AP-266 Aerostructural Optimization
Homework 02

Cap Eng **Ney** Rafael Secco
ney@ita.br

March 24th, 2019

Instructions

- Due date: April 5th, 2019. 11:59 PM.
- Send the solutions to ney@ita.br.
- Send the source code as well. The code should have comments.
- You may discuss solution approaches with other students, but you must write your own solution. Do NOT share your code.
- Late delivery penalty: 20% of the total score per late day.

1 Newton's Method

Consider the multidimensional quadratic function below:

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T\mathbf{P}\vec{x} + \vec{q}^T\vec{x} + r \tag{1}$$

where $\mathbf{P}$, $\vec{q}$, and r are constant, and $\vec{x}$ is a vertical vector. Also, $\mathbf{P}$ is a symmetric positive-definite matrix. Let $\vec{x}_0$ be the starting point for the Newton's Method. Prove that the Newton's Method converges to the MINIMUM of the quadratic in a single step.

2 Gradient-based Optimization

Let's continue using the Rosenbrock's function from the previous homework assignment:

$$f(\vec{x}) = \sum_{i=1}^{n-1}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \tag{2}$$

where $\vec{x} = [x_1,\ x_2,\ \cdots,\ x_n]$.

2.1 Part a)

Write Python3 programs to minimize the Rosenbrock's function **using analytical gradients** for any dimension n using:

1. The Conjugate Gradient (CG) Algorithm, starting at the point $\vec{x} = [0,\ 0,\ \cdots,\ 0]$
2. The BFGS Algorithm, starting at the point $\vec{x} = [0,\ 0,\ \cdots,\ 0]$

These two algorithms are already implemented in the `scipy.optimize.minimize` module, just like the Nelder-Mead Simplex method (see the documentation **here**).

Remember to create a function that provides analytical gradients to the `minimize` function. For example, if you have a function named `rosen_grad` that computes gradients, you should call the `minimize` function with:

```
result = minimize(objfun, x0, ..., jac=rosen_grad, ...)
```

Solve the optimization problem for $n = 2$, $n = 4$, and $n = 6$. Provide the final values of design variables, objective function, and number of function evaluations.

2.2 Part b)

Now let's check how these algorithms scale with the number of design variables. Create a plot showing the number of function evaluations with respect to the number of design variables of the problem for these algorithms. Plot all curves in the same plot. Also add the curves from the Nelder-Mead and Differential Evolution methods. Try reaching at least $n = 20$.

ATTENTION: The cost of computing jacobians with reverse AD is has roughly the same order of magnitude as the function evaluation. Therefore, use the number of functions evaluations (*results.nfev*) plus the number of jacobian evaluations (*results.njev*) as the total number of evaluations when comparing CG and BFGS agains the gradient-free methods

Do your best to make a fair comparison between the algorithms, for instance, by using the same solution tolerance. To check that, make another plot with the optimized objective function value versus the number of design variables for all optimization algorithms. Plot all curves in the same plot and use a log scale for the vertical axis.

The computational cost of these methods is roughly proportional to what power of n?

3 Algorithmic Differentiation

This exercise will help you continue coding the lifting-line optimization code that is required for the final report.

In the previous assignment, you implemented a Fortran subroutine to compute the velocity induced at an arbitrary point by a set of linked vortices. Recall that for n_v linked vortices, we get a 3 by n_v matrix of aerodynamic influence coefficients:

The lifting line theory requires that we compute the velocities induced by the linked vortices at a set of control points. Each horseshoe vortex has a corresponding control point at the mid point of the finite vortex segment, as shown in Fig. 1.

