The surrogate matrix methodology: A reference implementation for low-cost assembly in isogeometric analysis

Daniel Drzisga*, Brendan Keith, Barbara Wohlmuth

Lehrstuhl für Numerische Mathematik, Fakultät für Mathematik (M2), Technische Universität München, Garching bei München

Abstract

A reference implementation of a new method in isogeometric analysis (IGA) is presented. It delivers low-cost variable-scale approximations (surrogates) of the matrices which IGA conventionally requires to be computed by element-scale quadrature. To generate surrogate matrices, quadrature must only be performed on a fraction of the elements in the computational domain. In this way, quadrature determines only a subset of the entries in the final matrix. The remaining matrix entries are computed by a simple B-spline interpolation procedure. We present the modifications and extensions required for a reference implementation in the open-source IGA software library GeoPDEs. The exposition is fashioned to help facilitate similar modifications in other contemporary software libraries.

Method name: Surrogate matrix method for isogeometric analysis
Keywords: Surrogate numerical methods, isogeometric analysis, high order, reference implementation

Specification Table

Subject Area	Mathematics
More specific subject area:	Computational science & engineering
Method name:	Surrogate matrix method for isogeometric analysis
Name and reference of original method:	T. J. Hughes, J. A. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Comput. Methods Appl. Mech. Eng. 194 (2005) 41354195. S. Bauer, M. Mohr, U. Rüde, J. Weismüller, M. Wittmann, B. Wohlmuth, A two-scale approach for efficient on-the-fly operator assembly in massively parallel high performance multigrid codes, Appl. Numer. Math. 122 (2017) 1438. D. Drzisga, B. Keith, B. Wohlmuth, The surrogate matrix methodology: Low-cost assembly for isogeometric analysis, arXiv preprint (2019).
Resource availability:	Source code (Matlab) for full method implementation in GitHub repository (https://dx.doi.org/10.5281/zenodo.3402341).

Method details

In [1], we applied the surrogate matrix methodology to isogeometric analysis (IGA) in order to avoid *over-assembling* mass, stiffness, and divergence matrices. While doing so, we also showed how this methodology could be applied to other system matrices arising in IGA computations. In that article, large performance

*Corresponding author
Email addresses: drzisga@ma.tum.de (Daniel Drzisga), keith@ma.tum.de (Brendan Keith), wohlmuth@ma.tum.de (Barbara Wohlmuth)

gains were demonstrated for a number of important PDE scenarios while still maintaining solution accuracy. In the present article, we provide an accompanying reference implementation, together with explanations and examples. This should be considered an extension to the main theoretical article [1], which we strongly recommend to read beforehand.

Recall that [1] focuses on the Galerkin form of IGA [2, 3]. The resulting surrogate matrix methods perform quadrature for only a small fraction of the NURBS basis function interactions during assembly and then *approximate the rest by interpolation*. This leads to large sparse linear system matrices where the majority of entries have been computed by interpolation. Such interpolated matrices will generally not coincide with those which would otherwise be generated by performing quadrature for the complete basis, or on every element, but they can be interpreted as cost-effective surrogates for them.

This idea was first introduced in the context of first-order finite elements and massively parallel simulations by Bauer et al. in [4]. There, the computational run time improvement results from a two-scale strategy after which the method was named. Thereafter, several subsequent investigations were initiated [1, 5–7], of which this paper may be seen as a product. In particular, massively parallel applications in geodynamical simulations are presented in [5, 6] and a theoretical analysis for first order finite element methods is given in [7]. For moderately sized problems a two-scale strategy will possibly not result in an optimal performance gain. Therefore, we exploit a mesh-size dependent macro-element approach where the macro-mesh is closer related to the fine mesh.

