CFD with OpenSource software 2013

A course at Chalmers University of Technology taught by Håkan Nilsson

Coupling of Dakota and OpenFOAM for automatic parameterized optimization

Developed for OpenFOAM-1.6-ext

Adam Jareteg October 24, 2014
Reviewed by: Alejandro Lopez and Olivier Petit

Abstract

Shape optimization is an interesting option when faced with uncertainties of a design. It may help the designer come to a better understanding of the behavior of a product and even suggest an optimal solution based on a certain criteria. Since a lot of the methods for optimization are general there are softwares dedicated for solving optimization problems independent of the application. This means that as long as one can formulate a problem and the optimum of that problem in mathematical terms these softwares can assist in finding the optimal solution to that problem. This project is aimed at coupling such an optimization software, namely an Open-Source code called Dakota, with the fluid dynamics calculation toolbox OpenFOAM to enable parameterized shape optimization of objects in a flow.

This report will do mainly two things. Firstly it will describe and discuss the two software, their capabilities in terms of parameterized shape optimization and their limitations. It will describe a new boundary condition implemented in OpenFOAM to complement the existing functions to better allow for parameterized shape optimization. Secondly it will describe one way of coupling the softwares, what files are necessary and how to set up a simulation with the new boundary condition.

Contents

1 Introduction

1.1 Dakota

As indicated in the introduction Dakota is a general purpose optimization toolbox mostly developed by Sandia National Laboratories. It has several optimization strategies implemented and can interact with other software on different levels. Taken from their website it can, amongst others, handle gradient and non-gradient based optimization, reliability and stochastic methods. It can also handle surrogate models, sensitivity analysis and parameter studies. For this project, a very simple optimization algorithm, namely a parameter study, is used to show the principle and the most simple type of interaction is used. That means that Dakota will use OpenFOAM as a black box function that takes some input, specified by Dakota, and transform this to some output, used by Dakota to determine whether the input was ”good” or not.

Installing Dakota is very well described by Sandia and it is mostly a matter of downloading the source code and following a few instructions to that. The only thing to do that is not described is to compile it as a static library because otherwise some library names will collide with OpenFOAMs library names. This is done by adding the following lines to the file `BuildDakotaCustom.cmake` (the file by which the user sets the build options).

```
set(BUILD_STATIC_LIBS ON CACHE BOOL "Set to ON to build static libraries" FORCE)
set(BUILD_SHARED_LIBS OFF CACHE BOOL "Set to ON to build DSO libraries" FORCE)
```

Below follows some brief steps in the installation procedure

1. Download the source code from the Dakota projects website
2. At their website are also some instructions on which other programs that are needed and how they can be installed. Make sure those are installed before compiling.
3. In the downloaded source code there is a file called `INSTALL`. Towards the bottom of this document are the instructions on how to compile the code, follow these (but read the following bullet first).
4. When following the instructions in the `INSTALL` document one bullet will say something similar to ”Update /BuildCustomDakota.cmake to reflect your platform configuration ...”. While doing this make sure that the above lines for compiling Dakota as a static library are included at the bottom of the file. Here the path to the installation is also specified.
5. When the compilation is completed it is recommended to try it out with one of the test cases provided. For example the ”`rosen_multidim.in`” case found in ”`$DAK_INSTALL/examples/users/rosen_multidim.in`”

1.2 Altering shapes in OpenFOAM

There are several ways to get to change the geometry of a simulation, each with its own benefits and drawbacks. One simple way to do this would be to make the mesh with OpenFOAMs `blockMesh` utility. The input file can easily be parameterized and changed with a script. This also have some major drawbacks though. The most obvious is that one is limited to use a mesh that is generated with `blockMesh`. This limits the geometrical possibilities quite substantially. Another way to do it is to generate the mesh in another program. This could yield greater possibilities in terms of geometry but then requires the mesh program to allow for some parametrization.

Both these methods allow for big changes in the geometry since a new mesh is generated for each new case which ensures the mesh quality. One big drawback is that it costs to generate a new mesh. In some cases the cost is only minor while in some cases the cost can be substantial. Another drawback is that when the geometry is changed the old fields is not present anymore. This means that there will take more time to calculate since the initial guess is not as good as it could be.

A third way to change the geometry is to change the boundaries of an existing geometry to whatever is desired and then move the internal mesh accordingly. The movement of the mesh points is decided with a motion solver that ensures that the mesh quality remain as good as possible. The benefits of doing this is that all internal fields remains and no new mesh needs to be generated. The downside is that solving the mesh motion will take computational power and the mesh will most likely suffer in quality compared to the generation of a new one.

