SUPPLEMENTARY MATERIALS: FELICITY: A MATLAB/C++ TOOLBOX FOR DEVELOPING FINITE ELEMENT METHODS AND SIMULATION MODELING*

SHAWN W. WALKER†

SM1. FELICITY Demos. We demonstrate how to implement the example problems in subsection 4.1 and subsection 4.2 using FELICITY. In order to run these examples, FELICITY must be installed properly (see the wiki [SM8] for more info).

SM1.1. Laplace-Beltrami With Weak Boundary Conditions. We outline the implementation of the problem in subsection 4.1; the full example can be found in the FELICITY sub-directory: `./Demo/Laplace_Beltrami_Open_Surface`. Executing the script `test_Laplace_Beltrami_Open_Surface` at the Matlab prompt will run this example. Note: this script will also *compile* the `mex` files that this demo uses.

This example illustrates the following features in FELICITY:

- Surface finite element methods.
- Higher order meshes, e.g. piecewise quadratic.
- Finite element spaces defined on different sub-domains. E.g. implementing bilinear/linear forms that involve finite element functions from two different FE spaces defined over different sub-domains.
- Access to domain (and sub-domain) geometric terms when defining forms.

SM1.1.1. Finite Element Formulation. Let $\mathcal{T}_h$ be a piecewise quadratic surface triangulation (of mesh size h) of a surface $\Gamma := \text{int}\left(\cup_{T\in\mathcal{T}_h}\overline{T}\right)$; we assume Γ approximates a smooth surface Γ_s with smooth boundary $\partial\Gamma_s$ parameterized by

$$\boldsymbol{\Psi}(q_1,q_2) = \left(q_1, q_2, \frac{1}{2}(q_1^2 - q_2^2)\right)^T, \quad \text{for all } (q_1,q_2) \in \{q_1^2+q_2^2<1\}; \tag{SM1}$$

see Figure 1. Recalling subsection 2.1.5, we define a geometric element space

$$\mathbf{G}_h(\widehat{\Gamma}) = \{\mathbf{v}\in[C^0(\widehat{\Gamma})]^3 : \mathbf{v}|_{\widehat{T}} = \boldsymbol{\eta}\circ\mathbf{F}_{\widehat{T}}^{-1}, \text{ for } \widehat{T}\in\widehat{\mathcal{T}}, \boldsymbol{\eta}\in[\mathcal{L}_2(T_{\text{ref}})]^3\}, \tag{SM2}$$

where $\widehat{\mathcal{T}}$ represents $\widehat{\Gamma}$ (a piecewise linear approximation of Γ_s). Similar to (7), we define any quadratic triangle $T\in\mathcal{T}_h$ via $T = \boldsymbol{\Phi}_T(T_{\text{ref}})$, $\boldsymbol{\Phi}_T := \mathbf{g}_h\circ\mathbf{F}_{\widehat{T}}$, where $\widehat{T}$ is the corresponding linear triangle and $\mathbf{g}_h\in\mathbf{G}_h(\widehat{\Gamma})$ captures (SM1).

Next, we introduce the following Lagrange FE spaces

$$\begin{aligned} V_h &= \{v_h\in C^0(\Gamma) : v_h|_T = \eta\circ\boldsymbol{\Phi}_T^{-1}, T\in\mathcal{T}_h, \eta\in\mathcal{P}_2(T_{\text{ref}})\},\\ W_h &= \{\mu_h\in C^0(\partial\Gamma) : \mu_h|_E = \varphi\circ\boldsymbol{\Phi}_E^{-1}, E\in\mathcal{E}_h, \varphi\in\mathcal{P}_1(E_{\text{ref}})\}, \end{aligned} \tag{SM3}$$

where $\mathcal{E}_h = \{E\}$ is the set of conforming (quadratic) boundary edges that make up $\partial\Gamma$, i.e. $\partial\Gamma := \cup_{E\in\mathcal{E}}\overline{E}$, and $\boldsymbol{\Phi}_E : E_{\text{ref}}\to E$ is defined as follows. Given $E\in\mathcal{E}_h$, there exists a $T(E)\in\mathcal{T}_h$ such that $E\subset\partial T(E)$; equivalently, there exists $E_0\subset\partial T_{\text{ref}}$ and $\mathbf{Z}_E : E_{\text{ref}}\to\partial T_{\text{ref}}$ such that $E_0 = \mathbf{Z}_E(E_{\text{ref}})$. Whence, we obtain $\boldsymbol{\Phi}_E := \boldsymbol{\Phi}_{T(E)}\circ\mathbf{Z}_E$.

*Supplementary materials for SISC MS#M112874.

Funding: The author acknowledges funding support from NSF grants DMS-1418994 and DMS-1555222.

†Department of Mathematics, Louisiana State University, Baton Rouge, LA (walker@math.lsu.edu).

With this, the discrete variational formulation of (12) follows. Given data f_h in V_h, g_h in V_h, find u_h in V_h and λ_h in W_h such that

$$\begin{aligned} (\nabla_\Gamma u_h, \nabla_\Gamma v_h)_\Gamma + (\lambda_h, v_h)_{\partial\Gamma} &= (f_h, v_h)_\Gamma \,, \quad \forall v_h \in V_h, \\ (\mu_h, u_h)_{\partial\Gamma} &= (\mu_h, g_h)_{\partial\Gamma} \,, \quad \forall \mu_h \in W_h, \end{aligned} \tag{SM4}$$

where $(u_h, v_h)_\Gamma = \int_\Gamma u_h v_h$ and $(\mu_h, v_h)_{\partial\Gamma} = \int_{\partial\Gamma} \mu_h v_h$.

