m,\epsilon\mu\omega}\,\xi_{\alpha,m}\,\xi_{\epsilon,i}\,\xi_{\mu,j}\,\xi_{\omega,k} - x_{m,\epsilon\mu}\left(\xi_{\alpha,mk}\,\xi_{\epsilon,i}\,\xi_{\mu,j} + \xi_{\alpha,m}\,\xi_{\epsilon,ik}\,\xi_{\mu,j} + \xi_{\alpha,m}\,\xi_{\epsilon,i}\,\xi_{\mu,jk}\right). \tag{A.13}$$

Finally, the first, second, and third spatial derivatives of the basis functions are

$$N_{A,i} = N_{A,\alpha}\,\xi_{\alpha,i} \tag{A.14}$$

$$N_{A,ij} = N_{A,\alpha\beta}\,\xi_{\alpha,i}\,\xi_{\beta,j} + N_{A,\alpha}\,\xi_{\alpha,ij} \tag{A.15}$$

$$N_{A,ijk} = N_{A,\alpha\beta\gamma}\,\xi_{\alpha,i}\,\xi_{\beta,j}\,\xi_{\gamma,k} + N_{A,\alpha\beta}\left(\xi_{\alpha,ik}\,\xi_{\beta,j} + \xi_{\alpha,i}\,\xi_{\beta,jk} + \xi_{\alpha,ij}\,\xi_{\beta,k}\right) + N_{A,\alpha}\,\xi_{\alpha,ijk}. \tag{A.16}$$

Appendix B. Building and running the Bratu example

This section complements Section 3. It is meant to showcase additional features related to building and running codes using PETSc and PetIGA. In the following, we assume a POSIX environment such as GNU/Linux or OS X. Additionally, we assume that PETSc and PetIGA have been properly configured and built with an MPI implementation to enable parallel usage. As the usage of batch systems and job schedulers is quite specific to the distributed-memory computing environment users have access to, we only show parallel execution within a single compute node or desktop/laptop computer.

We set environment variables pointing to the PETSc and PetIGA directories as well as the PETSc build configuration. For recurrent use, these variables can be defined in the user's shell configuration file. When using the Bash shell, these definitions can be done as follows

```
$ export PETSC_DIR=/path/to/petsc
$ export PETSC_ARCH=arch-platform-c-opt
$ export PETIGA_DIR=/path/to/petiga
```

The `ls` command produces a listing with the content of the current working directory

```
$ ls -lh
-rw-rw-r--. 1 user users 1.7K Dec 31 23:58 Bratu.c
-rw-rw-r--. 1 user users  242 Dec 31 23:58 makefile
```

The `Bratu.c` file contains the C source code of the example under consideration. This code is available at the end of this section, see listing 1. The contents of `makefile` can be inspected with the `cat` command

```
$ cat makefile
Bratu: Bratu.PETIGA;
include $(PETIGA_DIR)/lib/petiga/conf/variables
include $(PETIGA_DIR)/lib/petiga/conf/rules
```

The `make` utility is arguably the most popular build framework in POSIX systems and is almost always available. PetIGA users can compile their codes by writing a trivial `makefile` and invoking `make` in the command line with no additional setup. This `makefile` is quite simple: the first line defines a single target and the following two lines include variable definitions and build rules found within the PetIGA directory.

We use the `make` command to build the example. For the sake of brevity, we omit the verbose output from the compiler and linker. We then use the `ls` command once again to verify that the `Bratu` binary executable has indeed been generated

```
$ make Bratu
...
$ ls -lh
-rwxrwxr-x. 1 user users 146K Dec 31 23:59 Bratu
-rw-rw-r--. 1 user users 1.7K Dec 31 23:58 Bratu.c
-rw-rw-r--. 1 user users  242 Dec 31 23:58 makefile
```