The method chosen for this project has been the last one. Partly because a structure for moving meshes in OpenFOAM is available that can be utilized, partly because the outlook of a possibly quicker solver and partly because there might be huge problems finding a meshing program with the right kind of parametrization properties that are needed.

2 Implementations

In OpenFOAM different motion solvers interpret a boundary condition in different ways. Some interpret it as a velocity of the boundary while some view it as a displacement. This is something to be careful with while choosing a solver since it will produce severely different results. For a shape optimization the natural thing would be to view it as a displacement of the boundary. I.e setting a boundary condition would be equal to move it a certain distance. This is also how the new `pointControlledDisplacement` boundary condition is intended. This sets some limitations on which solvers can be used out of the box. It is of course possible to take any velocity interpreting solver and just remove the time multiplication of the boundary condition to get it to behave like a displacement solver but some care should probably have to be taken depending on how the solver is implemented.

The new boundary condition has to be able to do a few things. Firstly it must allow for quite independent movement of different regions of the boundary. It must be able to, quite freely, alter the shape of the object to be able to properly cover a domain of shapes. Secondly it must be able to do this with only a few numbers set by Dakota. There are also here several ways of doing this. One could imagine to do this with a b-spline and fitting the boundary to it. The way this is done in this project is via a set of control points distributed along the boundary of interest. The movement of these points is then interpolated to a region of the boundary which follows these control points in some predetermined way. This is an established way to reduce the handling of a shape to the handling of a number of points. This allows for some great possibilities for big shape alterations since specific regions can be moved totally independent of other regions but it can also introduce some unwanted behavior depending on how the movement of the control points is interpolated to the boundary. As a simple example, if one were to change a straight line with one control point it can take the shape of a triangle if the movement is linearly interpolated or as a half circle if the movement is interpolated in a different way. This dependence on the interpolation scheme is something that needs to be considered when performing the optimization as it can introduce big limitations to the procedure.

To summarize a bit before the details begin, the `pointControlledDisplacement` condition will have to:

- Take a user specified list of control points
- For each control point determine an influence region
- When a control point moves interpolate this movement to the right boundary region

2.1 `pointControlledDisplacement` boundary condition

2.1.1 User input

There are three things the user will need to specify in the boundary condition. Firstly a list of the approximate location of the control points put in the `0/pointDisplacement` file otherwise also used by the displacement solver. The control points will be moved to the closest boundary point for better control of exactly how much it moves. The numbering is here important to keep track of since each control point will be referred to by the number it has in this list starting with 0. Below this done on rows 31-35, here five points are specified three of which will be able to move where as two will be used to determine the interpolation.

```
29      pointHandles
30      (
31        (-0.5 0 0)
32        (0.5 0 0)
33        (-0.4 0.25 0)
34        (0.4 0.25 0)
35        (0 0.5 0)
36      );
```

The second thing needed is a list which specifies which regions each point will influence and as with the previous list also put in the `0/pointDisplacement` file. This is done by, for each control point, specifying ”left” and ”right” bounding control point. There is a further explanation to the ”left” and ”right” and how the influence regions are determined in 2.1.2. This makes the movement of each control point only extend between two other control points and it is the feature that allows for very region specific movements. The first two numbers are the left and right bounding points while the third is a flag for telling if the points can be moved or used for the interpolation determination. A 1 means that the point is fixed while 0 means it can move. In this example the first two points are fixed while the three last ones can move and are bounded by some control points.

```
37      handleRelations
38      (
39        // (x y z), x-left boundary point, y-right boundary point, z-1=fix,0=variable
40        (0 1 1)
41        (0 1 1)
42        (0 3 0)
43        (2 1 0)
44        (0 1 0)
45      );
```

The third thing needed to complete the boundary condition is a new file called `pointMove` which contains a list of movements for each control point. It is this file that tells the code that a point will move and by how much. Like the previous two lists each entry corresponds to the numbering they have in the specification on there locations. As an example, on row 22 it can be seen that the third control point will be moved by a certain

distance. This file can be put in several time directories if one needs to change the shape during the simulation and this list will be the main way that Dakota communicates with OpenFOAM.

```
17 displacement
18 (
19   // Need one entry for every point, even fix points
20   (0 0 0)
21   (0 0 0)
22   (0.1 0 0)
23   (0 0 0)
24   (0 0 0)
25 );
```

2.1.2 Determining the influence region of a control point

As described in previous sections this boundary condition takes a set of control points and interpolates their movement onto the boundary of the geometry. To be able to do this it needs to sort out which control points will influence which regions of the boundary. As in the above section the user specifies this as a set of bounding points which are to be interpreted as the ”left” and ”right” neighbor. In figure 1 a contorl point and its neighbouring points is displayed.