SM1.1.2. Form Files. The abstract definition file for the forms in (SM4) is in the m-file: `MatAssem_LapBel_Open_Surface.m`, which is listed here:

```
function MATS = MatAssem_LapBel_Open_Surface()
%MatAssem_LapBel_Open_Surface

% define domain (2-D surface in 3-D)
Gamma  = Domain('triangle',3);
dGamma = Domain('interval') < Gamma; % subset

% define finite element spaces
V_h = Element(Gamma, lagrange_deg2_dim2); % piecewise quadratic
W_h = Element(dGamma,lagrange_deg1_dim1); % piecewise linear

% define functions in FE spaces
v_h = Test(V_h);
u_h = Trial(V_h);
mu_h = Test(W_h);

% define (discrete) forms
M = Bilinear(V_h,V_h);
M = M + Integral(Gamma, v_h.val * u_h.val );

K = Bilinear(V_h,V_h);
K = K + Integral(Gamma, v_h.grad' * u_h.grad );

B = Bilinear(W_h,V_h);
B = B + Integral(dGamma, mu_h.val * u_h.val );

% set the minimum number of quadrature points to use
Quadrature_Order = 10;
% define geometry representation:
%        Domain, piecewise quadratic representation
G_Space = GeoElement(Gamma,lagrange_deg2_dim2);
% define a set of matrices
MATS = Matrices(Quadrature_Order,G_Space);

% collect all of the matrices together
MATS = MATS.Append_Matrix(B);
MATS = MATS.Append_Matrix(K);
MATS = MATS.Append_Matrix(M);
```

The m-file defines the following three forms: `M` $\sim (u_h, v_h)_\Gamma$, `K` $\sim (\nabla_\Gamma u_h, \nabla_\Gamma v_h)_\Gamma$, and `B` $\sim (\mu_h, u_h)_{\partial\Gamma}$. Since `B` is an integral over $\partial\Gamma$ of functions defined on Γ and $\partial\Gamma$, the definition file specifies the two domains by

```
Gamma = Domain('triangle',3); % surface in 3-D
dGamma = Domain('interval') < Gamma; % subset of Gamma
```

i.e. `dGamma` is specified to be a *subset* of `Gamma`. Hence, FELICITY can automatically handle integrating the *trace* of a finite element function over a sub-domain.

Next, we have another definition file used to compute numerical errors to check convergence rates; see the m-file `Compute_Errors_LapBel_Open_Surface.m`, which is listed here:

```
function MATS = Compute_Errors_LapBel_Open_Surface()
%Compute_Errors_LapBel_Open_Surface

% define domain (2-D surface in 3-D)
Gamma = Domain('triangle',3);
dGamma = Domain('interval') < Gamma; % subset

% define finite element spaces
V_h = Element(Gamma, lagrange_deg2_dim2,1); % piecewise quadratic
W_h = Element(dGamma,lagrange_deg1_dim1,1); % piecewise linear
G_h = Element(Gamma, lagrange_deg2_dim2,3); % vector PW quadratic

% define functions in FE spaces
u_h = Coef(V_h);
lambda_h = Coef(W_h);

NV_h = Coef(G_h);
TV_h = Coef(G_h);

% geometry access functions
gf_Gamma = GeoFunc(Gamma);
gf_dGamma = GeoFunc(dGamma);

% define exact solns
u_exact = @(u,v) cos(v.*pi.*2.0).*sin(u.*pi.*2.0);
lambda_exact = <long formula>;

% define (discrete) forms
u_L2_Error_sq = Real(1,1);
u_L2_Error_sq = u_L2_Error_sq + Integral(Gamma,...
             (u_h.val - u_exact(gf_Gamma.X(1),gf_Gamma.X(2)))^2 );

lambda_L2_Error_sq = Real(1,1);
lambda_L2_Error_sq = lambda_L2_Error_sq + Integral(dGamma,...
 (lambda_h.val - lambda_exact(gf_dGamma.X(1),gf_dGamma.X(2)))^2 );

% check normal vector
NV_Error_sq = Real(1,1);
NV_Error_sq = NV_Error_sq + Integral(dGamma,...
              sum((gf_Gamma.N - NV_h.val).^2) );

% check tangent vector
TV_Error_sq = Real(1,1);
```

```
TV_Error_sq = TV_Error_sq + Integral(dGamma,...
              sum((gf_dGamma.T - TV_h.val).^2) );

% check Neumann data
xi_h = cross(gf_dGamma.T,gf_Gamma.N);
Neumann_L2_Error_sq = Real(1,1);
Neumann_L2_Error_sq = Neumann_L2_Error_sq + Integral(dGamma,...
                    ( lambda_h.val - ( -dot(xi_h,u_h.grad) ) )^2 );

% set the minimum order of accuracy for the quad rule
Quadrature_Order = 10;
% define geometry representation:
%      Domain, piecewise quadratic representation
G_Space = GeoElement(Gamma,lagrange_deg2_dim2);

% define a set of matrices
MATS = Matrices(Quadrature_Order,G_Space);

<similar conclusion code as before>
```