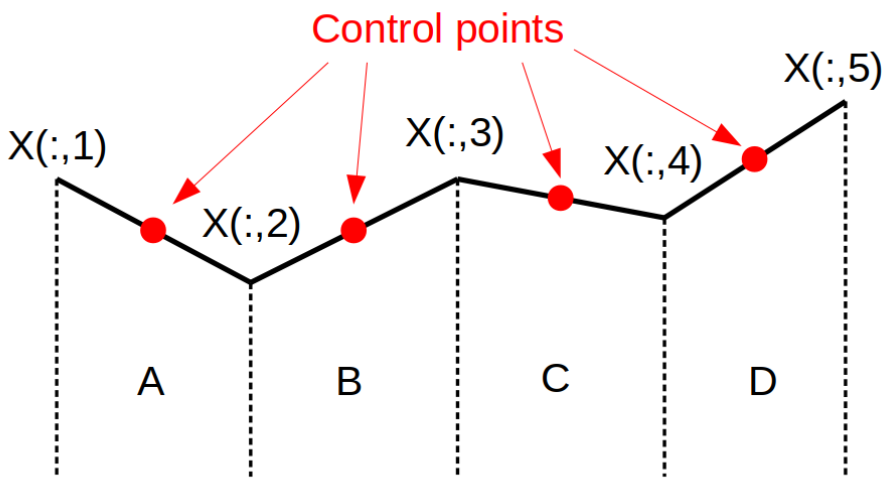

Figure 1: Control points of the horseshoe vortices.

As an example, the control point of the vortex B would be:

$$\vec{X}_B = \frac{\vec{X}_2 + \vec{X}_3}{2} \tag{3}$$

For n_v vortices, there are n_v corresponding control points. Therefore, we will have a set of n_v matrices of size 3 by n_v representing all combinations of vortices and control points. We can gather all these matrices and represent them as a tridimensional matrix **AIC** of size 3 by n_v by n_v as follows:

$$\mathbf{AIC} = \begin{bmatrix} \vec{\gamma}_{1,1} & \vec{\gamma}_{2,1} & \cdots & \vec{\gamma}_{n_v,1} \\ \vec{\gamma}_{1,2} & \vec{\gamma}_{2,2} & \cdots & \vec{\gamma}_{n_v,2} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{\gamma}_{1,n_v} & \vec{\gamma}_{2,n_v} & \cdots & \vec{\gamma}_{n_v,n_v} \end{bmatrix} \tag{4}$$

where $\vec{\gamma}_{i,j}$ is the aerodynamic influence vector induced by vortex i at the control point of vortex j.

The **AIC** matrix has three dimensions, as stated previously. The first dimension represents the component (x, y, or z) of the aerodynamic influence vector ($\vec{\gamma}$), the second dimension represents the index of the inducing vortex, and the third dimension represents the index of the induced control point. For example, `AIC(2,1,5)` should be the y component of the aerodynamic influence vector induced by the first horseshoe vortex over the control point of the fifth horseshoe vortex. Thus, `AIC(:,i,j)` should be equal to $\vec{\gamma}_{i,j}$.

3.1 Computing the AIC matrix

We will add a new subroutine to the *llt_module.F90* file to compute the matrix **A**. This subroutine should be named `get_AIC_matrix` and use the following sintax:

```
subroutine get_AIC_matrix(n_vort, X, Uinf, AIC)

! This subroutine computes the Aerodynamic Influence Coefficients (AIC)
! matrix that gives the adimensionalized induced velocities in
! all bound segments in X, that is: Vind = AIC*Gamma.
! Vind(1:3,ii) = AIC(1:3,:,ii)*Gamma(:)
!
! INPUTS
!
! n_vort: integer -> Number of horseshoe vortices.
! X: real(3,n_vort+1) -> Coordinates of the kinks of the horseshoe vortices.
! Uinf: real(3) -> Unitary vector along the free-stream direction.
!
! OUTPUTS
!
! AIC: real(3,n_vort,n_vort) -> Aerodynamic Influence matrix.
```

Here is a pseudo-code to guide you in the implementation:

1. for `ii` varying from 1 to n_v:
   (a) Extract from the matrix `X` the coordinates `X1ii` and `X2ii` of the kinks of the `ii`-th vortex.
   (b) Compute the position of the control point `Xc` of this vortex by taking the average between `X1ii` and `X2ii`.
   (c) for `jj` varying from 1 to n_v:
      i. Extract from the matrix `X` the coordinates `X1jj` and `X2jj` of the kinks of the `jj`-th vortex.
      ii. Compute the aerodynamic influence vector `gamma_ind` between vortices `ii` and `jj` using the `induced_vel` function written in the previous homework assignment.
      iii. Store the value of `gamma_ind` at `AIC[:,jj,ii]`.

test you implementation for the following case:

- $\vec{u}_\infty = [1, 0, 0]$ and $\mathbf{X} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 2.0 & 3.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$

The non-zero values of the `AIC` matrix for this case are:

$$\texttt{AIC(3,:,:)} = \begin{bmatrix} -0.31830988618379069 & 0.10610329539459691 & 2.1220659078919374E-002 \\ 0.10610329539459691 & -0.31830988618379069 & 0.10610329539459691 \\ 2.1220659078919374E-002 & 0.10610329539459691 & -0.31830988618379069 \end{bmatrix} \quad (5)$$

Give the aerodynamic influence matrices **AIC** for the following case:

- $\vec{u}_\infty = [1, 0, 0]$ and $\mathbf{X} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 2.0 & 3.0 \\ 0.0 & 0.1 & 0.2 & 0.3 \end{bmatrix}$

Print `AIC(1,:,:)`, then `AIC(2,:,:)`, and then `AIC(3,:,:)`.

3.2 Testing the LLT method functions

Take your versions of the `induced_vel` and `get_AIC_matrix` subroutines and add them to the `llt_module_incomplete.f90` file given in the homework folder. Then rename this file as `llt_module.f90`. You may have to include you dot product, cross product, and norm subroutines into the module as well.

This file has all necessary subroutines to compute the residual and functions of interest of the lifting-line theory. These tasks are performed by the `get_residuals` and the `get_functions` subroutines, which are already implemented.

Check if the implementation is correct by running the `get_residuals` for the following case:

- $\mathbf{X} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 2.0 & 3.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$
- `Gama = [2.0 2.0 2.0]`
- `alpha0 = [0.0 0.0 0.0]`
- `chords = [0.3 0.3 0.3]`
- `cl0 = [0.0 0.0 0.0]`
- `cla = [6.283 6.283 6.283]`
- `Vinf = [1.0 0.0 0.0]`
- `rho = 1.0`

You should get: `res = [6.4261187504344397, 5.0646641230067555, 6.4261187504344388]`
Now execute the `get_functions` subroutine for the same set of inputs. You should get:

- `Sref = 0.89999999999999991`
- `CL = 13.333333333333334`
- `CD = 4.3384458561346291`
- `L = 6.0000000000000000`
- `D = 1.9523006352605827`

Rerun the two subroutines for this new set of inputs:

- $\mathbf{X} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 2.0 & 3.0 \\ 0.2 & 0.0 & 0.0 & 0.2 \end{bmatrix}$
- `Gama = [1.0 2.0 1.0]`
- `alpha0 = [0.0 0.1 0.0]`
- `chords = [0.3 0.6 0.3]`
- `cl0 = [0.0 0.0 0.0]`
- `cla = [6.283 6.283 6.283]`
- `Vinf = [1.0 0.0 0.0]`
- `rho = 1.0`