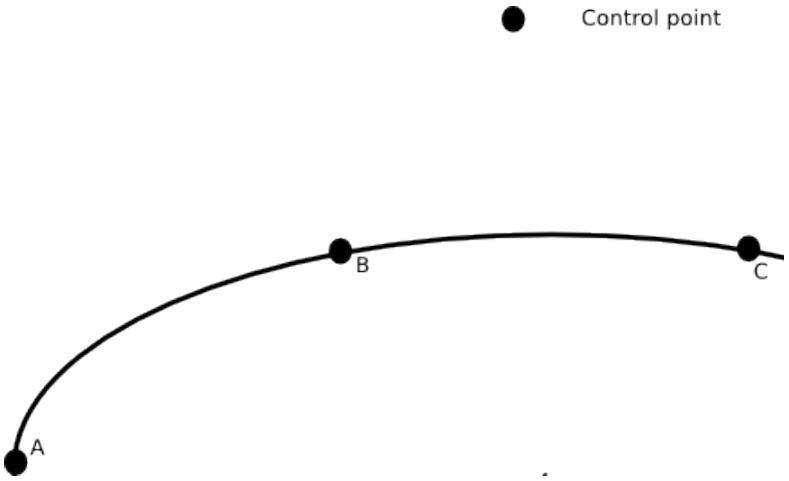

Figure 1: Schematic figure of a control point (B) and its neighbouring control points (A and C)

As stated many times each control point will have a region of influence which is determined via two bounding points. The interpolation will thus be done so that the movement of the boundary node closest to the control point will be moved with a factor one of the control points movement. This factor will then decrease for boundary nodes further away from the control points down to zero for boundary nodes that lie at or beyond the bounding points. Figure 2 shows an example of how this dependency might look where mesh node 1 have moved more than mesh node 2. The determination of

which nodes lie within the interval of the bounding points and which lie outside is a major part of the boundary condition.

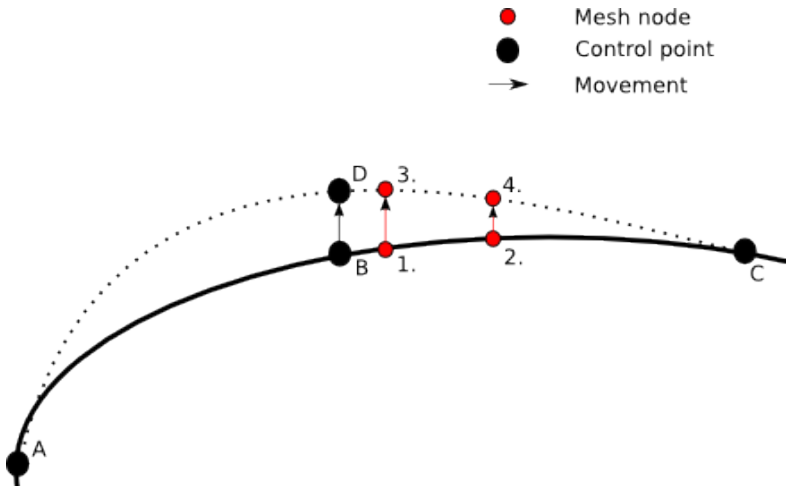

Figure 2: Schematic figure of movement of a control point (B to D) and the corresponding movement of mesh nodes (1. to 3. and 2. to 4.)

The nodes on the boundary are stored in a list which, for this project, is assumed to be in no specific order to make the code as general as possible. The algorithm to determine which boundary nodes will be affected by a control point is then based on distance (fig. 3).

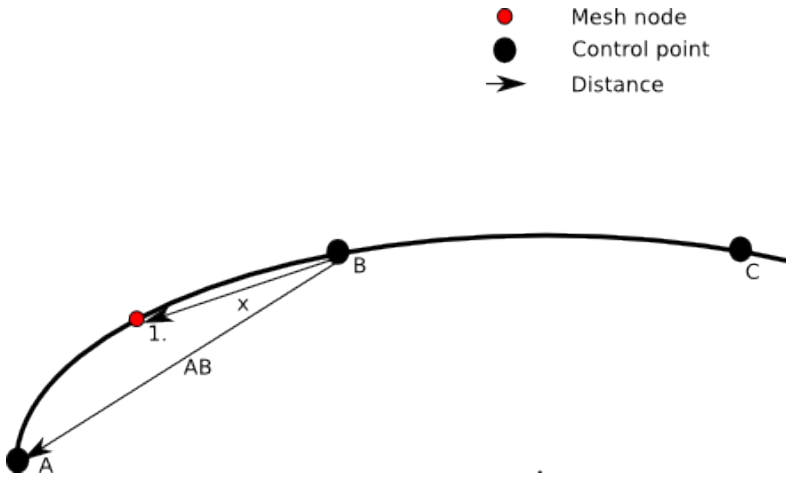

Figure 3: Schematic figure of distance from a control point (B) to mesh node 1. and to a neighbouring control point (A)

