iceFEM:
Open Source Package for Hydro-elasticity Problems

Balaje Kalyanaraman

Contents

1 Introduction

The package is intended for researchers aiming to solve Hydroelasticity problems using the finite element method. The principal idea behind the package is to use **`FreeFem++`** to solve the finite element problem and use **`MATLAB`** for visualization and other operations such as interpolation and cubic-spline constructions. It is necessary to have a basic **`FreeFem++`** installation to use this package and can be downloaded from the official website.

1.1 Installation

The package can be downloaded from `https://github.com/Balaje/iceFem`. The root directory contains the structure shown in Figure 1. The **`include`** folder contains a collection of `.idp` files which are `FreeFem++` scripts that contains pre-written functions and macros. To begin using the programs in the package, open the terminal and type the following.

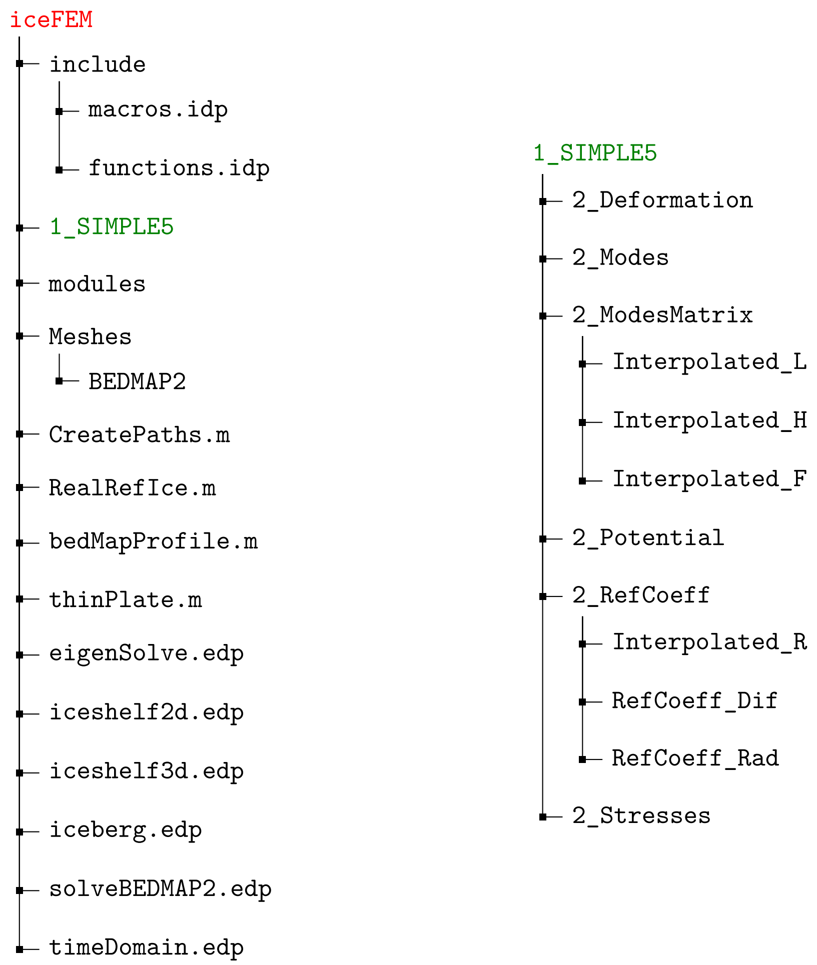


Figure 1: Directory structures of the main package (left).The `modules` folder contains a list of **`MATLAB`** scripts for interpolation and visualization. The **`meshes`** folder contains a sample geometry data from the BEDMAP2 dataset. A list of example FreeFem scripts using the iceFEM package (`*.edp`) and `MATLAB` scripts (`*.m`) available in the directory is also shown. A sample working directory 1_SIMPLE5 (right), generated using the directory generation script **`./genDir.sh 1_SIMPLE5`**.

```
export FF_INCLUDEPATH="$PWD/include"
```

This tells the **FreeFem++** compiler to add the **include** folder inside the package to the include path. Any new script could be added in the root directory. Then when writing scripts, the required **.idp** file can be imported by adding

```
include "macros.idp"
```

for example, to include the **macros.idp** file. In most cases, when using the predefined macros to solve the problem, adding **macros.idp** includes all the other **.idp** files in the package. When solving custom problems, individual **.idp** files can be included in the main program. Detailed description of the functions available can be found in Section 2.1. The **FF_INCLUDEPATH** variable must be set each time a new terminal session is started. One way to override this problem is to set the variable permanently by adding the line

```
export FF_INCLUDEPATH="/path/to/iceFem/include"
```

in **$HOME/.bashrc** or **$HOME/.bash_profile**. This ensures that the **FreeFem++** compiler locates the file each time a new terminal window is opened. Visualization can be done using **gnuplot** or the native plotter of **FreeFem++**. Newer versions of **FreeFem++** contains a routine to perform visualizations using ParaView.

A more convenient way to use the **FreeFem++** module is in conjunction with **MATLAB**. The **modules** folder consists of a set of **MATLAB** scripts that are used for visualization and validation of the **FreeFem++** code. This folder also contains routines that perform interpolation on certain quantities generated by the **FreeFem++** code. The list of functions available are listed below.

```
colscheme.m #For complex plot.

dispersion_elastic_surface.m #For computing the roots of an elastic-plate
    dispersion relation.

dispersion_free_surface.m #For computing the roots of a free-surface
    dispersion relation.

eigenFreqSW.m #To find the resonance frequencies of the shallow water problem
    using Newton Raphson.

export_fig.m #EXPORT_FIG package

findHInterpolated.m #Interpolating the H matrix at a specified frequency

findResonanceCplx.m #[depreciated]
getMatrices.m #[depreciated]

getProperties.m #Functions to get the properties of the ice

importfiledata.m #For FreeFem visualization in MATLAB to import the function
    values on the mesh points
```

```

importfilemesh.m #For FreeFem visualization in MATLAB to import the mesh
    points

interpolateFreq.m #Interpolate the system of equations on the new frequency
    space (real valued) and store the system of equations in 2_ModesMatrix/
    Interpolated_*

interpolateFreqComplex.m #Interpolate the system of equations on the new
    frequency space (complex valued) and store the system of equations in 2
    _ModesMatrix/Interpolated_*

interpolateRefCoeff.m #Interpolate the diffraction and radiation reflection
    coefficients and store in 2_RefCoeff/Interpolated_R

movingplate.m #Solve the thin-plate potential flow moving plate problem for
    uniform geometries.

pltphase.m #For complex plot

resonSW.m #To find the complex resonance frequencies of the shallow water
    problem. Uses eigenFreqSW.m.

shallowmovingplate.m #To solve the thin-plate shallow water problem for
    uniform geometries.

zdomain.m #For complex plot.
```

