ROBUST WAKE ROLLUP MODELLING USING DVES

by

Tejas Janardhan
Bachelor of Engineering, German University of Technology (2017)

A report

presented to Ryerson University

in partial fulfillment of the requirements

for the degree of

Master of Engineering

in the program of Aerospace Engineering

Toronto, Ontario, Canada, 2019

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A PROJECT

I hereby declare that I am the sole author of this project. This is a true copy of the project, including any required final revisions.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my project may be made electronically available to the public.

ROBUST WAKE ROLLUP MODELLING USING DVEs

Tejas Janardhan

Master of Engineering, Aerospace Engineering, Ryerson University, Toronto (2019)

ABSTRACT

This project report gives details on a modification of VAPTOR, a program that can predict the aerodynamic performance of aircrafts using a potential flow method with a relaxed wake model. In VAPTOR the wake is modelled using distributed vorticity elements (DVEs). DVEs can induce velocities at certain points used to relax the wake. A DVE has inbuilt singularity protections i.e. prevents the calculated velocity to approach infinity, but when two adjacent DVEs have a very low relative angle, these protections lead to an error in the calculation of the velocity at its shared midpoint during the relaxation process. In most cases these errors are negligible until a rotor is analysed during hover or vortex ring state. In these special cases the wake rollup is more intense leading to relatively small angles. The subsequent errors caused by the singularity protections cannot be ignored since they cause the solutions to be erratic and not smooth. It also causes the wake DVEs to deform disproportionally which is a visual indication of the errors.

The modification uses a method that involves splitting the DVE in order to eliminate the errors when calculating the velocity at the junction of two adjacent DVEs. The splitting is temporary and only applied during the calculation of the velocity at the junction. The algorithm for the splitting of the DVE and its implementation into MATLAB is provided in this report. The implementation is tested by ensuring that all conditions are kept the same except when splitting is enabled or disabled. A number of test runs were conducted, and an index called the Smoothness Index was created in order to quantify the improvements of the DVE splitting method. The results shown are promising as the solution with splitting enabled is twice as smooth as when the splitting is disabled. There is also a noticeable improvement during visual comparison of the wake diagrams when splitting is enabled and disabled. The results combined with the fact that the extra computation required to execute the DVE splitting method is negligible, the author recommends it be enabled in all cases. Having said that, the end user has full control whether he or she would like to use it or not. They can also change the parameters of splitting to suit their needs.

ACKNOWLEDGEMENTS

I am grateful to my supervisor, Dr. Bramesfeld for his guidance and support in this project. Through his detailed feedback and supervision, I can honestly say I have greatly improved various engineering and research-oriented skills. I would like to thank Devin Barcelos, a PhD student at RAALF, for introducing me and also giving me valuable insight into the inner workings of VAPTOR. I would also like to thank all RAALF members who maintain the lab server as that enabled me to complete my project and the members who gave me feedback on my presentation. And finally, I must thank my family without whose support none of this would have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

DVE	Distributed Vorticity Elements
Γ	Circulation
γ	Vorticity
$\overrightarrow{lv}$	Lateral vector
iDVE	Intermediate Distributed Vorticity Element
PF	Projection Factor
MF	Max Span Factor
θ	Relative angle between two adjacent DVEs
Λ_{LE}	Leading Edge angle
Cp	Coefficient of power
Ct	Coefficient of thrust
ΔCp	Change in coefficient of power
ΔCt	Change in coefficient of thrust
S_{Index}	Smoothness Index
Δn_{smp}	Number of shared midpoints created per timestep
Δn_{wdve}	Number of wake DVEs created per timestep
Δn_{mp}	Number of midpoints created per timestep
n_{step}	Number of timesteps

CHAPTER 1: INTRODUCTION

1.1 VAPTOR OVERVIEW

VAPTOR is a potential flow method with relaxed wake model that uses Distributed Vorticity Elements (DVEs), written in MATLAB. The use of DVEs has fewer issues with singularities in the flow field than conventional vortex-lattice or panel methods [1]. VAPTOR is used to predict the aerodynamic forces acting on lifting surfaces, such as wings and rotors.

1.2 DVE OVERVIEW

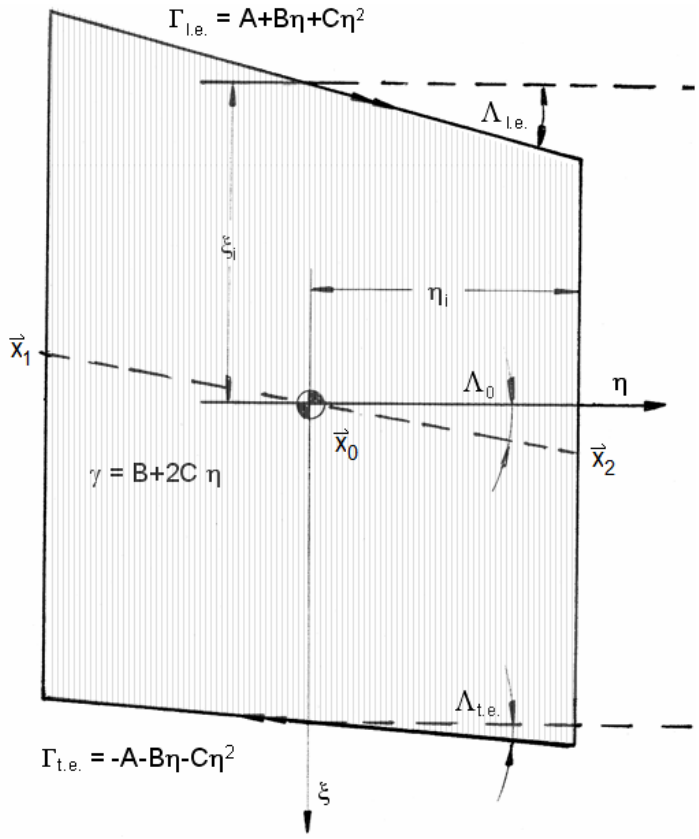

Figure 1-1 A distributed vorticity element. [2]

A distributed vorticity element (DVE) as shown in Fig. 1-1 is a trapezoidal element that is made of two opposite-strength coplanar vortex filaments and a vortex sheet that connects with the two filaments and is aligned with the ξ-axis.

A DVE's geometric characteristics include its chord length, span length i.e. distance between the two parallel side edges and finally the angle between the two vortex filaments make with the η axis.

The DVE filaments have quadratic spanwise circulation distributions of equal magnitude and opposite orientation as well as a vortex sheet with a linear vorticity distribution shown:

$$\Gamma_{LE} = A + B\eta + C\eta^2 \tag{1-1}$$

$$\Gamma_{TE} = -\Gamma_{LE} = -A - B\eta - C\eta^2 \tag{1-2}$$

$$\gamma_{LE} = \frac{d\Gamma_{LE}}{d\eta} = B + 2C\eta \tag{1-3}$$

The coefficients A, B and C are calculated based on three main conditions. Firstly, velocity (of fluid/air) must be tangent at a collocation point of a DVE i.e. the normal velocity at the point must be zero. Secondly, the circulation and vorticity must be continuous with neighbouring DVEs. Thirdly, the circulation at the tips of a wing/rotor must be zero since a pressure differential cannot be maintained at the tip.

1.2.1 Side edge singularities.

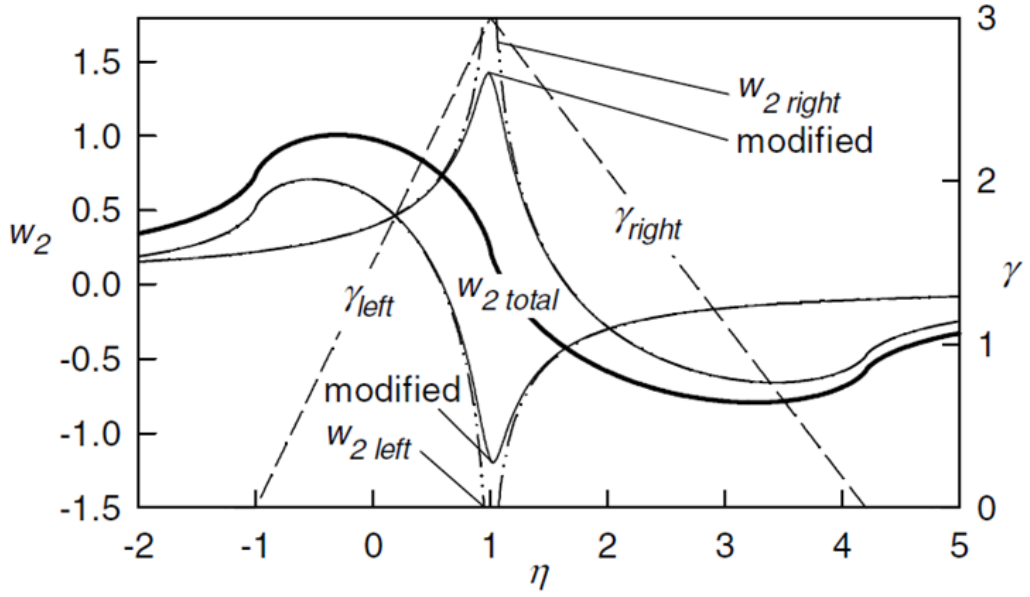

Figure 1-2 A spanwise distribution of the normal velocity that is induced in the plane of two semi-infinite vortex sheets. The dashed line denotes the spanwise vorticity distributions and the solid line denotes the total induced velocity. [1]

An important aspect of DVEs is how it deals with singularities at the side edges of the sheet. Singularity refers to how the induced velocity of a DVE becomes infinity as the side edge is approached [2].

In order to deal with the singularities when using DVEs in a numerical scheme, an additional singularity is added at the edge of the sheet to modify the original singularity so that an element's self-induced side edge velocity remains finite. Since the induced velocity remains finite, the velocities of the neighbouring element which share an edge have their velocities cancel out, thus removing the influence of the additional singularity as seen in Fig. 1-2 [2].

In VAPTOR [3] there are two main types of DVEs used in the code. Surface DVEs are used to model the surfaces of any object/vehicle. The subsequently used DVEs have two vortex filaments with a vortex sheet in between.

Wake DVES are only used to model the wake created by a lifting surface. The wake elements consist only of the finite vortex sheet without the vortex filaments at the leading and trailing edge of the element. The removal of the filaments is justified in a steady wake, where the vortex filaments of two DVEs that are next to each other in the streamwise direction, cancel each other. Wake DVEs are attached spanwise at their midchord and streamwise at their midspan.

1.3 Relaxation process Overview

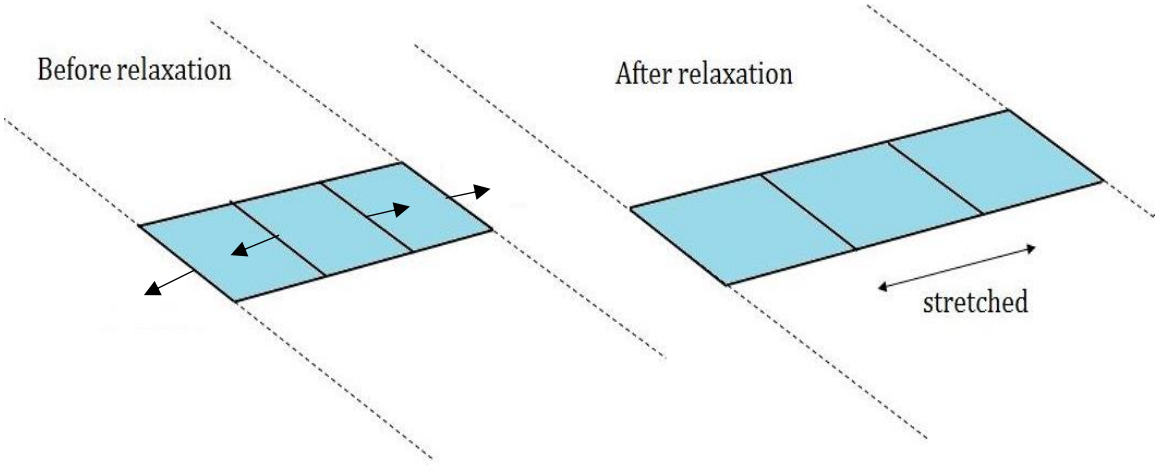

Figure 1-3 Relaxation process. Arrows show the direction of the induced velocity calculated at each midpoint. [4]

The relaxation process during each time step starts off with computing the induced velocities at the side edge midpoints of each wake element. These velocities are then used to calculate the displacement for each of the wake points based on the size of the time step. The wake elements are either compressed or stretched in the spanwise or streamwise direction as shown in Fig. 1-3. Each element must remain planar, and its side edges must remain parallel after the relaxation process.

The re-configuration of the elements requires the recalculation of the coefficients A, B and C. The recalculation is done based on two main conditions. The effective circulation must remain constant for a DVE before and after the relaxation process. Furthermore, the spanwise vorticity and circulation distributions of the wake elements have to remain continuous [4].