A few new items appear in this definition file: the geometry FE space $\mathbf{G}_h := [V_h]^3$, given *coefficient* functions `Coef` (analogous to `Test` and `Trial`), geometry access functions `GeoFunc`, and `Real` forms that are dense matrices where each entry is defined to be an integral of given data. Essentially, the definition file computes the following errors: (1) $\|u_h - u\|^2_{L^2(\Gamma)}$, where u is the exact solution given by $u(x, y) = \cos(2\pi x)\sin(2\pi y)$; (2) $\|\lambda_h - \lambda\|^2_{L^2(\partial\Gamma)}$, where λ is the corresponding exact solution; (3) $\|\boldsymbol{\nu}_h - I_h\boldsymbol{\nu}_s\|^2_{L^2(\partial\Gamma)}$, where $\boldsymbol{\nu}_s$ is the exact normal vector, I_h is the Lagrange interpolation operator of $\mathbf{G}_h$, and $\boldsymbol{\nu}_h$ is the discrete normal vector on Γ; (4) $\|\boldsymbol{\tau}_h - I_h\boldsymbol{\tau}_s\|^2_{L^2(\partial\Gamma)}$ (analogous to (3)); and (5) $\|\lambda_h - (-\boldsymbol{\xi}_h \cdot \nabla_\Gamma u_h)\|^2_{L^2(\partial\Gamma)}$, where $\boldsymbol{\xi}_h := \boldsymbol{\tau}_h \times \boldsymbol{\nu}_h$ (note that (5) measures the consistency of the Neumann data).

Getting access to the domain geometry is done by, first, including:

```
gf_Gamma = GeoFunc(Gamma);
gf_dGamma = GeoFunc(dGamma);
```

in the definition file. Then, when defining an `Integral`, we access the normal vector $\boldsymbol{\nu}_h$ and tangent vector $\boldsymbol{\tau}_h$ via `gf_Gamma.N` and `gf_dGamma.T`.

REMARK SM1.1 (access to geometry). *Other geometric quantities can be accessed [SM7] using* `gf_Gamma.<opt>`*, where* `<opt>` *can be* `Kappa` *(summed curvature),* `VecKappa` *(vector curvature),* `Kappa_Gauss` *(Gauss curvature), and* `Shape_Op` *(shape operator) [SM6]. The meaning of the different quantities depends on whether the domain is a surface or a curve. For instance, for a curve (e.g. $\partial\Gamma$),* `Kappa_Gauss` *is not valid and* `Kappa` *is the signed curvature; for a surface,* `Kappa` *is the sum of the two principle curvatures. Note that these options are only useful when a higher order representation of the surface is used (e.g. piecewise quadratic). For a piecewise linear triangulation, the curvature is zero (point-wise) within each triangular face.*

SM1.1.3. Execution. Running the simulation of (SM4) requires several steps: creating a mesh with sub-domains, generating a higher order surface approximation, defining FE spaces, assembling matrices, interpolating given data, solving the linear system, computing the errors, and visualizing the results. For brevity, we do not give

the complete code here, but outline the main parts (see `Execute_LapBel_Open_Surface.m` for the execution script).

We start by using `triangle_mesh_of_disk` (included in FELICITY) to define a quasi-uniform mesh of the unit disk centered at the origin. Next, apply (SM1) to create a piecewise *linear* surface triangulation of a saddle surface, followed by creating a `MeshTriangle` object, and adding $\partial\Gamma$ as a sub-domain. The code is summarized here:

```
Refine_Level = <choose>;
[Tri, Pts_q] = triangle_mesh_of_disk([0 0 0],1,Refine_Level);
% apply parameterization
Psi = @(q1,q2) [q1, q2, 0.5*(q1.^2 - q2.^2)];
Pts_x_y_z = Psi(Pts_q(:,1), Pts_q(:,2));
Mesh = MeshTriangle(Tri, Pts_x_y_z, 'Gamma');
% add the boundary
BDY  = Mesh.freeBoundary();
Mesh = Mesh.Append_Subdomain('1D','dGamma',BDY);
```

Next, define the space $\mathbf{G}_h \sim$ `G_Space` for the piecewise quadratic approximation:

```
P2_RefElem = ReferenceFiniteElement(lagrange_deg2_dim2());
G_Space = GeoElementSpace('G_h', P2_RefElem, Mesh);
G_DoFmap =...
     mex_LapBel_DoF_Alloc_G_Space(uint32(Mesh.ConnectivityList));
G_Space = G_Space.Set_DoFmap(Mesh,uint32(G_DoFmap));
```

Note that we use a mex file to create the DoFmap for `G_Space`. Next,

```
Geo_Points_hat =...
         G_Space.Get_Mapping_For_Piecewise_Linear_Mesh(Mesh);
```

where `Geo_Points_hat` are the nodal values representing $\widehat{\mathbf{g}}_h \in \mathbf{G}_h$, which is a piecewise quadratic interpolant of the underlying piecewise linear mesh stored in `Mesh`, i.e. $\widehat{\mathbf{g}}_h : \widehat{\Gamma} \to \widehat{\Gamma}$ is the identity map on $\widehat{\Gamma}$. The script file then processes $\widehat{\mathbf{g}}_h$ into $\mathbf{g}_h \in \mathbf{G}_h$, which is the identity map on Γ (recall (SM1); see the script for details). This gives a new variable `Geo_Points` $\sim \mathbf{g}_h$. Note: generating higher order curved elements that maintain convergence has been studied in [SM2, SM5].