1.2 A Quick Example

In this subsection, we describe the use of the **FreeFem++** code to solve a simple example. A set of reserved keywords used in the package are listed in Section 2. Once the **FF_INCLUDEPATH** is set, type

```
>> ./genDir.sh 1_TEST
>> mpirun -np 2 FreeFem++-mpi -v 0 simple.edp -Tr 4000 -hsize 0.08
```

in the command line. The first command generates the required directory structure used by the `simple.edp` script. This solves the ice-shelf problem using linear elasticity for the ice combined with potential flow for the fluid. The code produces the following output:

```
Dimension : 2
Dimension : 2
Imported Cavity Mesh (proc 1)...
Refining Cavity Mesh (proc 1) ...
Cavity : Before Refinement, NBV = 1828
Imported Ice Mesh (proc 0)...
Refining Ice Mesh (proc 0) ...
```

```
Ice : Before Refinement, NBV = 1746
Ice : After Refinement, NBV = 2423
Cavity : After Refinement, NBV = 3591


Reflection Coefficient = (0.631992,0.775192)
Absolute Value = 1.00017
```

For the ice-shelf problems, $|R| \approx 1$ due to energy conservation and it can be used to check the solution. When the macro **`setProblem`** is invoked, optional parameters can be specified to modify the problem.

```
 FreeFem++ -ne -v 0 [FILENAME].edp -L [LENGTH]
                                  -H [DEPTH OF OPEN OCEAN]
                                  -h [THICKNESS OF ICE]
                                  -N [MESH PARAM]
                                  -Tr [REAL(period)]
                                  -Ti [IMAG(period)]
                                  -iter [SOL. INDEX]
                                  -isUniIce [ON/OFF UNIFORM/NON UNIFORM ICE]
                                  -isUniCav [ON/OFF UNIFORM/NON UNIFORM CAVITY]
```

where the `[.]` indicates the corresponding numerical value of the optional parameters. The length, thickness of the ice and the depth of the ocean is specified in meters (m). The wave-period is specified in seconds (s). The `ON/OFF` values are specified in binary, i.e., 0 or 1. The **`iter`** variable is used to number the solution which aids in interpolation and other batch manipulations.

NOTE: For running the default FreeFem scripts, some default directories need to be generated. Simply run the command

```
>> for m in 1_BEDMAP2 1_SIMPLE5 2_ICEBERG 1_SIMPLE3D; do echo ./genDir.sh $m;
   done
```

1.3 MATLAB Interface

As mentioned earlier, **`MATLAB`** can be used to produce high quality graphics and the default functions in **`modules`** can be used. To use **`MATLAB`** seamlessly with **`FreeFem++`**, one needs to follow the instructions below carefully:

1.3.1 Setting Up

The user must find the location of the FreeFem++ compiler. This can be done by running

```
 which FreeFem++
```

in the command line. This produces an output like

```
/usr/local/ff++/openmpi-2.1/3.61-1/bin/FreeFem++
```

By default, the package comes with an initialization script called `CreatePaths.m` that tells the MATLAB compiler, the installation location of `FreeFem++` in the computer and also sets the `FF_INCLUDEPATH` variable for the current `MATLAB` session. The file also defines a set of variables that will be used to generate the plots.

```
    %% Filename: CreatePaths.m

    function CreatePaths
    clc
    close all

    fprintf('Run:\n\nwhich FreeFem++\n\nin your command line to get the path
        for FreeFem++.\n Set the full path in the variable `ff` in CreatePaths.
        m\n');
    addpath([pwd,'/modules/']);
    set(0,'defaultLegendInterpreter','latex');
    set(0,'defaulttextInterpreter','latex');
    set(0,'defaultaxesfontsize',20);
    envvar = [pwd,'/include'];
    setenv('FF_INCLUDEPATH',envvar);

    %% Should be set manually by the user.
    global ff
    ff='/usr/local/ff++/openmpi-2.1/3.61-1/bin/FreeFem++';
```

Once the compiler location is obtained, the user must add the **full path** in the `global ff` variable as shown in the code block above. Then the script `CreatePaths.m` should be run to set the global variable for the session.

1.3.2 Running FreeFem++ in MATLAB

First the function script `getProperties.m` can be used to get the default physical properties of ice and water. The usage is as follows

```
    [L, H, th, d, E, nu, rhow, rhoi, g, Ad] = getProperties();
```

where

```
    L: Length of the ice.           H: Depth of the open ocean.
    th: Thickness of the shelf.     d: Submergence of the ice.
    E: Young`s modulus.             nu: Poisson`s ratio.
    rhow: Density of Water.         rhoi: Density of Ice.
    g: Gravity Acceleration.        Ad: Amplitude of the wave.
```

To override certain parameters, use `~` at the desired entry. For example:

```
%% Get the Parameters of the ice.
[~,~,~,~,E,nu,rhow,rhoi,g,~] = getProperties();
H = 800;
L = 20000;
omega = 2*pi/(300); % Wave Period can be complex.
T = 2*pi/omega;
th = 200;
d = (rhoi/rhow)*th;
```

The most intuitive way to run the **FreeFem++** code is to call the script as an external program from **MATLAB**. The best way to do it is to use the following code block (with appropriate modifications).

```
%% Run the FreeFem++ code;
global ff
file = 'simple1.edp';
ffpp=[ff,' -nw -ne ', file];
cmd=[ffpp,' -Tr ',num2str(real(T)),' -Ti ',num2str(imag(T)),' -H ',num2str(H),
    ' -L ',num2str(L),' -h ' ,num2str(th),' -N ',num2str(3), ' -isUniIce ',
    num2str(0), ' -isUniCav ',num2str(0)];
[aa,bb1]=system(cmd);
if(aa)
    error('Cannot run program. Check path of FF++ or install it');
end
```

The global variable `ff` contains the full path to the **FreeFem++** compiler. If an error occurs in running the **FreeFem++** code, the variable `bb1` can be printed out to check the error. If the code runs successfully, then `aa=0` and the error message is not printed. The `num2str` function converts the numerical values to string and appends them to the full command, which is then passed to the `system` function.

1.3.3 Visualization

We use `pdeplot` command to plot the mesh. The macro `writeToMATLAB` in `include/macros.idp` contains a code snippet to write the **FreeFem++** data. In the **FreeFem++** code, call

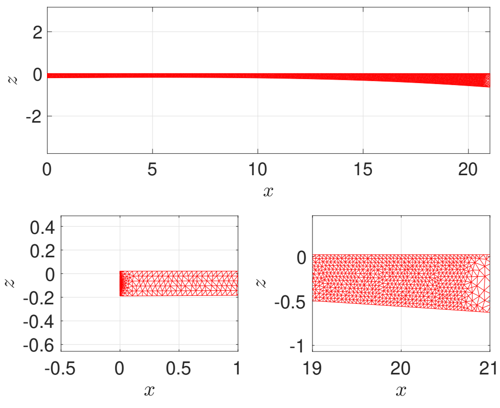
```
 writeToMATLAB(uh, Th, solfilename, meshfilename);
```

where `uh` denotes the finite element solution, `Th` denotes the finite element mesh and `solfilename, meshfilename` denotes string variables which contains the name of the solution file and the mesh file respectively.

