AERODYNAMIC ASSESSMENT OF HUMPBACK WHALE VENTRAL FIN SHAPES

Author: Damià Rita Espada

REPORT

Director: Roberto Maurice Flores Le Roux
September 2011
Index

List of figures iii
List of tables vii
List of abbreviations ix

1. Introduction 1
   1.1. Aim of the study 1
   1.2. Justification: UAV needs 1
   1.3. Background 3
   1.4. Scope of the study 13
   1.5. Basic specifications 14

2. Experimental analysis 15
   2.1. Experiment description 15
   2.2. Model and support description and fabrication techniques 21
   2.3. Results 28
   2.4. Conclusions drawn from experiments 44
   2.5. Difficulties encountered 45

3. Application proposal 47

4. FEMPAN 50
   4.1. GiD Presentation 50
   4.2. Panel Method Code 52
   4.3. FEM Code 58
   4.4. Interaction between parts: integrator 64
   4.5. Validation 68
   4.6. Analysis of the model 72

5. Technical and economical viability of the proposal 74

6. Environmental effects of the life cycle of the proposal 76
7. Conclusions  

8. Acknowledgements

9. Bibliography

10. Annexes

| 10.1. | Annex A - Furos Datasheet |
| 10.2. | Annex B – Experimental error assessment |
| 10.3. | Annex C – Photographs metadata |
| 10.4. | Annex D – Problemtype used by FEMPAN |
| 10.5. | Annex E – FEMPAN Code Source |
List of figures

Figure 1.1 Humpback whale. Source: (Welles, 2007) ........................................... 1
Figure 1.2 Ventral fin of a humpback whale. Source: (Svenheim, 2006) ............ 5
Figure 1.3 Flipper planform and sections .......................................................... 5
Figure 1.4 Comparison of manoeuvrability, cruising speed and anatomy of different whales. Source: (Woodward, et al., 2006) ................................... 6
Figure 1.5 Fins and flukes compared to body length. Source: (Woodward, et al., 2006) .............................................................. 7
Figure 1.6 Air view of bubble net. Source: (Alaska Fisheries Science Center, 2011) ................................................................................................ 7
Figure 1.7 Models used in (Miklosovic, et al., 2004) ........................................... 8
Figure 1.8 Experimental results of (Miklosovic, et al., 2004). Solid line is for typical flipper shape results and triangles for humpback whale shaped flipper. ........................................................................................... 9
Figure 1.9 Results of (Custodio, 2007) experiments. In the top row $C_L$ vs. $\alpha$ is plotted, in the central one, $C_D$ vs. $\alpha$ is plotted and in the bottom row, $C_L/C_D$ vs. $\alpha$ is plotted. Data is for 8X family hydrofoils at the left column and for the 4X family at the right one. .................................................. 11
Figure 1.10 Industrial cooling fan modified to include leading edge tubercles. Source: (Technology Development Associates) ......................... 13
Figure 2.1 Integration of the three fans to the square section of the conducts .............................................................................................................. 16
Figure 2.2 General picture of the wind tunnel. In the left, the test section is seen. In the middle, there is the green centrifugal compressor. In the right, the fan section. .............................................................. 16
Figure 2.3 Wind tunnel data acquisition system. (1) Differential pressure sensor. (2) Test chamber absolute pressure sensor. (3) Test chamber temperature sensor. (4) Test chamber humidity sensor. Red lines indicate pressure ducts and blue lines signal cables. Balance is not shown. .............................................................................. 17
Figure 2.4 Balance used in the measurements. Some of the load cells can be seen together with the PC output....................................................... 19
Figure 2.5 Measurement of model chord .................................................................. 21
Figure 2.6 Assembly of the ribs and spars .......................................................... 22
Figure 2.7 Wooden sheet where ribs have been machined ................................ 23
Figure 2.8 Complete assembly (rib number 4 has been removed) .................... 24
Figure 2.9 Three subassemblies of rib and polystyrene filler and a wooden prismatic piece before and after assembly ............................................. 25
Figure 2.10 Trailing edge of one model showing curvature produced by polystyrene pieces ................................................................................... 25
Figure 2.11 Completed model base and balsa wood ready to be attached on the leading edge ................................................................................. 26
Figure 2.12 Fabrication process of protuberances to be attached to the model ........................................................................................................... 26
Figure 2.13 Incidence changing device ................................................................ 27
Figure 2.14 Angle measuring scale. A pen is place to give an idea of the scale. ............................................................................................................. 28
Figure 2.15 C_L comparison at Re=1,2\cdot 10^5. It has to be remarked the high α at which the wing was tested without any step at lift................................. 29
Figure 2.16 C_D comparison at Re=1,2\cdot 10^5. Only a part of the α range is shown in order to ease visual comparison......................................................... 30
Figure 2.17 C_Mac comparison at Re=1,2\cdot 10^5 .................................................. 30
Figure 2.18 C_L comparison at Re=2,4\cdot 10^5 ...................................................... 31
Figure 2.19 C_D comparison at Re=2,4\cdot 10^5 ...................................................... 32
Figure 2.20 C_Mac comparison at Re=2,4\cdot 10^5 .................................................. 32
Figure 2.21 C_L comparison at Re=3,5\cdot 10^5 ...................................................... 33
Figure 2.22 C_D comparison for Re=3,5\cdot 10^5 .................................................... 34
Figure 2.23 C_Mac comparison at Re=3,5\cdot 10^5 .................................................. 34
Figure 2.24 Lift curve of clean wing as a function of Re ........................................ 35
Figure 2.25 Lift curve of whale wing as a function of Re ....................................... 35
Figure 2.26 Lift at α=15º for different Re values ................................................... 36
Figure 2.27 Lift at α=22º for different Re values ................................................... 37
Figure 2.28 China clay result at α=14º and Re=2,4\cdot 10^5. Left image is upper side of the wing and right one is lower side ............................................. 38
Figure 2.29 Close view of leading edge near the upper wing tip. Upper side of the airfoil seen. ........................................................... 39

Figure 2.30 Photography taken during the drying process with tunnel running................................................................. 40

Figure 2.31 China clay result at $\alpha=16^\circ$ and $Re=2,4\cdot10^5$. Left image is upper side of the wing and right one is lower side ............................................... 41

Figure 2.32 China clay result at $\alpha=18^\circ$ and $Re=2,4\cdot10^5$. Left image is top side of the wing and right one is bottom side ........................................... 42

Figure 2.33 China clay result at $\alpha=24^\circ$ and $Re=2,4\cdot10^5$. Left image is upper side of the wing and right one is lower side ........................................ 43

Figure 2.34 Close view of wing tip for $\alpha=24^\circ$ and $Re=2,4\cdot10^5$........................................ 44

Figure 3.1 Rear view of the membrane (red) and wing (yellow and transparent yellow).......................................................... 48

Figure 3.2 Front view of membrane (red) and wing (yellow and transparent yellow) ........................................................................ 48

Figure 4.1 Diagram of the workflow of GiD together with the solver programme. Source: (CIMNE, 2002) .......................................................... 51

Figure 4.2 A constant intensity doublet panel is equivalent to a vortex ring. Source: (Katz, et al., 1991) ................................................................. 54

Figure 4.3 Body axes used in FEMPAN .......................................................................................................................... 56

Figure 4.4 Panel calculations flow chart .......................................................................................................................... 57