Now we define the finite element spaces `V_Space` $\sim V_h$, `W_Space` $\sim W_h$:

```
V_Space = FiniteElementSpace('V_h', P2_RefElem, Mesh, 'Gamma');
V_Space = V_Space.Set_DoFmap(Mesh,uint32(G_DoFmap));

P1_RefElem = ReferenceFiniteElement(lagrange_deg1_dim1());
W_Space  = FiniteElementSpace('W_h', P1_RefElem, Mesh, 'dGamma');
W_DoFmap = mex_LapBel_DoF_Alloc_W_h(...
               uint32(Mesh.Get_Global_Subdomain('dGamma')));
W_Space  = W_Space.Set_DoFmap(Mesh,uint32(W_DoFmap));
```

We were able to "reuse" the DoFmap from `G_Space` in `V_Space` because they have the same reference finite element. For `W_Space`, we point out that `W_DoFmap` is allocated on the sub-domain $\partial\Gamma \sim$ `dGamma`.

Assembling the FE matrices requires that we know how `dGamma` is embedded in `Gamma`, because of the form `B` $\sim (\mu_h, u_h)_{\partial\Gamma}$. So we invoke the following lines

```
Domain_Names = {'Gamma'; 'dGamma'};
Gamma_Embed = Mesh.Generate_Subdomain_Embedding_Data(Domain_Names);
```

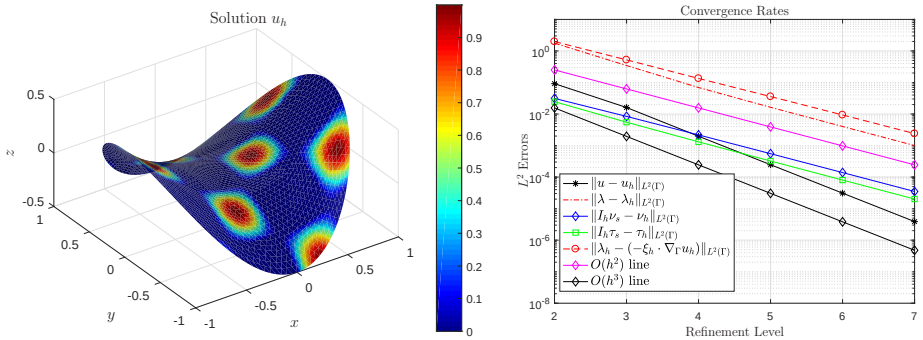

Fig. SM1: Solution of (SM4) and convergence rates for the numerical errors.

which yields the embedding data. Now call the mex file for assembling the matrices:

```
FEM = mex_MatAssem_LapBel_Open_Surface(...
              [],Geo_Points,G_Space.DoFmap,[],Gamma_Embed,...
              V_Space.DoFmap,W_Space.DoFmap);
```

We see that the matrix assembler needs the domain geometry description, how the sub-domains are embedded, and the `DoFmap`s of the FE spaces. Note: no two mex files for assembling different matrices (with different spaces) will have the same calling procedure. One must run the mex file at the Matlab prompt with no arguments to display information on what arguments (and their order) must be passed in.

REMARK SM1.2. *Nothing special needs to be done by the user to compute* traces *of finite element functions on sub-domains; it is done automatically by the matrix assembly mex file. This requires the sub-domain embedding information (e.g.* `Gamma_Embed`*), which can be automatically calculated with the* `Mesh` *method:* `Generate_Subdomain_Embedding_Data`.

FELICITY provides an accessor class for easily extracting the FE matrices:

```
LB_Mats = FEMatrixAccessor('Laplace-Beltrami',FEM);
M = LB_Mats.Get_Matrix('M');  % pull out matrices
K = LB_Mats.Get_Matrix('K');
B = LB_Mats.Get_Matrix('B');
```

Now combine the matrices into the block linear system and solve:

```
W_Num = W_Space.num_dof;
ZZ = sparse(W_Num,W_Num);
MAT   = [K, B';
         B, ZZ];
RHS   = [M * f_h; B * g_h];
Soln  = MAT \ RHS; % solve
```

Here, `f_h` $\sim f_h := I_h f \in V_h$, `g_h` $\sim g_h := I_h g \in V_h$ are interpolants of the data f, g in (12). Next, parse `Soln`:

```
V_Num = V_Space.num_dof;
u_h = Soln(1:V_Num,1);
lambda_h = Soln(V_Num+1:end,1);
```

where `u_h` $\sim u_h \in V_h$ and `lambda_h` $\sim \lambda_h \in W_h$.

The rest of the script visualizes the solution and computes the convergence rates of the error norms discussed in subsection SM1.1.2 (see Figure SM1).

SM1.2. EWOD. For brevity, we only outline the implementation of the problem in subsection 4.2; the full demo can be found in the FELICITY sub-directory: `../FELICITY/Demo/EWOD`. Executing the script `test_EWOD_FalkWalker` at the Matlab prompt will run this example. Note: this script will also *compile* the `mex` files that this demo uses.

This example illustrates the following features in FELICITY:

- Time dependent problems.
- Deforming meshes.
- Using $H(\mathrm{div},\Omega)$ elements.
- Finite element spaces defined on different sub-domains.
- Access to domain (and sub-domain) geometric terms when defining forms.