Finally, we run the `Bratu` binary in parallel with four processes using the `mpiexec` command. Additionally, we pass some command line options to the program.

```
$ mpiexec -n 4 ./Bratu -iga_elements 128 -iga_view -snes_monitor -ksp_type cg
IGA: dim=2 dof=1 order=2 geometry=0 rational=0 property=0
Axis 0: basis=BSPLINE[2,1] rule=LEGENDRE[3] periodic=0 nnp=130 nel=128
Axis 1: basis=BSPLINE[2,1] rule=LEGENDRE[3] periodic=0 nnp=130 nel=128
Partition - MPI: processors=[2,2,1] total=4
Partition - nnp: sum=16900 min=4096 max=4356 max/min=1.06348
Partition - nel: sum=16384 min=4096 max=4096 max/min=1
  0 SNES Function norm 5.266384548611e-02
  1 SNES Function norm 8.620220401724e-03
  2 SNES Function norm 2.054605014212e-03
  3 SNES Function norm 4.716226279209e-04
  4 SNES Function norm 8.916608064674e-05
  5 SNES Function norm 8.438014748365e-06
  6 SNES Function norm 1.155533195923e-07
  7 SNES Function norm 2.309601078808e-11
```

Command line option handling has been a key feature of PETSc since its inception. It allows users to control many aspects of the simulation avoiding the tedious edit–compile–run cycle. PetIGA builds on top of this infrastructure to support runtime specification of grid size, basis functions, and quadrature rule, as well as inspecting, monitoring, and input/output. The option `-iga_view` displays various details related to the discretization and parallel partitioning. From the first output line, we verify the problem is two-dimensional and the solution has a single component. The second and third line provide information about the discretization along each direction: the problem is being solved in a grid with 128×128 elements (as requested through the `-iga_elements` option), quadratic C^1 B-spline basis functions, and Gauss–Legendre quadrature with three points per direction. The option `-snes_monitor` monitors the nonlinear solver convergence by printing the iteration number and the 2-norm of the global residual vector. Finally, the option `-ksp_type` configures the linear solver to use the conjugate gradient method. Many more options are available to control the nonlinear solver, the linear solver, and the preconditioner. These options can be listed by passing the `-help` option.

Thanks to the call to the `VecViewFromOptions` routine, we visualize the solution at runtime in an X11 window

```
$ mpiexec -n 4 ./Bratu -iga_elements 128 -output draw:x -draw_pause 5
```

This feature allows users to quickly check for a sensible solution, as seen in Fig. B.15. This is particularly useful while in the initial setup of a time-dependent problem. To obtain higher quality visualizations, the option value can be changed to dump a VTK file to disk, which can then be post-processed with tools such as ParaView [80] as shown in Fig. B.15

```
$ mpiexec -n 4 ./Bratu -iga_elements 128 -output vtk:Bratu.vts
$ ls -lh Bratu.vts
-rw-rw-r--. 1 user users 2.1M Jan  1 00:01 Bratu.vts
$ paraview Bratu.vts
```

Appendix C. Numerical differentiation comparison

In this section we present some benchmark results measuring the computational overhead of two numerical differentiation approaches discussed in Section 2.8. We base our tests on the Bratu problem described in Section 3. However, we now focus on the three-dimensional counterpart to make the problem more computationally challenging. We use a modified version of the code listing 1 presented in Appendix B: instead of solving the nonlinear problem, we compute the global Jacobian matrix and report the minimum wall clock time from five successive computations. These computations are performed with three different methods.