Figure 4.5 Flowchart of the process towards the equilibrium position .......................................................................................... 65

Figure 4.6 Integrator scheme where ts means time step .......................................................................................... 66

Figure 4.7 Type of mesh used .............................................................................................................................................. 68

Figure 4.8 Convergence of the $C_L$ as a function of the number of divisions made to airfoils ........................................................................... 69

Figure 4.9 $C_p$ distribution and wake geometry obtained for $\alpha=0^\circ$ ............................................................................... 69

Figure 4.10 Panel Method result compared with wind tunnel data .......................................................................................... 70

Figure 4.11 $C_p$ distribution and wake geometry of the wing at $\alpha=15^\circ$ ........................................................................ 70

Figure 4.12 Convergence of the non-dimensional maximum displacement ........................................................................ 70

Figure 4.13 Deformated shape of the test membrane. Colours indicate vertical displacement .......................................................... 71
Figure 4.14 Bottom view of the inflated tubercles with an $\alpha=20^\circ$ and $q=0.3$ ....... 72

Figure 4.15 Side view of the inflated tubercles with an $\alpha=20^\circ$ and $q=0.3$........... 73

Figure 10.1 Distribution of measurements for N measure conditions and two test runs ............................................................................................................................. 85
List of tables

Table 1.1 Wavelength and Wave amplitude of hydrofoils tested by (Custodio, 2007) .................................................................................................. 10

Table 2.1 Characteristics of the differential pressure sensor (1) (General Electric, 2008) .................................................................................................. 17

Table 2.2 Characteristics of the absolute pressure sensor (2) (General Electric, 2011) .................................................................................................. 18

Table 2.3 Characteristics of Pt100 Temperature sensor (3) (Ingeco Z.S., 2005) ........................................................................................................... 18

Table 2.4 Characteristics of the test chamber humidity sensor (4) (Rense Instruments, 2008) .................................................................................. 18

Table 2.5 Characteristics of the load cells used in the balance(Interface - Advanced Force Measurements, 2011) .................................................. 19

Table 2.6 Settings of the PC card conditioning load cells signals .......... 19

Table 2.7 Characteristics and settings of the analogue to digital converter (National Instruments, 2010) ............................................................... 20

Table 2.8 Design values for geometric characteristics of the test models ................................................................................................................ 21

Table 2.9 Geometric characteristics of fabricated models ....................... 22

Table 2.10 Protuberances dimensions .......................................................... 22

Table 2.11 Error calculated for tests at Re=1,2·10^5 ................................. 28

Table 2.12 Error calculated for tests at Re=2,4·10^5 ................................. 31

Table 2.13 Error calculated for tests at Re=3,5·10^5 ................................. 33

Table 4.1 Results comparison for the Hencky problem ........................... 72

Table 10.1 Repetition error for each of the forces and moments ............. 88

Table 10.2 Repetition error for each of the coefficients ........................... 88

Table 10.3 Metadata of Figure 2.1 ........................................................... 90

Table 10.4 Metadata of Figure 2.2 ........................................................... 90

Table 10.5 Metadata of Figure 2.4 ........................................................... 90

Table 10.6 Metadata of Figure 2.5 ........................................................... 91
Table 10.7 Metadata of Figure 2.7 ........................................................... 91
Table 10.8 Metadata of Figure 2.1 ........................................................... 91
Table 10.9 Metadata of Figure 2.9 ........................................................... 92
Table 10.10 Metadata of Figure 2.10 ....................................................... 92
Table 10.11 Metadata of Figure 2.11 ....................................................... 92
Table 10.12 Metadata of Figure 2.12 ....................................................... 93
Table 10.13 Metadata of Figure 2.13 ....................................................... 93
Table 10.14 Metadata of Figure 2.14 ....................................................... 93
Table 10.15 Metadata of Figure 2.28 ....................................................... 94
Table 10.16 Metadata of Figure 2.29 ....................................................... 94
Table 10.17 Metadata of Figure 2.30 ....................................................... 94
Table 10.18 Metadata of Figure 2.31 ....................................................... 95
Table 10.19 Metadata of Figure 2.32 ....................................................... 95
Table 10.20 Metadata of Figure 2.33 ....................................................... 95
Table 10.21 Metadata of Figure 2.34 ....................................................... 96
## List of abbreviations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Angle of attack</td>
</tr>
<tr>
<td>$\alpha_d$</td>
<td>First coefficient of Rayleigh damping</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Sideslip angle</td>
</tr>
<tr>
<td>$\beta_d$</td>
<td>Second coefficient of Rayleigh damping</td>
</tr>
<tr>
<td>$\Delta p$</td>
<td>Pressure difference between sides of the membrane</td>
</tr>
<tr>
<td>AR</td>
<td>Aspect ratio</td>
</tr>
<tr>
<td>b</td>
<td>Wing span</td>
</tr>
<tr>
<td>b/2</td>
<td>Span of the semiwing</td>
</tr>
<tr>
<td>c</td>
<td>Chord of the airfoil</td>
</tr>
<tr>
<td>$C_D$</td>
<td>Drag coefficient of wing</td>
</tr>
<tr>
<td>CFD</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>$C_L$</td>
<td>Lift coefficient of wing</td>
</tr>
<tr>
<td>$C_{L_{max}}$</td>
<td>Maximum lift coefficient of wing</td>
</tr>
<tr>
<td>$C_{Mac}$</td>
<td>Aerodynamic moment coefficient at aerodynamic centre</td>
</tr>
<tr>
<td>$C_p$</td>
<td>Pressure coefficient</td>
</tr>
<tr>
<td>E</td>
<td>Aerodynamic efficiency</td>
</tr>
<tr>
<td>$E_m$</td>
<td>Young modulus of the membrane</td>
</tr>
<tr>
<td>MP/L</td>
<td>Maximum Payload</td>
</tr>
<tr>
<td>MTOW</td>
<td>Maximum Take-Off Weight</td>
</tr>
<tr>
<td>OEW</td>
<td>Operative Empty Weight</td>
</tr>
<tr>
<td>PIV</td>
<td>Particle Image Velocimetry</td>
</tr>
<tr>
<td>q</td>
<td>Dynamic pressure</td>
</tr>
<tr>
<td>$\tilde{q}$</td>
<td>Non-dimensional dynamic pressure</td>
</tr>
<tr>
<td>Q4</td>
<td>Quadrilateral element with 4 nodes</td>
</tr>
<tr>
<td>R</td>
<td>Radius of the membrane of the Hencky problem</td>
</tr>
<tr>
<td>Re</td>
<td>Reynolds number based on the chord</td>
</tr>
<tr>
<td>Re$_x$</td>
<td>Reynolds number based on the x coordinate</td>
</tr>
</tbody>
</table>
RHS  Right-hand-side vector, part of the linear system $[A](x) = \text{(RHS)}$
RTD  Resistance Temperature Detector
$t$   Membrane thickness
T3   Triangular element with three nodes
$U_\infty$ Test air speed at the wind tunnel
$w$  Membrane deflection at the centre of the Hencky problem
1. Introduction

1.1. Aim of the study

The ventral fins of the humpback whale (*Megaptera novaeangliae*) include a bulbous leading edge (Figure 1.1) acting as a natural high-lift device.