1.4 Problem Definition

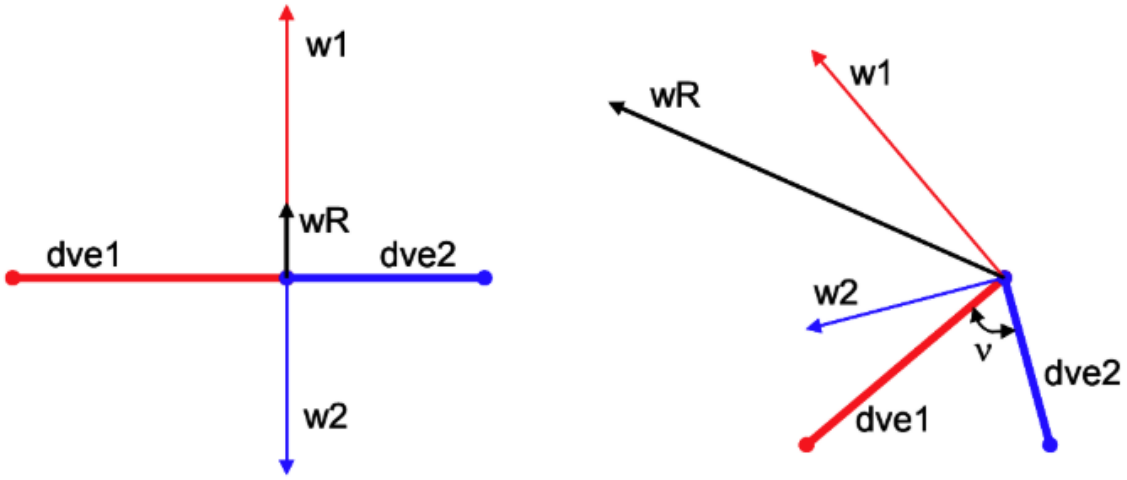

Figure 1-4. False Velocity Spike when DVE's are not coplanar [5]

The side edge singularity correction method explained in section 1.3 only works if the two adjacent DVEs are coplanar. Fig. 1-4 shows what happens when two adjacent DVEs are non-coplanar. Instead of the two velocities cancelling out (shown on the left) the velocities can add up causing a false velocity spike (shown on the right). The effect is inversely dependent on the relative angle between the two DVEs i.e. the smaller the relative angle the larger the velocity spike.

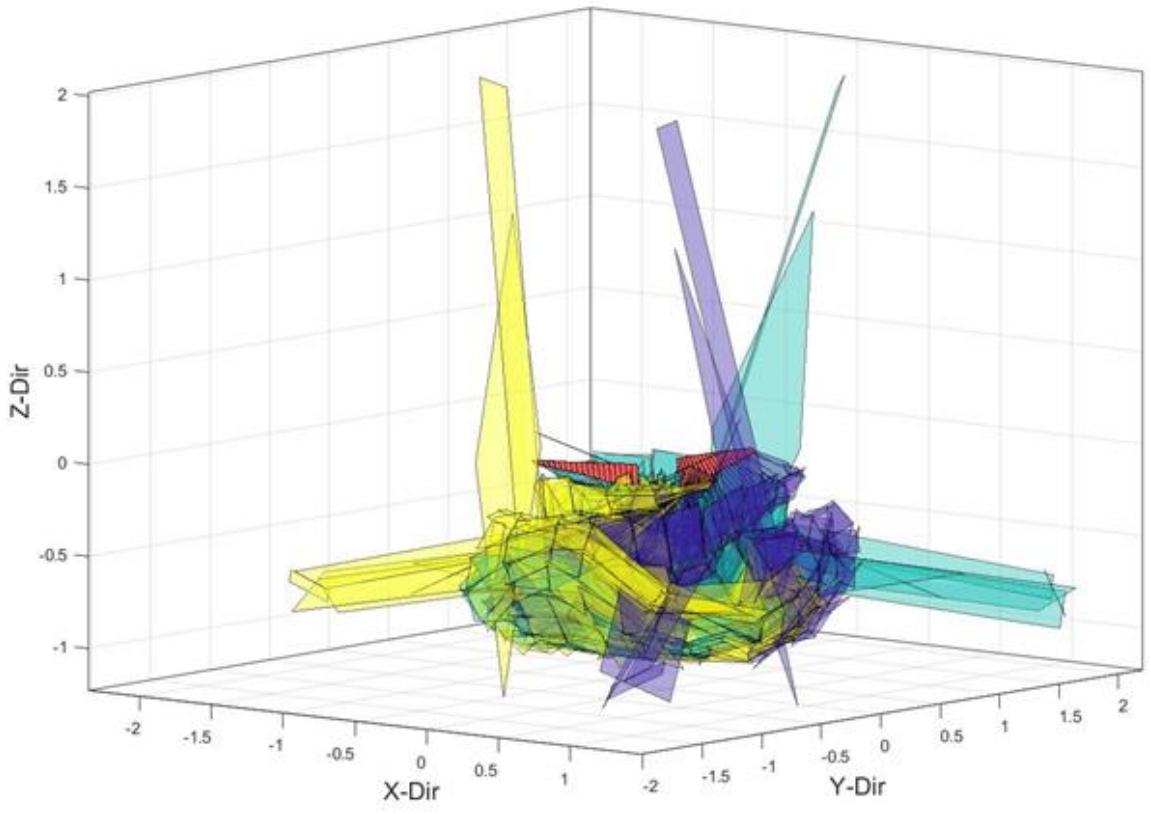

Figure 1-5 Rotor in hover.

A rotor simulated in hover, for example shown in Fig. 1-5, leads to a dense wake, which means an increase in the number of DVEs with small relative angles to adjacent DVEs. This leads to an increase in the magnitude of the false velocity spikes, thus introducing an error in the final force calculation that is dependent on the induced velocity. It also leads to the elongation of a few DVEs due to the disproportionally large velocities calculated during the wake relaxation process.

The increase in the number of DVEs with small relative angles to adjacent DVEs is also observed when a rotor is simulated in vortex ring state. Vortex ring state is a rotor flight condition where the relative velocity between the rotor and the wake is zero i.e. the rotor descends at the same rate as the wake. This will lead to a dense wake and thus forces errors in the final force calculation. Since it is convenient to use points along the side edges of elements in the wake relaxation procedure, it is important to address this issue especially when a rotor is in hover or vortex ring state. The objective of this project is to try and address this issue by applying the DVE splitting method in VAPTOR. [5]

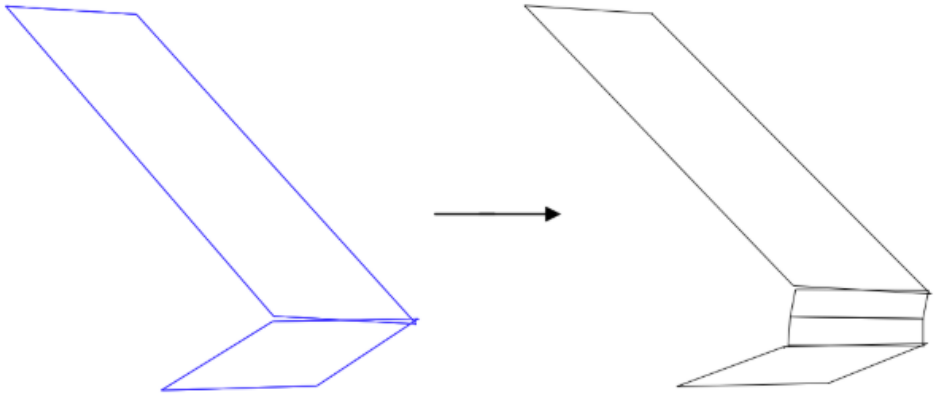

Figure 1-6 Splitting DVE's [5]

By splitting the two non-coplanar DVE's, shown on the left in Fig. 1-6 and forcing the two-inner split DVE's to be co-planar, shown on the right, it removes the false velocity spike. It is important to note that the splitting is temporary and its only done to calculate the velocity at the junction of every DVE pair during the wake-relaxation process. This paper discusses the implementation of the splitting in VAPTOR while ensuring it's not computationally intensive.

CHAPTER 2: METHODOLOGY

2.1 ALGORITHM USED

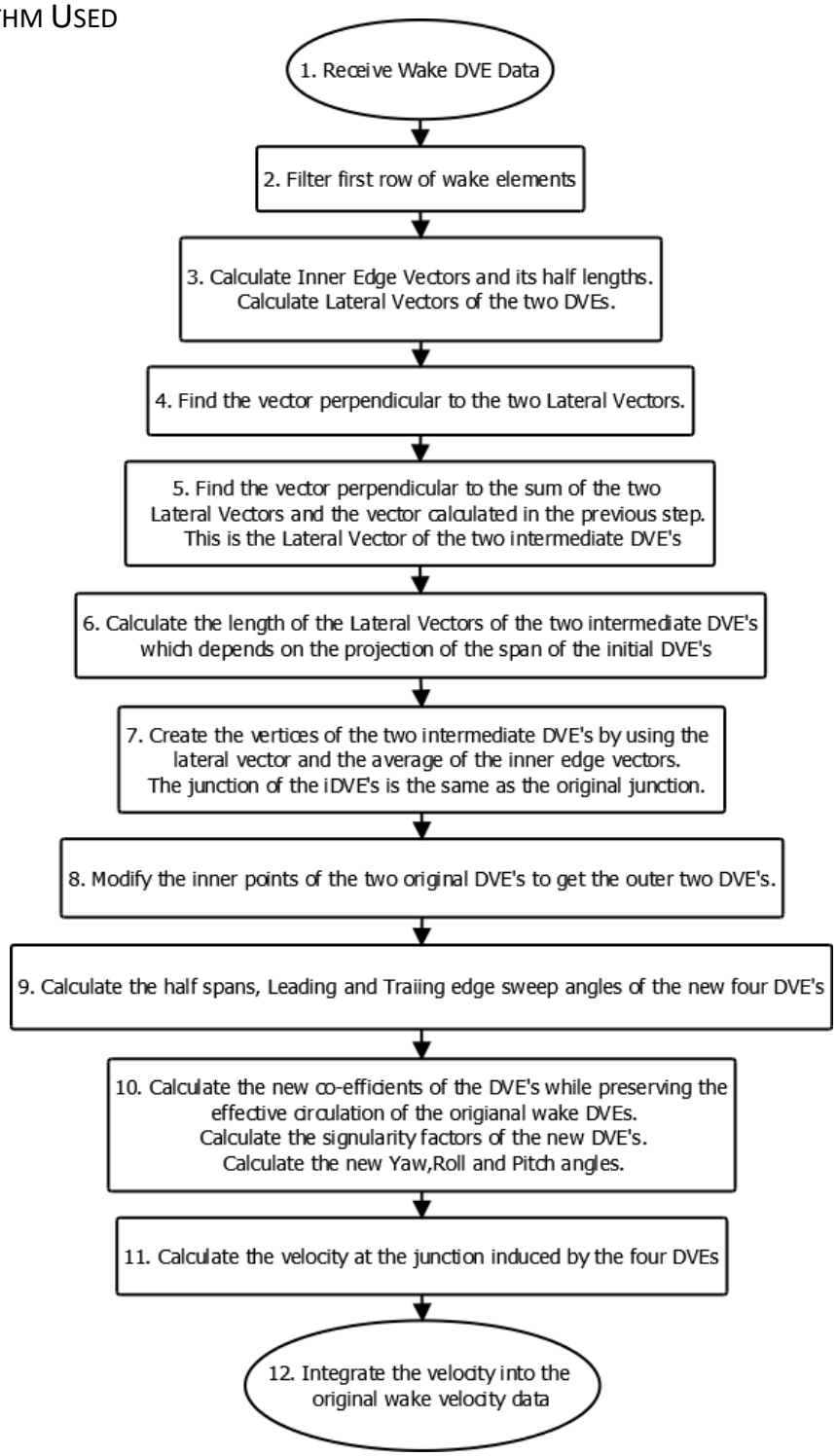

Figure 2-1. Flowchart

Fig. 2-1 shows the flowchart of the algorithm that was developed in order to implement the DVE splitting method. In Step 1, Wake Data refers to all the data that is related to the wake DVEs, for example as shown in Fig. 2-2 with the blue and yellow elements. In Step 2, the first-row wake elements, labelled in Fig. 2-2 must be filtered before all the pairs undergo the splitting process, this is because the pairs of adjacent DVE's do not connect at the midpoint. The method only works on DVE pairs connected at the midpoint.