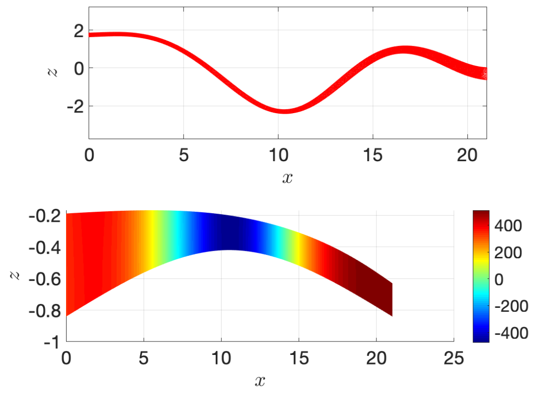
The **MATLAB**functions that will be used here are

```
[pts,seg,tri] = importfilemesh(filename);
uh = importfiledata(filename);
```

The variable `pts` is a $2 \times N$ array containing the $x-$ and $y-$ coordinate of the points. The variables



(a) Finite element mesh



(b) Displacement and Velocity Potential.

Figure 2: Figure showing the plots generated by the **MATLAB** script.

[seg,tri] stores the mesh connectivity information which will be used by **pdeplot**. To plot the mesh in **MATLAB**, we write

```
figure;
pdeplot(pts,seg,tri);
axis equal
grid on
```

and to plot a finite element function in **MATLAB**, we write

```
figure;
pdeplot(pts,seg,tri,'XYData',real(uh)','colormap','jet');
axis equal
grid on
```

An example **MATLAB** script demonstrating the interface has been added in the package named **eg1.m** and the results are shown in Figure 2. As we observe, the visualization is precise using **MATLAB** than the native plotter, whose functionalities are limited in comparison. For generating high-quality PDF plots, it is recommended to use the **export_fig** package which can be found in `https://github.com/altmany/export_fig`.

Visualization can also be done using ParaView, for example,

```
int[int] Order=[1];
savevtk(FileName,MeshName,FuncName,dataname="VecFun ScalFun",Order=Order);
```

saves a **.vtk**/**.vtu** file for ParaView visualization. FreeFem offers extensive support for ParaView and examples of the figures generated using ParaView can be found in the `README.md` file located in the main repository. For more details on the ParaView visualization, refer to the FreeFem manual.

NOTE: In later versions of iceFEM, ParaView would become the default method of visualization.

2 Macros and Keywords

2.1 Keywords

The iceFEM package consists of a few reserved keywords that can be changed within any program. The package also consists of a list of macros that can be used to solve certain Hydroelasticity problems. They can be found in the script file `macros.idp`. A list of currenlty available reserved keywords are discussed below.

- `isUniIce, isUniCav`. Data Type: `bool`. Variable to switch between of uniform/non-uniform profiles. `Can be changed. Set to default as true`.
- `NModes`. Data Type: `int`. Sets the number of modes in the modal expansion in the open-ocean solution. This is used in the construction of the non-local boundary condition at the ocean/cavity interface. `Set to default as 3.`
- `nev`. Data Type: `int`. Sets the number of in-vacuo modes of vibration of the ice-shelf. This is also the dimension of the reduced system obtained in the final step. `Set to default as 20.`
- `iter`. Data Type: `int`. A variable used to index the solution for batch operations in `MATLAB` such as interpolation. `Default set to 0.`
- `Lc, tc`. Data Type: `real`. The characteristic length and time computed for non-dimensionalization. If not using non-dimensionalization, leave the values of L_c and t_c to be equal to 1. Automatically set if `setProblem` is called.
- `omega`. Data Type: `complex`. The incident frequency computed from the wave period. $\omega = 2\pi/T$.
- `rhoi, rhow, ag, densRat`. Data Type: `real`. The denisities of ice and water, the acceleration due to gravity g and the ratio of densities ρ_i/ρ_w, respectively. Obtained from the `getProperties` function.
- `LL, HH, dd, tth`. Data Type: `real`. The non-dimensional values length of the ice-shelf, cavity-depth, submergence, shelf-thickness. Can be set manually, but consistency needs to be ensured, if meshes are imported from an external source.

  `lambdahat, muhat`. The ratio λ/L_c^2 and μ/L_c^2 where λ, μ are the Lamé parameters.

  `gammahat, deltahat`. The ratio ρ_w/L_c and $\rho_w g/L_c$. `tt`. Data Type: `complex`. Value of the incident wave period, `Tr + 1i*Ti`. Computed from the user-input values of `-Tr, -Ti`. Set automatically.
- `Ap`. Data Type: `complex`. The amplitude of the incident velocity potential. Computed from the incident wave period. Set automatically.

- `ThIce, ThCavity`. Data Type: `mesh`. The variables containing the mesh data for the ice-shelf and the sub-shelf cavity, respectively. `Can be modified to any valid mesh file. See the FreeFem++ manual for more details.`
- `Wh, Vh, Xh`. Data Type: `fespace`. Finite element spaces for the cavity (W_h) and the ice-shelf (V_h, X_h). The space X_h is a vectorial finite element space. `Depends on the finite element mesh. Modified if the mesh is modified.`
- `WhBdy, VhBdy`. Data Type: `fespace`. Boundary Finite element spaces for the cavity (`WhBdy`) and the ice–shelf (`VhBdy`). Used to speed–up the construction of the reduced system by the `macro buildReducedSystemOptim`.
- `k, kd`. Data Type: `complex[int]`. Complex 1D arrays containing the wave-numbers obtained after solving the free-surface dispersion equation with depths H and $H-d$, respectively. The length of the arrays is `NModes+1` and is computed by the `dispersionfreesurface` function.
- `fh`. Data Type: `func`. An external function that is used to specify the non-homogeneous part of the Dirichelt/Neumann boundary condition. Should be specified in the program before obtaining the matrices using `getLaplaceMat` macro.
- `STIMA, BMASSMA, LHS`. Data Type: `matrix<complex>`. Contains the stiffness matrix on the cavity mesh, boundary mass matrix on the ocean-cavity interface. The quantities are computed by `macro getLaplaceMat()` and is generally not changed. Once computed the user can set the `LHS` matrix, for example, `LHS=STIMA+(BMASSMA)`.
- `RHS`. Data Type: `Wh<complex>` . Contains the RHS function corresponding to the function `fh`. The finite element solution, say, `uh` is computed using

```
uh[]=LHS^-1*RHS[];
```

- `B, K, AB, Hmat`. Data Type: `complex[int,int]`. Complex 2D arrays that are the components of the final reduced system. `F`. Data Type: `complex[int]`. Right hand side of the reduced system. The quantities are computed by `macro buildReducedSystem*()`.
- `xi`. Data Type: `complex[int]`. Solution of the reduced system set by the `macro solveReducedSystem` Do not modify.
- `mu`. Data Type: `real[int]`. Variable to store the eigenvalues of the in-vacuo Euler Bernoulli problem. Generated by `macro solveEigenEB()`. Do not Modify.

2.2 Macros

In this subsection, we list a set of predefined macros that can be used when writing a program. The usage of the macros will be discussed in the Tutorial section.

- `macro setProblem()`:

  The `macro setProblem` receives the length of the ice-shelf L, thickness h and submergence d of the ice-shelf, depth of the ocean H and the incident wave frequency ω. The macro also computes the value of the non-dimensional constants along with `lambdahat, muhat, gammahat, deltahat`.

- `macro solveDispersion():`

  Macro to obtain the roots of the dispersion equations

$$-k \tan kH = \alpha, \quad \text{and}$$
$$-k_d \tan k_d(H-d) = \alpha$$

  which updates the arrays `k,kd`.