![Humpback whale](image)

*Figure 1.1 Humpback whale. Source: (Welles, 2007)*

It has been suggested that application of this concept to wing design may yield advantages over traditional shapes (Miklosovic, et al., 2004). During the course of this project, the aerodynamic performance of whale fin models will be compared with conventional wing shapes. Based on the results of the study new wing design paradigms will be developed to improve the performance of flight vehicles operating under similar Reynolds numbers. Namely, the UAV developed by CATUAV, Furos (see Annex A - Furos Datasheet), will be used as reference aircraft.

In order to implement this new leading edge shape, a light and self-operated device will be proposed. It consists of a membrane attached to the leading edge in such a way that it inflates at high angles of attack reproducing the whale’s bulbous shape. Finally, a design tool will be presented to calculate the shape of the inflated membrane by numerical means.

1.2. Justification: UAV needs

Unmanned air vehicles (UAVs) are becoming more and more popular for low altitude and low cost terrain observation. They benefit from the fact that they do not carry the equipments needed for pilot ergonomics nor the pilot. Thus, UAVs are generally smaller than piloted flying vehicles.
There is a great range in UAV sizes. Smallest ones are a few centimetres long and fly at extremely low Reynolds numbers (Re). On the other hand, there are UAVs of the size of a fighter plane that operate at high Re. Their characteristics of the air flow are similar to that of traditional aircrafts. There is, however, a large segment of medium sized UAVs. They operate at Re of the order of $10^5$ or $10^6$. This is the kind of aircrafts this study is directed to.

For moderate values of the Reynolds number, the boundary layer begins as laminar and large transitional regions develop. This contrasts to large Re flows, which can be considered fully turbulent. The lack of turbulence in the boundary layer has a negative effect at high angle of attack ($\alpha$) as it has less energy to overcome adverse pressure gradients. Consequently, medium sized UAVs suffer from premature stall and have to use wing sections especially designed for good stall characteristics.

Furthermore, the fact that the boundary layer is transitional over a large region of the wing makes it more difficult to use Computational Fluid Dynamics (CFD) techniques. This is because CFD codes usually require as input the position of the transition band. This information is only available for simple and well-known geometries.

The small size of the wings does not allow easy implementation of traditional high-lift devices at the leading and trailing edges. Mechanical systems and actuators become too small and their weight, compared to that of the complete platform, is high. Finally, this kind of devices needs to receive orders from the flight computer. Typically the flight computer is, by far, the most expensive part of an UAV. Therefore, it is desirable to keep it as simple as possible.

Most of the characteristics presented here are common with other slow aircrafts with short chords. These low-Re conditions also appear at gliders, for example.

Poor airfoil behaviour at stall, difficult numerical simulation and low usability of typical flaps and slats make it necessary to have a low weight high-lift device specifically tailored to UAVs needs. Given the difficulties encountered trying to predict the system behaviour, a sensible option is to examine alternative solutions already available in nature. The inflatable membrane device proposed in this study follows this line of thought.
1.3. Background

1.3.1. Leonardo’s and Darwin’s views

The genius of Renaissance, Leonardo da Vinci, had already been interested in studying nature as a source of knowledge to help his designs during the XV century. He performed in-depth studies on the movement of water, the flight of birds and anatomy among others. In flight mechanics, for example, he had already discovered the relationship between banking and changing trajectory. This was a great breakthrough, as physics and mathematics were not yet linked as they are today.

Later on, in the XIX century, another great scientist, Charles Darwin, explained the mechanism by which nature achieves its great success at developing new concepts. He did so by observing the change of domestic species (cultivated vegetables, animals grown for food…) in time and the effect of isolation of different populations of the same species. He developed the Theory of Evolution. It states that nature produces random changes on individuals and those mutations that best adapt to the local environment are more likely to be transmitted to future generations becoming thus, with time, widespread and improving the species.

Leonardo and Darwin teach us that it is a good idea to find inspiration in nature, since it has been trying millions of different solutions for ages and selecting the best ones. Humans, on the other hand, do not have such a long time available for design tasks. A designer cannot test every different possibility he comes up with until the optimal solution is found. He can, however, see what nature has done to solve some similar problems.

1.3.2. Biomimetics

As it has been said, nature has already tested a myriad of possible modifications. Only those offering a larger survival probability propagate, constantly improving and adapting species. Human have looked at nature to create new designs. Some genial breakthroughs have been:

- Velcro: It was invented imitating the fruits of burdock that attach to the hair of animals with tiny hooks.
- Road reflectors: They were copied from the cat eyes, which reflect light very efficiently at the *tapetum lucidum*.
- Self-cleaning paints: Lotus leaf presents a surface where nor water nor dirt attach. It was used as inspiration for certain self-cleaning paints.
- Neural Networks: Nowadays, when numerical predictions have to be done and there is no clear way to design the algorithm, neural networks are
used. Basically, the computer simulates a network of simple calculators and adapts its connections so that the response resembles the solution sought. Neural Networks have, thus, to be trained. This kind of device is, obviously, based on the brain of animals.

- Computer viruses: They use similar techniques to real ones to reproduce themselves.

The list of human developments based on nature is very long. However, we still find difficulties to fully copy nature because of:

- Use of materials: Nature is capable of producing incredibly complex substances with finely adapted characteristics whereas humans have a very limited catalogue of available materials. Furthermore, human designs tend to use rigid materials that have to keep properties during their service life.

- “Intelligence distribution”: In nature, every single part of a system knows what its functions are and is capable of reacting to environment changes and damage. This is possible because every function is done by a community of equal partners capable of duplicating themselves: cells of the body, individuals of a community, parts of a cell … On the other hand, when humans design system all the intelligence is concentrated on the designer and in the design process.

- Complex reaction to complex inputs: Natural systems are capable to react to external changes no matter how complex they are. This is not possible in human designed systems that are limited to a relatively small number of input variables. The capability of a bird to react to local changes of air flow on its wings cannot be compared to that of a pilot on a plane. The capability of a forest to react to the fall of a tree is much bigger to that of an urbanist to react to the fall of a building.

- Natural systems are highly integrated in greater systems. This allows that parts that do not comply with their function become automatically to other parts of the system. When a human product becomes a waste in modern society, it cannot be reintegrated to the system easily. A process, consuming large amounts of energy and resources, is required. While a dead leave falls to the ground and feeds the soil ecosystem, a used solar panel has to undergo very expensive processes to have its materials reused. Human focus on solving specific problems and tend to lose the systemic view.
1.3.3. Humpback Whale case

The humpback whale (*Megaptera novaeangliae*) shows very large ventral fins. These fins are reported to have an elevated aspect ratio and a semi-span of 0.25 to 0.3 times the length of the body (Fish, et al., 1995). Additionally, fins have large protuberances on the leading edge as it is seen in Figure 1.2. These protuberances or tubercles correspond anatomically to the cartilages of the manus. Different reasons for these peculiar dimensions have been hypothesized. Among others, it has been suggested that they serve to increase wet surface in order to improve thermal regulation (Brodie, 1977), to produce acoustic signals by slapping on the water surface (Tyack, 1981), to mate (Evans, 1987) and to manoeuvre (Edel, et al., 1978).

The high aspect ratio and rounded tip of the humpback whale are shown in Figure 1.3. These are key factors when flipper are used as hydrofoils. In addition to the planform, and the size, the predating methods of the *Megaptera novaeangliae*, which call for high manoeuvrability, give an idea of the importance of the fins in the whale’s life.