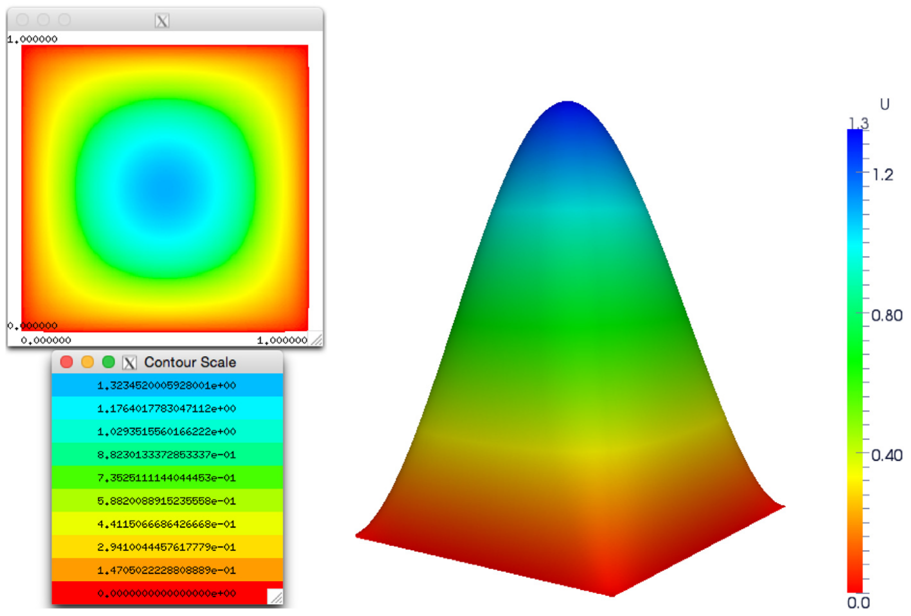
- *Explicit* uses the explicitly coded, user-provided Jacobian routine, as described in Section 3,
- *Coloring* uses the global colored finite differences implementation available in PETSc,
- *Local ND* uses the local (i.e., at quadrature points) numerical differentiation implementation available in PetIGA.

The benchmarks were run in a workstation with two Intel Xeon CPU E5-2680 v2 (10 cores, 2.80 GHz, 25 MB cache, 8 GT/s QPI) and 128 GB of memory running Linux 4.0.4 (Fedora 21) and used the in-development versions of PETSc and PetIGA compiled with GCC 4.9.2 using the `-Ofast` optimization flag.

Listing 1: C source code for the Bratu example

```
#include <petiga.h>

#define dim 2
#define dot(a,b) (a[0]*b[0]+a[1]*b[1])

typedef struct {
  double lambda;
} Params;

int Residual(IGAPoint p,const double U[],double R[],void *ctx) {
  int    a,nen      = p->nen;
  double (*N0)      = (typeof(N0)) p->shape[0];
  double (*N1)[dim] = (typeof(N1)) p->shape[1];
  double u,grad_u[dim],lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  IGAPointFormGrad (p,U,grad_u);
  for (a=0; a<nen; a++)
    R[a] = dot(N1[a],grad_u) - lambda*exp(u)*N0[a];
  return 0;
}

int Jacobian(IGAPoint p,const double U[],double J[],void *ctx) {
  int    a,b,nen    = p->nen;
  double (*N0)      = (typeof(N0)) p->shape[0];
  double (*N1)[dim] = (typeof(N1)) p->shape[1];
  double u,lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  for (a=0; a<nen; a++)
    for (b=0; b<nen; b++)
      J[a*nen+b] = dot(N1[a],N1[b]) - lambda*exp(u)*N0[a]*N0[b];
  return 0;
}

int main(int argc, char *argv[]) {
  PetscInitialize(&argc,&argv,NULL,NULL);

  IGA iga;
  IGACreate(PETSC_COMM_WORLD,&iga);
  IGASetDim(iga,dim);
  IGASetDof(iga,1);
  IGASetFromOptions(iga);
  IGASetUp(iga);

  int direction,side;
  for (direction=0; direction<dim; direction++) {
    for (side=0; side<2; side++) {
      int field = 0; double value = 0.0;
      IGASetBoundaryValue(iga,direction,side,field,value);
    }
  }

  Params params;
  params.lambda = IGAGetOptReal(NULL,"-lambda",6.80);
  IGASetFormFunction(iga,Residual,&params);
  IGASetFormJacobian(iga,Jacobian,&params);

  SNES snes;
  IGACreateSNES(iga,&snes);
  SNESSetFromOptions(snes);

  Vec U;
  IGACreateVec(iga,&U);
  SNESSolve(snes,NULL,U);

  VecViewFromOptions(U,NULL,"-output");

  VecDestroy(&U);
  SNESDestroy(&snes);
  IGADestroy(&iga);
  PetscFinalize();
  return 0;
}
```