Provide the values you obtained for: `res`, `Sref`, `CL`, `CD`, `L`, and `D`

3.3 Differentiating the code

We need to differentiate the `get_residuals` and the `get_functions` subroutines to apply the adjoint method. The problem is that Tapenade's web interface accepts only a single subroutine as starting point for the differentiation process. Therefore, we need to create a *dummy* subroutine named `tapenade_main` that calls the two subroutines we want to differentiate. Create a `tapenade_main` subroutine at the end of the `llt_module.f90` file with the following syntax:

```
subroutine tapenade_main(n_vort, X, Gama, alpha0, chords, cl0, cla, Vinf, rho, &
                         res, Sref, CL, CD, L, D)

! Dummy subroutine to allow differentiation of get_residuals and
! get_functions in a single Tapenade call.
!
! INPUTS
!
! n_vort: integer -> Number of horseshoe vortices.
! X: real(3,n_vort+1) -> Coordinates of the kinks of the horseshoe vortices.
! Gama: real(n_vort) -> Circulation intensity of each horseshoe vortex.
! alpha0: real(n_vort) -> Array of local incidence angles.
! chords: real(n_vort) -> Array of local chords.
! cl0: real(n_vort) -> cl0 of each 2D section
! cla: real(n_vort) -> lift curve slope of each 2D section [1/rad]
! Vinf: real(3) -> Free-stream velocity vector
! rho: real -> Air density
!
! OUTPUTS
!
! res: real(n_vort) -> Residuals of the LLT method
! Sref: real -> Area of all panels
! CL: real -> Lift coefficient (adimensionalized by Sref)
! CD: real -> Drag coefficient (adimensionalized by Sref)
! L: real -> Lift force
! D: real -> Drag force
```

Once the subroutine is implemented, use the `send_to_tapenade.py` script to generate differentiated versions of the `llt_module.f90` file in forward and reverse modes. The differentiation should be done with respect to the following variables:

- Input variables: `X`, `Gama`, `alpha0`, `chords`.
- Output variables: `res`, `Sref`, `CL`, `CD`, `L`, `D`.

3.4 Finite difference test

Let's use the finite difference method to compute the directional derivative of the residuals `res` with respect to the inputs variables.

In our case, since we have 4 input variables (`X`, `Gama`, `alpha0`, and `chords`), we need to define 4 perturbation directions (`Xd`, `Gamad`, `alpha0d`, and `chordsd`) to compute a direction derivative. For this exercise, use the **first set of inputs** from Sec. 3.2 and the following perturbation directions:

- $\mathbf{Xd} = \begin{bmatrix} -0.1 & 0.5 & 0.2 & 0.0 \\ 1.0 & -0.7 & 0.1 & -0.9 \\ 0.3 & 0.0 & -0.6 & 0.2 \end{bmatrix}$
- `Gamad = [-1.0 0.5 0.3]`
- `alpha0d = [0.0 0.1 -0.1]`
- `chordsd = [1.0 -0.2 0.7]`

You should perturb each input variable in a small amount along these direction, compute the residuals at this new location, and use finite differences to estimate the directional derivative:

$$\texttt{res} = \texttt{get\_residuals(X, Gama, alpha0, chords)} \tag{6}$$

$$\texttt{res1} = \texttt{get\_residuals(X + h*Xd, Gama + h*Gamad, alpha0 + h*alpha0d, chords+h*chordsd)} \tag{7}$$

$$\text{directional deriv. of } \texttt{res} = \frac{1}{h}\left[\texttt{res1} - \texttt{res}\right] \tag{8}$$

NOTE: the directional derivative of `res` should have the same number of elements as the variable `res`.

We can also use the forward AD version of the `get_residuals` subroutine to get the directional derivative by using the perturbation directions as derivative seeds:

$$\begin{aligned}\texttt{res}, \text{ directional deriv. of } \texttt{res} \quad = \quad & \texttt{get\_residuals\_d(X, Xd, Gama, Gamad,} \\ & \texttt{alpha0, alpha0d, chords, chordsd)}\end{aligned} \tag{9}$$

Test different values of $h = 10^n$ and find the step size magnitude (n) that gives the smallest difference between the directional derivatives estimated with these two methods. You do not need to run an optimization to find the best h, just give the order of magnitude. Use the norm of the difference between arrays as metric:

$$\texttt{difference} = |\text{FD directional deriv.} - \text{AD directional deriv.}| \tag{10}$$

Give the values of the directional derivative computed by the AD code.

