Using the code for optimising airfoils

Botond Oreg, bzo20@cam.ac.uk

September 1, 2017

1 Different airfoils

1.1 General properties of the airfoil classes

These classes contain the information about the geometry of the domain. Most importantly the mesh is made upon initialising these classes.
A proper airfoil should have the following initialising arguments:

- `resolution`: The number of points on the airfoil. Increasing this number makes the mesh finer as well. It probably shouldn't be lower than 50 and computational time becomes quite large above 300.
- `outerGeometry`: The properties of the outer bounding box of the airfoil at the centre. It is a dictionary with the following entries:
  - `"xFrontLength"`: The distance between the centre and the inflow boundary.
  - `"yHalfLength"`: The distance between the centre and the bottom and top boundaries.
  - `"xBackLength"`: The distance between the centre and the outflow boundary.

  All three of these distances are positive.
- `airfoilParameters`: The list parameters describing the airfoil. It should be always a list even if there is just one parameter.

After initialisation the airfoil should have the following fields:

- `mesh`: The mesh of the domain. It is probably the most important field because it s called every time a function space is declared.

- `dim`: The 'dimension' of the airfoil i.e. the number of its parameters. The list airfoilParameters should have dim number of elements.
- `area`: The area of the airfoil.
- `airfoilPoints`: The list of (x,y) coordinate tuples of airfoil coordinate points.
- `deltax`: The width of the airfoil in the x direction. That is the difference between the x coordinates of the rightmost point and the leftmost point.
- `deltay`: The thickness of the airfoil in the y direction. That is the difference between the y coordinates of the uppermost point and the lowermost point.
- `dSurf`: The object one can use to integrate on a specific boundary. For example to integrate on the airfoil only multiply the function with `dSurf(1)` instead of `dolfin.ds`. The surfaces are marked as:
  - *1*: Airfoil boundary.
  - *2*: Left or inflow boundary.
  - *3*: Bottom and top or wall boundary.
  - *4*: Right or outflow boundary.
- `n`: The surface normal on the boundaries pointing outwards.

An airfoil should have the following callable methods:

- `leftBoundary(x, on_boundary)`: Returns True if x is on the inflow boundary.
- `rightBoundary(x, on_boundary)`: Returns True if x is on the outflow boundary.
- `bottomTopBoundary(x, on_boundary)`: Returns True if x is on the wall boundary.
- `airfoilBoundary(x, on_boundary)`: Returns True if x is on the airfoil boundary.
- `surfaceDeformation(paramID)`: Returns the $\mathbf{V}$ deformation vector space on the surface of the airfoil with respect to changes of paramID$^{\text{th}}$ parameter of the airfoil.

- `areaDerivative()`: Returns a list of the derivatives of the area with respect to the airfoil parameters. The i^{th} element of the list is the derivative of the airfoil area with respect to the i^{th} parameter.

1.2 CircleAirfoil.py

A circular obstacle. It has one parameter its radius (dim = 1).

1.3 JoukowskyAirfoil.py

A Joukowsky-like airfoil. It has three parameters (dim = 3). These are the real and imaginary parts of the parameter c and an angle of rotation (in radians, to the positive direction). The first parameter needs to be smaller than 0.025 otherwise the airfoil becomes overlapping (like an ∞ shape).

1.4 GivenAirfoil.py

This is a special type of parameter free airfoil which is defined by its surface points. For this airfoil the third initialising argument (which would normally be `airfoilParameters`) is a list of the (x,y) coordinates of the airfoil points. For this airfoil dim = 0. The functions `surfaceDeformation()` and `areaDerivative()` are not implemented for this airfoil as it is parameter free. At the moment it is best to use it only for calculating the flow around it. In the future this airfoil might be used for parameter free optimisation.

2 FlowAroundAirfoil.py

This class contains everything about the flow fields.
The initialising arguments are:

- `airfoil`: An initialised airfoil object.
- `v_in`: The v_{in} inflow velocity.
- `mu`: The μ dynamic viscosity.
- `rho`: The ρ density of the fluid.

The class has the following fields:

- `uFull`: The complete flow. Its first two dimensions are the x and y components of the **u** velocity field and its third component is the p pressure field.

- `u_x, u_y`: The x and y components of the $\mathbf{u}$ velocity.
- `u`: The $\mathbf{u}$ velocity vector field.
- `p`: The p pressure field.
- `eigenvalues`: A list of eigenvalues. `eigenvalues[i]` is the tuple (sReal, sImag) the real and imaginary parts of the eigenvalue. The field exists only if either `findEigenvalues()` or `getMaxEigenvalue()` was called beforehand.
- `eigenflows`: A list of eigenflows. `eigenflows[i]` is the tuple (wRealFull, wImagFull) the real and imaginary parts of the complete eigenflows. For both functions the first two components are the x and y components of velocity fields and the third component is the pressure field. The field exists only if either `findEigenvalues()` or `getMaxEigenvalue()` was called beforehand.