In this article, we restrict our attention to Poisson's boundary value problem on a single patch geometry $\Omega = \boldsymbol{\varphi}(\widehat{\Omega})$, where $\widehat{\Omega} = (0,1)^n$, $\boldsymbol{\varphi} : \widehat{\Omega} \to \mathbb{R}^n$ is a fixed diffeomorphism of sufficient regularity, and $n = 2, 3$. Represented on the reference domain $\widehat{\Omega}$, the bilinear form appearing in the standard weak form of this problem is defined

$$\widehat{a}(\widehat{w}, \widehat{v}) = \int_{\widehat{\Omega}} \widehat{\nabla}\widehat{w}(\widehat{\boldsymbol{x}})^\top K(\widehat{\boldsymbol{x}})\, \widehat{\nabla}\widehat{v}(\widehat{\boldsymbol{x}})\, \mathrm{d}\widehat{\boldsymbol{x}}\,, \qquad K = \frac{D\boldsymbol{\varphi}^{-1} D\boldsymbol{\varphi}^{-\top}}{|\det(D\boldsymbol{\varphi}^{-1})|}\,,$$

for arbitrary $\widehat{w}, \widehat{v} \in H^1(\widehat{\Omega})$. Adopting further notation from [1], this reference domain description leads to the definition of a set of smooth *stencil functions* $\Phi_{\boldsymbol{\delta}} \colon \widehat{\Omega} \supsetneq \widetilde{\Omega} \to \mathbb{R}$, $\boldsymbol{\delta} \in \mathcal{D}$. In turn, these functions can be interpolated onto a finite-dimensional space of multivariate B-splines using their values sampled from a subset $\{\widetilde{\boldsymbol{x}}_i^{\mathrm{s}}\} \subseteq \widetilde{\Omega} \cap \{\widetilde{\boldsymbol{x}}_i\}$ of a lattice $\{\widetilde{\boldsymbol{x}}_i\}$. Such interpolants are called *surrogate stencil functions* $\widetilde{\Phi}_{\boldsymbol{\delta}} \colon \widetilde{\Omega} \to \mathbb{R}$. Taking on the convention $\boldsymbol{\delta} = \widetilde{\boldsymbol{x}}_j - \widetilde{\boldsymbol{x}}_i$, we define the surrogate stiffness matrix $\widetilde{\mathsf{A}} \in \mathbb{R}^{N \times N}$ as follows:

$$\widetilde{\mathsf{A}}_{ij} = \begin{cases} \widetilde{\Phi}_{\boldsymbol{\delta}}(\widetilde{\boldsymbol{x}}_i) & \text{if } \widetilde{\boldsymbol{x}}_i, \widetilde{\boldsymbol{x}}_j \in \widetilde{\Omega} \text{ and } i < j, \\ \widetilde{\mathsf{A}}_{ji} & \text{if } \widetilde{\boldsymbol{x}}_i, \widetilde{\boldsymbol{x}}_j \in \widetilde{\Omega} \text{ and } i > j, \\ \mathsf{A}_{ij} & \text{in all other cases where } i \neq j, \\ -\sum_{k \neq i} \widetilde{\mathsf{A}}_{ik} & \text{if } i = j. \end{cases}$$

This definition preserves the symmetry and the kernel of the true IGA stiffness matrix $\mathsf{A} \in \mathbb{R}^{N \times N}$.

The reference implementation described in this paper is built on the GeoPDEs package for Isogeometric Analysis in Matlab and Octave [8, 9]. This package provides a framework for implementing and testing new isogeometric methods for PDEs. Our reference implementation is available in the git repository [10], which is itself a fork of the GeoPDEs repository. It is important to note that similar modifications can be made to other present-day software libraries and so the implementation presented in this article should foremost serve as a template. For this reason, we tried to find a balance between comprehensibility and performance. Most notably, the performance of our implementation may be greatly improved by performing quadrature for individual basis function interactions instead of element-wise quadrature. However, for most IGA software libraries, this feature would require significant software restructuring.

Before continuing, let us establish some more mathematical notation which appears in the article, cf. [1]. Let n be the space dimension, p be the maximum degree of the discrete IGA B-spline spaces, and h be the mesh size. By M, we denote the skip parameter which defines the sampling length $H = M \cdot h$ on which we perform interpolation of the stencil functions. Let q be the spline degree of the interpolation space. We

assume that the number of elements in each spatial dimension is always equal and we denote its value by N_{el}. Finally, recall that B-spline and NURBS bases are built from tensor products of univariate B-splines. For convenience, we assume that the associated number of knots and the B-spline degrees do not depend on the Cartesian coordinates.