Repeat the same process to verify the directional derivatives of the `get_functions` subroutine. Use the same set of perturbation directions and step size h from the previous test. Compute the difference between directional derivatives estimated with the finite difference method and the forward AD code. Also give the directional derivatives of: `Sref`, `CL`, `CD`, `L`, and `D`.

3.5 Dot product test

Now we will use the dot product test to verify the reverse AD code.

Run the forward AD version of the `get_residuals` subroutine with this set of arbitrary derivative seeds:

- $\mathbf{Xd} = \begin{bmatrix} -0.1 & 0.5 & 0.2 & 0.0 \\ 1.0 & -0.7 & 0.1 & -0.9 \\ 0.3 & 0.0 & -0.6 & 0.2 \end{bmatrix}$
- `Gamad = [-1.0 0.5 0.3]`
- `alpha0d = [0.0 0.1 -0.1]`
- `chordsd = [1.0 -0.2 0.7]`

Give the derivative seeds of the outputs: `resd`.

Next, execute the reverse AD version of `get_residuals` subroutine with this set of arbitrary derivative seeds:

- `resb = [0.1 -0.2 0.3]`

Give the derivative seeds of the inputs: `Xb`, `Gamab`, `alpha0b`, `chordsb`.

Now do the dot product test by adding the contributions of the input variables and by subtracting the contributions of the output variables:

$$\begin{aligned}\texttt{dotprod} \quad = \quad & \texttt{sum(Xd*Xb) + sum(Gamad*Gamab) + sum(alpha0d*alpha0b) + sum(chordsd*chordsb)} \\ & \texttt{- sum(resd*resb)}\end{aligned} \tag{11}$$

Write the value you obtained for `dotprod`. The final value of the dot product should be zero (to machine precision).

Now we will repeat the same process for the `get_functions` subroutine. Use the same input derivative seeds and give the corresponding derivative seeds you got for `Srefd`, `CLd`, `CDd`, `Ld`, and `Dd`.

Then execute the reverse AD version of `get_functions` subroutine with this set of arbitrary derivative seeds:

- `Srefb = 0.6`
- `CLb = -0.2`
- `CDb = 0.1`

- `Lb = -0.3`
- `Db = 0.7`

Give the derivative seeds of the inputs: `Xb`, `Gamab`, `alpha0b`, `chordsb`.

Now do the dot product test by adding the contributions of the input variables and by subtracting the contributions of the output variables:

```
dotprod  =  sum(Xd*Xb) + sum(Gamad*Gamab) + sum(alpha0d*alpha0b) + sum(chordsd*chordsb)
            - Srefd*Srefb - CLd*CLb - CDd*CDb - Ld*Lb - Dd*Db
```
(12)

Write the value you obtained for `dotprod`. The final value of the dot product should be zero (to machine precision).

3.6 Troubleshooting

If the dot product is not zero, it may due to a couple of things:

- If you are using the `-fdefault-real-8` flag when compiling Fortran code, you shoud also set the `real4toreal8` flag to `True` in the `send_to_tapenade.py` script.
- The reverse AD code modifies the content of the reverse AD seeds of the output variables. Therefore, make a copy of these derivative seeds before calling the reverse AD code, so that you still have the original seeds for the dot product test.
- Remember to set the reverse AD seeds of the input variables to `0.0` before calling the reverse AD code, since the code just accumulate values to these seeds. Previous values stored in the memory will interfere with your results if you do not initialize the seeds.
- Sometimes the differentiation does not work when you use the same variable twice as arguments of a function. For instance, assume that you have a subroutine `dot(a,b,adotb)` that computes the dot product of the vector `a` and `b`. You can use this same subroutine to compute the square of the norm of vector `a` by calling: `dot(a,a,norma2)`. The reverse differentiation may not work well because Tapenade may not recognize that the two arguments of `dot` are in fact the same variable. To get around this case, it is better to define a new dedicated subroutine `norm(a,norma2)` to compute the norm of a vector.