The class has the following callable functions:

- `saveSolution((uFile,pFile), label=None)`: Save the calculated $(\mathbf{u}, p)$ flow field into the given files `uFile` (for velocity) and `pFile` (for pressure). If label is specified then the given label is added to the saved flow fields. It can be time, frame number etc. The given label needs to be convertible to a float.
- `dragForceDirect(directionVector)`: The component of the force on the airfoil pointing towards `directionVector`. It is assumed that `directionVector` has unit length. The force component is calculated directly by integrating on the airfoil boundary. It is better to use the function `dragForceGauss()` as that produces good results even for lower resolutions. It is kept mainly to be able to compare the two methods.
- `dragForceGauss(directionVector)`: The component of the force on the airfoil pointing towards `directionVector`. It is assumed that `directionVector` has unit length. The force component is calculated as a volume integral using Gauss' law. For example `dragForceGauss((1,0))` gives the (x directional) drag force on the airfoil and `dragForceGauss((0,1))` gives the (y directional) lift force.
- `forceDerivative(directionVector)`: A list of the derivatives of the component of the force on the airfoil pointing towards `directionVector`. It is assumed that `directionVector` has unit length. The i^{th} element

of the list is the derivative of the force component with respect to the i^{th} shape parameter.

- `costLD()`: The Lift-Drag ratio.
- `costLDDerivative()`: The derivative of the Lift-Drag ratio. A list is returned and its i^{th} component is the derivative of the Lift-Drag ratio with respect to the i^{th} parameter.
- `findEigenvalues(nEigenvalues=60)`: Find the first `nEigenvalues` number of eigenvalues ordered by magnitude. This number should probably be increased as the most unstable mode (which one usually seeks to find) is sometimes found only when `nEigenvalues` is larger. The function creates the fields `eigenvalues` and `eigenflows`.
- `getMaxEigenvalue(nonZeroFreq=False, nEigenvalues=60)`: Find the most unstable eigenvalue i.e. the one with the largest real part. The tuple `(maxSReal, maxSImag), maxSIndex` is returned where `maxSReal` and `maxSImag` are the real and imaginary parts of the most unstable eigenvalue and `maxSIndex` is the index of the eigenvalue. It is true that `eigenvalues[maxSIndex] = (maxSReal, maxSImag)`. If nonZeroFreq is True than the searched eigenvalue has positive imaginary part. The maximum eigenvalue is searched among the first `nEigenvalues` number of eigenvalues ordered by magnitude. The function creates the fields `eigenvalues` and `eigenflows`.
- `eigenvalueDerivative(eigenIndex=None)`: Find the derivative of an eigenvalue. If `eigenIndex` is specified then the derivative of the eigenvalue of that index is searched if not than the derivative of the found most unstable eigenvalue is searched. The tuple `(sRealDer, sImagDer)` is returned where both `sRealDer` and `sImagDer` are list of the derivatives of the real and imaginary parts. Their i^{th} element is the derivative with respect to the i^{th} parameter.

There is another function `costFunctionHessian(stepSize)` which is not working currently. It originally calculated the Hessian matrix of `costLD` with respect to shape parameter changes.

3 Scripts using the code

The following scripts illustrate how to use the programs.

3.1 flow_stab.py

The script optimises the airfoil. It first optimises to minimal drag with the area kept constant. Then the Lift-Drag ratio is maximised with the area kept constant. Then the angle of attack is changed until the flow becomes stable enough. Then the Lift-Drag ratio is maximised while keeping the area and the real part of the most unstable eigenmode constant. The flow is saved during every step and the important parameters are written out into a file.

3.2 methods.py

This script has three methods which can be useful.

- `shapeTransform(originalShape, finalShape, nFrames=150)`: `originalShape` is a list of (x,y) coordinates of the original airfoil and `finalShape` is a similar list of the final airfoil. The function gives a list of list of coordinates. These coordinate lists correspond to a frame during the transformation of the original shape to the final shape. The function might be used upon animating the transformation of one airfoil to the other.

- `constrainedStep(costDer, constraintDer, constraintVal, stepSize, maximise=True)`: Calculating the step in multidimensional space towards the maximum of a function (or the minimum if `maximise=False`) with one constraint. The constraint is that for a function $f = 0$. `costDer` is the gradient of the cost function, `constraintDer` is the gradient of the constraint function f. `constraintVal` is the value of the function at the current point. This is not necessarily zero (but it is usually small) and the step is calculated to make the necessary corrections. `stepSize` gives how large step should be taken. If `maximise` is True then the step towards the maximum value is calculated otherwise the step towards the minimum value is calculated.

- `twoConstrainedStep(costDer, constraintDer_1, constraintVal_1, constraintDer_2, constraintVal_2, stepSize, maximise=True)`: Similar to `constrainedStep` but with two constraints.

3.3 plot_flow.py

This script varies the angle of the airfoil and finds the eigenvalues of the flow. It writes out the real and imaginary parts of each eigenvalue at each angle of incidence. At the end of the file it writes out the most unstable eigenvalues at each angle of incidence separately.

3.4 plot_perturbation.py

This script calculated the frames of a video. The base flow is first calculated and shown for 1.5 *sec*. Then the most unstable mode is added to the base flow and shown for 10 *sec*. With the original input arguments the perturbation should be at the edge of stability and the amplitude is not changed. Then the angle of attack is reduced and the flow is stabilised. It is shown how the perturbation dies away for another 30 *sec*.