- `macro setMeshIce(bottom_right_y, middle_x, middle_y):`

  Macro to set simple non-uniform meshes using 3–point cubic spline technique. Updates `ThIce`

- `macro setMeshCav(middle_x, middle_y, bottom_right_y):`

  Macro to set simple non-uniform meshes using 3–point cubic spline technique. `ThCavity`

- `macro solveEigen():`

  Macro to solve the Eigenvalue problem to obtain the in–vacuo modes. Always preceded by:

```
   Xh[int][VX,VY](nev);
   real[int] ev(nev);
```

  and updates the variables `VX,VY,ev`.

- `macro writeEigen():`

  Macro to write the eigenmodes to the disk. Saved in `iceFEM/[WORKING_DIR]/2_Modes`.

- `macro readEigen():`

  Macro to read the Eigenvalue problem to from `iceFEM/[WORKING_DIR]/2_Modes`. Always preceded by:

```
   Xh[int][VX,VY](nev);
   real[int] ev(nev);
```

  and updates the variables `VX,VY,ev`.

- `macro getLaplaceMat(a,b,c):`

  Computes the matrices `LHS, STIMA, BMASSMA`. The `STIMA` is the stiffness matrix, i.e.,

$$[\texttt{LHS}] = \int_\Omega \nabla u_h \cdot \nabla v_h \, dx,$$

  the boundary mass matrix `BMASSMA` corresponding to the boundary with the `integer label 2` i.e., the inner–product

$$[\texttt{BMASSMA}]_{jk} = \int_{\Gamma_2} u_h \, v_h \, ds.$$

Also computes the `RHS` vector corresponding to the boundary with the `integer label 4` and `3` i.e., the inner-product

$$[\texttt{RHS}]_k = \int_{\Gamma_4} (-\texttt{fh})\, v_h\, ds + \int_{\Gamma_3} -i * \texttt{omega} * \texttt{Lc} * \left(\texttt{a} \cdot \texttt{N.x} + \texttt{b} \cdot \texttt{N.y} + \texttt{c} \cdot \texttt{N.z}\right) ds$$

//IF 3D.

and

$$[\texttt{RHS}]_k = \int_{\Gamma_4} (-\texttt{fh})\, v_h\, ds + \int_{\Gamma_3} -i * \texttt{omega} * \texttt{Lc} * \left(\texttt{a} \cdot \texttt{N.x} + \texttt{b} \cdot \texttt{N.y}\right) ds$$

//IF 2D. (`c` can be set to 0).

The boundary labelled `3` corresponds to the shelf/cavity interface and the boundary labelled `4` (Inlet) corresponds to the ocean–cavity interface.

- `macro getLaplaceMatEB(m,rad)`

  Same as `getLaplaceMat(a,b,c)` but to compute the solution of the Laplace's equation using the roots of the Euler–Bernoulli equation `mu`. Argument `m` denotes the mth radiation potential and `rad=0` for rigid–ice ($\int_{\Gamma_3} = 0$) and `rad=1` for moving-ice ($\int_{\Gamma_3} \neq 0$).

- `macro getLaplaceMatDBC(m,rad):`

  Same as `getLaplaceMatEB(m,rad)` but for Dirichlet boundary condition

```
on(4, phih=fh)
```

  on Γ_4, instead of the Neumann boundary condition

```
\int1d(ThCavity, 4)(fh*vh)
```

  on the inlet.

- `macro buildReducedSystem(Xh[int,int] [VX, VY] ,Wh<complex> phi0,Wh<complex>[int] phij(nev)):`

  Build the reduced system using the modal functions `VX, VY, phi0, phij`. Refer to the tutorial on how to assign the functions for a sample ice–shelf problem.

- `macro buildReducedSystemEB(real[int] mu ,Wh<complex> phi0, Wh<complex>[int]phij(nev), complex alpha,real beta, real gamma):`

  Same as `buildReducedSystem`, but for the thin–plate problem. The in–vacuo modes are characterized by the eigenvalues `mu`, but contains extra parameters `alpha, beta, gamma`. Refer to the tutorial on how to assign the functions for a sample ice–shelf problem.

- `macro buildReducedSystemOptim():`

  Optimized version of the `buildReducedSystem` macro where the construction of the reduced system is done using the boundary value of the solution only. The user needs to define a new function `phi00`:

```
WhBdy<complex> phi00=phi0;
```

and an array of boundary functions

```
WhBdy<complex>[int] phijj(nev);
for(int m=0; m<nev; m++)
{  : //Compute phij[m]
   phijj[m]=phij[m];
}
```

Once `phi00, phijj, VX, VY` set, then simply call

```
buildReducedSystem;
```

For more advanced example using `MPI`, see the examples `iceshelf2d.edp, iceshelf3d.edp` etc.

- `macro solveReducedSystem():`

  Solve the reduced system built using the previous macros and store the solution in the variable `xi`.

- `macro writeToMATLAB(uh, Th, solfilename, meshfilename)`

3 Tutorial

In this section, we describe how to solve two ice-shelf problems using the iceFEM package. The first example is the ice-shelf modelled using the Euler Bernoulli beam theory.

3.1 Euler-Bernoulli beam

In this subsection, we solve the example in [1]. The complete example is provided in the package as `simple3.edp`. First, we invoke the necessary modules by importing `macros.idp`.

```
verbosity=0.; //Sets the level of output.
include "macros.idp"
```

Next we set the problem by specifying the number of modes in the series expansions and solve the Dispersion equation to obtain the wave numbers.

```
nev=20; //Number of in-vacuo modes
NModes=5; //Number of open-ocean modes

//Set the problem.
setProblem;

//Solve the Dispersion equation to obtain the wave number arrays k, kd
solveDispersion;
```

The next step is to build the mesh for the cavity. To specify the shelf/cavity interface, we first build a uniform mesh for the ice-shelf. However, this mesh will not be used to compute the solution.

```
isUniformIce=true;//Force uniform mesh for the ice to obtain the shelf-cavity
    interface.
setMeshIce(0,0,0);

isUniformCav=false;
//  A three point cubic spline is used: Args. (midX, midY, endY)
setMeshCav(LL/2., -0.5*HH, -HH);
```

If `isUniformIce/isUniformCav=true`, any option given as arguments will be overridden to default values. The next step is to solve the eigenvalue problem to obtain the in-vacuo modes of the ice-shelf. This is done by simply calling,

```
solveEigenEB; //Solves the Eigenvalue problem to obtain the in-vaco EB modes.
```

The next step is to obtain the non-local boundary condition which is of the form:

$$\partial_x \phi = \underbrace{\mathrm{Q}\phi}_{\text{:Matrix}} + \underbrace{\chi}_{\text{:Vector}} .$$

Two functions are available in the `nonLocal.idp` file to calculate the necessary matrix and vector. The following code block is used to obtain the boundary condition. For more details, see [1].

```
//Matrix MQ stores the Q-operator
matrix<complex> MQ;
//ctilde stores the vector corresponding to the incident wave
complex[int] ctilde(NModes+1);

//Obtain the matrix and vector.
getQphi(4,MQ);
getChi;

//Obtain the function by combining the modes.
Wh<complex> chi1;
for(int m=0; m<NModes+1; m++)
  chi1 = chi1+ctilde[m]*cos(kd[m]*(y+HH))/cos(kd[m]*(HH-dd));
```