SM1.2.1. Finite Element Formulation. Let $\mathcal{T}_h^i$ be a piecewise linear triangulation of $\Omega^i \subset \mathbb{R}^2$ (i denotes a time-index) with $\mathcal{E}_h^i$ being the set of boundary edges conforming to $\mathcal{T}_h^i$, i.e. $\Gamma^i = \partial\Omega^i$ and $\Gamma^i = \cup_{E\in\mathcal{E}_h^i}\overline{E}$. Next, introduce the following FE spaces on the "current" domain Ω^i, Γ^i:

$$
\begin{aligned}
\mathbf{V}_h^i &= \mathrm{BDM}_1(\Omega^i),\\
Q_h^i &= \{q_h \in L^2(\Omega^i) : q_h|_T = \eta\circ\mathbf{F}_T^{-1}, T\in\mathcal{T}_h^i, \eta\in\mathcal{P}_0(T_{\mathrm{ref}})\},\\
M_h^i &= \{\mu_h \in C^0(\Gamma^i) : \mu_h|_E = \eta\circ\mathbf{F}_E^{-1}, E\in\mathcal{E}_h^i, \eta\in\mathcal{P}_1(E_{\mathrm{ref}})\},\\
\mathbf{Y}_h^i &= \{\mathbf{y}_h \in [C^0(\Gamma^i)]^2 : \mathbf{y}_h|_E = \boldsymbol{\eta}\circ\mathbf{F}_E^{-1}, E\in\mathcal{E}_h^i, \boldsymbol{\eta}\in[\mathcal{P}_1(E_{\mathrm{ref}})]^2\},\\
\mathbf{G}_h^i &= \{\mathbf{r}_h \in [C^0(\Omega^i)]^2 : \mathbf{r}_h|_T = \boldsymbol{\eta}\circ\mathbf{F}_T^{-1}, T\in\mathcal{T}_h^i, \boldsymbol{\eta}\in[\mathcal{P}_1(T_{\mathrm{ref}})]^2\},
\end{aligned}
\tag{SM5}
$$

where $\mathbf{V}_h^i$ is the lowest order Brezzi-Douglas-Marini space of piecewise linear vector functions [SM1, SM4] over Ω^i. The space $\mathbf{G}_h^i$ is the geometric element space for representing the geometry of the domain Ω^i.

With this, the discrete variational formulation of (14) follows. Given Lipschitz domains Ω^i, Γ^i, a domain parameterization $\mathbf{x}_h^i \equiv \mathrm{id}_{\Gamma^i}$ in $\mathbf{Y}_h^i$, previous velocity $\mathbf{u}_h^i$ in $\mathbf{V}_h^{i-1}$, and electro-wetting force E_h^{i+1} in M_h^i, find $\mathbf{u}_h^{i+1}$ in $\mathbf{V}_h^i$, p_h^{i+1} in Q_h^i, $\mathbf{x}_h^{i+1}$ in $\mathbf{Y}_h^i$, and λ_h^{i+1} in M_h^i such that

$$
\begin{aligned}
\left(\frac{\alpha}{\delta t}+\beta\right)\left(\mathbf{u}_h^{i+1},\mathbf{v}\right)_{\Omega^i} - \left(p_h^{i+1},\nabla\cdot\mathbf{v}\right)_{\Omega^i} + \left(\lambda_h^{i+1},\mathbf{n}^i\cdot\mathbf{v}\right)_{\Gamma^i} &= \\
\frac{\alpha}{\delta t}\left(\mathbf{u}_h^i,\mathbf{v}\right)_{\Omega^i} - \left(E_h^{i+1},\mathbf{n}^i\cdot\mathbf{v}\right)_{\Gamma^i}, &\quad \forall\mathbf{v}\in\mathbf{V}_h^i,\\
-\left(\nabla\cdot\mathbf{u}_h^{i+1},q_h\right)_{\Omega^i} = 0, &\quad \forall q_h\in Q_h^i,\\
\left(\nabla_{\Gamma^i}\mathbf{x}_h^{i+1},\nabla_{\Gamma^i}\mathbf{y}_h^{i+1}\right)_{\Gamma^i} - \left(\lambda_h^{i+1},\mathbf{n}^i\cdot\mathbf{y}_h\right)_{\Gamma^i} = 0, &\quad \forall\mathbf{y}_h\in\mathbf{Y}_h^i,\\
-\frac{1}{\delta t}\left(\mathbf{x}_h^{i+1}\cdot\mathbf{n}^i,\mu_h\right)_{\Gamma^i} + \left(\mathbf{u}_h^{i+1}\cdot\mathbf{n}^i,\mu_h\right)_{\Gamma^i} = -\frac{1}{\delta t}\left(\mathbf{x}_h^i\cdot\mathbf{n}^i,\mu_h\right)_{\Gamma^i}, &\quad \forall\mu_h\in M_h^i,
\end{aligned}
\tag{SM6}
$$

where $(\mathbf{u}_h,\mathbf{v}_h)_{\Omega^i} = \int_{\Omega^i}\mathbf{u}_h\cdot\mathbf{v}_h$ is the $L^2(\Omega^i)$ inner product and $(\mathbf{x}_h,\mathbf{y}_h)_{\Gamma^i} = \int_{\Gamma^i}\mathbf{x}_h\cdot\mathbf{y}_h$ is the $L^2(\Gamma^i)$ inner product. After solving one time-step, we define the new domain Γ^{i+1} via the parameterization $\mathbf{x}_h^{i+1}$; this induces a displacement from Γ^i to Γ^{i+1} which we use to update Ω^i to Ω^{i+1} via a harmonic extension [SM3].