The benchmark results are summarized in Table C.3. Parameters P, n_{el}, p, and k refer to number of processes, number of elements, degree and continuity order of the polynomial space, respectively. All cases used full Gauss–Legendre quadrature with $p + 1$ quadrature points per direction. Taking the explicitly coded Jacobian computations as a baseline, colored finite differences is over an order of magnitude slower whereas the local numerical differentiation approach is only about four times slower.



```
1.3234520005928001e+00
1.1764017783047112e+00
1.0293515560166222e+00
8.8230133372853337e-01
7.3525111144044453e-01
5.8820088915235558e-01
4.4115066686426668e-01
2.9410044457617779e-01
1.4705022228808889e-01
0.0000000000000000e+00
```

(a) Visualizing at runtime. (b) Post-processing with ParaView.

Fig. B.15. Visualizing the solution of the Bratu equation.

Table C.3
Benchmark of numerical differentiation schemes. Parameters P and n_{el} denote number of processes and elements, respectively, while p and k denote polynomial degree and continuity order, respectively. We report the minimum wall clock time from five successive computations of the global Jacobian matrix corresponding to the 3D Bratu problem.

P	n_{el}	p	k	Time (s)		
				Explicit	Coloring	Local ND
8	32^3	2	0	0.31	7.67	1.21
			1	0.33	7.79	1.24
		3	2	3.71	105.08	15.27
16	64^3	2	0	1.31	34.44	5.15
			1	1.35	34.84	5.26
		3	2	15.41	452.67	64.93
	128^3	1	0	0.52	10.35	1.72
		2	1	10.74	307.06	41.46

References

[1] T.J.R. Hughes, J.A. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Comput. Methods Appl. Mech. Engrg. 194 (39–41) (2005) 4135–4195. http://dx.doi.org/10.1016/j.cma.2004.10.008.

[2] J.A. Cottrell, T.J.R. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward Integration of CAD and FEA, John Wiley & Sons, Ltd., 2009, http://dx.doi.org/10.1002/9780470749081.

[3] M. Bercovier, G. Berold, Solving design problems by integration of CAD and FEM software, in: B. Ford, F. Chatelin (Eds.), Proceedings of IFIP TC 2/WG 2.5 Working Conference on Problem Solving Environments for Scientific Computing, Sophia Antipolis, France, 17–21 June, 1985, North Holland, 1987, p. 309.

[4] A. Sheffer, T. Blacker, M. Bercovier, Steps towards smooth CAD–FEM integration, in: Proceedings of 6th International Conference on Numerical Grid Generation in Computational Field Simulations, 1998, pp. 705–714.

[5] A. Sheffer, M. Bercovier, CAD model editing and its applications (Ph.D. thesis), Hebrew University of Jerusalem, 1999.

[6] K. Höllig, Finite Element Methods with B-splines, Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, 2003, http://dx.doi.org/10.1137/1.9780898717532.

[7] H. Gomez, V.M. Calo, Y. Bazilevs, T.J.R. Hughes, Isogeometric analysis of the Cahn–Hilliard phase-field model, Comput. Methods Appl. Mech. Engrg. 197 (49–50) (2008) 4333–4352. http://dx.doi.org/10.1016/j.cma.2008.05.003.

[8] H. Gomez, T.J.R. Hughes, X. Nogueira, V.M. Calo, Isogeometric analysis of the isothermal Navier–Stokes–Korteweg equations, Comput. Methods Appl. Mech. Engrg. 199 (25–28) (2010) 1828–1840. http://dx.doi.org/10.1016/j.cma.2010.02.010.

[9] P. Vignal, L. Dalcin, D.L. Brown, N. Collier, V.M. Calo, An energy-stable convex splitting for the phase-field crystal equation, Comput. Struct. 158 (2015) 355–368. http://dx.doi.org/10.1016/j.compstruc.2015.05.029.

[10] D.J. Benson, Y. Bazilevs, M.-C. Hsu, T.J.R. Hughes, A large deformation, rotation-free, isogeometric shell, Comput. Methods Appl. Mech. Engrg. 200 (13–16) (2011) 1367–1378. http://dx.doi.org/10.1016/j.cma.2010.12.003.