The next step is to obtain the diffraction potential in the sub-shelf cavity.

```
// Solve for the diffraction potential
Wh<complex> phi0; //Declare a complex FE function the cavity region.
//Set the external function.
func fh=chi1;

//Call the routine to compute the FE matrices for the problem.
getLaplaceMatEB(0,0); //Indicates the routine to solve for the Diffraction
    potential.

//Set the LHS matrix and solve the problem.
LHS=STIMA+(MQ); //Add the Q-matrix.
set(LHS,solver=sparsesolver);
phih[]=LHS^-1*RHS[];
phi0=phih;

//Plot the result.
plot(phi0,wait=1,fill=1,value=1);
```

Similarly, the radiation potentials can be obtained by

```
//4) Solve for the radiation potential
Wh<complex>[int] phij(nev);
for(int m=0; m<nev; m++)
  {
    fh=0; //0 for radiation potential.
    getLaplaceMatEB(m,1); //Indicates the routine to input the mth mode of
        vibration.
    LHS=STIMA+(MQ);
    set(LHS,solver=sparsesolver);
    phih[]=LHS^-1*RHS[];
    phij[m]=phih;
  }
```

The next step is to build the reduced system which will be solved to obtain the final solution. The mathematics can be found in the Appendix.

Build the Appendix.

```
//Parameters for the system.
complex ndOmega=2*pi/tt;
complex alpha = HH*ndOmega^2;
real beta = 1;
real gamma = densRat*tth;

//Call the routine to construct the reduced system
buildReducedSystemEB(mu, phi0, phij, alpha, beta, gamma);

//Solve the reduced system.
complex[int] xi(nev);
solveReducedSystem; //The solution is stored in xi
```

Finally we solve the reduced system to obtain the modal contributions. The final solution is the linear combinatinon of these coefficients with the corresponding bases for the displacement and potential. The solution can be visualized using any of the means discussed in Section 1. Further, if the user wants to compute the reflection coefficients, a macro **`getRefCoeff`** is available in the module **`macros.idp`**. The following code block computes the reflection coefficient for the problem.

```
complex Ref;
getRefCoeff(4,phi,Ref);

//Print the value.
cout.precision(16);
cout<<"Reflection Coefficient = "<<Ref<<endl<<"|R| = "<<abs(Ref)<<endl;
```

This concludes the first tutorial. In the next tutorial, we will discuss the second model, where the ice-shelf is modelled using 2D linear elasticity equations under plane strain assumptions.

3.2 2D Linear Elasticity

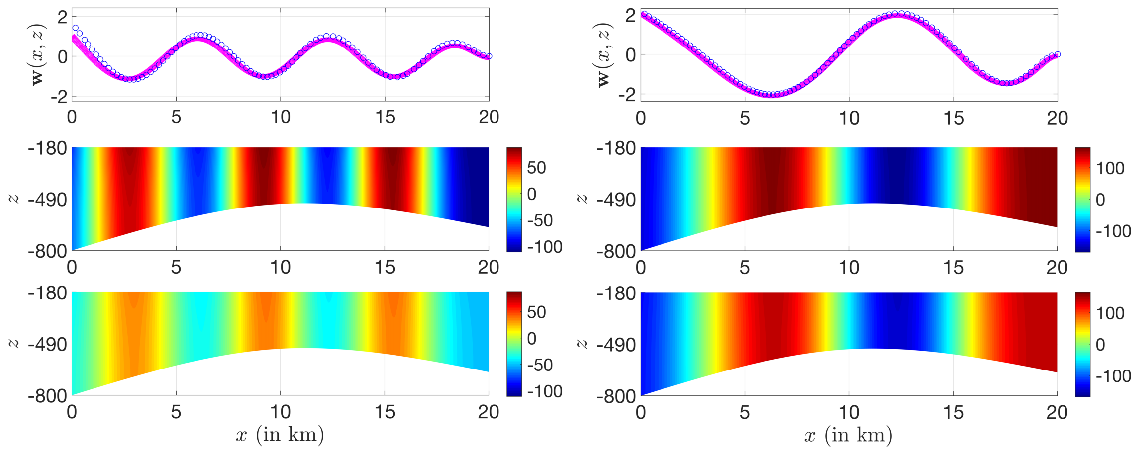
The second example is when the ice is modelled using the 2D elasticity equations under plane strain conditions. In this subsection, the same problem can be solved using 2D linear elasticity for the ice-shelf. The code follows along the same line except for slight modifications. The full code can be found below.

```
verbosity=0.;

macro dimension 2//EOM"
macro fspace 1//EOM"

include "macros.idp"

//Sets up an example problem. Can control input using CMD line args
nev=20;
setProblem;

//Solve the dispersion equation -k tan(k h) = \alpha. -k tan(k (h-d)) = \alpha
solveDispersion;

//Build the meshes.
real botRight=-3.*tth, midPX=3.7*LL/4, midPY=-2.5*tth;
setMeshIce(botRight, midPX, midPY);
real midx=LL/2., midy=-0.5*HH, endy=-HH;
setMeshCav(midx, midy, endy);


//  1) Solve the in-vacuo eigenvalue problem.
Xh[int][VX,VY](nev); //Define an array of fe-function to store in-vacuo modes.
real[int] ev(nev); //Define a real array for the eigenvalues.
solveEigen;
```

```


//  2) Get the Non-local boundary condition
matrix<complex> MQ;
getQphi(4,MQ);
getChi;
for(int m=0; m<NModes+1; m++)
  chi1 = chi1+ctilde[m]*cos(kd[m]*(y+HH))/cos(kd[m]*(HH-dd));

//  3) Solve for the diffraction potential.
Wh<complex> phi0;
func fh=chi1;//Store in fh, the right-hand side function on the ocean-cavity
    interface.
getLaplaceMat(0,0,0);
LHS=STIMA+(MQ);
set(LHS,solver=sparsesolver);
phih[]=LHS^-1*RHS[];
phi0=phih;//Store in phi0;


//  4) Solve for radiation potential.
Wh<complex>[int] phij(nev);
for(int m=0; m<nev; m++)
  {
    func fh=0;
    getLaplaceMat(VX[m],VY[m],0);
    LHS=STIMA+(MQ);
    set(LHS,solver=sparsesolver);
    phij[m][]=LHS^-1*RHS[];
  }

//Build the reduced system and solve it.
buildReducedSystem(VX,VY,phi0,phij);
solveReducedSystem;

//Compute the solution.
Vh <complex> etax, etay;
Wh<complex> phi;
phi = phi0;
for(int m=0; m<nev; m++)
  {
    phi = phi + xi[m]*phij[m];
    etax = etax + xi[m]*VX[m];
    etay = etay + xi[m]*VY[m];
  }

//Compute the reflection coefficient.
complex Ref;
getRefCoeff(4,phi,Ref);
cout.precision(16);
```

```
if(mpirank==0){
savevtk(SolutionDir+"/sol1_"+iter+".vtk",ThIce,[real(etax),real(etay)],dataname
    ="ReDisp",order=Order);
savevtk(SolutionDir+"/sol2_"+iter+".vtk",ThCavity,[real(phi),imag(phi)],
    dataname="ReDisp",order=Order);
    cout<<"Reflection Coefficient = "<<Ref<<endl<<"|R| = "<<abs(Ref)<<endl;
}

//Write data to MATLAB
//iter could be used to index the solution.
if(mpirank==0){
    Vh ux=real(etax), uy=real(etay);
    writeToMATLAB(ux,ThIce,"xDisp"+iter+".bb","meshIce"+iter+".msh")
    writeToMATLAB(uy,ThIce,"yDisp"+iter+".bb","meshIce"+iter+".msh");
    writeToMATLAB(rphi,ThCavity,"potentialCav"+iter+".bb","meshCav"+iter+".msh"
        );
}
```

As guessed, the linear elasticity solution coincides with the Euler Bernoulli solution for thin ice-shelves. The thinness is determined with respect to the incident wavelength that the ice-shelf is subject to. Figure 3 shows the two solutions for a uniform ice-shelf of length 20 km subject to two different incident wave-forcing.