SM1.2.2. Form File. The abstract definition file for the forms in (SM6) is in the m-file: `MatAssem_EWOD_FalkWalker.m`, which is listed here:

```
function MATS = MatAssem_EWOD_FalkWalker()
%MatAssem_EWOD_FalkWalker
```

```
% define domain (2-D domain with 1-D boundary)
Omega = Domain('triangle');
Gamma = Domain('interval') < Omega; % subset

% define finite element spaces
V_h = Element(Omega,brezzi_douglas_marini_deg1_dim2); % BDM_1
Q_h = Element(Omega,lagrange_deg0_dim2); % piecewise constant
M_h = Element(Gamma,lagrange_deg1_dim1); % piecewise linear
Y_h = Element(Gamma,lagrange_deg1_dim1,2); % vector PW linear

% space that represents the geometry of the domain
G_h = Element(Omega,lagrange_deg1_dim2,2); % vector PW linear

% define functions in FE spaces
u_h = Trial(V_h);
v_h = Test(V_h);

q_h = Test(Q_h);

mu_h = Test(M_h);

x_h = Trial(Y_h);
y_h = Test(Y_h);

s_h = Trial(G_h);
r_h = Test(G_h);

% geometric functions
gf = GeoFunc(Gamma);

% define (discrete) forms
M = Bilinear(V_h,V_h);
M = M + Integral(Omega, v_h.val' * u_h.val );

K = Bilinear(Y_h,Y_h);
K = K + Integral(Gamma, sum(sum(x_h.grad .* y_h.grad)) );

B = Bilinear(Q_h,V_h);
B = B + Integral(Omega, q_h.val * u_h.div );

C = Bilinear(M_h,V_h);
C = C + Integral(Gamma, mu_h.val * (u_h.val' * gf.N) );

D = Bilinear(M_h,Y_h);
D = D + Integral(Gamma, mu_h.val * (x_h.val' * gf.N) );

chi = Linear(M_h);
chi = chi + Integral(Gamma, mu_h.val * (gf.X' * gf.N) );
```

```
A = Bilinear(G_h,G_h);
A = A + Integral(Omega, sum(sum(s_h.grad .* r_h.grad)) );

% set the minimum order of accuracy for the quad rule
Quadrature_Order = 5;

% define geometry representation (piecewise linear)
G_Space = GeoElement(Omega);

% define a set of matrices
MATS = Matrices(Quadrature_Order,G_Space);

<similar conclusion code as before>
```

The m-file defines several bilinear/linear forms: `M` $\sim (\mathbf{u}_h, \mathbf{v}_h)_{\Omega^i}$, `K` $\sim (\nabla_\Gamma \mathbf{x}_h, \nabla_\Gamma \mathbf{y}_h)_{\Gamma^i}$, `B` $\sim (q_h, \nabla \cdot \mathbf{u}_h)_{\Omega^i}$, `C` $\sim (\mu_h, \mathbf{u}_h \cdot \mathbf{n}_h)_{\Gamma^i}$, `D` $\sim (\mu_h, \mathbf{x}_h \cdot \mathbf{n}_h)_{\Gamma^i}$, `A` $\sim (\nabla \mathbf{s}_h, \nabla \mathbf{r}_h)_{\Omega^i}$ for $\mathbf{s}_h, \mathbf{r}_h$ in $\mathbf{G}_h^i$, and `chi` $\sim (\mu_h, \mathbf{x} \cdot \mathbf{n}_h)_{\Gamma^i}$. Similar to subsection SM1.1.2, the definition file specifies two sub-domains by

```
Omega = Domain('triangle');
Gamma = Domain('interval') < Omega; % subset
```

So this is another example of how FELICITY can automatically handle integrating the *trace* of a finite element function over a sub-domain.

SM1.2.3. Execution. An implementation of a time-dependent simulation of (SM6) is given in `Execute_EWOD_FalkWalker.m`. For brevity, we do not give the complete code here but only outline the main parts.

The problem parameters are given by: `alpha` $= 0.01$, `beta` $= 1.0$, `dt` $= 0.1$. The initial velocity is $\mathbf{u}_h^0 = \mathbf{0}$. The initial domain Ω^0 is taken to be a unit disk centered at the origin, with boundary $\Gamma^0 := \partial\Omega^0$, and is stored as a `MeshTriangle` object named `Mesh`. Throughout this section, the time-index i refers to the *current* time-index in the time-stepping loop below.

Next, define the space $\mathbf{G}_h^i \sim$ `G_Space` to handle the piecewise *linear* mesh:

```
P1_RefElem_2D = ReferenceFiniteElement(lagrange_deg1_dim2());
G_Space = GeoElementSpace('G_h',P1_RefElem_2D,Mesh);
G_Space = G_Space.Set_DoFmap(Mesh,uint32(Mesh.ConnectivityList));
```

This is simpler than in subsection SM1.1.3 because we can use the mesh connectivity directly as the `DoFmap`. We also set a fixed sub-domain of `G_Space` to be $\Gamma \sim$ `Gamma`:

```
G_Space = G_Space.Append_Fixed_Subdomain('Gamma');
```
(SM7)

this is used later to extend the boundary displacement to the entire domain Ω by a Laplace extension (see (SM9)).

With this, we get the domain coordinates that represent Ω^i:

```
x_h_i = G_Space.Get_Mapping_For_Piecewise_Linear_Mesh(Mesh);
x_h_evolve(Num_Steps+1).data = []; % init
x_h_evolve(1).data = x_h_i;
```

where we allocated a `struct` named `x_h_evolve` to store the domain coordinates for several time-steps. Note that `x_h_i` represents the identity map on Ω^i.