If the distance from a node on the boundary to a control point (”x” in fig. 3) is smaller than the distance from a bounding point to a control point (”AB” in fig. 3) then the node is to be affected from the control point. For this to work as intended a few restrictions,

or checks, have to be made. First some sort of check that the node is on the ”correct” side of the boundary. Imagine a very slender geometry, then there is great risk that a distance based algorithm will find points on two different sides of the geometry. The nodes on two of the sides of the geometry will be close to each other compared to the nodes on the other two sides. This restriction is implemented with the help of a user specified ”stopping line”. The user will have to specify two points that are fixed in space and that are to be used to ensure that a control point will only affect nodes on the correct side. A schematic picture is shown in figure 4 with stopping line ”S”.

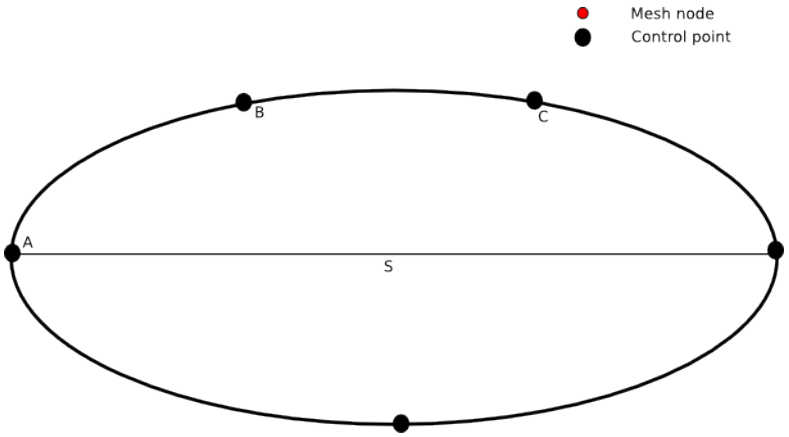

Figure 4: Schematic figure of a slender geometry with a stopping line

Secondly boundary nodes will have to be checked on which side of a control point they lie. The reason is that the distance to the control points will be compared to the distance to a bounding point and the correct bounding point has to be compared to. When all is sorted out, the interpolation will be done via a cos^2 function with the period such that the movement of a node at the position at the bounding point will be zero and increasing to one at a node at the control point. The choice of that particular function is due to that the movement of boundary part between two control points with the same movement will be the movement of those points. It also does not introduce as much sharp edges as a linear function would.

In figure 5 the entire basic idea of the boundary condition is visualized. The ellipse is the geometry to change with a few control points (A,B,C) spread along its edge. Only two mesh nodes (1. and 2.) are shown to illustrate the behavior. The ”stopping line” (S) can as well be seen which makes sure the movement is isolated to the correct side. When the points have moved (to 3. and 4.) via the control point (B to D) the movement is interpolated between the two bounding points (A and C).