Implementation

Our implementation preserves the local element quadrature approach present in most standard IGA and finite element software. This is not optimally efficient, but performance advantages can still be easily achieved because quadrature is usually not required on every element. In order to avoid performing quadrature on specific elements, we made some minor modifications to the following core functions in GeoPDEs: `msh_evaluate_col.m`[1], `@sp_scalar/sp_evaluate_col.m`[2], and `@sp_scalar/sp_evaluate_col_param.m`[3].[4] In addition to these minor modifications, we added two functions which handle the assembly of the surrogate stiffness matrices in 2D and 3D, respectively. Namely, we added `op_gradu_gradv_surrogate_2d`[5] and `op_gradu_gradv_surrogate_3d`[6]. These functions are based on `@sp_scalar/op_gradu_gradv_tp`[7] wherein the global matrix is assembled in a column-wise fashion [9]. In this section, we outline the modifications made to these core routines and describe the new 2D function in detail by breaking it down into coherent pieces of code.

Code modifications

In GeoPDEs, column-wise assembly which exploits the tensor product structure in IGA is used for performance [9]. For instance, in 2D, the functions listed above are called for each column of elements[8] in a patch during the column-wise assembly of the global stiffness matrix. Here, the function `msh_evaluate_col` collects the quadrature rules for the physical domain in a column and the function `sp_evaluate_col` returns a `struct` variable representing the discrete function space in that column.

Let us denote the elements where quadrature in necessary as *active elements*, cf. Fig. 1. We added the possibility to perform quadrature on only a subset of the elements in a given column by providing a list of indices called a *mask* for the active elements. We implemented this feature by extending the core functions mentioned above by an additional optional argument named `element_mask` which, as the name suggests, consists of a cell array with vectors of indices of active elements. Particularly, the `element_mask` is a cell array with a mask for the second dimension in the first component and, if required, contains an additional mask for the third dimension in the second component.

Some typical patterns of active elements are depicted in Fig. 1. Here, we have illustrated two different element masks corresponding to the cases $p = 2, 3$, $M = 10$, and $N_{\text{el}} = 39$. Let us focus on the case $p = 2$. Here, the first and last four columns correspond to near-boundary elements. That is, each whole column of elements is made up of active elements; cf. the red column in Fig. 1. In this scenario, we should employ unmodified function calls to `msh_evaluate_col` and `sp_evaluate_col`.

Another scenario is when only the top and bottom near-boundary elements in a column are active; cf. the green elements in Fig. 1. Here, the element mask is given by the set $\mathcal{B} = \{1, \dots, 2p\} \cup \{N_{\text{el}} - (2p-1), \dots, N_{\text{el}}\}$, which yields the element mask $\{1, 2, 3, 4, 36, 37, 38, 39\}$ for $p = 2$.

In the third and final scenario, both elements in the interior and near the top and bottom boundary are active; cf. the blue elements in Fig. 1. Each interior active element patch consists of $p+1$ elements and we start

[1] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/msh/%40msh_cartesian/msh_evaluate_col.m

[2] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/sp_evaluate_col.m

[3] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/sp_evaluate_col_param.m

[4] The interested reader may refer to the diff between the surrogate and the GeoPDEs master branch in order to view the precise modifications.

[5] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/op_gradu_gradv_surrogate_2d.m

[6] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/op_gradu_gradv_surrogate_3d.m

[7] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/op_gradu_gradv_tp.m

[8] In 3D, each column in the first dimension corresponds to a plane in the remaining dimensions, thus multiple element masks are required to cover all the active elements.