Figure 1.2 Ventral fin of a humpback whale. Source: (Svenheim, 2006)

Figure 1.3 Flipper planform and sections
Comparing the blue whale (*Balaenoptera musculus*), humpback whale (*Megaptera novaeangliae*), gray whale (*Eschrichtius robustus*), and right whale (*Eubalaena glacialis*) (Woodward, et al., 2006) it is found that humpback whale is the most manoeuvrable (Figure 1.4) and the one with the highest fin surface. As for the planform, the humpback whale is found to have the highest ventral fin aspect ratio (to generate high lifts) and the largest flukes. However, the aspect ratio of the humpback whale caudal fin is slightly lower than that of the right and blue whales. This indicates that it uses its flukes to generate high thrust but only during short periods, as efficiency is sacrificed.

Figure 1.4 Comparison of manoeuvrability, cruising speed and anatomy of different whales. Source: (Woodward, et al., 2006)
When preying, humpback whales are reported to be the most acrobatic whales (Miklosovic, et al., 2004). One of their fishing techniques is the bubble net. It is a cooperative preying technique performed by groups of around 12 whales. It consists of swimming in circles around a school of fishes while releasing air through their spiracles. They do so in order to group the fishes before some preying partners eat the entire school at once. Turning diameter during the use of the bubble net technique is between 50m and 1.5m (Woodward, et al., 2006). Since their mean body length is 13.5m (Woodward, et
al., 2006), it can be considered an extreme manoeuvrability. A low-resolution aerial image of whales using fishing with bubble net can be seen in Figure 1.6.

In order to find out what applications tubercles leading edge technology could be useful for, it is important to know the Re at which Humpback whales swim. Aircraft flying at such Re would benefit from similar advantages. Humpback whales swim at a speed in the neighbourhood of 2.6m/s (Miklosovic, et al., 2004). The mean chord of the flipper is 0.5m (Miklosovic, et al., 2004). Therefore, the Re at sea water under these conditions is Re≈10^6. One can expect that during tight manoeuvres (when the high-lift capability is used), speed and Re decrease. Therefore, Re values near 5⋅10^5 are the focus of interest.

There have already been some studies on the aerodynamic benefits of humpback tubercles on leading edge. Most of them mentioned in (Bavassano, 2010). Next, some of them are outlined as they motivated this project and support the general conclusions.

1.3.3.1. Previous studies

The great number of articles released by Dr. Frank E. Fish, from the Department of biology of the West Chester University deserves special mention. He has done different studies from the anatomic, geometric, numeric and experimental points of view. He is co-author of the first experimental analysis presented in (Miklosovic, et al., 2004). There, geometric data from (Fish, et al., 1995) was used to build a couple of models to compare their characteristics in a wind tunnel. One model was a copy of the humpback whale ventral fin while the other one was an equivalent shape but with a smooth leading edge. Models can be seen in Figure 1.7. NACA 0020 airfoils were used, which are symmetrical and have a 20% relative thickness. Experiments were performed at Re between 5⋅10^5 and 5.2⋅10^5.

Results of these experiments are shown in Figure 1.8.
Little change at low $\alpha$ was seen. An increase in maximum lift and maximum efficiency were observed and stall happened at higher $\alpha$. However, the lift loss caused by stall is larger. Since humpback whales feel the flow around their fins, they are not expected to allow their flippers to stall as it could happen to a human pilot. The performance change observed is, then, suitable for use by whales.

These results have not been confirmed by other experiments. Those validations tests have found a different stall and post-stall behaviour. In general, maximum lift remains unchanged, but the lift curve does not show a stepped descend after stall.

In (Custodio, 2007), a more extensive study to see the effect of the wavelength of the sinusoidal leading edge and its amplitude was carried out. He used 7 models defined as shown in Table 1.1.
Baseline model has a NACA 634-021 airfoil, which is similar to a humpback whale cross section. Its chord is $c=102\text{mm}$ and its span, $b=203\text{mm}$. The other six models where designed with the same span and planform surface. Spanwise chord changes do not affect the shape of the wing at the rear end because the airfoil is modified in order to confine all changes into the forward area. The experiment was conducted at $Re=1,83\cdot10^5$. Results are shown in Figure 1.9.

**Table 1.1 Wavelength and Wave amplitude of hydrofoils tested by (Custodio, 2007)**

<table>
<thead>
<tr>
<th>Hydrofoil</th>
<th>Wavelength</th>
<th>Wave amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>--</td>
<td>0·c</td>
</tr>
<tr>
<td>4S</td>
<td>0,050·c</td>
<td>0,025·c</td>
</tr>
<tr>
<td>4M</td>
<td>0,050·c</td>
<td>0,050·c</td>
</tr>
<tr>
<td>4L</td>
<td>0,050·c</td>
<td>0,120·c</td>
</tr>
<tr>
<td>8S</td>
<td>0,250·c</td>
<td>0,025·c</td>
</tr>
<tr>
<td>8M</td>
<td>0,250·c</td>
<td>0,050·c</td>
</tr>
<tr>
<td>8L</td>
<td>0,250·c</td>
<td>0,120·c</td>
</tr>
</tbody>
</table>
Figure 1.9 Results of (Custodio, 2007) experiments. In the top row $C_L$ vs. $\alpha$ is plotted, in the central one, $C_D$ vs. $\alpha$ is plotted and in the bottom row, $C_L/C_D$ vs. $\alpha$ is plotted. Data is for 8X family hydrofoils at the left column and for the 4X family at the right one.

As it can be seen, maximum lift is larger for the baseline hydrofoil, but modified ones do not show an important loss in $C_{L\text{max}}$. Models with smaller tubercles have a smaller loss, but stall is more abrupt. Stall behaviour can be smoothed by increasing the number of tubercles (reducing its wavelength). Some comment has to be done on the reference surface used to calculate non-dimensional coefficients. In these experiments, models are considered equivalent when they have the same planform surface. However, for this project purposes,
tubercles will be considered as deployable high-lift devices (as traditional flaps are). This is because an inflatable configuration will be proposed. Thus, when models with tubercles are tested, the reference surface used is that of the clean wing (the original surface without taking into account the tubercles). If this convention had been used, the results in Figure 1.9 would have probably shown an increase in lift due to the enlarged surface.

Finally, in (Stanway, 2008), other static experiments are done with models copied from those in (Miklosovic, et al., 2004). Re values tested are lower: 44.648; 59.530; 89.295 and 119.060. It is found there that, for the model with tubercles, $C_{L_{\text{max}}}$ increases as the linear region of the lift curve becomes longer. At the highest Re tested, $C_{L_{\text{max}}}$ becomes slightly higher to that of (Miklosovic, et al., 2004). For the baseline model, $C_{L_{\text{max}}}$ does not change significantly with Re and stall is deeper than that of the new model. The difference with the results of (Miklosovic, et al., 2004) is explained by the different stall mechanism due to the change in Re. At Stanway’s experiment stall starts at the trailing edge (seen by PIV visualization), while at (Miklosovic, et al., 2004) the lift curve suggests a leading edge bubble burst phenomenon. Figures are not reproduced as copyright rights are reserved for this article by express desire of the author.