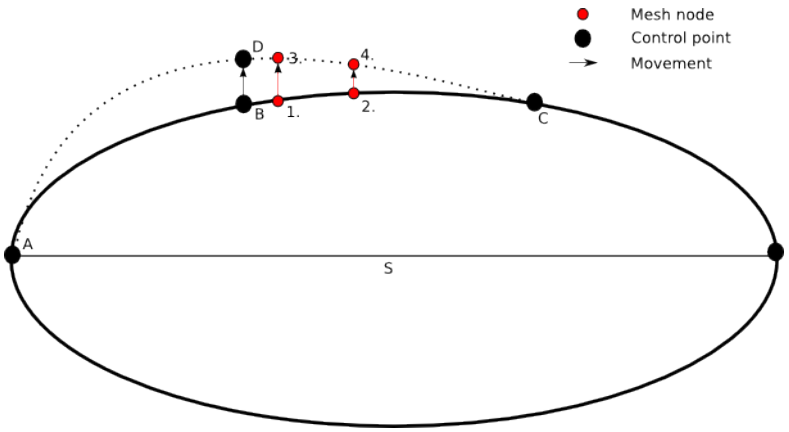

Figure 5: Schematic figure of how the boundary condition works

2.1.3 `pointControlledDisplacement` implementation

First the constructor:

```
53 pointControlledDisplacement::
54 pointControlledDisplacement
55 (
56     const pointPatch& p,
57     const DimensionedField<vector, pointMesh>& iF,
58     const dictionary& dict
59 )
60 :
61     fixedValuePointPatchVectorField(p, iF, dict),
62   movePoints_(dict.lookup("pointHandles")),
63   handleRelations_(dict.lookup("handleRelations")),
64   weights(movePoints_.size()),
65   movement(this->patch().localPoints().size(),vector(0,0,0)),
66   displacements_(movePoints_.size(),vector(0,0,0))
67 {
68   snapToBoundary();
69   setWeights();
70   updateCoeffs();
71 }
```

The constructor needs to find two new keywords in the file `pointDisplacement`, which is a file used by the displacement solver. `pointHandles` which is the location of the control points and `handleRealtions` which is how the control points will relate to one another. The constructor will also run the functions `snapToBoundary` which moves the control points to their closest boundary nodes, `setWeights` which determines how much of the control points movements will be transfered to each individual boundary node and `updateCoeffs` which is the function the displacement solver calls to update the boundary movement.

The `snapToBoundary` function simply loops over all boundary nodes and all control points and finds the closest boundary node for each control point.

```
164 void pointControlledDisplacement::snapToBoundary()
165 {
166   vectorField lP(this->patch().localPoints());
167   scalarField closestPoint(movePoints_.size(),1);
168
169   forAll(lP,i)
170   {
171     forAll(movePoints_,j)
172     {
173       if (
174           mag(movePoints_[j]-lP[closestPoint[j]])>mag(movePoints_[j]-lP[i])
175         )
176       {
177         closestPoint[j]=i;
178       }
179     }
180   }
181
182   forAll(movePoints_,k)
183   {
184     movePoints_[k]=lP[closestPoint[k]];
185   }
186 }
```

`setWeights` is the function that handles the interpolation, or how much of a control points movement shall be transfered to the boundary nodes. The first part of it is a loop that finds the ”stopping line” specified.

```
188 void pointControlledDisplacement::setWeights()
189 {
190   vector s(0,0,0);  // Normalized stop vector
191   vector O(0,0,0);  // Origo vector
192   vector a(0,0,0);  // 90 deg to stop vec
193   vector p(0,0,0);  // Current handle relative 0
194   vector lo(0,0,0); // Current point relative 0
195   vector bpl(0,0,0);  // Left bounding point
196   vector bpr(0,0,0);  // Right bounding point
197   vectorField lp=this->patch().localPoints();
198   int A=0;
199   int B=0;
200
201   // Set stopping line
202   forAll(handleRelations_,i)
203   {
204     if ( handleRelations_[i].z() > 0)
205     {
206       A = handleRelations_[i].x();
207       B = handleRelations_[i].y();
208       if (A != B)
209       {
210         s = movePoints_[B]-movePoints_[A];
211         s = s/mag(s);
212         O = movePoints_[A];
213         break;
214       }
215     }
216   }
```

The second, and last, part of `setWeights` loops over all control points and for each control point checks which boundary nodes should be affected by it. This creates a list containing one list for each control points with the amount of influence the control point has on the boundary nodes ranging from zero to one. These lists will later be multiplied with the corresponding movement for each control point and then added together to form the overall movement of the boundary. Noteworthy is also the last condition in the if statements $lo.z() < 0.0001$. This is due to the way OpenFOAM handles two dimensionality. Since it is a three dimensional domain but with only one cell in the third direction, the boundary condition must ensure that the displacement only appears on one side of the cells and the displacement solver will then move the rest of the nodes. The `snapToBoundary` will make the control points move to one side of the domain and the check ensures that the movement occurs at that side since the influence of a control point is distance based.

```
218   // Set weights
219   forAll(movePoints_,i)
220   {
221
222     List<scalar> tempList(lp.size(),0.0);
223
224     if (handleRelations_[i].z()<1)
225     {
226       p = movePoints_[i] - O;
227       a = p - (p & s)*s;
228
229
230       // This needs the points to be given sorted
231       bpl = movePoints_[handleRelations_[i].x()]-O;
232       bpr = movePoints_[handleRelations_[i].y()]-O;
233
234       forAll(lp,j)
235       {
236         lo = lp[j]-O;
237
238         if ( correctSideOfStop(a,lo) && leftSideOfPoint(lo,p,s,bpl) && lo.z()<0.0001 )
239         {
240           // Compare to bpl
241           tempList[j]=linearWeight(p-lo,p-bpl);
242
243         }
244
245         else if (correctSideOfStop(a,lo) && rightSideOfPoint(lo,p,s,bpr) && lo.z()<0.0001)
246         {
247           //Compare to bpr
248           tempList[j]=linearWeight(p-lo,p-bpr);
249         }
250
251       }
252     }
253
254     weights[i].append(tempList);
255   }
256
257 }
```