(a) $T = 80\,\mathrm{s}$ (b) $T = 200\,\mathrm{s}$

Figure 3: Comparison results for the ice-shelf vibration for two different wave-periods. The thinness of the ice-shelf is determined with respect to the incident wavelengths. The longer the incident wave (higher T), the better the agreement is, since the front thickness is negligibly small compared to long wavelengths. Hence more deviation can be observed for $T = 80\,\mathrm{s}$ case.

3.3 An example using MPI

In this section an example using MPI and the macro **`buildReducedSystemOptim`** is illustrated. The code snippet inside the macro is written assuming that the eigenfunctions corresponding to the in-vacuo modes and the velocity potentials are restricted to the boundary of the domain. The the construction of the reduced system is parallelized, if the user enables splitting of the meshes. The complete program along with the comments is shown below.

```
//Simple example to demonstrate the ice-shelf toolbox;
//NOTES:
//1)   The functions/macros/keywords from the toolbox is denoted by [iceFEM]
    next to it.
//2)   For a list of KEYWORDS and FUNCTIONS, refer to the manual located in the
     Repository.

verbosity=0.;
real cpu=mpiWtime();
bool debug=true;

macro dimension 2//EOM" (Sets up the dimension of the problem);
macro fspace 2//EOM" (Sets up the FESpace);

include "macros.idp"

real timeTaken;

SolutionDir="1_SIMPLE5/";

macro Sigma(u,v)[2*muhat*dx(u)+lambdahat*(dx(u)+dy(v)),
                 2*muhat*dy(v)+lambdahat*(dx(u)+dy(v)),
                 muhat*(dy(u)+dx(v))]//EOM (Macro to define the stress tensor)"

//Sets up an example problem. Can control input using CMD line args.
//For a detailed list of default args. Refer the manual.
setProblem;

//Solve the dispersion equation -k tan(k h) = \alpha. -k tan(k (h-d)) = \alpha
solveDispersion;

real starttime=mpiWtime();
//Build the meshes.
real botRight=-3.*tth, midPX=3.7*LL/4, midPY=-2.5*tth;

setMeshIce(botRight, midPX, midPY);//[iceFEM] For a list of ARGS, refer to the
    manual

real midx=LL/2., midy=-0.5*HH, endy=-HH;

setMeshCav(midx, midy, endy);//[iceFEM] For a list of ARGS, refer to the manual

```

```
refineMesh;//[iceFEM] Refine Mesh.
splitMesh(isSplit);//[iceFEM] Split Mesh for domain decomposition (isSplit is
    defaulted to 0).

real endtime=mpiWtime();
real difTime=endtime-starttime;//Time the code.
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(mpirank==0)
    cout<<"Time taken for Meshing = "<<timeTaken<<" s"<<endl;


//The respective finite element spaces are
//Vh, Vh<complex> -> P1/P2 on ICE.
//Wh, Wh<complex> -> P1/P2 on CAVITY.
//Xh, Xh<complex> -> [P1,P1]/[P2,P2] on ICE2d
//                -> [P1,P1,P1]/[P2,P2,P2] on ICE3d

Xh[int][VX,VY](nev); //[iceFEM] Define an array of fe-function to store in-
    vacuo modes. (Should be named: [VX,VY])
real[int] ev(nev); //[iceFEM] Define a real array for the eigenvalues. (Should
    be named: ev[])
starttime=mpiWtime();

solveEigen; //[iceFEM] Solve the Eigenvalue problem;


endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(mpirank==0)
    cout<<"Time to solve Eigenvalue = "<<timeTaken<<" s"<<endl;
//readEigen;


//  2) Get the Non-local boundary condition
Wh<complex> chi1; //[iceFEM] Define a function chi1 on the cavity domain for
    the incident wave.
matrix<complex> MQ;//[iceFEM] Define a matrix MQ for the Q-operator.

getQphi(4,MQ);//[iceFEM] Build Q-Operator on boundary 4 of cavity.
getChi;//[iceFEM] Build Function Chi.

for(int m=0; m<NModes+1; m++)
    chi1 = chi1+ctilde[m]*cos(kd[m]*(y+HH))/cos(kd[m]*(HH-dd));

//  3) Solve for the diffraction potential.
Wh<complex> phi0;
func fh=chi1;//[iceFEM] Define and store in keyword fh, the right-hand side
    function on the ocean-cavity interface.
getLaplaceMat(0,0,0);//[iceFEM]
```

```
LHS=STIMA+(MQ);
set(LHS,solver=UMFPACK,eps=1e-20);
phih[]=LHS^-1*RHS[];
phi0=phih;//Store in phi0;
//Interpolate to boundary.
WhBdy<complex> phi00=phi0;


//Solve for radiation potential. [PARALLEL RUN is OPTIONAL]
Wh<complex>[int] phij(nev);
WhBdy<complex>[int] phijj(nev);//Define array of boundary functions
buildParti(nev); //[iceFEM] Build partition depending on the number of CPUs.
complex[int,int] PHIJ(WhBdy.ndof,nev),PHIJProc(WhBdy.ndof,partisize);
starttime=mpiWtime();
for(int m=start; m<=stop; m++)
 {
     func fh=0;
     getLaplaceMat(VX[m],VY[m],0);//[iceFEM]
     LHS=STIMA+(MQ);
     set(LHS,solver=UMFPACK,eps=1e-20);
     phih[]=LHS^-1*RHS[];
     phij[m]=phih; //The full solution is used to visualize.
     phijj[m]=phih; //Interpolate on the BOUNDARY alone and store in matrix.
     PHIJProc(:,m-start)=phijj[m][];
 }
int[int] rcounts1=rcounts*WhBdy.ndof, dspls1=dspls*WhBdy.ndof;
mpiAllgatherv(PHIJProc,PHIJ,rcounts1,dspls1); //Gather the solutions.
endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiSUM);
if(mpirank==0)
    cout<<"Time taken to solve potentials = "<<timeTaken/mpisize<<" s"<<endl;

//Unpack and set to phij locally in all procs (Only boundary)
for(int m=0; m<nev; m++)
    phijj[m][]=PHIJ(:,m);


//Build Reduced System using the modes. [H]{\lambda}={f}
K.resize(nev,nev);
B.resize(nev,nev);
AB.resize(nev,nev);
F.resize(nev);
starttime=mpiWtime();
buildReducedSystemOptim;//[iceFEM]

endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(mpirank==0){
    complex[int] Refm(nev), Reft(nev);

    writeReducedSystem;//[iceFEM] Write the reduced system to a file.
    cout<<"Time taken to build reduced system = "<<timeTaken<<" s"<<endl;

 }

//Solve the reduced system;
solveReducedSystem; //[iceFEM]

//Compute the solution.
Vh<complex> etax, etay, etaxProc, etayProc;
Wh<complex> phi, phiProc;
for(int m=start; m<=stop; m++)
 {
     phiProc = phiProc + xi[m]*phij[m];
     etaxProc = etaxProc + xi[m]*VX[m];
     etayProc = etayProc + xi[m]*VY[m];
 }

mpiReduce(phiProc[],phi[],processor(0),mpiSUM);
mpiReduce(etaxProc[],etax[],processor(0),mpiSUM);
mpiReduce(etayProc[],etay[],processor(0),mpiSUM);

if(mpirank==0){
    //Compute the reflection coefficient.
    cout<<"\n\n\n";
    cout<<"Non-Dim parameter Lc,Tc = "<<Lc<<","<<tc<<endl;
    phi=phi+phi0;
    complex Ref;

    getRefCoeff(4,phi,Ref);//[iceFEM] Compute the Reflection Coefficient

    cout.precision(16);
    cout<<"Reflection Coefficient = "<<Ref<<endl<<"|R| = "<<abs(Ref)<<endl;

    //Write data to Paraview
    //iter is used to index the solution.
    int[int] Order=[1,1];
    savevtk(SolutionDir+"/sol1_"+iter+".vtk",ThIce,[real(etax),real(etay)],[
        imag(etax),imag(etay)], dataname="ReDisp ImDisp",order=Order);
    savevtk(SolutionDir+"/sol2_"+iter+".vtk",ThIce,[real(Sigma(etax,etay)[0]),
        real(Sigma(etax,etay)[1]), real(Sigma(etax,etay)[2])],[imag(Sigma(etax,
        etay)[0]), imag(Sigma(etax,etay)[1]), imag(Sigma(etax,etay)[2])],
        dataname="ReSigma ImSigma",order=Order);
    savevtk(SolutionDir+"/solCavity"+iter+".vtk",ThCavity,[real(phi),imag(phi)
        ],dataname="RePhi ImPhi",order=Order);
 }

starttime=cpu;
```