1.3.3.2. Practical uses

Humpback whale tubercles have already been used in commercial applications. Technology Development Associates use this technology to produce more efficient rotating-wing devices. They claim that devices using their technology are capable of producing more lift and reduce tip vortices, thus causing less drag, noise and vibrations. At the moment, Technology Development Associates have designed and produced industrial cooling fans (Figure 1.10), wind turbines blades, computer fans and sailboat rudders with humpback-whale-like foils (Technology Development Associates).
1.4. Scope of the study

This study does not pursue the final design of a high-lift device for a specific UAV. It consists of three main parts: experimental assessment of aerodynamic improvements brought by humpback whale tubercles, conceptual design of the application system and production of a numeric design tool. The list of the required tasks is:

- Research of an available and suitable wind tunnel to carry out the experiments.
- Design and construction of the models needed for wind tunnel testing.
- Measurement of the static aerodynamic polars for \( \text{Re}=1.2 \cdot 10^5 \), \( \text{Re}=2.4 \cdot 10^5 \) and \( 3.5 \cdot 10^5 \) of a plain wing and an equivalent wing with rigid tubercles.
- Flow visualization on the model with tubercles at \( \text{Re}=2.4 \cdot 10^5 \) by the China clay technique.
- CAD conceptual design of a inflatable membrane high-lift device for the wing of the reference UAV: Furos (see: Annex A - Furos Datasheet)
- Design and production of an analysis program suitable to calculate the shape of a membrane under ideal flow. It will be based on Panel Method (ideal flow) and FEM. The language used will be FORTRAN 90.

While the code structure shall be documented, no user’s manual will be provided. No viscous CFD computations of the flow over tubercles will be done. This, as it has been previously mentioned, is because of the nature of the boundary layer at the selected Re. No aeroelastic study of the membrane will be done nor will it be tested in the wind tunnel.
1.5. Basic specifications

1.5.1. Wind tunnel measurements

- The error in the measured coefficients will not be more than: 0.075 for lift, 0.015 for aerodynamic moment and 0.035 for drag.

1.5.2. Conceptual design of high-lift device

- It will not have any aerodynamic detrimental effect in cruise conditions.
- No mechanical parts will be needed for deployment.
- No electronic devices will have to manage the device
- Maximum payload (MP/L) will be increased, as capability to fly at higher $C_L$ will overcome the small increase in weight.

1.5.3. Code

- It will be compatible with pre/post-processor GiD v.10 allowing for a user-friendly interface.
- Triangular and quadrilateral elements will be available both for rigid and flexible surfaces
- User will be allowed to choose the position of air intakes and membrane supports.
- User will be allowed to enforce a symmetry condition.
- User will be able to plot pressure coefficient ($C_p$) field, velocity vectors, wake shape and intensity, membrane deformation using the GiD post-processor module.
2. Experimental analysis

Since the beginning of the project, it became apparent that wind tunnel tests would be needed. As an alternative, CFD simulations could have been used in order to study benefits produced by tubercles. However, given the complex flow dynamics involved this approach was not considered practical.

At the Re at which calculations would have been done, the boundary layer could not be considered completely laminar or completely turbulent. Neither defining the line where transition happens was a good idea. This is because an ample transition region was expected and the addition of tubercles would have a major effect on its shape. It would not be sensible to prescribe the position of the transition band because it is a function of the shape of the tubercles. Unless the transition is left free the real effect of the tubercles cannot be assessed.

The Dipartimento di Ingegneria Aerospaziale of the Politecnico di Milano, very kindly allowed the tests I proposed at GVPM.

2.1. Experiment description

The main objective of this experiment is to assess the aerodynamic performance changes produced by humpback whale tubercles. The focus is not on the value of $C_{L,\text{max}}$ or any other characteristic of any model, but in the change they experiment when tubercles are added. To do this, two models where fabricated with the same wing section and planform. Wooden pieces simulating whale tubercles were to be attached to one of them. However, as more time than initially expected was finally available for experiments, only one model was tested: first in the clean configuration and then, with tubercles attached. Furthermore, then model was not detached from the balance when gluing the tubercles. This suppressed two sources of error that could have affected measured performance: model differences and changes in distance to the tunnel walls.

The experiment was conducted at the wind tunnel of the Dipartimento di Ingegneria Aerospaziale located at the Bovisa facilities of the Politecnico di Milano. It is a closed circuit wind tunnel powered by three fans, placed as shown in Figure 2.1, delivering a combined power of 100kW. Its maximum speed at the test section is 55m/s. The test section is 1m wide, 1.5m tall and 3m long (in the direction of the flow). The measured turbulence is 1‰. The wind tunnel was equipped with a centrifugal compressor, shown in Figure 2.2, in order to maintain static pressure in the test section equal to ambient level. This is achieved by injecting air into the low speed part of the tunnel, which was at a higher static
pressure than the outside. To compensate for air losses the compressor works continuously, thus acting as an air exchanger. Therefore, heat exchangers were not needed.

Figure 2.1 Integration of the three fans to the square section of the conducts

Figure 2.2 General picture of the wind tunnel. In the left, the test section is seen. In the middle, there is the green centrifugal compressor. In the right, the fan section.
The scheme of the wind tunnel data acquisition system, depicted in Figure 2.3, shows the sensors used during the experiments. Additionally, a 6 components balance was used. It is not shown in Figure 2.3.

![Wind tunnel data acquisition system diagram](image)

Figure 2.3 Wind tunnel data acquisition system. (1) Differential pressure sensor. (2) Test chamber absolute pressure sensor. (3) Test chamber temperature sensor. (4) Test chamber humidity sensor. Red lines indicate pressure ducts and blue lines signal cables. Balance is not shown.

The differential pressure sensor (1) was used to calculate the dynamic pressure (q) of the flow in the test chamber. This was done by an experimentally determined correlation. This function was very precise because the wind tunnel was, at that moment of the experiments, being certified to calibrate anemometers. The characteristics of the differential pressure sensor are summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Druck LPM 9481</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Number</td>
<td>2377361</td>
</tr>
<tr>
<td>Range</td>
<td>0 to 5000 Pa</td>
</tr>
<tr>
<td>Output</td>
<td>0 to 10 V</td>
</tr>
<tr>
<td>Error</td>
<td>0,1% of the full scale</td>
</tr>
</tbody>
</table>

Table 2.1 Characteristics of the differential pressure sensor (1) (General Electric, 2008)

The test chamber absolute pressure sensor (2) provided feedback information to the control system of the centrifugal compressor while also serving to calculate air density in combination other measurements. This sensor is characterized in Table 2.2.
Test chamber temperature and humidity, provided by sensors (3) and (4) respectively, were used to calculate air density. Combining this value with the dynamic pressure reading the air speed could be obtained. The test chamber temperature sensor was based on a RTD (Resistance Temperature Detector) of the type Pt100. The characteristics are listed in Table 2.3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Pt100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material</td>
<td>Platinum</td>
</tr>
<tr>
<td>Resistance at 0°C</td>
<td>100Ω</td>
</tr>
</tbody>
</table>

Table 2.3 Characteristics of Pt100 Temperature sensor (3) (Ingeco Z.S., 2005)

Humidity was measured with a Rense HT 922-I-01 sensor (4) whose characteristics are presented in Table 2.4.