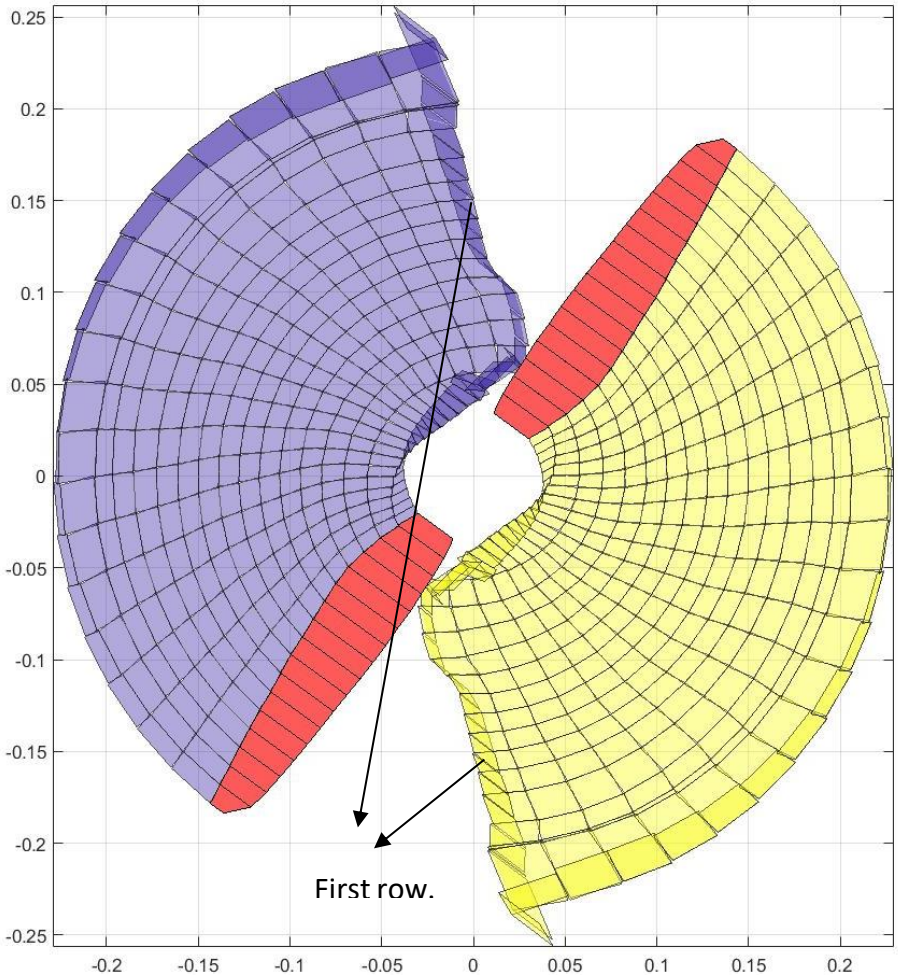

Figure 2-2 Wake DVEs created by a rotor during each timestep. First row wake DVEs are also shown.

Lateral Vector ($\overrightarrow{lv}$) referred in Step 3 is the vector joining the mid-chord points on the side edges of a DVE. Figure 2-3 shows the vector, it is important that the lateral vector of a DVE always points away from the shared midpoint.

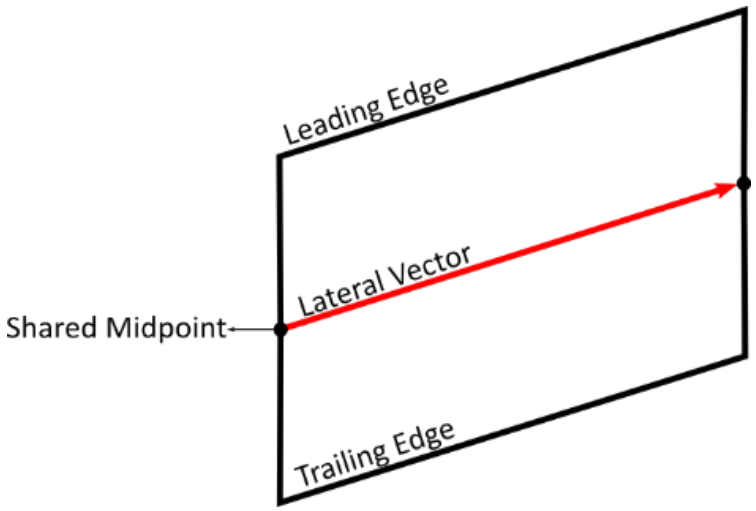

Figure 2-3 Lateral Vector. Side edges refers to the unlabeled edges.

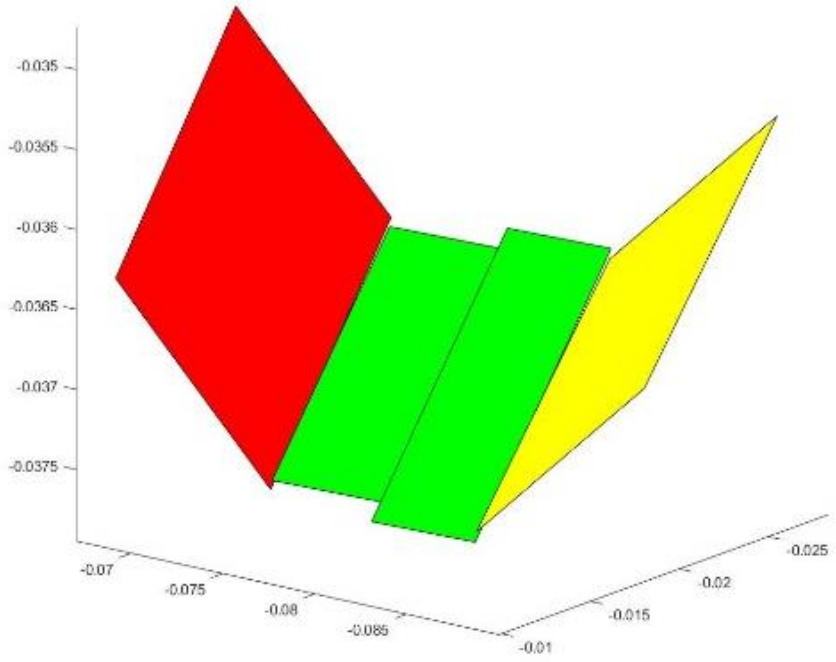

Figure 2-4 intermediate DVE's (iDVEs). The red and yellow DVEs are the outer iDVEs and the green are the inner iDVEs.

An important point to note is the lengths of the lateral vectors of the two-inner intermediate DVE's (iDVEs Fig. 2-4) are equal to a fraction of the orthogonal projection of the span of the original DVE's, called the Projection Factor (PF):

$$\left|\overrightarrow{lv_{iDVE}}\right| = PF(span_{DVE} \sin\theta) \qquad (2\text{-}1)$$

The upper limit on the value of the length of a lateral vector is equal to the fraction of the span of the original DVEs. This upper limit is called the Max Span Factor (MF):

$$|\overrightarrow{lv_{iDVE}}| = \begin{cases} PF(span_{DVE} \sin\theta), & if\ |\overrightarrow{lv_{iDVE}}| < MF(span_{DVE}) \\ MF(span_{DVE}), & if\ |\overrightarrow{lv_{iDVE}}| > MF(span_{DVE}) \end{cases} \quad (2\text{-}2)$$

θ in equations 2-1 and 2-2 is the relative angle between two adjacent DVES. The lengths of the side edges of the inner iDVEs are the same as the lengths of the inner side edges of the original DVEs, where 'inner' refers to the side closest to the shared midpoint i.e. the chord lengths of the both the inner and outer iDVEs are the same as its original DVE. The span of the inner iDVEs are calculated based on the leading-edge angles of the inner iDVEs:

$$span_{iDVE} = |\overrightarrow{lv_{iDVE}}| \cos\Lambda_{LE} \quad (2\text{-}3)$$

This means the factors have a direct effect on the spans of the inner iDVEs. The midpoint locations have to be preserved when creating the inner iDVEs.

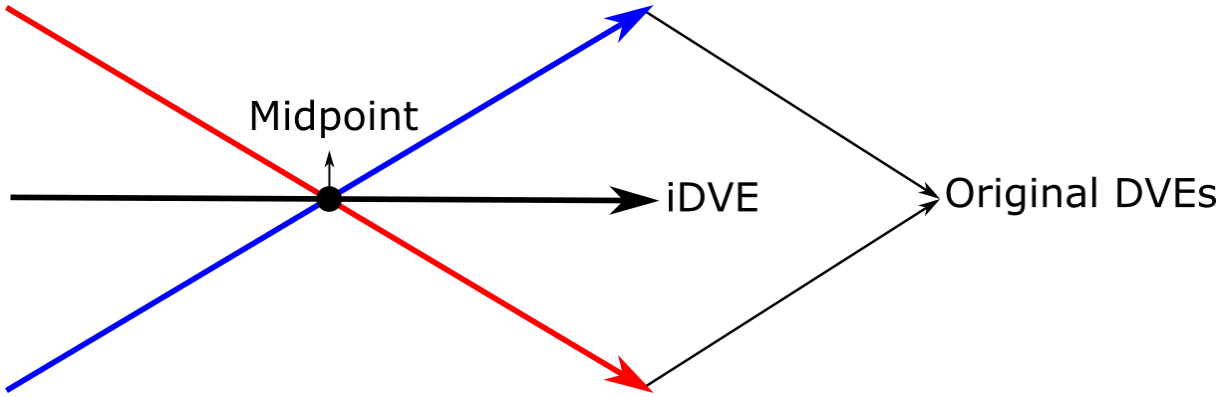

Figure 2-5 Side view of a pair of DVEs. Also shows the side edge vectors of the three DVEs

In Step 7, the inner iDVEs are created using the average of the two original DVE side edge vectors shown in Fig. 2-5. The iDVE mentioned in Fig. 2-5 are the inner iDVEs. Both inner iDVEs in a split DVE pair share the same side edge vector while the outer iDVEs have the same side edge vector as its original DVE.

2.1.1 Coefficients.

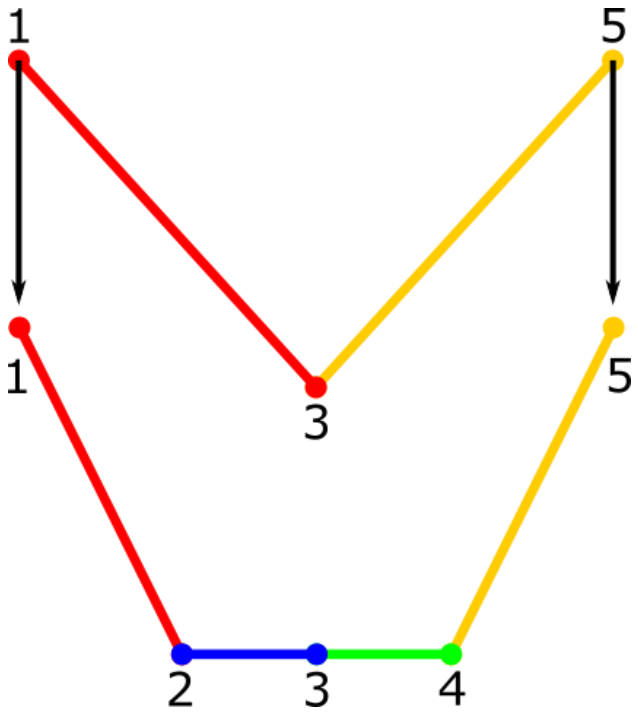

Figure 2-6 Shows how the old and new DVEs are related, this relation is used to calculate the new coefficients.

Just like the relaxation process, explained in section 1.3, the coefficients A, B and C must be calculated for the four iDVEs since, not only are the iDVEs different in their geometry but there are two new DVEs created which renders the coefficients of the two original DVEs unusable to calculate the induced velocities.

In Step 10 in the algorithm shown in Fig. 2-1, the coefficients of the four iDVEs are calculated by using 12 equations to solve 12 unknows, three coefficients for each of the four iDVEs. 10 independent equations are provided by continuity equations, Fig. 2-6 shows the points 1 and 5 which maintain continuity to the other DVEs. Points 2,3 and 4 are used to maintain continuity between all the four iDVEs. The continuity equations refer to ensuring the circulation and vorticity distributions are continuous along all four iDVEs. The remaining two equations preserve the effective circulation of the two original DVEs. This is done by maintaining the effective circulation of the each DVE with its corresponding pair of inner and outer iDVEs. A complete set of the equations are included in Appendix II.