[11] J. Kiendl, M.-C. Hsu, M.C.H. Wu, A. Reali, Isogeometric Kirchhoff–Love shell formulations for general hyperelastic materials, Comput. Methods Appl. Mech. Engrg. 291 (2015) 280–303. http://dx.doi.org/10.1016/j.cma.2015.03.010.

[12] R. Bouclier, T. Elguedj, A. Combescure, An isogeometric locking–free NURBS-based solid–shell element for geometrically nonlinear analysis, Internat. J. Numer. Methods Engrg. 101 (10) (2015) 774–808. http://dx.doi.org/10.1002/nme.4834.

[13] I. Akkerman, Y. Bazilevs, V.M. Calo, T.J.R. Hughes, S. Hulshoff, The role of continuity in residual-based variational multiscale modeling of turbulence, Comput. Mech. 41 (3) (2008) 371–378. http://dx.doi.org/10.1007/s00466-007-0193-7.

[14] I. Akkerman, Y. Bazilevs, C.E. Kees, M.W. Farthing, Isogeometric analysis of free-surface flow, J. Comput. Phys. 230 (11) (2011) 4137–4152. http://dx.doi.org/10.1016/j.jcp.2010.11.044. Special issue High Order Methods for CFD Problems.

[15] A. Buffa, J. Rivas, G. Sangalli, R. Vázquez, Isogeometric discrete differential forms in three dimensions, SIAM J. Numer. Anal. 49 (2) (2011) 818–844. http://dx.doi.org/10.1137/100786708.

[16] J.A. Evans, T.J.R. Hughes, Isogeometric divergence-conforming B-splines for the unsteady Navier–Stokes equations, J. Comput. Phys. 241 (2013) 141–167. http://dx.doi.org/10.1016/j.jcp.2013.01.006.

[17] L. Beirão da Veiga, A. Buffa, J. Rivas, G. Sangalli, Some estimates for h–p–k-refinement in isogeometric analysis, Numer. Math. 118 (2011) 271–305. http://dx.doi.org/10.1007/s00211-010-0338-z.

[18] J.A. Evans, Y. Bazilevs, I. Babuška, T.J.R. Hughes, n-Widths, sup–infs, and optimality ratios for the k-version of the isogeometric finite element method, Comput. Methods Appl. Mech. Engrg. 198 (21–26) (2009) 1726–1741. http://dx.doi.org/10.1016/j.cma.2009.01.021. Advances in Simulation-Based Engineering Sciences—Honoring J. Tinsley Oden.

[19] N. Collier, L. Dalcin, V.M. Calo, On the computational efficiency of isogeometric methods for smooth elliptic problems using direct solvers, Internat. J. Numer. Methods Engrg. 100 (8) (2014) 620–632. http://dx.doi.org/10.1002/nme.4769.

[20] N. Collier, D. Pardo, L. Dalcin, M. Paszynski, V.M. Calo, The cost of continuity: A study of the performance of isogeometric finite elements using direct solvers, Comput. Methods Appl. Mech. Engrg. 213–216 (2012) 353–361. http://dx.doi.org/10.1016/j.cma.2011.11.002.

[21] N. Collier, L. Dalcin, D. Pardo, V.M. Calo, The cost of continuity: Performance of iterative solvers on isogeometric finite elements, SIAM J. Sci. Comput. 35 (2) (2013) A767–A784. http://dx.doi.org/10.1137/120881038.

[22] R. Baxter, N.C. Hong, D. Gorissen, J. Hetherington, I. Todorov, The research software engineer, in: Digital Research Conference, Oxford, 2012, pp. 1–3.

[23] G. Wilson, Those who will not learn from history..., Comput. Sci. Eng. 10 (3) (2008) 5–6. http://dx.doi.org/10.1109/MCSE.2008.86.

[24] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, L. Curfman McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page [online], 2016. URL http://www.mcs.anl.gov/petsc.

[25] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, L. Curfman McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016. URL http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf.

[26] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.

[27] K. Rupp, M.G. Knepley, B.F. Smith, D.A. May, P. Sanan, Extreme-scale multigrid components within PETSc, ArXiv e-prints, 2016, 1–21. arXiv:1604.07163.