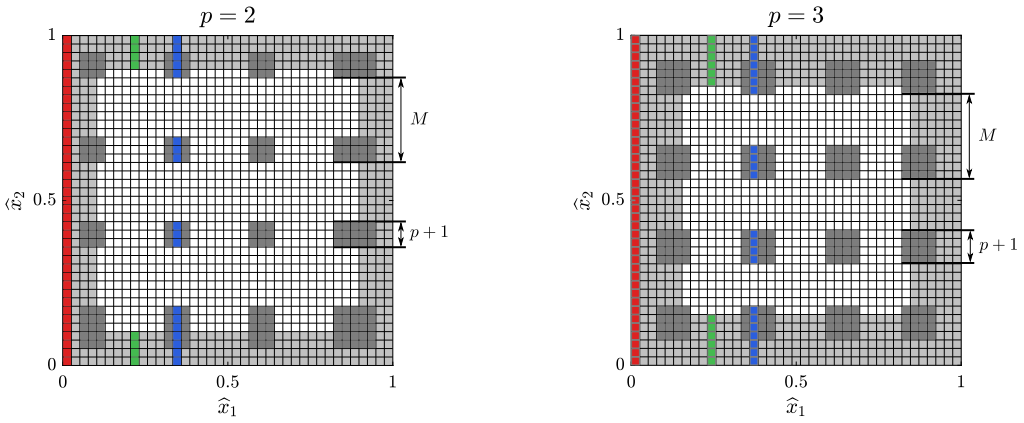

Figure 1: The active elements (shown in gray) involved in the surrogate assembly for $M = 10$ with forty knots in each Cartesian direction. The light gray elements correspond to the active boundary elements and the dark gray elements correspond to the inner active elements required for the sampling of the stencil functions. The red, green, and blue elements correspond to the three different active element masks.

sampling at the $p+1$ element with an increment of M. Since we want to avoid extrapolation in the subsequent spline interpolation, we also include the active elements near the interior boundary. With $K = N_{\text{el}} - 3p - 1$, the interior indices are thus given by $\mathcal{I} = \bigcup_{0 \leq k \leq \lfloor \frac{K}{M} \rfloor} \{kM+p+1, \ldots, kM+2p+1\} \cup \{K+p+1, \ldots, K+2p+1\}$. The final element mask is obtained by also including the near-boundary elements, i.e., $\mathcal{I} \cup \mathcal{B}$. In our case $p = 2$, the element mask is described by the set $\{1, 2, 3, 4, 5, 13, 14, 15, 23, 24, 25, 33, 34, 35, 36, 37, 38, 39\}$.

Code extensions

Below is the signature of `op_gradu_gradv_surrogate_2d`. This function has the following input arguments: a `space` of type `sp_scalar`, a `mesh` of type `msh_cartesian`, a function handle `coeff` of the coefficient, the skip paramater `M` and the surrogate interpolation degree `q`. The final surrogate matrix `K_surr` (in sparse format) is the sole output.

```
36 function [K_surr] = op_gradu_gradv_surrogate_2d(space, msh, coeff, M, q)
```

At the beginning of the function, some self-explanatory sanity checks are performed which verify that the correct input parameters have been passed.

```
38 if msh.ndim ~= 2
39   error('op_gradu_gradv_surrogate_2d: This function only supports 2D');
40 end
41
42 if ~all(space.ndof_dir == space.ndof_dir(1))
43   error('op_gradu_gradv_surrogate_2d: Dofs in each dimension must be equal');
44 end
45
46 if q == 1
47   method = 'linear';
48 elseif q == 3
49   method = 'spline';
50 else
51   error('op_gradu_gradv_surrogate_2d: q = %d is not supported', q);
52 end
```

In the next few lines, some helper variables holding the number of degrees of freedom (dofs) in the univariate B-spline basis are defined. The total number may be expressed as $N_{\text{el}} + p$. Removing $2p$ dofs from both the left and right boundary yields the number of interior dofs $N_{\text{el}} - 3p$ present in one Cartesian direction in the stencil function domain.

```
iga_degree = min(space.degree);
num_1D_basis = space.ndof_dir(1);
num_1D_basis_inner = num_1D_basis - 4 * iga_degree;
```

The following lines compute the indices of the rows in the global stiffness matrix corresponding to the stencil function domain. This is done by collecting all available indices and removing the $4p$ indices nearest the boundary in each dimension. In the 3D case, the array `row_indices` has 3 dimensions.

```
row_indices = 1:num_1D_basis;
row_indices = repmat(row_indices, num_1D_basis, 1);
for i = 2:num_1D_basis
  row_indices(i,:) = row_indices(i-1,:) + num_1D_basis;
end
ind = 2*iga_degree+1:num_1D_basis-2*iga_degree;
row_indices = row_indices(ind, ind);
```