<table>
<thead>
<tr>
<th>Model</th>
<th>Rense HT 922-I-01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Number</td>
<td>RI201303</td>
</tr>
<tr>
<td>Range</td>
<td>0 to 100% RH</td>
</tr>
<tr>
<td>Output</td>
<td>4 to 20 mA</td>
</tr>
<tr>
<td>Error</td>
<td>±2% RH</td>
</tr>
</tbody>
</table>

Table 2.4 Characteristics of the test chamber humidity sensor (4) (Rense Instruments, 2008)

Forces on the model were measured by an in-house made balance. It consisted on 7 load cells mounted in a compact box where models can be attached. The balance is portrayed in Figure 2.4. It has been designed for high accuracy measurements at high loads. Since small loads were measured, the relative error can be significant in low speed test runs.
Figure 2.4 Balance used in the measurements. Some of the load cells can be seen together with the PC output

Load cells were of the type MB-100, its characteristics are specified in Table 2.5.

<table>
<thead>
<tr>
<th>Model</th>
<th>MB-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Number</td>
<td>D65883 and others</td>
</tr>
<tr>
<td>Range</td>
<td>0 to 100lb (444N)</td>
</tr>
<tr>
<td>Output</td>
<td>0 to 5V</td>
</tr>
<tr>
<td>Non linearity Error</td>
<td>0.03%</td>
</tr>
<tr>
<td>Derive Error</td>
<td>0.025% at 20min</td>
</tr>
</tbody>
</table>

Table 2.5 Characteristics of the load cells used in the balance (Interface - Advanced Force Measurements, 2011)

The load cells were connected to a PC card that supplied power and conditioned cell output. Its settings are shown in Table 2.6. The power supply control system used a feedback cable to measure voltage at the cells. No current was allowed to flow through this wire in order to avoid interference with the load measurements. Current through the cells was measured to obtain a force reading. Signal conditioning consisted on a low pass filter and an amplifier.

<table>
<thead>
<tr>
<th>Model</th>
<th>SCXI 1520</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply tension</td>
<td>5V</td>
</tr>
<tr>
<td>Low Pass cut Freq.</td>
<td>10Hz</td>
</tr>
<tr>
<td>Gain</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 2.6 Settings of the PC card conditioning load cells signals

All sensor outputs were passed through an analogue-to-digital converter that transferred all the information to a computer. Converter is identified in Table 2.7
All the information was processed in the computer by a labVIEW program that calculated derived variables (dynamic pressure, air density, air speed) and wrote text files including: angle of attack (introduced manually from the visual scale), values of 3 forces and 3 moments, dynamic pressure, absolute pressure, air temperature and humidity, air speed and air density. All this information, together with the reference surface was used to calculate aerodynamic coefficients and Re for each measurement.

Tests were conducted as follows. The clean configuration model was tested at \( \mathbf{U}_\infty = 10 \text{m/s} \), \( \mathbf{U}_\infty = 20 \text{m/s} \) and \( \mathbf{U}_\infty = 30 \text{m/s} \) (corresponding to \( \text{Re} = 1.2 \cdot 10^5 \), \( \text{Re} = 2.4 \cdot 10^5 \) and \( \text{Re} = 3.5 \cdot 10^5 \)). For each of these wind speeds, \( \alpha \) was varied from 0º to the stall angle of attack. \( \alpha \) change from measure to measure was of 1º. Three polars were thus obtained. In order to get an idea of the repeatability of these measurements, 13 combinations of \( \text{Re} \) and \( \alpha \) were measured again. This information has been used in “10.2 Annex B – Experimental error assessment” to estimate the experimental errors. Next, for \( \alpha = 15^\circ \), measures were done at speeds ranging from \( \mathbf{U}_\infty = 10 \text{m/s} \) (corresponding to \( \text{Re} = 1.2 \cdot 10^5 \)) to \( \mathbf{U}_\infty = 30 \text{m/s} \) (corresponding to \( \text{Re} = 3.5 \cdot 10^5 \)). Speed increment from measurement to measurement was 2m/s (corresponding to a change in \( \text{Re} \) of \( 2.4 \cdot 10^4 \)). This was done to study the effect of \( \text{Re} \). At \( \alpha = 22^\circ \) the \( \text{Re} \) effect study was repeated.

Later, humpback whale tubercles were attached to the leading edge of the model and all of the aforementioned measurements were repeated. The only difference is that when very high angles-of-attack were explored the increment in \( \alpha \) was larger than 1º.

Finally, for \( \text{Re} = 2.4 \cdot 10^5 \) the china clay flow visualization technique (Barlow, et al., 1999) was used to study the flow at \( \alpha = 14^\circ \), \( \alpha = 16^\circ \), \( \alpha = 18^\circ \) and \( \alpha = 24^\circ \).

Before plotting the model polars, some wind tunnel corrections were applied (Barlow, et al., 1999). In particular, corrections for solid blockage and wake blockage were incorporated.

<table>
<thead>
<tr>
<th>Model</th>
<th>NI PXI-6284</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolution</td>
<td>18 bits</td>
</tr>
<tr>
<td>Maximum Sample Rate</td>
<td>625 kHz</td>
</tr>
<tr>
<td>Used Sample Rate</td>
<td>10kHz</td>
</tr>
<tr>
<td>Sample Time</td>
<td>10s</td>
</tr>
</tbody>
</table>

Table 2.7 Characteristics and settings of the analogue to digital converter (National Instruments, 2010)
2.2. Model and support description and fabrication techniques

2.2.1. Model

As it has been said, two models were fabricated although only one was used in the wind tunnel. Fabrication techniques used considered different factors: price, ease of manufacture, geometric repeatability (ability to produce equally shaped models), surface finish quality and realistic airfoil for UAV models. The latter is the main difference with experiments in existing literature. Wing sections similar to those of the humpback whale had been tested instead of airfoils used by the UAV industry. Planform was, selected to attain the desired Re (high c) while avoiding high bending moments at the wing root (low semispan, b/2) and keeping a reasonable aspect ratio (AR). The design parameters are listed in Table 2.8:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0.2 m</td>
</tr>
<tr>
<td>b/2</td>
<td>0.636 m</td>
</tr>
<tr>
<td>AR</td>
<td>6.36</td>
</tr>
</tbody>
</table>

Table 2.8 Design values for geometric characteristics of the test models.

Once the model was manufactured, it was measured to know precisely the reference surface (see Figure 2.5).

Finally, the measured size of the model was as specified in Table 2.9:
Protuberance size and spacing (relative to the chord) where chosen to match those found in whale flippers. Particular values are displayed in Table 2.10. This dimensions are equivalent to those of the 8L at (Custodio, 2007) experiments.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>0.19374 m</td>
</tr>
<tr>
<td>$b/2$</td>
<td>0.5948 m</td>
</tr>
<tr>
<td>$AR$</td>
<td>6.14</td>
</tr>
</tbody>
</table>

Table 2.9 Geometric characteristics of fabricated models.

To fabricate the models, a ribs and spars concept was selected. It assured good airfoil shape control and stiffness. Ribs were made of wood and were 4mm thick. 4 steel spars were used and had a 12mmx3mm rectangular cross section. The general setup is pictured in Figure 2.6.

![Figure 2.6 Assembly of the ribs and spars.](image)

Four different types of ribs were fabricated, as it is seen in Figure 2.6 and Figure 2.7:

- The rib farthest from the viewer in Figure 2.6 (number 1) is a simple NACA 4415 airfoil.
- The contour of the second rib type is obtained by offsetting 1mm inwards the surface of the NACA 4415 airfoil. Additionally, four 12mmx3mm cuts are made in order to house the steel bars on the airfoil.
• The third type has the same shape as number 2, but a rectangular cavity is cut in the inside.
• The rib closest to the viewer (number 4) has a NACA 4415 airfoil and a circular hole 10mm in diameter at the aerodynamic centre

All the ribs were machined from a single wood panel using a CNC milling machine. It is depicted in Figure 2.7.