[28] S. Zampini, PCBDDC: a class of robust dual-primal methods in PETSc, SIAM J. Sci. Comput. (2016). Special Issue on CS&E Software, in press.

[29] L. Beirão da Veiga, L.F. Pavarino, S. Scacchi, O.B. Widlund, S. Zampini, Isogeometric BDDC preconditioners with deluxe scaling, SIAM J. Sci. Comput. 36 (3) (2014) A1118–A1139. http://dx.doi.org/10.1137/130917399.

[30] R.D. Falgout, U.M. Yang, hypre: A library of high performance preconditioners, in: P.M.A. Sloot, A.G. Hoekstra, C.J. Kenneth Tan, J.J. Dongarra (Eds.), Computational Science—ICCS 2002, in: Lecture Notes in Computer Science, vol. 2331, Springer, Berlin, Heidelberg, 2002, pp. 632–641. http://dx.doi.org/10.1007/3-540-47789-6_66.

[31] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, K.S. Stanley, An overview of the Trilinos project, ACM Trans. Math. Software 31 (3) (2005) 397–423. http://dx.doi.org/10.1145/1089014.1089021.

[32] P.R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Hybrid scheduling for the parallel solution of linear systems, Parallel Comput. 32 (2) (2006) 136–156. http://dx.doi.org/10.1016/j.parco.2005.07.004. Parallel Matrix Algorithms and Applications.

[33] W. Bangerth, R. Hartmann, G. Kanschat, deal.II—a general purpose object oriented finite element library, ACM Trans. Math. Software 33 (4) (2007) 24/1–-24/27. http://dx.doi.org/10.1145/1268776.1268779.

[34] A. Logg, K.-A. Mardal, G. Wells, Automated Solution of Differential Equations by the Finite Element Method, Springer, Berlin, Heidelberg, 2012.

[35] B.S. Kirk, J.W. Peterson, R.H. Stogner, G.F. Carey, `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations, Eng. Comput. 22 (3–4) (2006) 237–254. http://dx.doi.org/10.1007/s00366-006-0049-3.

[36] V.E. Sonzogni, A.M. Yommi, N.M. Nigro, M.A. Storti, A parallel finite element program on a beowulf cluster, Adv. Eng. Softw. 33 (7–10) (2002) 427–443. http://dx.doi.org/10.1016/S0965-9978(02)00059-5. Engineering Computational Technology & Computational Structures Technology.