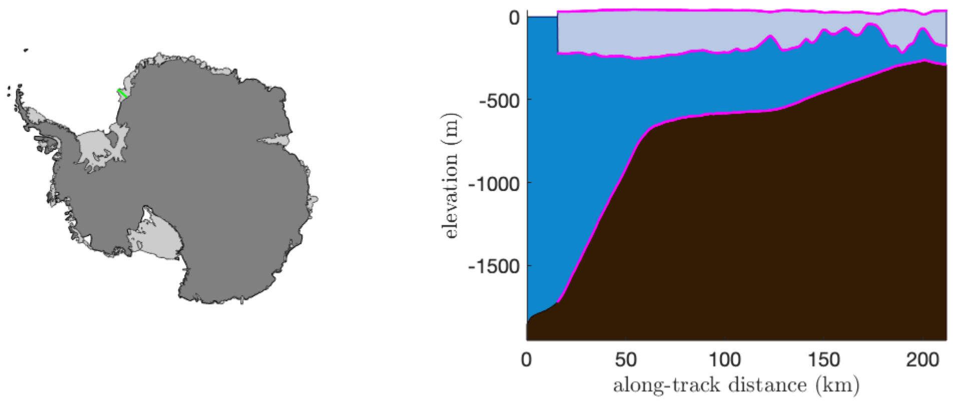


Figure 4: Sample output of **`bedmap2_profile`** command and the user-defined path (Left), shown in Green. The marked path shows the cross section of the Brunt ice-shelf (Right).

```
endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(mpirank==0)
    cout<<"Total time = "<<timeTaken<<" s"<<endl;
```

3.4 Real shelf profiles using BEDMAP2

The last example is solving the linear elasticity problem with data obtained from the BEDMAP2 dataset. In this example the cavity region is assumed to extend into the open–ocean. This module of the software requires **`MATLAB`** and the Antaractic Mapping Tools (AMT) along with the BEDMAP2 dataset to run. Once the requisite packages are installed, launch **`MATLAB`** and run

```
antmap
load coast
patchm(lat,long, [0.5,0.5,0.5]);
bedmap2 patchshelves
[hice,hbed,hwater]=bedmap2_profile();
```

This lauches a **`MATLAB`** figure window showing the map of Antarctica. The user needs to click two points on the map to define a path as shown. Undo points by hitting **`Backspace`**. When you're satisfied with a path you've drawn, hit **`Enter`** to create a profile. To quit the user interface without creating a profile, hit **`Esc`**. The structures **`hice,hbed,hwater`** contain the coordinates of the ice–shelf, sea–floor and the open–ocean profiles, respectively inside the variable **`Vertices`**. The spline data can be extracted from the coordinates using **`MATLAB`** and this will be used by **`FreeFem++`** to reconstruct the cubic spline. The **`MATLAB`** script **`bedMapProfile2`** obtains the profile from the user and the script **`solveBEDMAP2.edp`** solves the problem. The results are shown in Figure 5. The code for the same is

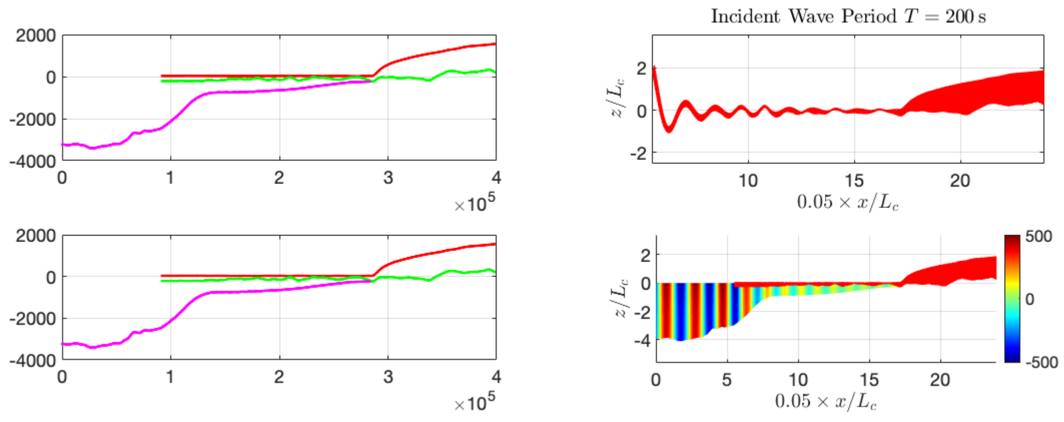


Figure 5: Figure showing the cubic-spline curve representing the Brunt Ice-Shelf and cavity region (left) and the vibration of the ice-shelf subject to an incident wave forcing of $T = 200\,\mathrm{s}$ (Right). The region of the ice-shelf beyond the cavity region is assumed to be fixed to the ground along with the grounding line.