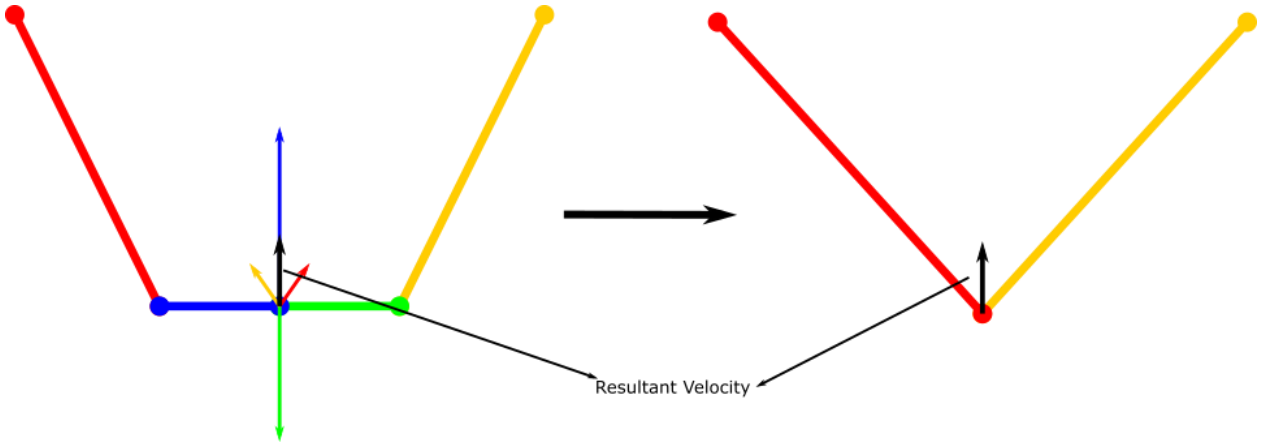

Figure 2-7 Calculation of Resultant Velocity from the 4 iDVEs

And finally, in Steps 11 and 12, the velocity is calculated at the junction of the two inner iDVEs. Once this calculation is completed the four iDVEs are discarded and the velocity will be used to relax the original two wake DVEs. Fig. 2-7 shows how the resultant velocity is calculated. On the left it shows the velocity induced by all four iDVEs and on the right it shows the resultant velocity used to move the midpoint during the wake-relaxation process. It is important to note that this is not the only velocity component that is used in the relaxation process, but rather it will be is only part of the total velocity that is the compound of all velocities induced by the rest of the DVEs, which in turn is used to displace the midpoint.

2.2 MATLAB INTEGRATION

The implementation of the algorithm, that is outlined in Fig. 2-1, in MATLAB takes advantage of the vectorized programming capability of MATLAB. Vectorizing in MATLAB is a method of parallel processing large datasets, thus improving computational performance. It is a capability that is built into MATLAB and is highly optimized.

The method of writing parallel code is vastly different from writing a serial one. For example, when doing an operation on a wake DVE, all DVEs must be considered for the operation as it is done in parallel. It is important to note that vectorizing is only possible if the data sets have no inner dependencies.

The splitting is an optional functionality and can be turned on or off. This is done by creating a separate function called fcnINDVEL_RELAX and placing it in fcnRELAXWAKE. The _RELAX suffix is used as the splitting is only done to calculate the velocity used to relax a wake. Inside fcnINDVEL_RELAX is one of the two main functions used to implement the splitting technique which is fcnWDEVEL_RELAX used to calculate the velocities of the wake DVEs.

The velocities of the wake DVEs are first calculated like it is done normally. The subsequent velocities are sent to the function fcnWDVEJVEL_SPLIT that calculates all the induced velocities of the adjacent DVE pairs at their junctions while implementing the splitting routine and replace the older velocities that were in the previous step without splitting. All the split DVEs stay within fcnWDVEJVEL_SPLIT and are deleted once the function is no longer needed. This method of replacing velocities was done to minimize the modifications done to the existing VAPTOR functions

2.3 TESTING

2.3.1 DVE Visualization

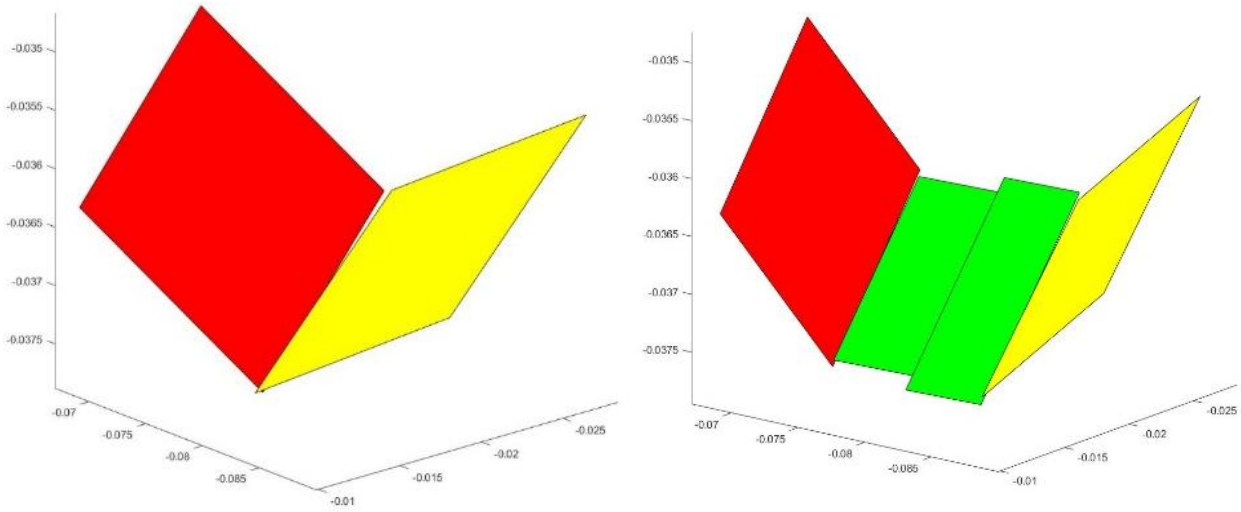

Figure 2-8. Before and After the Splitting Process.

In order to verify the code, DVE pairs were visualised in order to do a visual examination of the splitting process. *Figure 2-8* shows a pair of DVEs before and after the splitting process. This method of visualizing was used to verify the splitting geometry. It was a valuable diagnostic tool

when debugging the code. A short script written in MATLAB was used to implement the graphical output. This was especially helpful in the early stages of the implementation process as it was used as a validation check in order to ensure the code was doing what it was supposed to do.

2.3.2 iDVE Orientation Deviation

When splitting DVEs, it is ideal to ensure that the orientation of the outer two iDVEs stay as close as possible to the original DVEs. For that purpose, another test was scripted in MATLAB which saved the data of the original and intermediate DVEs at a certain time step and checked how much the outer iDVEs had deviated from the original adjacent DVE pair. This was done by taking the mean of the absolute value of the deviation in pitch, roll, yaw and leading-edge angles. The pairs with the largest deviation in any of the angles was identified and isolated, visualised and inspected in order to understand the reason for such a large deviation and debug the algorithm.

2.3.3 Test Case

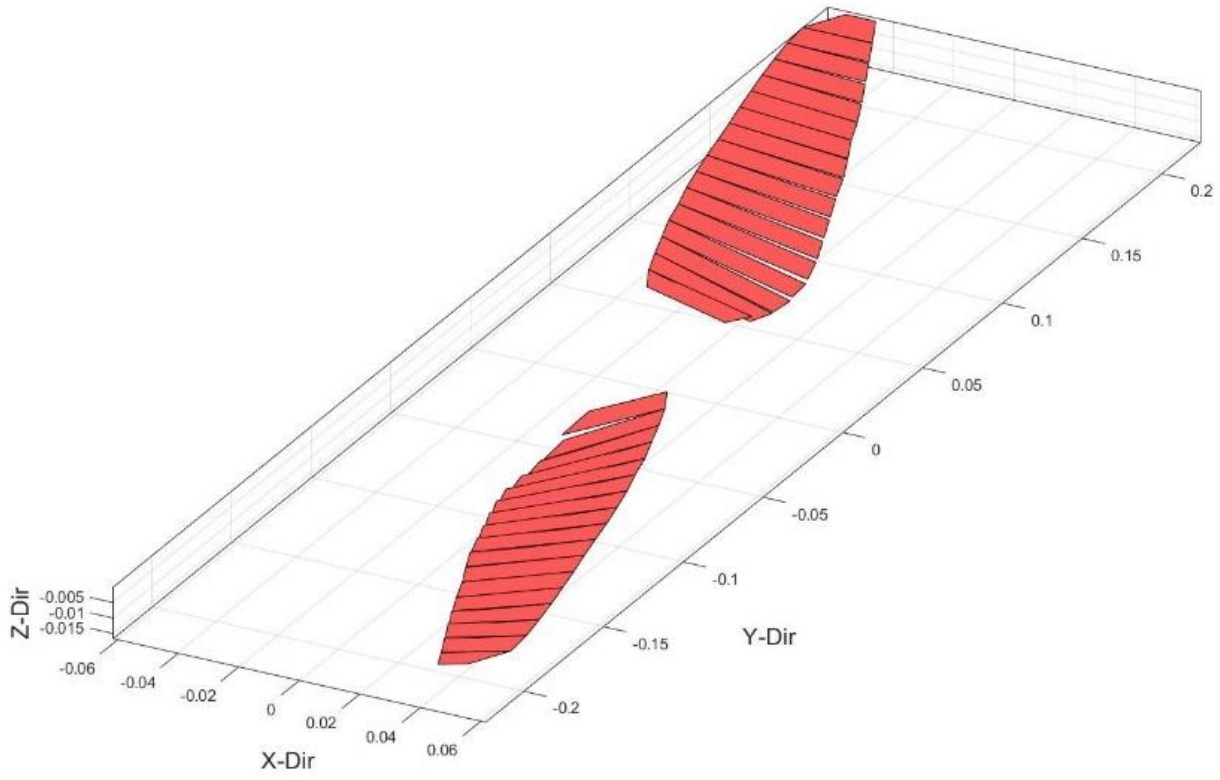

Figure 2-9. TMotor split into 34 elements [6]

A test case was used as a base reference for evaluating the effects of splitting DVEs. All parameters were kept the same as the test case except for the fact that splitting was enabled. The test case was a propeller in hover conditions.

The propeller used was the T-Motor rotor as seen in Fig. 2-9. The rotational velocity was 3000 RPM. All test runs were for runs are 200 timesteps. Each time step was 0.0004 sec long, which equates to 5 full rotations of the propeller. These parameters were chosen as a good balance between computational time and amount of data needed to properly analyse the effects. Each test run took approximately 4 hours to complete using an Intel Xeon processor, model number E5-2620 v4. Thus, 200-time steps were the upper limit of what was feasible due to the large number of runs needed. All test runs were inviscid simulations in order to limit the computational effort and since the modification does not affect the viscous portion of VAPTOR.

The final result which was used to determine the impact of the DVE splitting method on the solution will be the change in the coefficient of power (ΔCp) and change in coefficient of thrust (ΔCt).

2.3.4 Model Solution

The solution, mentioned in the previous section, refers to coefficient of power (Cp) and coefficient of thrust (Ct) calculated at each timestep. The ideal solution of Cp and Ct would be one with no abrupt changes i.e. the solution curve is smooth and with no bumps. To evaluate if a solution is ideal, we use change of Cp and Ct to check for the 'smoothness' of a solution. An ideal solution usually has the values ΔCp and ΔCt at each time step as small as possible with no disproportionally large values. In order to quantify the 'smoothness' of a solution we use the method explained in the next section.

2.3.5 Parameter Optimisation

The DVE splitting method requires the determination of the Projection Factor and Maximum Span Factor, as explained in the Section 2.1. In order to find the best values for these parameters, a Smoothness Index is devised in order to rank each test run. Smoothness Index quantifies the quality of both solutions, that is Cp and Ct. The aim is to find a run with the lowest Smoothness Index.

$$S_{Index} = \left[\frac{MeanΔCp\left(1 + \frac{StdΔCp}{MeanCp}\right)}{MeanΔCp_{NS}\left(1 + \frac{StdΔCp_{NS}}{MeanΔCp_{NS}}\right)} + \frac{MeanΔCt\left(1 + \frac{StdΔCt}{MeanCt}\right)}{MeanΔCt_{NS}\left(1 + \frac{StdΔCt_{NS}}{MeanΔCt_{NS}}\right)} \right] \Big/ 2 \quad (2\text{-}4)$$

Equation 2-4 is used to calculate smoothness index, S_{index}. The mean and standard deviation of the absolute values of ΔCp and ΔCt of a particular test run is used to calculate the smoothness index. Since the S_{Index} will be used to compare each run with reference to the no split case, the values are normalized by dividing them by the values from the test run with no splitting of DVEs, hence the suffix NS (no splitting). The average of the S_{Index} of Cp and Ct is taken to calculate the final S_{Index} of the run.

2.3.6 Computation Time testing

In order to ascertain the computational impact splitting has on VAPTOR, a simple comparison test is devised using MATLAB's built-in function profiler. MATLAB's function profiler finds the execution time of each individual function and gives a good view into the computational impact of all the functions of VAPTOR.

The test conditions are MATLAB 2017b, core i7-4720HQ processor, no other applications running at the same time and the internet is disabled. The runs are completed using the test case, one run with splitting enabled and the other run with splitting disabled.