The next step is computing the sample point coordinates in the interior of the stencil function domain $\widetilde{\Omega}$ described in [1]. For practical reasons, we map this domain to the unit domain $[0,1]^n$ and take every M^{th} point in each direction. To avoid extrapolating any values later, we reinsert the sample points on boundary of $[0,1]^n$ which may have been skipped.

```
x = linspace(0, 1, num_1D_basis_inner);
[X, Y] = meshgrid(x);
ind = unique([1:M:num_1D_basis_inner, num_1D_basis_inner]);
X_sample  = X(ind, ind);
Y_sample  = Y(ind, ind);
```

Prior to computing the element mask, we save the indices of the matrix rows corresponding to the sample points `row_indices_subset`. These indices are required in order to determine the active elements with the GeoPDEs function `sp_get_cells`. In order to make the coming call to `sp_get_cells` faster, we filter the `row_indices_subset` to only include the rows up to index $(2p+1)(N_{\text{el}}+p)$. The last line filtering the rows is optional and may be omitted.

```
row_indices_subset = row_indices(ind, ind);
row_indices_subset = row_indices_subset(:);
row_indices_subset = row_indices_subset(row_indices_subset <= (2*iga_degree + 1) * num_1D_basis);
```

The following snippet sets up an element mask which skips the quadrature at all non-active elements. Here, we employ the `sp_get_cells` function which returns the indices of all elements within the support of the basis functions corresponding to the rows filtered in the previous snippet. Some additional filtering enforces that only the indices up to N_{el} are included. Since quadrature needs to be performed on all of the remaining near-boundary elements, we ensure that the elements with first or second indices in $\mathcal{B}$ are also active.

```
element_mask{1} = sp_get_cells(space, msh, row_indices_subset');
element_mask{1} = element_mask{1} - iga_degree * msh.nel_dir(1);
element_mask{1} = element_mask{1}(element_mask{1} <= msh.nel_dir(1));
element_mask{1} = [(1:(2*iga_degree))'; element_mask{1};
 ↪  ((msh.nel_dir(1)-(2*iga_degree-1)):msh.nel_dir(1))'];
element_mask{1} = unique(element_mask{1});
```

Additionally, a second mask with only the near-boundary element indices is required. Clearly, this mask only includes the indices in $\mathcal{B}$.

```
boundary_mask{1} = [1:(2*iga_degree), (msh.nel_dir(1)-(2*iga_degree-1)):msh.nel_dir(1)];
boundary_mask{1} = unique(boundary_mask{1});
```

For performance reasons, we pre-allocate memory for the sparse matrix with an estimated number of non-zero entries per row.

```
K_surr = spalloc(space.ndof, space.ndof, (2*iga_degree + 1)^2*space.ndof);
```

Below is the first main loop which only performs quadrature on the active elements; i.e., those required for the subsequent stencil interpolation. It is very similar to the original column-wise loop in `op_gradu_gradv_tp`, but it employs the modified `msh_evaluate_col` and `sp_evaluate_col` (see previous subsection) with the previously determined masks.

```
for iel = 1:msh.nel_dir(1)
  if ismember(iel, boundary_mask{1}) % Case corresponding to the red elements in the article
    msh_col = msh_evaluate_col(msh, iel);
    sp_col = sp_evaluate_col(space, msh_col, 'value', false, 'gradient', true);
  elseif ismember(iel, element_mask{1}) % Case corresponding to the blue elements in the article
    msh_col = msh_evaluate_col(msh, iel, 'element_mask', element_mask);
    sp_col = sp_evaluate_col(space, msh_col, 'value', false, 'gradient', true, 'element_mask',
     ↪  element_mask);
  else % Case corresponding to the green elements in the article
    msh_col = msh_evaluate_col(msh, iel, 'element_mask', boundary_mask);
    sp_col = sp_evaluate_col(space, msh_col, 'value', false, 'gradient', true, 'element_mask',
     ↪  boundary_mask);
  end

  for idim = 1:msh.rdim
    coords{idim} = reshape(msh_col.geo_map(idim,:,:), msh_col.nqn, msh_col.nel);
  end
  coeffs = coeff(coords{:});

  K_surr = K_surr + op_gradu_gradv(sp_col, sp_col, msh_col, coeffs);
end
```