The actual movement of nodes will be done by the displacement solver but it uses the displacement of the boundary as a boundary condition when solving and it asks for that via the `updateCoeffs`. The function first checks if the boundary is already updated and if not, it tries to read the file called `pointMove`. As stated in 2.1.1 it is in this file where the movement of each individual control point will be.

```
110 void pointControlledDisplacement::updateCoeffs()
111 {
112
113   if (this->updated())
114     {
115         return;
116     }
117
118     const polyMesh& mesh = this->dimensionedInternalField().mesh()();
119
120
121   IOdictionary movementdict
122   (
123     IOobject
124     (
125       "pointMove",
126       mesh.time().timeName(),
127       mesh,
128       IOobject::READ_IF_PRESENT,
129       IOobject::AUTO_WRITE
130     )
131   );
```

The last part of this function first checks if something was read from the file. If it was, the function first resets the boundary displacement to its original shape. It then adds the contribution from each control point to the total displacement of the boundary and lastly updates the boundary displacement.

```
134   if (movementdict.readIfPresent("displacement",displacements_))
135   {
136       movement.resize(movement.size(),vector(0,0,0));
137
138     forAll(displacements_,i)
139     {
140
141       forAll(movement,j)
142       {
143         movement[j] = movement[j] + weights[i][j]*displacements_[i];
144       }
145     }
146   }
147
148   Field<vector>::operator=(movement);
149   fixedValuePointPatchVectorField::updateCoeffs();
150 }
```

2.1.4 Limitations

Due to the scope of this project there are some strong limitations to this implementation. Firstly it is a two dimensional implementation and even though a three dimensional version is possible as an extension to this it is quite a lot of work to get it there. Secondly the boundary condition requires quite ”nice” geometries. This means geometries that do not change in too strange ways in between the control points since the algorithm to find which nodes each control point will influence is distance based. Thirdly it requires a geometry where the ”stopping line” can be set in a reasonable way since if it intersects a region associated to a control point a part of that region will not be moved with the control point since it is then regarded as the ”wrong” side of the geometry (see 2.1.2).

2.2 The Dakota to OpenFOAM communication

2.2.1 Dakota user input file

Dakota uses, much like OpenFOAM, a control file for its simulations. In this specific case setup a simple parameter study will be done to show the principle of simulation. This document will only describe the settings used for this project and not generally how Dakota is set up. For a description on the different uses of Dakota the reader is referred to other documentation such as Sandias user guide of Dakota.

```
# Usage:
#   dakota -i xxx.in -o run.out > stdout.out

strategy,
    graphics
    tabular_graphics_data
        tabular_graphics_file = 'table_out.dat'
    single_method

method,
  multidim_parameter_study
    partitions = 1 1 2

model,
  single

variables,
  continuous_design = 3
    lower_bounds   -0.1   -0.1   -0.4
    upper_bounds   0.1    0.1    0.4
    descriptors    'x1'   'x2'   'x3'

interface,
  fork
    analysis_driver = 'a_driver'
    parameters_file = 'params.in'
    results_file    = 'results.out'
    work_directory directory_tag
    template_directory = 'temp_dir'
    named 'workdir' file_save  directory_save

responses,
  num_objective_functions = 1
```

```
34   no_gradients
35   no_hessians
```

`strategy`
: This section contains the general settings for the simulation. `graphics` makes Dakota show the result in its graphical window, `tabular_graphics_data` specifies that the results should be saved and the next line specifies to what file. `single_method` specifies that only one method will be used.

`method`
: This section contains information on what optimization method will be used. In this case `multidim_parameter_study` is a simple parameter study of several parameters. The `partions` keyword specifies how many intervals the variable domains should be split into. In this case the domains of the first and second parameter should be split into one interval, meaning that two values will be tried one in each end of the interval. While the domain of the third parameter will split in two equidistant intervals, meaning three values will be tried.

`model`
: This section specifies how many and which models will be used, or in other words how Dakota will map the input to the response. In this case just a single model will be used, which is the default behavior.

`variables`
: The `variables` section handles the variables of the optimization. In this case there will be three variables with names $x1 - x3$ and some lower and upper bounds. Here some care has to be taken when performing the simulation because these variables will be imported to OpenFOAM and used to move the boundary. In this case there will be three control points moving and they will have certain limits on how much they can move before the simulation crashes.