CHAPTER 3: RESULTS

3.1 SPLITTING EFFECTS

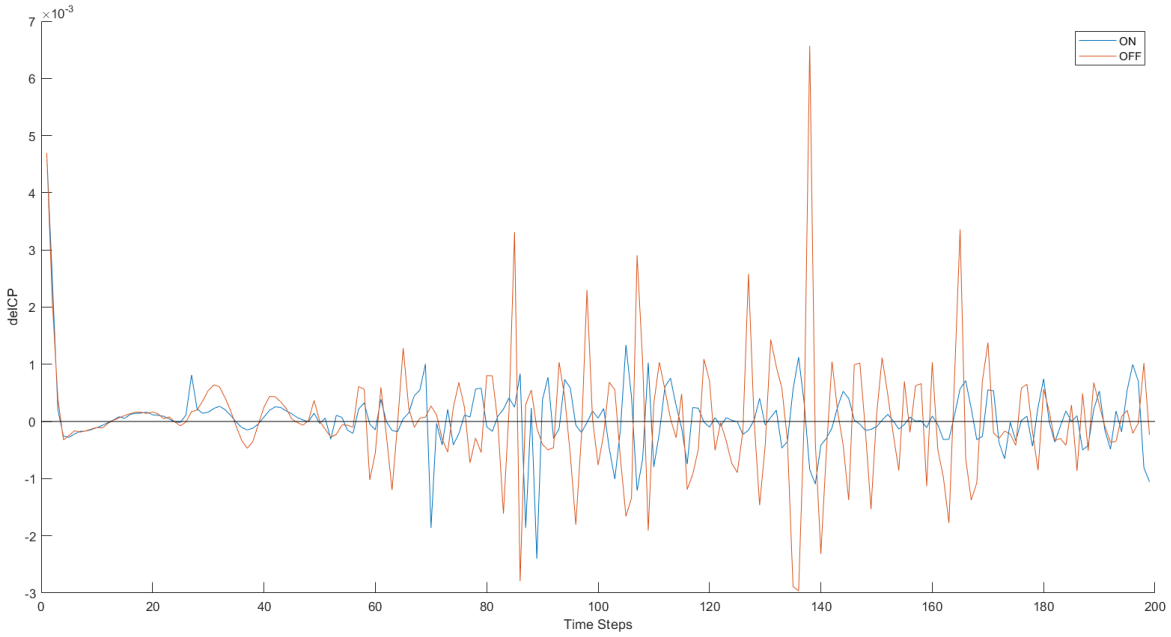

Figure 3-1 ΔCp, with splitting on and off

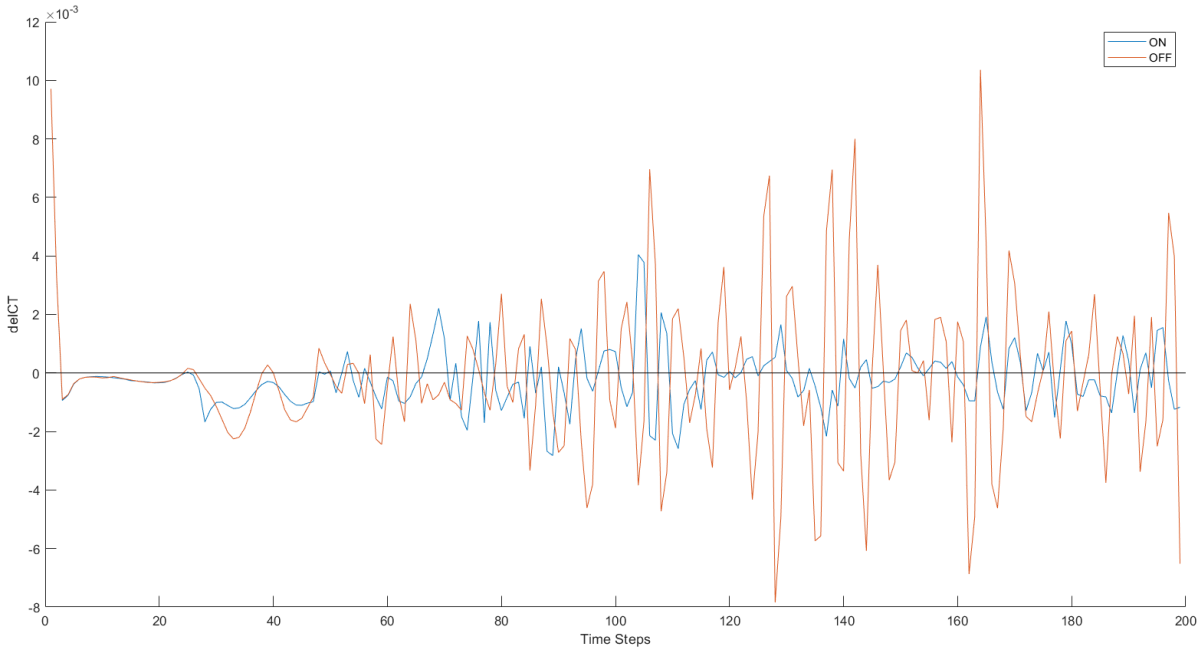

Figure 3-2 ΔCt, with splitting on and off

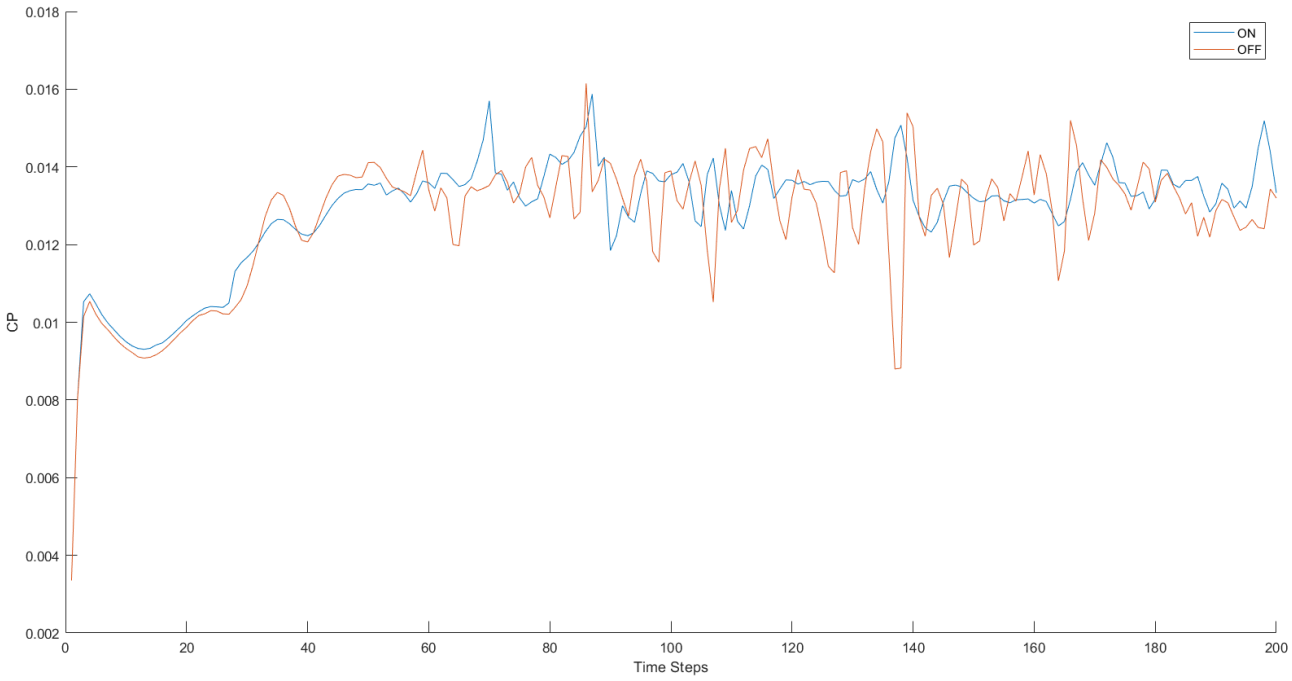

Figure 3-3 Cp, with splitting on and off

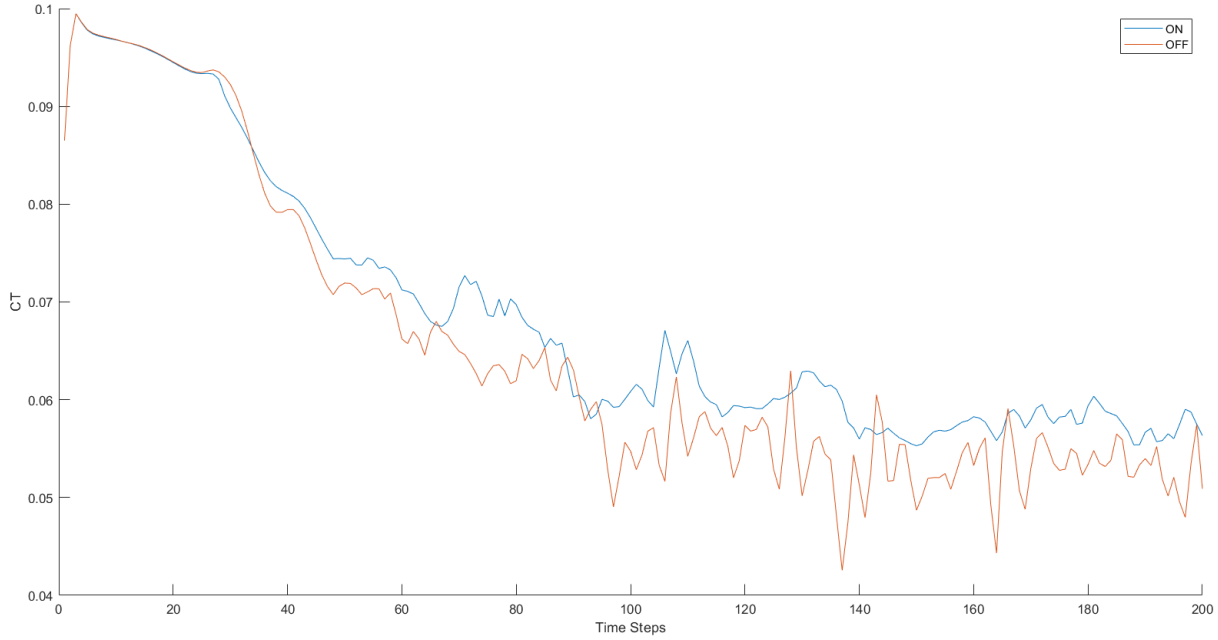

Figure 3-4 Ct, with splitting on and off

Figure 3-1 and Fig. 3-2 show the variation of ΔCp and ΔCt with the time step when splitting is on and off, while Fig. 3-3 and Fig. 3-4 show the variation of Cp and Ct with the time step when splitting is on and off, with Projection Factor or PF = 0.8 and Maximum Span Factor or MF = 0.275.

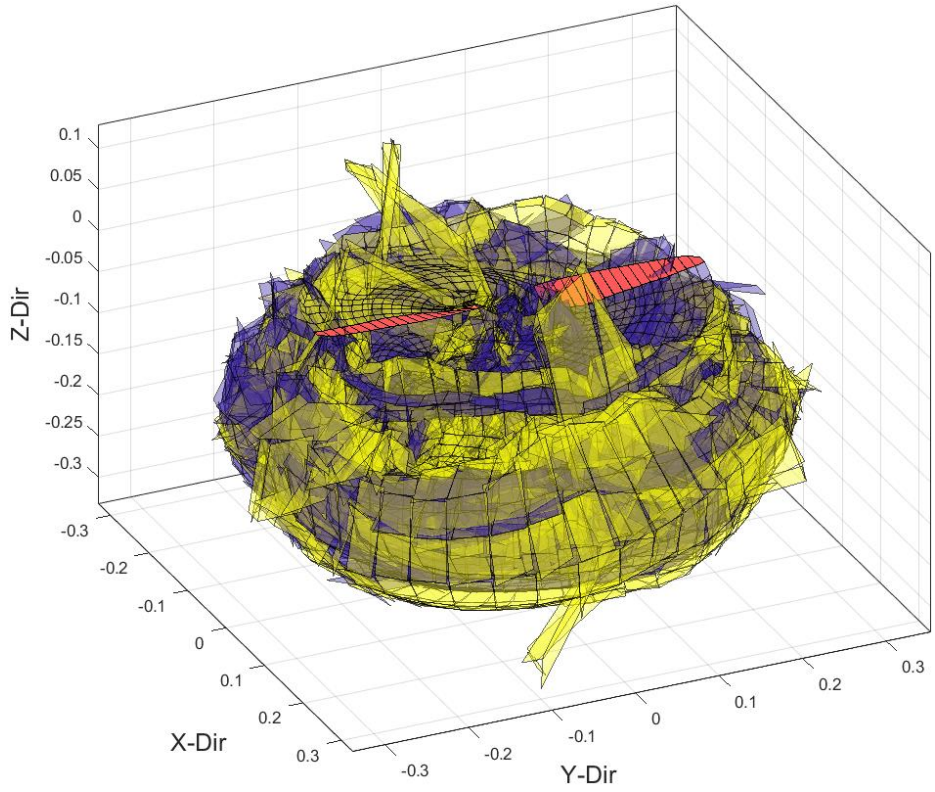

Figure 3-5 Wake DVEs with splitting off.