We define three vectors, which will hold the columns, rows, and values of the surrogate matrix. This format allows for faster construction of a sparse matrix in a compressed sparse column format.

```
sp_i = [];
sp_j = [];
sp_v = [];
```

The following snippet contains the main loop of the surrogate assembly algorithm. In practice, there are $|\mathcal{D}| = (2p+1)^n$ surrogate stencil functions $\widetilde{\Phi}_{\boldsymbol{\delta}}(\cdot)$, so the statements in the inner-most loop are reached $(2p+1)^n$ times. First, the position (i.e., `shift`) of the stencil function relative to the diagonal entries in the matrix is computed. This "shift" is in one-to-one correspondence with the translations $\boldsymbol{\delta} \in \mathcal{D}$ described in [1]. Due to symmetry, we only need to compute the surrogates for the upper-diagonal matrix, thus we skip all cases where the shift is smaller or equal to zero by simply continuing to the next iteration. If the shift is larger than zero, we extract all the sample points from the associated rows of the partly assembled stiffness matrix. Having all the sample points $\widetilde{\boldsymbol{x}}_i^{\mathrm{s}}$ at hand, we can easily perform the interpolation of the remaining matrix entries $\widetilde{\Phi}_{\boldsymbol{\delta}}(\widetilde{\boldsymbol{x}}_i)$. In order to remove any dependence on external software, the built-in Matlab function `interp2` is used here. Note that this function only supports the spline interpolation orders $q = 1$ and $q = 3$. In [1], we used the `RectBivariateSpline` function provided by the SciPy Python package [11], which supports any spline interpolation up to order 5. In the last three lines, the rows, columns, and values of the surrogate matrix are added to the vectors required for the sparse matrix creation at the end of the routine. Note that the values of the upper-diagonal are added to the lower-diagonal in order to enforce symmetry.

```
for i=-iga_degree:iga_degree
  for j=-iga_degree:iga_degree
```

```
121
122     shift = i + num_1D_basis * j;
123
124     % Skip if not upper part in the symmetric case
125     if shift <= 0
126       continue;
127     end
128
129     % Obtain stencil function values for the sample points
130     stencilfunc = K_surr(sub2ind(size(K_surr), row_indices(:), row_indices(:) + shift));
131     stencilfunc = reshape(stencilfunc, num_1D_basis_inner, num_1D_basis_inner);
132     sf_sample = full(stencilfunc(ind, ind));
133
134     % Interpolate missing values
135     tmp = interp2(X_sample, Y_sample, sf_sample, X, Y, method);
136
137     % Add contribution to sparse vectors
138     sp_i = [sp_i, row_indices(:)', row_indices(:)' + shift];
139     sp_j = [sp_j, row_indices(:)' + shift, row_indices(:)'];
140     sp_v = [sp_v, tmp(:)', tmp(:)'];
141   end
142 end
```

For performance reasons, we pre-allocate the diagonal of the surrogate stiffness matrix with dummy values which are overwritten later.

```
145 sp_i = [sp_i, 1:length(K_surr)];
146 sp_j = [sp_j, 1:length(K_surr)];
147 sp_v = [sp_v, ones(1,length(K_surr))];
```

In the penultimate step, we combine the matrix components obtained through interpolation with those obtained through standard quadrature. This is done by creating a matrix `K_interp` with only the interpolated components. We then zero all of the entries in `K_surr` which have non-zero values in `K_interp` and finally sum the two matrices. This may seem complicated at first, but it yields good performance because the sum of both sparse matrices may be computed efficiently.

```
150 K_interp = sparse(sp_i, sp_j, sp_v, length(K_surr), length(K_surr));
151 idx_interp = find(K_interp);
152 idx_surr = find(K_surr);
153 idx_intersect = intersect(idx_interp,idx_surr);
154 K_surr(idx_intersect) = 0;
155 K_surr = K_surr + K_interp;
```

In the final step, the zero row-sum property is enforced by setting the diagonal value to the negative sum of the off-diagonal values in each row.

```
158 K_surr(logical(speye(size(K_surr)))) = diag(K_surr) - sum(K_surr, 2);
```

Remark 1. The `op_gradu_gradv_surrogate_3d` function is in many ways similar to its 2D counterpart. For instance, clearly some of the 2D arrays need to be changed to 3D arrays, the 3D interpolation function `interp3` needs to be used, and the number of stencil functions increases. However, performing quadrature only on the active elements is more difficult, since in the column-wise iteration GeoPDEs uses, each column is now made up of a matrix of elements (instead of a vector). Therefore, a much more complicated set of element masks need to be used; see lines 98 to 115 in `op_gradu_gradv_surrogate_3d.m`[9].