Next, we define the finite element spaces `V_Space` $\sim \mathbf{V}_h^i$, `Q_Space` $\sim Q_h^i$, `Y_Space` $\sim \mathbf{Y}_h^i$, `M_Space` $\sim M_h^i$, as in subsection SM1.1.3, in addition to other processing. For example, we create the space $\mathbf{V}_h^i$ by:

```
BDM1_RefElem =...
     ReferenceFiniteElement(brezzi_douglas_marini_deg1_dim2());
V_Space  = FiniteElementSpace('V_h', BDM1_RefElem, Mesh, 'Omega');
V_DoFmap = mex_EWOD_DoF_Alloc_V_h(uint32(Mesh.ConnectivityList));
V_Space  = V_Space.Set_DoFmap(Mesh,uint32(V_DoFmap));
```

REMARK SM1.3. *The* ***DoFmaps*** *for the finite element spaces do not depend on the time index because the topology of the underlying mesh is assumed not to depend on time. Of course, new* ***DoFmaps*** *must be generated if the mesh topology is ever regenerated; this is necessary when mesh elements are close to singular or invalid (e.g. inverted elements).*

The next part of the code computes the map from DoF indices in M_h^i to DoF indices in $\mathbf{G}_h^i$, and similarly the map from DoF indices in $\mathbf{Y}_h^i$ to DoF indices in $\mathbf{G}_h^i$. This is necessary for purposes of interpolation, as well as extending the boundary velocity to the interior bulk mesh in order to smoothly update the domain Ω (see (SM9)). We start by putting the coordinates of the DoFs for $\mathbf{G}_h^0$ into a quadtree:

```
BB = 2*[-1.001, 1.001, -1.001, 1.001]; % bounding box
QT = mexQuadtree(x_h_i,BB);
```

Recall `x_h_i` from earlier. Then we find the coordinates of the DoFs for the initial finite element spaces M_h^0 and $\mathbf{Y}_h^0$:

```
M_Points = M_Space.Get_DoF_Coord(Mesh);
Y_Points = Y_Space.Get_DoF_Coord(Mesh);
```

We then search for the closest DoF in $\mathbf{G}_h^0$ to each point in M_h^0 ($\mathbf{Y}_h^0$):

```
[M_h_to_G_h, QT_dist] = QT.kNN_Search(M_Points,1);
[Y_h_to_G_h, QT_dist] = QT.kNN_Search(Y_Points,1);
delete(QT); % delete quadtree object
```

where we used a nearest neighbor search with the quadtree object. The column vector `M_h_to_G_h` has the following meaning. Given the `i`th DoF in M_h^0, the corresponding DoF index in $\mathbf{G}_h^0$ is given by `M_h_to_G_h(i)`; a similar format holds for `Y_h_to_G_h`.

REMARK SM1.4. *Note that the above code is evaluated before the time-stepping loop below because the DoF index maps do not depend on time (so long as the mesh connectivity does not change).*

We also store the DoF indices of $\mathbf{G}_h^i$ that do **not** lie on Γ (recall (SM7)):

(SM8)
```
G_h_Free_DoFs = G_Space.Get_Free_DoFs(Mesh,'all');
```

Assembling the FE matrices requires the mesh embedding information (c.f. subsection SM1.1.3):

```
Domain_Names = {'Omega'; 'Gamma'};
Omega_Embed = Mesh.Generate_Subdomain_Embedding_Data(Domain_Names);
```

In addition, the BDM_1 space requires choosing a consistent orientation of all facets (i.e. edges) in the mesh [SM1, SM4]. This is accomplished by:

```
Edges = Mesh.edges;
[~, Omega_Orient] = Mesh.Get_Facet_Info(Edges);
```

Note that both `Omega_Embed` and `Omega_Orient` only depend on mesh topology, so they do not need to be recomputed at each time-step.

Next, start the time-stepping loop and assemble matrices on Ω^i:

```
for ii = 1:Num_Steps

    FEM = mex_MatAssem_EWOD_FalkWalker(...
             [],x_h_i,G_Space.DoFmap,Omega_Orient,Omega_Embed,...
             G_Space.DoFmap,M_Space.DoFmap,Q_Space.DoFmap,...
             V_Space.DoFmap,Y_Space.DoFmap);
```

where we use the current domain coordinates `x_h_i`. We then extract the matrices, `M`, `K`, `B`, `C`, `D`, `chi`, as we did in subsection SM1.1.3, and form the matrix system for (SM6):

```
VN = V_Space.num_dof;
QN = Q_Space.num_dof;
YN = 2*Y_Space.num_dof;
MN = M_Space.num_dof;
MAT =...
[((alpha/dt)+beta)*M,           -B', sparse(VN,YN),            C';
                  -B, sparse(QN,QN), sparse(QN,YN), sparse(QN,MN);
       sparse(YN,VN), sparse(YN,QN),             K,           -D';
                   C, sparse(MN,QN),    -(1/dt)*D, sparse(MN,MN)];
RHS=[(alpha/dt)*M*u_h-C'*E_h; zeros(QN,1);
                              zeros(YN,1); -(1/dt)*chi];
```

Then we solve: `Soln = MAT \ RHS` and parse the solution, analogously to subsection SM1.1.3, to obtain `u_h` $\sim \mathbf{u}_h^{i+1}$, `p_h` $\sim p_h^{i+1}$, `x_h_new_bdy` $\sim \mathbf{x}_h^{i+1}$, `lambda_h` $\sim \lambda_h^{i+1}$. Again, nothing special had to be done for the embedded sub-domain Γ; recall Remark SM1.2.