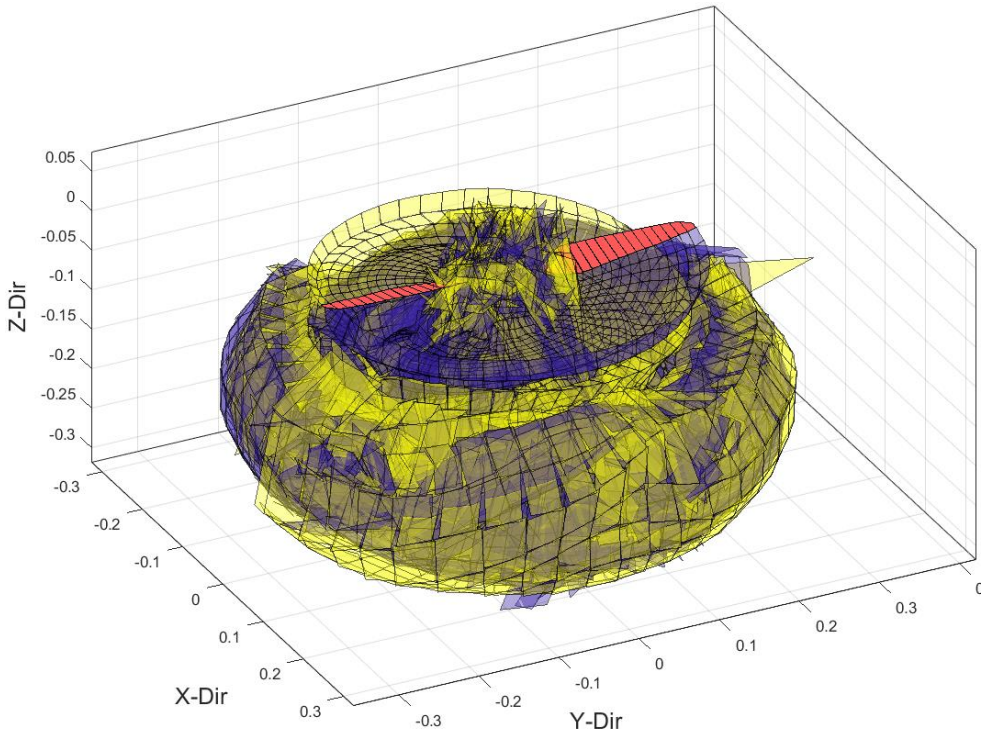

Figure 3-6 Wake DVEs with splitting on.

Figure 3-5 and Fig. 3-6 shows the wake elements of the test runs with splitting on and off. The yellow wake DVEs are created by one rotor blade while the blue wake DVEs are created by the other rotor blade.

3.2 Splitting Parameter Study

Table 3-1 Top ten optimization runs

PF	MF	mean ΔCp	std ΔCp	mean ΔCt	std ΔCt	Smoothness Index
0.8	0.275	3.38E-04	4.92E-04	8.11E-04	9.39E-04	5.11E-01
0.8	0.3	3.34E-04	4.74E-04	9.31E-04	0.001146	5.46E-01
0.8	0.4	3.60E-04	5.37E-04	9.09E-04	0.001112	5.69E-01
0.35	0.15	3.81E-04	5.55E-04	9.07E-04	0.001104	5.81E-01
0.838	0.4	3.96E-04	5.65E-04	8.84E-04	0.001083	5.84E-01
0.775	0.4	3.61E-04	5.39E-04	9.46E-04	0.001316	6.02E-01
0.9	0.4	4.12E-04	5.20E-04	0.001024477	0.001179	6.05E-01
0.7	0.4	3.99E-04	5.11E-04	0.001056945	0.001214	6.06E-01
0.75	0.4	4.44E-04	5.72E-04	0.001015067	0.001045	6.15E-01
0.825	0.41	3.87E-04	5.48E-04	0.001014707	0.001294	6.20E-01

Table 3-1 lists the top 10 runs when ordered by its Smoothness Index in ascending order. The Smoothness Index was calculated using equation 2-4. PF is the Projection Factor and MF is the Maximum Span Factor. The entire table is available in Appendix III – Extra Results. Table 3-1 allows one to be able to quickly ascertain which combination of parameters work best as the entries at the top have lower Smoothness indices which indicate better smoothness of solution.

Table 3-2 No Splitting Run

mean delCP	std delCP	mean delCt	std delCt
6.47E-04	8.27E-04	0.001895665	0.001918

The values shown in Table 3-2 are the ones used to normalize all the test runs using equation 2-4.

3.3 COMPUTATION TIME

Table 3-3 Function profile with Splitting Disabled.

Function Name	Number of Calls	Total Time (s)	Self Time (s)
fcnVSIND	3858	5945.034	5945.034
fcnDVEIND_CHUNKS	1197	9006.787	2213.164
fcnDVEVEL	1196	10050.094	1043.348
fcnSTARGLOB	5990	495.855	495.855
fcnGLOBSTAR	5660	339.066	339.066
fcnWDVEVEL	598	9976.588	175.054

Total time of run - 10288.346 s

Table 3-4 Function profile with Splitting Enabled.

Function Name	Number of Calls	Total Time (s)	Self Time (s)
fcnVSIND	4254	5859.729	5859.729
fcnDVEIND_CHUNKS	1395	8807.554	2106.494
fcnDVEVEL	1394	9666.341	858.817
fcnSTARGLOB	6584	485.959	485.959
fcnGLOBSTAR	6452	341.811	341.811
fcnWDVEVEL_RELAX+fcnWDVEVEL	598	9671.301	184.196
fcnWDVEJVEL_SPLIT	198	47.39	42.373

Total time of run - 10228.241 s

Table 3-3 and Table 3-4 only lists the functions with the most impact i.e. these functions make up more than 96% of the total time. In addition to that Table 3-4 lists the fcnWDVEJVEL_SPLIT even though its low impact. Number of calls refers to the number of times the function was executed. Self Time refers to the total time minus the time taken by nested functions to be executed.

CHAPTER 4: DISCUSSION

4.1 SPLITTING IMPROVEMENTS

Table 4-1 Best Run

PF	MaxF	mean delCP	std delCP	mean delCt	std delCt	Smoothness Index
0.8	0.275	3.38E-04	4.92E-04	8.11E-04	9.39E-04	5.11E-01

From Table 3-1 we see the best run has a Smoothness Index of 0.511, which is also included in Table 4-1. This means there is on average a two times improvement over not splitting since lower smoothness indices are desired. It is interesting to note all runs with index >1 means it is worse than the no split case. From Fig. 3-1 and Fig. 3-2, which represents the best run, it is easy to visually notice the difference when splitting is enabled. There is a noticeable decrease in the variations of ΔCp and ΔCt.

From Fig. 3-3 and Fig. 3-4 the improved smoothness can be visually seen from the solution i.e. Cp and Ct which is what the implementation of DVE splitting is trying to achieve. Figure 3-5 and Fig 3-6. shows the wake diagrams and a clear difference can be visually noticed between them. Figure 3-5 shows the wake with splitting disabled and its wake elements are deformed with no real pattern with some elements disproportionally larger and this is to be expected. Figure 3-6 on the other hand has splitting enabled and the wake elements are a lot smoother, with much less disproportional deformation albeit some elements still exhibit this unwanted behaviour.

When examining the results of the computation test, there is a difference of 0.6% between the two scenarios with splitting actually lower in computation time. This implies that the difference is negligible and variations between the two are more due to availability of computational resources rather than the fact that splitting was enabled.

The impact is negligible because of two main reasons. Firstly, fcnWDVEJVEL_SPLIT, which splits the DVEs takes 47 seconds to execute for a run of 200 timesteps which is a negligible fraction or 0.46% of the total time. Secondly, when the impact of the increased induced wake DVE velocity computation is studied using the following equation:

$$\%increase \sim \frac{1200\Delta n_{smp}}{\Delta n_{dve}\Delta n_{wmp}\left(2n_{step}+1\right)}\% \quad (4\text{-}1)$$

In equation 4-1, Δn_{smp} is the number of shared midpoints per time step, Δn_{wmp} is the number of midpoints per time step, Δn_{dve} is the number of wake DVEs per time step, n_{step} is the number of time steps.

Thus, for a run of 200-time steps and 32 shared midpoints ,34 wake DVEs and 36 midpoints per time step implies a 0.08% increase in induced wake DVE velocity calculation, which again is negligible. So large datasets see a negligible impact and small datasets see a significant impact, but since the amount of time taken is low for small datasets, the increase in time in either scenario is acceptable. Derivation for equation 4-1 is available in Appendix IV.

4.2 Parameter Impact Uncertainty

When splitting is enabled, the user has to provide the two parameters which are the projection factor and max splitting factor explained in section 2.1. While its ideal to keep them as low as possible, it's hard to predict its exact impact on the smoothness of the solution. There is no apparent pattern amongst all the runs as seen in Table 3-1 and Table III-1 (Appendix III) between the S_{index} and the two parameters. A reason for this could be that since S_{index} considers the average values of all the time steps and each time step is influenced by splitting at its time step and the ones in the previous time steps, that is, it is a complex system and knowing the exact effects is very hard and beyond the scope of this project.

But the user can overcome this uncertainty as they can tune the parameter to his/her liking based on the type of problem to get the best S_{index} possible. In general, the goal is to make sure there is minimal deviation from the original DVEs to the outer iDVEs but the inner iDVEs should also have a certain amount of span to actually have a benefit. This sweet spot is hard to calculate and is unique to each solution. But having said that, there is still an improvement over a vast range of parameters, it is just hard to predict which parameter values and combination gives the best improvement possible.

CHAPTER 5: CONCLUSION

In conclusion, this project includes one function and makes minor modifications to the other functions in VAPTOR. The modifications implement DVE splitting method to create a robust wake DVEs that do not give errors during wake rollup. This method is especially important for analysing rotors during hover and vortex ring state.

The test cases indicated twice the increase in smoothness. Although it might vary with different test cases, it should still, on average, be twice the smoothness in the desired solutions. The parameters can always be adjusted/tuned based on the user's preference and the type of problem being dealt with.

Since the computation required to implement the DVE splitting method is negligible it is recommended to enable it for all cases. The DVE splitting method can be enabled or disabled based on the user's preference.