[37] L. Dalcin, N. Collier, PetIGA: High performance isogeometric analysis [online], 2016. URL https://bitbucket.org/dalcinl/petiga.
[38] C. de Falco, A. Reali, R. Vázquez, GeoPDEs: A research tool for isogeometric analysis of PDEs, Adv. Eng. Softw. 42 (12) (2011) 1020–1034. http://dx.doi.org/10.1016/j.advengsoft.2011.06.010.
[39] M.S. Pauletti, M. Martinelli, N. Cavallini, P. Antolin, Igatools: An isogeometric analysis library, SIAM J. Sci. Comput. 37 (4) (2015) C465–C496. http://dx.doi.org/10.1137/140955252.
[40] B. Jüttler, U. Langer, A. Mantzaflaris, S.E. Moore, W. Zulehner, Geometry + simulation modules: Implementing isogeometric analysis, PAMM 14 (1) (2014) 961–962. http://dx.doi.org/10.1002/pamm.201410461.
[41] R. Cimrman, SfePy—write your own FE application, in: P. de Buyl, N. Varoquaux (Eds.), Proceedings of the 6th European Conference on Python in Science, EuroSciPy 2013, 2014, pp. 65–70. URL http://arxiv.org/abs/1404.6391.
[42] R. Cimrman, Enhancing SfePy with isogeometric analysis, in: P. de Buyl, N. Varoquaux (Eds.), Proceedings of the 7th European Conference on Python in Science, EuroSciPy 2014, 2014, pp. 65–72. URL http://arxiv.org/abs/1412.6407.
[43] MFEM Web page [online], 2016. URL http://mfem.org/.
[44] M.G. Cox, The numerical evaluation of B-splines, IMA J. Appl. Math. 10 (2) (1972) 134–149. http://dx.doi.org/10.1093/imamat/10.2.134.
[45] C. de Boor, On calculation with B-splines, J. Approx. Theory 6 (1) (1972) 50–62. http://dx.doi.org/10.1016/0021-9045(72)90080-9.
[46] L. Piegl, W. Tiller, The NURBS Book, in: Monographs in Visual Communication, Springer, 1995.
[47] J. Liu, L. Dedè, J.A. Evans, M.J. Borden, T.J.R. Hughes, Isogeometric analysis of the advective Cahn-Hilliard equation: Spinodal decomposition under shear flow, J. Comput. Phys. 242 (2013) 321–350. http://dx.doi.org/10.1016/j.jcp.2013.02.008.
[48] D.A. Knoll, D.E. Keyes, Jacobian-free Newton–Krylov methods: a survey of approaches and applications, J. Comput. Phys. 193 (2) (2004) 357–397. http://dx.doi.org/10.1016/j.jcp.2003.08.010.
[49] A.H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Rev. 47 (4) (2005) 629–705. http://dx.doi.org/10.1137/S0036144504444711.
[50] M. Pernice, H.F. Walker, NITSOL: A Newton iterative solver for nonlinear systems, SIAM J. Sci. Comput. 19 (1998) 302–318.
[51] T.J.R. Hughes, A. Reali, G. Sangalli, Efficient quadrature for NURBS-based isogeometric analysis, Comput. Methods Appl. Mech. Engrg. 199 (5–8) (2010) 301–313.
[52] R. Ait-Haddou, M. Bartoň, V.M. Calo, Explicit Gaussian quadrature rules for cubic splines with non-uniform knot sequences, ArXiv e-prints, 2014, 1–15. arXiv:1410.7196.
[53] M. Bartoň, R. Ait-Haddou, V.M. Calo, Gaussian quadrature rules for C^1 quintic splines, ArXiv e-prints, 2015, pp. 1–20. arXiv:1503.00907.
[54] M. Bartoň, V.M. Calo, Gaussian quadrature for splines via homotopy continuation: rules for C^2 cubic splines, ArXiv e-prints, 2015, pp. 1–22. arXiv:1505.04391.
[55] G.E. Karniadakis, S.J. Sherwin, Spectral/hp Element Methods for Computational Fluid Dynamics, second ed., Oxford University Press, 2013.
[56] F. Auricchio, L. Beirão da Veiga, T.J.R. Hughes, A. Reali, G. Sangalli, Isogeometric collocation methods, Math. Models Methods Appl. Sci. 20 (11) (2010) 2075–2107. http://dx.doi.org/10.1142/S0218202510004878.
[57] L. Dalcin, N. Collier, igakit [online], 2016. URL https://bitbucket.org/dalcinl/igakit.
[58] N. Collier, L. Dalcin, PetIGA and igakit tutorial [online], 2016. URL https://petiga-igakit.readthedocs.org.
[59] J. Jacobsen, K. Schmitt, The Liouville-Bratu-Gelfand problem for radial operators, J. Differential Equations 184 (1) (2002) 283–298. http://dx.doi.org/10.1006/jdeq.2001.4151.
[60] W. Schroeder, K. Martin, B. Lorensen, The Visualization Toolkit, fourth ed., Kitware Inc., 2006.
[61] VTK File Formats, www.vtk.org/VTK/img/file-formats.pdf, 2016.
[62] L.M. Bernal, V.M. Calo, N. Collier, G.A. Espinosa, F. Fuentes, J.C. Mahecha, Isogeometric analysis of hyperelastic materials using PetIGA, Procedia Comput. Sci. 18 (2013) 1604–1613. http://dx.doi.org/10.1016/j.procs.2013.05.328. 2013 International Conference on Computational Science.
[63] J.C. Simo, T.J.R. Hughes, Computational Inelasticity, Springer, 1998.
[64] O. Gonzalez, A.M. Stuart, A First Course in Continuum Mechanics, Cambridge University Press, 2008.
[65] K.E. Jansen, C.H. Whiting, G.M. Hulbert, A generalized-α method for integrating the filtered Navier–Stokes equations with a stabilized finite element method, Comput. Methods Appl. Mech. Engrg. 190 (3–4) (2000) 305–319. http://dx.doi.org/10.1016/S0045-7825(00)00203-6.
[66] Y. Bazilevs, V.M. Calo, J.A. Cottrell, T.J.R. Hughes, A. Reali, G. Scovazzi, Variational multiscale residual-based turbulence modeling for large eddy simulation of incompressible flows, Comput. Methods Appl. Mech. Engrg. 197 (1–4) (2007) 173–201. http://dx.doi.org/10.1016/j.cma.2007.07.016.
[67] Y.G. Motlagh, H.T. Ahn, T.J.R. Hughes, V.M. Calo, Simulation of laminar and turbulent concentric pipe flows with the isogeometric variational multiscale method, Comput. & Fluids 71 (2013) 146–155. http://dx.doi.org/10.1016/j.compfluid.2012.09.006.
[68] Texas Advanced Computing Center [online], 2016. URL http://www.tacc.utexas.edu/.
[69] M. Woźniak, K. Kuźnik, M. Paszyński, V.M. Calo, D. Pardo, Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers, Comput. Math. Appl. 67 (10) (2014) 1864–1883. http://dx.doi.org/10.1016/j.camwa.2014.03.017.
[70] R. Yokota, J. Pestana, H. Ibeid, D. Keyes, Fast multipole preconditioners for sparse matrices arising from elliptic equations, Comput. Res. Repository (2014) 1–32. arXiv:1308.3339.
[71] N. Collier, A.-L. Haji-Ali, F. Nobile, E. von Schwerin, R. Tempone, A continuation multilevel Monte Carlo algorithm, BIT 55 (2) (2015) 399–432. http://dx.doi.org/10.1007/s10543-014-0511-3.
[72] H. Casquero, C. Bona-Casas, H. Gomez, A NURBS-based immersed methodology for fluid–structure interaction, Comput. Methods Appl. Mech. Engrg. 284 (2015) 943–970. http://dx.doi.org/10.1016/j.cma.2014.10.055. Isogeometric Analysis Special Issue.
[73] S. Rudraraju, A. Van der Ven, K. Garikipati, Three-dimensional isogeometric solutions to general boundary value problems of toupin's gradient elasticity theory at finite strains, Comput. Methods Appl. Mech. Engrg. 278 (2014) 705–728. http://dx.doi.org/10.1016/j.cma.2014.06.015.