found below and is titled **`solveBEDMAP2.edp`** in the package. Sample output data can be found in **`BEDMAP_Samples`** folder.

```
/*
  WARNING:
  Should be run after splitting the meshes
  ff-mpirun -np 4 splitMesh.edp -hsize 0.02 -isBEDMAP 1
*/

verbosity=0;

macro dimension 2//EOM"
macro fspace 2//EOM"

include "macros.idp"

macro extractFS(fefunc, febdymesh, filename){
  ofstream file(filename);
  for(int m=0; m<febdymesh.nv; m++){
    if(febdymesh(m).y==0)
      file<<real(fefunc(febdymesh(m).x,0))<<"\t"<<imag(fefunc(febdymesh(m).x,0)
          )<<"\n";
  }
}//EOM" End of macro to extract the Free Surface of the function .

macro Sigma(u,v)[2*muhat*dx(u)+lambdahat*(dx(u)+dy(v)),
                 2*muhat*dy(v)+lambdahat*(dx(u)+dy(v)),
                 muhat*(dy(u)+dx(v))]//EOM (Macro to define the stress tensor)"
```

```

real timeTaken;

NModes=5;
SolutionDir="1_BEDMAP2/";
real cpu=mpiWtime();
int isMesh= getARGV("-isMesh",1);
int nborders= getARGV("-nborders",4);

macro writeSolution(var, filename){
  ofstream file(filename);
  file<<var<<endl;
}//Tiny macro to write files

//Construct/Load the meshes.
real starttime=mpiWtime();
iceBEDMAP2(6,isMesh);
if(isMesh){
  refineMesh; //refine the meshes and overwrite the original meshes.
 }
splitMesh(isSplit);
real endtime=mpiWtime();
real difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(nborders!=4)
  dd=0.;

//Solve the dispersion equation.
solveDispersion;
if(mpirank==0)
  cout<<"Solved Dispersion Equation ... "<<endl;

matrix<complex> MQ;
Wh<complex> chi1;
getQphi(4,MQ);//[iceFEM] Build Q-Operator on boundary 4 of cavity.
getChi;//[iceFEM] Build Function Chi.
for(int m=0; m<NModes+1; m++)
  chi1=chi1+ctilde[m]*cos(kd[m]*(y+HH))/cos(kd[m]*(HH-dd));

if(mpirank==0)
  cout<<"Obtained the nonlocal boundary condition"<<endl;

//Solve EigenValue problem
Xh[int][VX,VY](nev);
real[int] ev(nev);
starttime=mpiWtime();
solveEigen;
endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
```

```
if(mpirank==0)
  cout<<"Time to solve Eigenvalue = "<<timeTaken<<" s"<<endl;

//Solve for the Diffraction potential
Wh<complex> phi0;
func fh=chi1;
getLaplaceMat(0,0,0);
BMASSMA=alpha*BMASSMA;
LHS=STIMA+(MQ)+(-BMASSMA);
set(LHS,solver=UMFPACK,eps=1e-20);
phih[]=LHS^-1*RHS[];
phi0=phih;
WhBdy<complex> phi00=phi0;

Wh<complex>[int] phij(nev);
WhBdy<complex>[int] phijj(nev);
buildParti(nev);
complex[int,int] PHIJ(WhBdy.ndof,nev),PHIJProc(WhBdy.ndof,partisize);
starttime=mpiWtime();
for(int m=mpirank*parti; m<mpirank*parti+parti; m++)
  {
    func fh=0;
    getLaplaceMat(VX[m],VY[m],0);
    BMASSMA=alpha*BMASSMA;
    LHS=STIMA+(MQ)+(-BMASSMA);
    set(LHS,solver=UMFPACK,eps=1e-20);
    phih[]=LHS^-1*RHS[];
    phij[m]=phih;
    phijj[m]=phih;
    PHIJProc(:,m-start)=phijj[m][];
  }
int[int] rcounts1=rcounts*WhBdy.ndof, dspls1=dspls*WhBdy.ndof;
mpiAllgatherv(PHIJProc,PHIJ,rcounts1,dspls1);
endtime=mpiWtime();
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiSUM);
if(mpirank==0)
  cout<<"Time taken to solve potentials = "<<timeTaken/mpisize<<" s"<<endl;

//Unpack and set to phij locally in all procs
for(int m=0; m<nev; m++)
  phijj[m][]=PHIJ(:,m);

F.resize(nev);
B.resize(nev,nev);
K.resize(nev,nev);
AB.resize(nev,nev);
starttime=mpiWtime();
buildReducedSystemOptim;
endtime=mpiWtime();
```

```
difTime=endtime-starttime;
mpiReduce(difTime,timeTaken,processor(0),mpiMAX);
if(mpirank==0){
    complex[int] Refm(nev), Reft(nev);
    cout<<"Time taken to build reduced system = "<<timeTaken<<" s"<<endl;
    writeReducedSystem;
 }

//Solve the reduced system.
solveReducedSystem;

//Compute the solution.
Vh <complex> etax, etay, etaxProc, etayProc;
Wh<complex> phi, phiProc;
for(int m=start; m<=stop; m++)
  {
    phiProc = phiProc + xi[m]*phij[m];
    etaxProc = etaxProc + xi[m]*VX[m];
    etayProc = etayProc + xi[m]*VY[m];
  }

mpiReduce(phiProc[],phi[],processor(0),mpiSUM);
mpiReduce(etaxProc[],etax[],processor(0),mpiSUM);
mpiReduce(etayProc[],etay[],processor(0),mpiSUM);

if(mpirank==0){
  //Compute the reflection coefficient.
  cout<<"\n\n\n";
  cout<<"Non-Dim parameter Lc,Tc = "<<Lc<<","<<tc<<endl;
  phi=phi0+phi;
  complex Ref;
  getRefCoeff(4,phi,Ref);
  cout.precision(16);
  cout<<"Reflection Coefficient = "<<Ref<<endl<<"|R| = "<<abs(Ref)<<endl;

  //Write data to MATLAB
  //iter could be used to index the solution.

  int[int] Order=[1,1];
  savevtk(SolutionDir+"sol1_"+iter+".vtk",ThIce,[real(etax),real(etay)],[imag(
      etax),imag(etay)], dataname="ReDisp ImDisp",order=Order);
  savevtk(SolutionDir+"sol2_"+iter+".vtk",ThIce,[real(Sigma(etax,etay)[0]),
      real(Sigma(etax,etay)[1]), real(Sigma(etax,etay)[2])],[imag(Sigma(etax,
      etay)[0]), imag(Sigma(etax,etay)[1]), imag(Sigma(etax,etay)[2])],dataname
      ="ReSigma ImSigma",order=Order);
  savevtk(SolutionDir+"solCavity"+iter+".vtk",ThCavity,[real(phi),imag(phi)],
      dataname="RePhi ImPhi",order=Order);
 }

if(mpirank==0)
```

```
cout<<"\nP: "<<mpirank<<" T = "<<mpiWtime()-cpu<<"\t s"<<endl;
```

References

[1] M. Ilyas, M. H. Meylan, B. Lamichhane, and L. G. Bennetts. Time-domain and modal response of ice shelves to wave forcing using the finite element method. *J. Fluids Struct.*, 80:113–131, 2018.