APPENDIX I – SOURCE CODE

```
function [w_ind] = fcnWDVEJVEL_SPLIT(w_ind,dvetype,len,valTIMESTEP,
WAKE, SURF, FLAG)

%Splits DVE's and finds the velocity at the junction of two DVE's and
%places it in the original w_ind matrix.

re_adj=WAKE.matWADJE(WAKE.matWADJE(:,2)==2 &
WAKE.matWADJE(:,1)>WAKE.valWSIZE,[1 2 3]);

dve_id   = reshape(re_adj(:,[1 3])',[],1);
vert_id  = reshape(WAKE.matWDVE(dve_id,:)',[],1);
adve     = WAKE.matWVLST(vert_id,:);
n_pair   = length(re_adj(:,1));
coeff    = WAKE.matWCOEFF(dve_id,:);
a_hspn_o = WAKE.vecWDVEHVSPN(dve_id,:);

midp_id = uint32(WAKE.matWDVEMPIDX(re_adj(:,1),2));
i_midp = WAKE.matWDVEMP(midp_id,:);
% clear re_adj

ledge_hlen = sum((adve(2:8:end-6,:)-adve(3:8:end-5,:)).^2,2).^0.5/2;
redge_hlen = sum((adve(5:8:end-3,:)-adve(8:8:end,:)).^2,2).^0.5/2;
aledge_dir = (adve(2:8:end-6,:)-adve(3:8:end-5,:))./(2*ledge_hlen) ;
aredge_dir = (adve(5:8:end-3,:)-adve(8:8:end,:))./(2*redge_hlen) ;

iedge_dir  = (aledge_dir+aredge_dir);iedge_dir =
iedge_dir./sum(iedge_dir.^2,2).^0.5;

laspn_dir = (adve(1:8:end-7,:)+adve(4:8:end-4,:))/2-(adve(2:8:end-
6,:)+adve(3:8:end-5,:))/2; laspn_dir =
laspn_dir./sum(laspn_dir.^2,2).^0.5;
raspn_dir = (adve(6:8:end-2,:)+adve(7:8:end-1,:))/2-(adve(5:8:end-
3,:)+adve(8:8:end,:))/2;   raspn_dir =
raspn_dir./sum(raspn_dir.^2,2).^0.5;

ispn_dir  = cross(cross(laspn_dir,raspn_dir),(laspn_dir+raspn_dir));
ispn_dir  = ispn_dir./sum(ispn_dir.^2,2).^0.5;

ang       =
reshape(repmat(acosd(dot(laspn_dir,raspn_dir,2))/2,1,2)',[],1);

i_hspn   = WAKE.valISPNF(1).*(a_hspn_o).*sind(ang);
i_hspn(i_hspn<WAKE.valISPNF(2)*a_hspn_o) =
WAKE.valISPNF(2)*a_hspn_o(i_hspn<WAKE.valISPNF(2)*a_hspn_o);
i_hspn(i_hspn>WAKE.valISPNF(3)*a_hspn_o) =
WAKE.valISPNF(3)*a_hspn_o(i_hspn>WAKE.valISPNF(3)*a_hspn_o);

% a_hspn_min   = min([a_hspn_o(1:2:end-1) a_hspn_o(2:2:end)],[],2);
```

```
% i_hspn       = WAKE.valISPNF(1).*a_hspn_min.*sind(ang);
% i_hspn(i_hspn<WAKE.valISPNF(2)*a_hspn_min) =
WAKE.valISPNF(2)*a_hspn_min(i_hspn<WAKE.valISPNF(2)*a_hspn_min);
% i_hspn(i_hspn>WAKE.valISPNF(3)*a_hspn_min) =
WAKE.valISPNF(3)*a_hspn_min(i_hspn>WAKE.valISPNF(3)*a_hspn_min);
% i_hspn = reshape(repmat(i_hspn,1,2)',[],1);

clear laspn_dir raspn_dir a_hspn_min

idve = zeros(n_pair*8,3);
idve(1:8:end-7,:) = i_midp - ispn_dir.*2.*i_hspn(1:2:end-1) +
iedge_dir.*ledge_hlen;
idve(4:8:end-4,:) = i_midp - ispn_dir.*2.*i_hspn(1:2:end-1) -
iedge_dir.*ledge_hlen;
idve(2:8:end-6,:) = i_midp + iedge_dir.*ledge_hlen;
idve(3:8:end-5,:) = i_midp - iedge_dir.*ledge_hlen;
idve(5:8:end-3,:) = i_midp + iedge_dir.*redge_hlen;
idve(8:8:end,:)   = i_midp - iedge_dir.*redge_hlen;
idve(6:8:end-2,:) = i_midp + ispn_dir.*2.*i_hspn(2:2:end) +
iedge_dir.*redge_hlen;
idve(7:8:end-1,:) = i_midp + ispn_dir.*2.*i_hspn(2:2:end) -
iedge_dir.*redge_hlen;

adve_o=adve;
adve(2:8:end-6,:) = i_midp - ispn_dir.*2.*i_hspn(1:2:end-1) +
aledge_dir.*ledge_hlen;
adve(3:8:end-5,:) = i_midp - ispn_dir.*2.*i_hspn(1:2:end-1) -
aledge_dir.*ledge_hlen;
adve(5:8:end-3,:) = i_midp + ispn_dir.*2.*i_hspn(2:2:end)   +
aredge_dir.*redge_hlen;
adve(8:8:end,:)   = i_midp + ispn_dir.*2.*i_hspn(2:2:end)   -
aredge_dir.*redge_hlen;

ale_dir   = adve(2:4:end-2,:) - adve(1:4:end-3,:);
ale_dir   = ale_dir./sum(ale_dir.^2,2).^0.5;
aedge_dir = reshape([aledge_dir aredge_dir]',3,[])';

clear aledge_dir aredge_dir ledge_hlen

a_leswp      = acos(dot(aedge_dir,ale_dir,2))-pi/2;
a_hspn = sum((adve(1:4:end-3,:)-adve(2:4:end-
2,:)).^2,2).^0.5.*cos(a_leswp)/2;
i_hspn = i_hspn.*reshape(repmat(cos(acos(dot(ispn_dir,iedge_dir,2))-
pi/2),1,2)',[],1);

% coeff_o=coeff;
i=[1,1,1,2,2,3,3,3,3,3,3,4,4,4,4,5,5,5,5,5,5,6,6,6,6,7,7,7,7,7,7,8,8,8
,8,9,9,9,10,10,11,11,11,11,12,12,12,12]'+(0:12:(n_pair-1)*12);i=i(:);
j=[1,2,3,2,3,1,2,3,4,5,6,2,3,5,6,4,5,6,7,8,9,5,6,8,9,7,8,9,10,11,12,8,
9,11,12,10,11,12,11,12,1,3,4,6,7,9,10,12]'+(0:12:(n_pair-
1)*12);j=j(:);
```

```
A=[ones(n_pair,1), -a_hspn(1:2:end-1), a_hspn(1:2:end-1).^2,
ones(n_pair,1), -2*a_hspn(1:2:end-1),     ones(n_pair,1),
a_hspn(1:2:end-1),...
   a_hspn(1:2:end-1).^2, -ones(n_pair,1), i_hspn(1:2:end-1), -
i_hspn(1:2:end-1).^2,      ones(n_pair,1), 2*a_hspn(1:2:end-1),...
   -ones(n_pair,1), 2*i_hspn(1:2:end-1),        ones(n_pair,1),
i_hspn(1:2:end-1), i_hspn(1:2:end-1).^2, -ones(n_pair,1),
i_hspn(2:2:end), -a_hspn(2:2:end).^2,          ...
   ones(n_pair,1), 2*i_hspn(1:2:end-1), -ones(n_pair,1),
2*i_hspn(2:2:end),         ones(n_pair,1), i_hspn(2:2:end),
i_hspn(2:2:end).^2, ...
   -ones(n_pair,1), a_hspn(2:2:end), -a_hspn(2:2:end).^2,
ones(n_pair,1), 2*i_hspn(2:2:end), -ones(n_pair,1), 2*a_hspn(2:2:end),
ones(n_pair,1), a_hspn(2:2:end), a_hspn(2:2:end).^2,        ...
   ones(n_pair,1), 2*a_hspn(2:2:end),           a_hspn(1:2:end-
1)./(a_hspn(1:2:end-1)+i_hspn(1:2:end-1)), a_hspn(1:2:end-
1).^3./(3*(a_hspn(1:2:end-1)+i_hspn(1:2:end-1))), i_hspn(1:2:end-
1)./(a_hspn(1:2:end-1)+i_hspn(1:2:end-1)), i_hspn(1:2:end-
1).^3./(3*(a_hspn(1:2:end-1)+i_hspn(1:2:end-1))),      ...
   i_hspn(2:2:end)./(a_hspn(2:2:end)+i_hspn(2:2:end)),
i_hspn(2:2:end).^3./(3*(a_hspn(2:2:end)+i_hspn(2:2:end))),a_hspn(2:2:e
nd)./(a_hspn(2:2:end)+i_hspn(2:2:end)),
a_hspn(2:2:end).^3./(3*(a_hspn(2:2:end)+i_hspn(2:2:end)))]';A=A(:);
A=sparse(i,j,A);
coeff=[coeff(1:2:end-1,1)-coeff(1:2:end-1,2).*a_hspn_o(1:2:end-
1)+coeff(1:2:end-1,3).*a_hspn_o(1:2:end-1).^2,...
       coeff(1:2:end-1,2)-2*coeff(1:2:end-1,3).*a_hspn_o(1:2:end-
1),zeros(n_pair,6),...

coeff(2:2:end,1)+coeff(2:2:end,2).*a_hspn_o(2:2:end)+coeff(2:2:end,3).
*a_hspn_o(2:2:end).^2,...
       coeff(2:2:end,2)+2*coeff(2:2:end,3).*a_hspn_o(2:2:end),...
       coeff(1:2:end-1,1)+coeff(1:2:end-1,3).*a_hspn_o(1:2:end-
1).^2/3,...
       coeff(2:2:end,1)+coeff(2:2:end,3).*a_hspn_o(2:2:end).^2/3
]';coeff=coeff(:);
clear a_hspn_o
coeff=A\coeff;
coeff = reshape(coeff,3,[])';
i=uint32([1;4]+(0:4:n_pair*4-4));i=i(:);
j=uint32([2;3]+(0:4:n_pair*4-4));j=j(:);
coeff=[coeff(i,:);coeff(j,:)];

i_hchrd = sum(((idve(1:4:end-3,:)+idve(2:4:end-2,:))./2-(idve(3:4:end-
1,:)+idve(4:4:end,:))./2).^2,2).^0.5/2;
a_hchrd = sum(((adve(1:4:end-3,:)+adve(2:4:end-2,:))./2-(adve(3:4:end-
1,:)+adve(4:4:end,:))./2).^2,2).^0.5/2;

n_adve = cross(ale_dir,aedge_dir);
n_adve=n_adve./sum(n_adve.^2,2).^0.5;
```

```
n_idve =
reshape(repmat(cross(ispn_dir,iedge_dir),1,2)',3,[])';n_idve=n_idve./s
um(n_idve.^2,2).^0.5;

a_roll   = -atan2(n_adve(:,2),n_adve(:,3));
a_pitch  = asin(n_adve(:,1));
aloc_dir = fcnGLOBSTAR((adve(1:4:end-3,:)+adve(2:4:end-2,:))./2-
(idve(3:4:end-
1,:)+adve(4:4:end,:))./2,a_roll,a_pitch,zeros(n_pair*2,1));
a_yaw    = atan2(-aloc_dir(:,2),-aloc_dir(:,1));
i_roll   = -atan2(n_idve(:,2),n_idve(:,3));
i_pitch  = asin(n_idve(:,1));
iloc_dir = fcnGLOBSTAR((idve(1:4:end-3,:)+idve(2:4:end-2,:))./2-
(idve(3:4:end-
1,:)+idve(4:4:end,:))./2,i_roll,i_pitch,zeros(n_pair*2,1));
i_yaw    = atan2(-iloc_dir(:,2),-iloc_dir(:,1));

i_leswp      = idve(2:4:end-2,:)-idve(1:4:end-3,:);
i_leswp      = i_leswp./sum(i_leswp.^2,2).^0.5;
i_leswp(:,1) =
acos(dot(reshape(repmat(iedge_dir,1,2)',3,[])',i_leswp,2))-
pi/2;i_leswp=i_leswp(:,1);

singf = repmat(WAKE.vecWK(dve_id),2,1);
dvenum = reshape([1;n_pair*2+1;2;n_pair*2+2]+(0:2:n_pair*2-2),[],1);
fpg = reshape(repmat(i_midp,1,4)',3,[])';
dvetype = dvetype((dve_id(1:2:end-1)-1)*len+1);dvetype =
reshape(repmat(dvetype,1,4)',[],1);

t_dve_id = reshape(uint32((1:1:n_pair*4*4)'),4,[])';
w_ind_j = fcnDVEVEL(dvenum, fpg, dvetype, t_dve_id , [adve;idve] ,
coeff , singf , [a_hspn;i_hspn], [a_hchrd;i_hchrd] , [a_roll;i_roll] ,
[a_pitch;i_pitch] , [a_yaw,i_yaw] , [a_leswp;i_leswp] ,
[a_leswp;i_leswp] , SURF.vecDVESYM, FLAG.GPU);
w_ind_j = reshape((w_ind_j(1:2:end-1,:)+w_ind_j(2:2:end,:))',[],1);

dve_id = reshape(repmat(dve_id,1,3)',[],1);
midp_id = reshape((uint32([1;2;3;1;2;3])+(midp_id-1)'*3),[],1);
w_id = sub2ind(size(w_ind),dve_id,midp_id);
w_ind(w_id) = w_ind_j;

end
```

dvetest.m script

```
%Create Quad
f1 = figure('Name','Before Split');
hold on
% d_id = [re_adj(:,1),re_adj(:,3)]';d_id=d_id(:);
% pitch_o = WAKE.vecWDVEPITCH(d_id);
% roll_o = WAKE.vecWDVEROLL(d_id);
% yaw_o = WAKE.vecWDVEYAW(d_id);
```

```
n=6100; %Enter which pair of DVEs you wanna observe.
o=(1:4)+8*(n-1);
a=(1:4)+8*(n-1);
b=1:2+2*(n-1);
fill3(adve_o(a,1),adve_o(a,2),adve_o(a,3),'r');
fill3(adve_o(a+4,1),adve_o(a+4,2),adve_o(a+4,3),'y');
%
quiver3(center(b,1),center(b,2),center(b,3),normal(b,1),normal(b,2),no
rmal(b,3));
%
quiver3(midp(a(1),1),midp(a(1),2),midp(a(1),3),rollaxis(b(1),1),rollax
is(b(1),2),rollaxis(b(1),3),0.2);

hold off
% aaang = acosd(dot(laspn_dir,raspn_dir,2));
%Roll angle rot  axis parallel to x passing through center,
%Pitch angle rot axis dve span, Yaw angle rot axis normal.

%Create Quad
f2 = figure('Name','After Split');
hold on
fill3(adve(a,1),adve(a,2),adve(a,3),'r');
fill3(adve(a+4,1),adve(a+4,2),adve(a+4,3),'y');
fill3(idve(o,1),idve(o,2),idve(o,3),'g');
fill3(idve(o+4,1),idve(o+4,2),idve(o+4,3),'g');
% %
quiver3(center_n(b,1),center_n(b,2),center_n(b,3),normal_n(b,1),normal
_n(b,2),normal_n(b,3));
% %
quiver3(midp(a(1),1),midp(a(1),2),midp(a(1),3),rollaxis(b(1),1),rollax
is(b(1),2),rollaxis(b(1),3),0.2);

hold off
```