[74] P. Vignal, A. Sarmiento, A.M.A. Côrtes, L. Dalcin, V.M. Calo, Coupling Navier-Stokes and Cahn-Hilliard equations in a two-dimensional annular flow configuration, Procedia Comput. Sci. 51 (2015) 934–943. http://dx.doi.org/10.1016/j.procs.2015.05.228. International Conference On Computational Science, {ICCS} 2015 Computational Science at the Gates of Nature.

[75] L.F.R. Espath, A.F. Sarmiento, P. Vignal, B.O.N. Varga, A.M.A. Cortes, L. Dalcin, V.M. Calo, Energy exchange analysis in droplet dynamics via the Navier–Stokes–Cahn–Hilliard model, J. Fluid Mech. 797 (2016) 389–430. http://dx.doi.org/10.1017/jfm.2016.277. arXiv:1512.02249.

[76] T.E. Oliphant, A Guide to NumPy, Trelgol Publishing, 2006.

[77] J.D. Hunter, Matplotlib: A 2D graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95. http://dx.doi.org/10.1109/MCSE.2007.55.

[78] F. Pérez, B.E. Granger, IPython: a system for interactive scientific computing, Comput. Sci. Eng. 9 (3) (2007) 21–29. http://dx.doi.org/10.1109/MCSE.2007.53.

[79] T.J.R. Hughes, The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Dover Publications, 2000.

[80] A. Henderson, ParaView Guide, A Parallel Visualization Application, Tech. Rep. Revision 4.1, Kitware Inc., 2014.