[9] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/space/%40sp_scalar/op_gradu_gradv_surrogate_3d.m

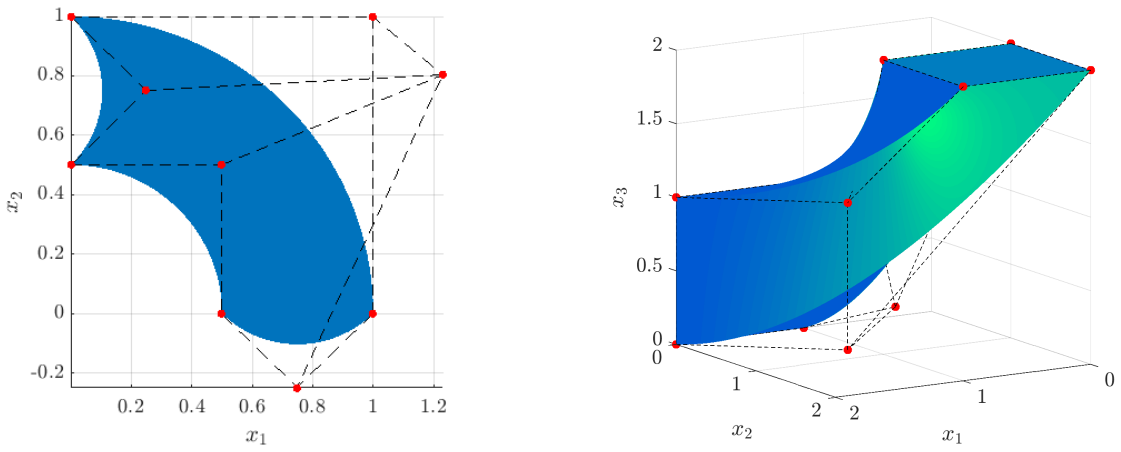

Figure 2: Left: Domain Ω considered in the 2D example. Right: Domain Ω considered in the 3D example.

Verification

To facilitate easy verification, we added the demo script `geopdes_surrogate_examples.m`[10], which provides the ability to run the problems considered in [1, Section 7.3]. The code for the 2D and 3D examples may be found in `ex_surrogate_poisson_2d.m`[11] and `ex_surrogate_poisson_3d.m`[12], respectively. Again, recall Poisson's problem

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= g \quad \text{on } \partial\Omega, \end{aligned}$$

on a domain $\Omega \subseteq \mathbb{R}^n$. The 2D and 3D domains we considered are depicted in Fig. 2 and their respective NURBS/B-spline descriptions are in `GeomQuarterAnnulusWithBumps.m`[13] and `GeomBentTwistedBox.m`[14]. In order to test the quality of the discrete solutions, we employ the following manufactured solution and load in 2D

$$\begin{aligned} u(x,y) &= \sin(20\,\pi\,x)\,\sin(20\,\pi\,y), \\ f(x,y) &= 800\,\pi^2\sin(20\,\pi\,x)\,\sin(20\,\pi\,y), \end{aligned}$$

and the following in 3D

$$\begin{aligned} u(x,y,z) &= \sin(20\,\pi\,x)\,\sin(20\,\pi\,y)\,\sin(20\,\pi\,z), \\ f(x,y,z) &= 1200\,\pi^2\sin(20\,\pi\,x)\,\sin(20\,\pi\,y)\,\sin(20\,\pi\,z). \end{aligned}$$

The Dirichlet datum g is chosen as the restriction of the manufactured solution to the boundary $\partial\Omega$.