`interface`
: The section `interface` is the part where the user specifies to Dakota how the variables will be mapped to a response. The `fork` keyword tells Dakota that the response will be provided by some external source, in this case OpenFOAM. Here a few things are necessary, first a script that is run for each simulation in this case called "`a_driver`". This script will be responsible for handling the input file "`params.in`" that Dakota creates for each set of variables tested and making sure that a response to that file will be present in a file called "`results.out`". Secondly Dakota will do each iteration of the simulation in a new directory, line 28 and 30 just specifies their names and that they are to be saved after the simulation is complete. The `template_directory` allows the user to have a directory linked into each of the working directories if there are some files the simulation will always need to see. When coupling Dakota and OpenFOAM in this project most of the coupling will be in the analysis driver and in the use of the template directory. As

shown later, the template directory will contain all the files for a base case setup and will be the starting point for each iteration step.

`response`
: Here the user specifies what kind of response Dakota will receive. In this case one response with no gradient or hessians.

2.2.2 The driving script, `a_driver`

The script that Dakota runs each time will first receive the `params.in` file from Dakota and use a utility provided by Dakota called `dprepro`. This takes a file, in this case `pointMove.template`, finds all occurrences of the variable names specified in the control file of Dakota and replace them with the, for the iteration, current values. The script will then move the file that `dprepro` just created, in this case called `pointMove.in` into the time directory that OpenFOAM will start its simulation from. The OpenFOAM simulation then starts and lastly the script calls another script which calculates the desired output and saves it in the file given by the driver.

```
1  #!/bin/sh
2
3  # $1 is params.in FROM Dakota
4  # $2 is results.out returned to Dakota
5
6  # Replacing the variables with the current values
7  dprepro $1 pointMove.template pointMove.in
8
9  # Setting up the OpenFOAM run
10 cp pointMove.in 0.1/pointMove
11 icoDyMFoam > log
12
13 # Calculating output to Dakota
14 ./calc_out.py $2
```

2.2.3 Calculating the response

The response function in this project is the drag coefficient of the geometry. OpenFOAM has functionality for calculating this which can be specified in the `controlDict` of the simulation. In this case that will look like the following:

```
53 functions
54 {
55   forces
56   {
57     type         forceCoeffs;
58     functionObjectLibs ( "libforces.so" );
59     outputControl timeStep;
60     outputInterval 1;
61     patches
62     (
63       cylinder
64     );
65     pName        p;
66     UName        U;
```

```
67      rhoName rhoInf;
68      log         true;
69      rhoInf      1;
70      CofR        ( 0 0 0 );
71      liftDir     ( 0 1 0 );
72      dragDir     ( 1 0 0 );
73      pitchAxis   ( 0 0 0 );
74      magUInf     1;
75      lRef        1;
76      Aref        1;
77    }
78  }
```

The output from the above function is then manipulated some to finally come to the desired response for the iteration. The script for doing this first begins with importing some libraries and it then reads in the file with the data to process. In the data file one drag coefficient have been calculated for each timestep but since Dakota only wants one response value the mean of almost all outputs are taken. A few values in the beginning are being disregarded due to that the calculations will have to settle some from changing the geometry. The last thing the calculation script does is to save the calculated value to the specified file.

```
1  #!/usr/bin/python
2
3  import numpy as np
4  import sys
5
6  arr=np.loadtxt('forces/0.1/forceCoeffs.dat', delimiter='\t')
7  cd=np.mean(arr[10:,1]);
8
9  f=open(str(sys.argv[1]),"w");
10 f.write(str(cd))
11 f.close();
```

2.2.4 The template directory

As mentioned in 2.2.1, Dakota can use a directory as a template for each iteration in the simulation. This will be linked into each of the working directories and can therefore always be used. In this project it will contain the entire OpenFOAM setup and for each iteration a new calculation will be done with OpenFOAM starting from the base case. The base case is first run up to a certain time to get a nicely settled flow as a good starting point for all coming simulations. The `controlDict` is then updated to start from this time and the case is set to run for another, longer, time. In the template directory lies also the template file `pointMove.template` used by `dprepro` to create each iteration specific movement file.