Using `x_h_new_bdy`, we update the domain coordinates `x_h_i` for the next time step using a harmonic extension, which we now describe. Note that the coordinates of the new boundary Γ^{i+1} is in `x_h_new_bdy`, which is ordered according to the DoF indices in $\mathbf{Y}_h^i$. So, upon recalling the DoF index map `Y_h_to_G_h` from earlier, we extract the coordinates of the previous boundary Γ^i by

```
x_h_old_bdy = x_h_i(Y_h_to_G_h,:);
```

which is also ordered according to the DoF indices in $\mathbf{Y}_h^i$. Therefore, we can compute the displacement of the boundary by:

```
Displace_bdy = x_h_new_bdy - x_h_old_bdy;
```

Next, we create a two-column matrix that will eventually contain the vector-valued displacement of the domain Ω:

```
Displace = G_Space.Get_Zero_Function; % init
Displace(Y_h_to_G_h,:) = Displace_bdy;
```

where we set the boundary values of the displacement to `Displace_bdy`. We then reshape the two-column matrix into a long column vector:

```
D_Soln = Displace(:);
```

We now extend the boundary displacement via Laplace's equation, i.e. we solve the following PDE (given in strong form):

$$\begin{aligned} -\Delta \mathbf{d}^{i+1} &= 0, && \text{in } \Omega^i, \\ \mathbf{d}^{i+1} &= \mathbf{d}_{\Gamma^i}^{i+1}, && \text{on } \Gamma^i, \end{aligned} \tag{SM9}$$

where `Displace_bdy` $\sim \mathbf{d}_{\Gamma^i}^{i+1}$.

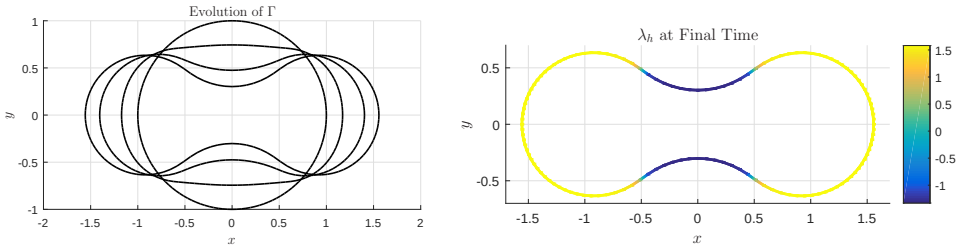

Fig. SM2: Illustration of the evolution of Γ^i, from solving (SM6), at $i = 0, 1, 4, 30$. Solution λ_h^{30} is also shown.

This extension is achieved by the following commands:

```
A = EWOD_Mats.Get_Matrix('A');
R1 = -A * D_Soln; % set boundary conditions
D_Soln(G_h_Free_DoFs,1) =...
      A(G_h_Free_DoFs,G_h_Free_DoFs) \ R1(G_h_Free_DoFs,1);
Displace(:) = D_Soln; % put back into two-column matrix format
```

Recall (SM8). This now yields `Displace` $\sim \mathbf{d}^{i+1}$, which is used to update the Ω coordinates:

```
x_h_new = x_h_i + Displace;
```

Then, we store the new domain coordinates and update the coordinates for the next time-step:

```
% store solution
x_h_evolve(ii+1).data = x_h_new; % store it
x_h_i = x_h_new; % update for next time-step
```

Finally, we close the time-stepping loop: `end`.

Plots of the evolution of the droplet boundary, and λ_h, are shown in Figure SM2.

REFERENCES

[SM1] D. Boffi, F. Brezzi, and M. Fortin, *Mixed Finite Element Methods and Applications*, vol. 44 of Springer Series in Computational Mathematics, Springer-Verlag, New York, NY, 2013.

[SM2] P. Ciarlet and P.-A. Raviart, *Interpolation theory over curved elements, with applications to finite element methods*, Computer Methods in Applied Mechanics and Engineering, 1 (1972), pp. 217 – 249, https://doi.org/10.1016/0045-7825(72)90006-0, http://www.sciencedirect.com/science/article/pii/0045782572900060.

[SM3] R. S. Falk and S. W. Walker, *A mixed finite element method for EWOD that directly computes the position of the moving interface*, SIAM Journal on Numerical Analysis, 51 (2013), pp. 1016–1040.

[SM4] G. N. Gatica, *A Simple Introduction to the Mixed Finite Element Method: Theory and Applications*, SpringerBriefs in Mathematics, Springer, Jan 2014.

[SM5] M. Lenoir, *Optimal isoparametric finite elements and error estimates for domains involving curved boundaries*, SIAM Journal of Numerical Analysis, 23 (1986), pp. 562–580.

[SM6] S. W. Walker, *The Shapes of Things: A Practical Guide to Differential Geometry and the Shape Derivative*, vol. 28 of Advances in Design and Control, SIAM, 1st ed., 2015.

[SM7] S. W. Walker, *FELICITY: manual*, 2017, https://www.mathworks.com/matlabcentral/fileexchange/31141-felicity/.

[SM8] S. W. Walker, *Felicity wiki documentation*, 2017, https://github.com/walkersw/felicity-finite-element-toolbox/wiki.