Each example script assembles the standard IGA matrix A first and solves the corresponding system. Afterwards, the surrogate matrix $\widetilde{\mathsf{A}}$ is assembled and the surrogate system is solved. Finally, the relative L^2 and H^1 errors for each case are computed and written to the console. In addition, both the maximum absolute value of the difference in the two stiffness matrices (i.e., $\|\mathsf{A} - \widetilde{\mathsf{A}}\|_{\max}$) and the achieved speed-up are displayed.

In the following, we present the output of running the `ex_surrogate_poisson_2d` script in Matlab 2018b. The script ran on a workstation equipped with an Intel® Xeon® E5-1630 v3 processor with a nominal frequency of 3.7 GHz.

[10] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/examples/geopdes_surrogate_examples.m
[11] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/examples/surrogate/ex_surrogate_poisson_2d.m
[12] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/examples/surrogate/ex_surrogate_poisson_3d.m
[13] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/examples/surrogate/data_files/GeomQuarterAnnulusWithBumps.m
[14] https://github.com/drzisga/geopdes/blob/v3.1-surrogate/geopdes/inst/examples/surrogate/data_files/GeomBentTwistedBox.m

```
Initializing problem...
Assembling standard IGA matrix...
Standard assembly time: 5.022883 s
Solving standard IGA problem...
Standard solve time: 0.270706 s
Assembling surrogate IGA matrix...
Surrogate assembly time: 1.577257 s
Solving surrogate IGA problem...
Surrogate solve time: 0.273799 s
Computing errors...
Relative error in standard IGA
L2-norm: 1.335554e-03
H1-norm: 1.407778e-02

\|A-\tilde{A}\|_{\max} = 9.877796e-04

Relative error in surrogate IGA
L2-norm: 1.335619e-03
H1-norm: 1.407779e-02

Assembly speed-up: 218.46%
```

Remark 2. The default parameters in the demo script are $p = 2$, $q = 3$, and $M = 10$. The number of knots in each dimension is 160 in 2D and 40 in 3D. These parameters may be easily modified by resetting the appropriate variables at the beginning of each script.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 800898. This work was supported by the German Research Foundation (DFG) and the Technical University of Munich (TUM) within the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA), by grant WO671/11-1, and in the framework of the Open Access Publishing Program.

References

[1] D. Drzisga, B. Keith, B. Wohlmuth, The surrogate matrix methodology: Low-cost assembly for isogeometric analysis, arXiv preprint arXiv:1904.06971 (2019).

[2] J. A. Cottrell, T. J. Hughes, Y. Bazilevs, Isogeometric analysis: toward integration of CAD and FEA, John Wiley & Sons, 2009.

[3] T. J. Hughes, J. A. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Comput. Methods Appl. Mech. Eng. 194 (2005) 4135–4195.

[4] S. Bauer, M. Mohr, U. Rüde, J. Weismüller, M. Wittmann, B. Wohlmuth, A two-scale approach for efficient on-the-fly operator assembly in massively parallel high performance multigrid codes, Appl. Numer. Math. 122 (2017) 14–38.

[5] S. Bauer, M. Huber, S. Ghelichkhan, M. Mohr, U. Rüde, B. Wohlmuth, Large-scale simulation of mantle convection based on a new matrix-free approach, J. of Comput. Science 31 (2019) 60–76.

[6] S. Bauer, M. Huber, M. Mohr, U. Rüde, B. Wohlmuth, A new matrix-free approach for large-scale geodynamic simulations and its performance, in: Int. Conf. on Comput. Science, Springer, 2018, pp. 17–30.

[7] D. Drzisga, B. Keith, B. Wohlmuth, The surrogate matrix methodology: a priori error estimation, arXiv preprint arXiv:1902.07333 (2019).

[8] C. de Falco, A. Reali, R. Vázquez, GeoPDEs: a research tool for isogeometric analysis of PDEs, Adv. Eng. Software 42 (2011) 1020–1034.

[9] R. Vázquez, A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0, Comput. & Math. with Appl. 72 (2016) 523–554.

[10] Fork of the GeoPDEs git repository including the surrogate branch, https://dx.doi.org/10.5281/zenodo.3402341, 2019. doi:10.5281/zenodo.3402341, [Online; accessed September 7th 2019].

[11] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, http://www.scipy.org, 2001–. [Online; accessed February 11th 2019].