2.3 Summary of the overall simulation procedure and all required files

In this project the procedure for doing a shape optimization with Dakota and OpenFOAM is the following:

1. Create and run a base case for the optimization.
2. Start Dakota and for each step in the iteration it will:
   (a) Write the current variables to the file `params.in`
   (b) Call the script `a_driver`
   (c) `a_driver` will call `dprepro` which inserts the current variables to the right place
   (d) Run an OpenFOAM case with the current variables
   (e) The response is calculated and saved to `results.out`
3. Post-process the results

Except for the normal OpenFOAM case setup the following files will be required (with the name of the files used in the case SEC REF):

- Dakota control file, `cyl_show.in`
- Analysis driving script, `a_driver`
- Response calculation, `calc_out.py`
- `pointDisplacement` for setting up the new boundary condition
- `pointMove.template` for setting a movement of the new boundary condition for each iteration

For this simple parameter case study this could have easily been done in other ways but it is very simple to change to a, for example, gradient based optimization. It is a matter of changing a few lines in the Dakota control file and the procedure will still be the same. When using the template directory one also speeds up the simulation since the flow for most of the domain will be close to equilibrium. Important to notice is that `a_driver` will put the new movement in a certain time directory and it is important that OpenFOAM starts the simulation from that specific directory. It is also important to put reasonable values for the variables in the Dakota control file since OpenFOAM will move the mesh and cannot handle too big mesh deformation before it will crash.

3 Running a case

The following requires access to Dakota

First of all download the case files from the homepage to the course in which this report is written, http://www.tfd.chalmers.se/ hani/kurser/OS_CFD_2013/.

Load the OpenFOAM-1.6-ext environment, `OF16ext` and move the downloaded files to `$WM_PROJECT_USER_DIR`. Then run the following commands to unpack and compile.

```
cd $WM_PROJECT_USER_DIR
tar -xvf pointCTut.tar
cd src
wmake libso
run
cd pointCTut
```

The case can be directly run by typing:

```
dakota -i cyl_show.in -o run.out > run.log
```

A script for assembling a directory for better visualize the change of geometry is provided and can be run by:

```
./mk_pltdir.py
```

To view the results either open the output file `table_out.dat` for a table of the used input and received response or run the following for looking at the geometry changes:

```
cd pltdir
paraFoam
```

In figure 6 the initial shape of the tutorial case can be seen. In figure 7 and figure 8 the shapes have been changed with the new boundary condition.

Figure 6: Initial geometry

Figure 7: Shape after a while in the optimization

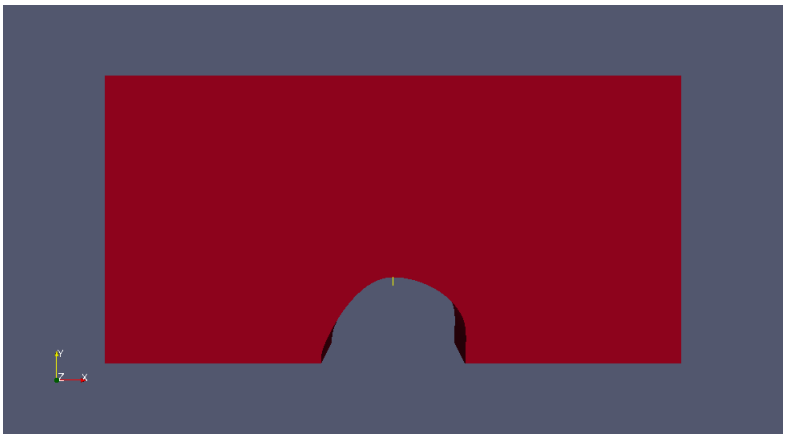

Figure 8: Shape after a while in the optimization

4 Discussion

This report has taken one approach to how it would be possible to couple Dakota and OpenFOAM. There are surely several other ways this can be done and which way to do it is highly application specific. In this project a new boundary condition has been implemented that lets the user change the shape of a boundary via a set of control points. This has then been used to do a simple show of principle to how one could do a shape optimization using these two softwares. Some limitations exist, mostly due to the boundary condition, which are important to be aware of. First of all the boundary condition is implemented only for a two dimensional case. Secondly it can not handle any arbitrary shape. It needs to be able to set the ”stopping line” previously written about and it needs shapes that do not behave too crazy in between the control point since the algorithm is distance based. What it do can handle is a lot of simpler geometries such as wings or convex polygons. It is also important to notice that the project has not been about doing an actual shape optimization of an application. When doing that some care would probably have to be taken regarding using OpenFOAM as a black box without further thought. Generally speaking, flows are more or less chaotic and small perturbations to a geometry could have major impact on responses. This could lead to great difficulties in analyzing the response function due to it being very noisy. But if care is taken and the simulations are carefully set up, shape optimization could be of great aid in exploring possibilities and this project shows that it is very possible to do with OpenSource softwares with quite limited resources.