dev.m script

```
clc
clear
load('tm_200_0.8_0.01-0.27n0.28.mat');
id=1;
Cp = OUTP(id).vecCP;
Ct = OUTP(id).vecCT;

delCp = Cp(2:end)-Cp(1:end-1);
delCt = Ct(2:end)-Ct(1:end-1);

mean_delp = mean(abs(delCp));
std_delp  = std(abs(delCp));

mean_delt = mean(abs(delCt));
std_delt  = std(abs(delCt));
```

```
% deldelCp = delCp(2:end)-delCp(1:end-1);
% deldelCt = delCt(2:end)-delCt(1:end-1);
%
% mean_deldelp = mean(abs(deldelCp));
% std_deldelp  = std(abs(deldelCp));
%
% mean_deldelt = mean(abs(deldelCt));
% std_deldelt  = std(abs(deldelCt));

% delCp = Cp(191:end)-Cp(190:end-1);
% delCt = Ct(191:end)-Ct(190:end-1);
%
% mean_cdelp = mean(abs(delCp));
% std_cdelp  = min(std(abs(delCp)));
%
% mean_cdelt = min(mean(abs(delCt)));
% std_cdelt  = min(std(abs(delCt)));

FCp = Cp(end);
FCt = Ct(end);
```

APPENDIX II – 12 COEFFICIENT EQUATIONS

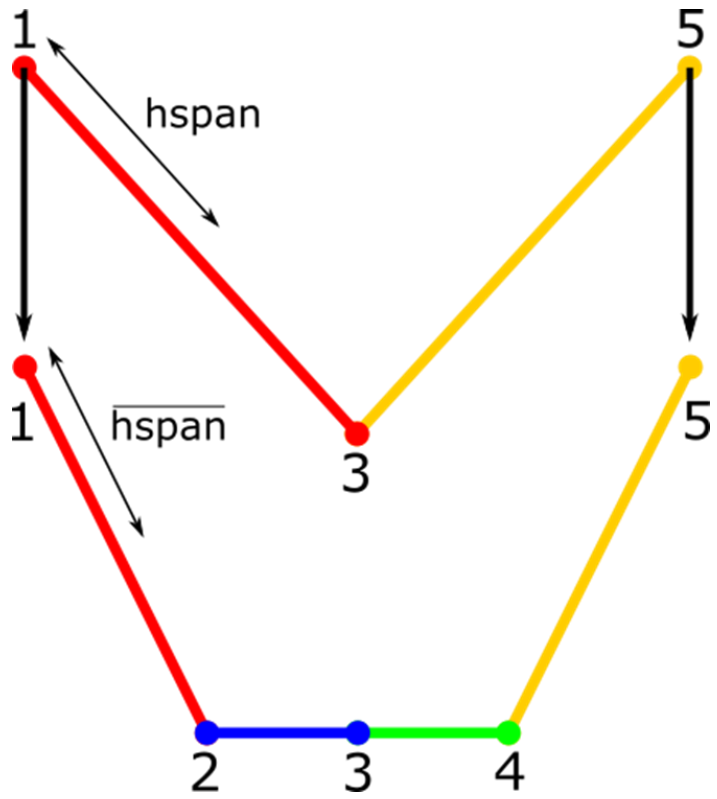

Figure II-1 Shows how the old and new DVEs are related, this relation is used to calculate the new coefficients.

Knowns – $\overline{A_l}, \overline{B_l}, \overline{C_l}, \overline{A_r}, \overline{B_r}, \overline{C_r}, \overline{hspan_l}, \overline{hspan_r}, hspan_l, hspan_r, hspan_{ml}, hspan_{mr}$

Unknowns - $A_l, B_l, C_l, A_{ml}, B_{ml}, C_{ml}, A_{mr}, B_{mr}, C_{mr}, A_r, B_r, C_r$

Point 1

$$A_l - B_l hspan_l + C_l hspan_l{}^2 = \overline{A_l - B_l hspan_l + C_l hspan_l{}^2} \quad (II\text{-}1)$$

$$B_l - 2C_l hspan_l = \overline{B_l - 2C_l hspan_l} \quad (II\text{-}2)$$

Point 2

$$A_l + B_l hspan_l + C_l hspan_l{}^2 - A_{ml} + B_{ml} hspan_{ml} - C_{ml} hspan_{ml}{}^2 = 0 \quad (II\text{-}3)$$

$$B_l + 2C_l hspan_l - B_{ml} + 2C_{ml} hspan_{ml} = 0 \quad (II\text{-}4)$$

Point 3

$$A_{ml} + B_{ml}hspan_{ml} + C_{ml}hspan_{ml}{}^2 - A_{mr} + B_{mr}hspan_{mr} - C_{mr}hspan_{mr}{}^2 = 0 \quad \textit{(II-5)}$$

$$B_{ml} + 2C_{ml}hspan_{ml} - B_{mr} + 2C_{mr}hspan_{mr} = 0 \quad \textit{(II-6)}$$

Point 4

$$A_{mr} + B_{mr}hspan_{mr} + C_{mr}hspan_{mr}{}^2 - A_r + B_rhspan_r - C_rhspan_r{}^2 = 0 \quad \textit{(II-7)}$$

$$B_{mr} + 2C_{mr}hspan_{mr} - B_r + 2C_rhspan_r = 0 \quad \textit{(II-8)}$$

Point 5

$$A_r + B_rhspan_r + C_rhspan_r{}^2 = \overline{A_r + B_rhspan_r + C_rhspan_r{}^2} \quad \textit{(II-9)}$$

$$B_r + 2C_rhspan_r = \overline{B_r + 2C_rhspan_r} \quad \textit{(II-10)}$$

Effective Circulation is preserved.

$$A_l\frac{hspan_l}{hspan_l + hspan_{ml}} + C_l\frac{hspan_l{}^3}{3(hspan_l + hspan_{ml})} + A_{ml}\frac{hspan_{ml}}{hspan_l + hspan_{ml}} + C_{ml}\frac{hspan_{ml}{}^3}{3(hspan_l + hspan_{ml})} = \overline{A_l + C_l\frac{hspan_l{}^2}{3}} \quad \textit{(II-11)}$$

$$A_{mr}\frac{hspan_{mr}}{hspan_r + hspan_{mr}} + C_{mr}\frac{hspan_{mr}{}^3}{3(hspan_l + hspan_{mr})} + A_r\frac{hspan_r}{hspan_r + hspan_{mr}} + C_r\frac{hspan_r{}^3}{3(hspan_r + hspan_{mr})} = \overline{A_r + C_r\frac{hspan_r{}^2}{3}} \quad \textit{(II-12)}$$

APPENDIX III – EXTRA RESULTS

Table III-1 Optimization runs

PF	MF	mean ΔCP	std ΔCP	mean ΔCt	std ΔCt	Smoothness Index
0.8	0.275	3.38E-04	4.92E-04	8.11E-04	9.39E-04	5.11E-01
0.8	0.3	3.34E-04	4.74E-04	9.31E-04	0.001146	5.46E-01
0.8	0.4	3.60E-04	5.37E-04	9.09E-04	0.001112	5.69E-01
0.35	0.15	3.81E-04	5.55E-04	9.07E-04	0.001104	5.81E-01
0.838	0.4	3.96E-04	5.65E-04	8.84E-04	0.001083	5.84E-01
0.775	0.4	3.61E-04	5.39E-04	9.46E-04	0.001316	6.02E-01
0.9	0.4	4.12E-04	5.20E-04	0.001024477	0.001179	6.05E-01
0.7	0.4	3.99E-04	5.11E-04	0.001056945	0.001214	6.06E-01
0.75	0.4	4.44E-04	5.72E-04	0.001015067	0.001045	6.15E-01
0.825	0.41	3.87E-04	5.48E-04	0.001014707	0.001294	6.20E-01
1	0.35	4.14E-04	5.56E-04	0.001014094	0.001247	6.26E-01
0.83	0.4	4.23E-04	5.18E-04	0.001123188	0.001217	6.26E-01
0.4	0.35	4.17E-04	5.66E-04	0.001019081	0.001226	6.28E-01
0.85	0.4	4.33E-04	5.32E-04	0.001068601	0.00125	6.31E-01
0.826	0.4	4.33E-04	5.55E-04	1.09E-03	0.001315	6.51E-01
0.812	0.4	4.03E-04	5.73E-04	0.001086061	0.001365	6.53E-01
0.824	0.4	3.94E-04	6.57E-04	9.74E-04	0.001418	6.70E-01
0.825	0.4	4.30E-04	6.20E-04	0.001060992	0.00141	6.80E-01
0.4	0.2	4.36E-04	6.96E-04	9.78E-04	0.001383	6.94E-01
0.8	0.35	4.37E-04	6.22E-04	0.001151571	0.001429	6.98E-01
1	0.4	4.94E-04	6.64E-04	0.001128998	0.001327	7.15E-01
0.82	0.4	5.02E-04	6.11E-04	0.001251373	0.001377	7.22E-01
0.8	0.45	5.18E-04	6.55E-04	0.001296776	0.001516	7.67E-01
0.825	0.39	4.55E-04	7.33E-04	0.001086442	0.001709	7.70E-01
0.6	0.35	5.11E-04	6.64E-04	0.001263612	0.001725	7.91E-01
0.8	0.325	6.55E-04	8.09E-04	0.001630962	0.002157	9.93E-01
0.6	0.4	7.12E-04	9.64E-04	0.001561025	0.00173	1.00E+00

APPENDIX IV – DERIVATIONS

To calculate the approximate increase in the Wake DVE velocity computation, first look at the approximate increase per time step shown:

$$\%increase\ per\ time\ step \sim \frac{400 n_{smp} tiv_{step}}{n_{dve} n_{mp} tiv_{step}} = \frac{400 n_{smp}}{n_{dve} n_{mp}} \quad (IV\text{-}1)$$

In equation IV-1, n_{smp} is the number of shared midpoints, tiv_{step} is the time taken to calculate the induced velocity on a point by a DVE, n_{dve} is the number of wake DVEs and n_{mp} is the number of midpoints. Equation IV-1 is the ratio between the number of times a velocity has to be calculated for split DVEs and the number of times the velocity has to be calculated when splitting is disabled. To calculate the cumulative/total increase in the computation time, the effects of all-time steps are summed up, shown here:

$$\%increase \sim \frac{400 \Delta n_{smp} tiv_{step} \sum_{x=1}^{x=n_{step}} x}{\Delta n_{dve} \Delta n_{mp} tiv_{step} \sum_{x=1}^{x=n_{step}} x^2} = \frac{400 \Delta n_{smp} tiv_{step} \frac{n_{step}(n_{step}+1)}{2}}{\Delta n_{dve} \Delta n_{mp} tiv_{step} \frac{n_{step}(n_{step}+1)(2n_{step}+1)}{6}} \quad (IV\text{-}2)$$

Simplification of equation IV-2 gives the final equation:

$$\%increase \sim = \frac{1200 \Delta n_{smp}}{\Delta n_{dve} \Delta n_{mp}(2n_{step}+1)} \quad (IV\text{-}3)$$

REFERENCES

[1] G. Bramesfeld and M. D. Maughmer, "Relaxed-Wake Vortex-Lattice Method Using Distributed Vorticity Elements," *Journal Of Aircraft,* vol. 45, pp. 560-568, 2008.

[2] G. Bramesfeld, *A Higher Order Vortex-Lattice Method with a Force-Free Wake, Phd Thesis,* University Park: The Pennsylvania State University, 2006.

[3] RAALF, "VAP 3.1 Source Code," Toronto, 2018.

[4] T. Choephel, "AERODYNAMIC ANALYSIS OF HELICOPTER ROTORS USING A HIGHER-ORDER, FREE-WAKE METHOD," Phd ThrsisThe Pennsylvania State University, 2016.

[5] B. J. Basom, *Inviscid Wind-Turbine Analysis Using Distributed Vorticity Elements, MSc Thesis* The Pennsylvania State University, 2010.

[6] A. Kolaei, D. Barcelos and G. Bramesfeld, "Experimental Analysis of a Small-Scale Rotor at Various Inflow Angles," *International Journal of Aerospace Engineering,* vol. 2018, Article ID 2560370, p. 14, 2018.