In order to fill the empty spaces, polystyrene pieces were also manufactured. They were cut out of a 125cmx60cmx5cm sheet by a numeric control hot wire machine. Polystyrene gave stability to the system and allowed the thin skin to transmit forces to the structure. Figure 2.8. shows the complete assembly.
The empty space visible in Figure 2.8 served to house a prismatic piece of wood that would transfer stresses from the skin through the ribs and the polystyrene filling to a steel round bar with a 10mm diameter used to attach the model to the balance. On top of the polystyrene, a 1mm balsa wood skin was placed. Its function is to cover irregularities produced by spars and ribs. In order to reduce surface roughness the ensemble was covered with an adhesive plastic film.

The assembly process consisted on gluing every polystyrene piece to the rib located on its left side with special polystyrene adhesive as it can be seen in Figure 2.9. Next, the subassemblies were glued together in groups of six to form the base structures for two models. The base structure had a very low stiffness as it can be seen in Figure 2.10. Cylindrical bar and spars were added increasing the overall stiffness of the part. In order to attach the balsa wood skin to the model it had to be previously immersed in water to prevent the leading edge from cracking. Once the balsa wood panel had been curved it was left to dry into its final shape. The result of these processes can be seen in Figure 2.11. Finally, the balsa wood was glued and covered with plastic film to obtain the complete model.
Figure 2.9 Three subassemblies of rib and polystyrene filler and a wooden prismatic piece before and after assembly.

Figure 2.10 Trailing edge of one model showing curvature produced by polystyrene pieces
To produce the tubercles, 4cm x 4cm x 4cm pieces of balsa wood were used. Placing sandpaper on the model leading edge, the contact surface was machined by abrasion. Then, using a box cutter, the wooden cube was given a rounded shape. Surface finish was improved with sandpaper. A picture of the process can be seen in Figure 2.12.

2.2.2. Support

At the wind tunnel facilities, there was no device available to support the model and allow, at the same time, changing incidence without entering the test
chamber. A new system had to be provided which would be screwed to the balance on one side and attached to the bar of the model on the other allowing for relative rotation between them. To accomplish this function, a 15mm steel plate with the necessary holes was welded to a bike part. The part used is the attachment of the telescoping seat to the bicycle frame. The device can be seen in Figure 2.13. The needle seen was used to measure the change in angle of attack. In order to produce an angle scale, a protractor was photocopied as seen in Figure 2.14.

This scale allowed to measure angle of attack difference to the reference position. The reference position was established by placing into the tunnel a reference plane that was swept by laser beams in order to visualize it. It was placed aligned to the air flow and then, the model was aligned with it.
2.3. Results

2.3.1. Coefficients comparison

Next, $C_L$, $C_D$ and $C_{Mac}$ comparison will be shown at $Re=1,2 \cdot 10^5$, $2,4 \cdot 10^5$ and $3,5 \cdot 10^5$. Plots contain data from the baseline model, the model with tubercles and a second test run done with the baseline wing to check the repeatability of the experiment. This additional test run is plotted using red crosses and give visual information on the confidence of the experiment. The procedure to estimate errors using the repeatability test run is explained in Annex B – Experimental error assessment.

2.3.1.1. $Re=1,2 \cdot 10^5$

The error calculated in “10.2 Annex B – Experimental error assessment” for this speed is that expressed in Table 2.11.

<table>
<thead>
<tr>
<th></th>
<th>$C_L$</th>
<th>$C_D$</th>
<th>$C_{Mac}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>$\pm 0.075$</td>
<td>$\pm 0.035$</td>
<td>$\pm 0.015$</td>
</tr>
</tbody>
</table>

Table 2.11 Error calculated for tests at $Re=1,2 \cdot 10^5$
Figure 2.15 $C_L$ comparison at $Re=1.2\cdot10^5$. It has to be remarked the high $\alpha$ at which the wing was tested without any step at lift.

At this Re, as Figure 2.15 shows, the baseline wing presents an abrupt stall. The wing with tubercles was tested up to 70° of incidence without observing any sudden change in $C_L$. As for $C_{L\text{max}}$, a substantial loss in maximum lift is observed. As it will be explained in 2.4 Conclusions drawn from experiments, part of this loss can be attributed to an incorrect tubercle shape design, which causes flow separation near the leading edge.
Figure 2.16 $C_D$ comparison at $Re=1,2 \cdot 10^5$. Only a part of the $\alpha$ range is shown in order to ease visual comparison.

Figure 2.16 shows $C_D$ behaviour. It can be seen that, until the moment when the clean airfoil stalls (at $\alpha=15^\circ$, losing part of its induced drag) the difference in drag is very small compared to the repeatability error.

Figure 2.17 $C_{Mac.}$ comparison at $Re=1,2 \cdot 10^5$.

Aerodynamic moment (plotted in Figure 2.17) has different behaviour at each model. However, for $\alpha=0^\circ$, moment value is the same for both models. At
low angle of attack, aerodynamic moment increases at the wing with tubercles and decreases at the clean one. This makes the clean wing more stable. However, when stall arrives, an additional recovery moment appears at the wing with tubercles and increases with $\alpha$. This makes this wing safer near stall.

For this Re value aerodynamic efficiency ($E$) is always higher for the clean wing.

2.3.1.2. $Re=2.4\cdot10^5$

The errors estimated in “10.2 Annex B – Experimental error assessment” for this speed are presented in Table 2.12.

<table>
<thead>
<tr>
<th>Error</th>
<th>CL</th>
<th>CD</th>
<th>CMac</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>±0.02</td>
<td>±0.0075</td>
<td>±0.0035</td>
</tr>
</tbody>
</table>

Table 2.12 Error calculated for tests at $Re=2.4\cdot10^5$

In Figure 2.18, the baseline wing shows a less abrupt stall than that seen at $Re=1.2\cdot10^5$. In this case, stall is of the trailing edge type, as it will be seen in 2.3.3 Flow visualization. The change in behaviour suggests that at lower Re stall could be of the explosive bubble type (Meseguer Ruiz, et al., 2005). The wing with tubercles shows again a very smooth stall with no sudden drop in lift.
As shown in Figure 2.19, drag once again shows minimal change before stall happens. Then, clean wing stalls and it loses induced drag compared to the model with tubercles.

Aerodynamic moment behaviour, as shown Figure 2.20, is very similar to that encountered at Re=1.2·10^5. In this case, efficiency is also higher for the conventional wing.
2.3.1.3. \( \text{Re}=3,5\cdot10^5 \)

The errors estimated in 10.2 “Annex B – Experimental error assessment” for this speed can be found in Table 2.13.

<table>
<thead>
<tr>
<th></th>
<th>( C_L )</th>
<th>( C_D )</th>
<th>( C_{Mac} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>±0.008</td>
<td>±0.0035</td>
<td>±0.0015</td>
</tr>
</tbody>
</table>

Table 2.13 Error calculated for tests at \( \text{Re}=3,5\cdot10^5 \)

Figure 2.21 shows that, at relatively high \( \text{Re} \), the reference wing shows a smooth stall, although the whale-like wing still displays a more progressive response.
In this case, the effect of the tubercles on drag prior to stall is negligible even though they increase wet surface, see Figure 2.22.

General behaviour of aerodynamic moment presented in Figure 2.23 is similar to that seen at lower Re.
2.3.2. Effect of Re

It is interesting to investigate how Re affects the performance of both wings. This gives insight on the kind of applications humpback whale tubercles could be suitable for.

![Figure 2.24 Lift curve of clean wing as a function of Re](image)

![Figure 2.25 Lift curve of whale wing as a function of Re](image)
Figure 2.24 and Figure 2.25 give information on the changes of the lift curve with Re. For the conventional wing, $C_{L_{\text{max}}}$ decreases slightly with stall becoming less abrupt as Re increases. On the other hand, $C_{L_{\text{max}}}$ for the whale wing increases with Re while the shape of the lift curve remains unchanged.

It is also interesting to check if the effect of the Reynolds number is progressive. There could be a sudden change in behaviour at a critical Re value, as it happens in other phenomena when boundary layer changes from laminar to turbulent. Since Re values used previously are widely spaced (compared to the first Re tested, the second one is double and the third one, triple) additional measurements were taken two specific values of $\alpha$: $\alpha=15^\circ$ (close to $C_{L_{\text{max}}}$) and $\alpha=22^\circ$ (fully stalled).

![Graph of $C_{L(15^\circ)}$ vs. Re](image)

Figure 2.26 Lift at $\alpha=15^\circ$ for different Re values
Figure 2.26 shows how the lift of models set at 15° of incidence changed when Re was progressively increased. The clean wing is stalled for Re=1.2×10^5. This is not consistent with Figure 2.15, where α=15° is below stall. This happens because stall shows a marked hysteresis. For the first curve α was being increased, so the wing started from an attached flow condition. On the other hand, for the Re sweep speed was ramped-up. In this case, the flow starts detached at low speed and recovery is delayed by the hysteresis. It can also be observed that once the flow reattaches the lift of the conventional wing decreases slightly with Re while, on the other hand, the whale wing gains lift.

Figure 2.27 shows how, at 22°, the whale wing does not experiment a large change in lift as Re increases. The clean wing, on the other hand, shows a sudden increase in lift as stall type changes from a steep drop to a progressive reduction (see Figure 2.24).

2.3.3. Flow visualization

The flow around a wing with tubercles is much more complex than that around a conventional wing. For the latter case, a 2D model adapted to account for finite span gives a reasonable description of the major features. In order to get a better understanding of the flow and the process responsible for delaying stall, flow visualization techniques were used. Flow visualization was done by applying China clay on the model with tubercles and testing it at Re=2.4×10^5 and values of α close to the stall: 14°, 16°, 18° and 24°.
Wings were tested in vertical position due to the test chamber proportions: height was larger than width. Some liquid was retained behind the tubercles by surface tension causing the release of drops during the test. Under the effect of gravity, the larger drops drifted downwards. The images below show wide trails curving downwards as they approach the trailing edge. These are attributed to liquid drops affected by gravity (in the pictures the gravity points to the bottom of the page).

2.3.3.1. $\alpha=14^\circ$

Figure 2.28 China clay result at $\alpha=14^\circ$ and $Re=2.4\cdot10^5$. Left image is upper side of the wing and right one is lower side.

Figure 2.28 shows china clay traces for $\alpha=14^\circ$. According to the lift curve, this point is on the area where the slope is still positive but smaller than in the linear region. This is consistent with the image showing how the rear part of the upper side has regions of reversed flow. This is the onset of a trailing edge type stall. Observing the lower side, it is apparent that near the tips there is an
increased spanwise velocity component, which is responsible for the development of the wing tip vortex.

The close-up view of Figure 2.29 shows that, aft of a tubercle, drops are not entrained downstream. This suggests a low shear stress region where flow has detached. Immediately behind the detached region, there is an area where little red dye remains. This is because it was swept by a high shear stress before the liquid evaporated.

There is a spanwise change in flow properties over a characteristic distance equal to the tubercle spacing. Figure 2.28 is consistent with this as reverse flow regions have their peaks just behind the valleys between tubercles.
Figure 2.30, again, shows how air velocity changes spanwise. Here, the lower side of the wing is seen through a glass window while tunnel is running. Regions behind a tubercle dry first which means that they experience a higher shear stress.
2.3.3.2. $\alpha=16^\circ$

Figure 2.31 China clay result at $\alpha=16^\circ$ and $Re=2.4\cdot10^5$. Left image is upper side of the wing and right one is lower side.

The general look of Figure 2.31 is the same as that for $\alpha=14^\circ$. The change observed is that small regions of reverse flow have coalesced into larger features. According to lift curve $\alpha=16^\circ$ corresponds to $C_{L_{\text{max}}}$. 
Figure 2.32 China clay result at $\alpha=18^\circ$ and $Re=2.4\times10^5$. Left image is top side of the wing and right one is bottom side.

For $\alpha=18^\circ$, Figure 2.32 shows a large reverse flow region has developed over the middle section of the wing. According to the lift curve, for this angle of attack, the wing is completely stalled and the lift curve slope is virtually zero.
For \( \alpha = 24^\circ \), Figure 2.33 shows that the region where reverse flow occurs has grown and now extends all the way to the wing tip. However, \( C_L \) remains above 1. A closer view of wing tip is presented in Figure 2.34.
At the wing tip, air flows from the lower side of the wing to the upper side thus creating the wing tip vortex. Flow detachment is probably caused by wing tip edges that are not rounded in this model. Tests with more realistic wing tips would give information on whether this explanation is correct or it is an effect of tubercles.

2.4. Conclusions drawn from experiments

Figure 2.15, Figure 2.18 and Figure 2.21 show the effect on the lift curve due to the tubercles placed on the wing leading edge. They show that stall becomes remarkably smoother. They also show that $C_{L_{\text{max}}}$ decreases when tubercles are included. As it has been mentioned during the flow visualization analysis, local detachments take place behind the tubercles. This validates the previous experiments shown in 1.3.3.1 Previous studies (Custodio, 2007).
In (Custodio, 2007), 8L model (equivalent to the model tested here) and other models with large tubercles show a loss in $C_{L_{\text{max}}}$ and a much smoother stall. Smaller tubercles presented a $C_{L_{\text{max}}}$ similar to the reference model but degraded stall characteristics. An improvement can be achieved by increasing the number of tubercles. It would be interesting to see to what extend stall can be delayed by reducing tubercle size and wavelength, as it is suggested in (Custodio, 2007) experiments.

The optimum lift curve depends on the intended application. In some cases, a wide $\alpha$ margin to stall has to be used with traditional wings. The $\alpha$ tolerance gained by the progressive stall behaviour might then outweigh the loss in $C_{L_{\text{max}}}$. For other applications, where the angle of attack can be controlled precisely, it may be preferable to increase $C_{L_{\text{max}}}$ regardless of the post-stall characteristics. In any case, observing Figure 1.9, it is apparent that stall behaviour can be controlled by installing leading edge protuberances on the leading edge.

Drag does not experience a substantial increase over the linear region of the lift curve. The combination of drag increase and lift loss causes a reduction in efficiency for cruise conditions.

$C_{Mac}$ indicates that the centre of pressure moves forward when tubercles are installed. This effect becomes more pronounced when $\alpha$ increases. At $\alpha=0^\circ$ it is virtually negligible. Tubercles are very beneficial near stall, as aerodynamic moment has, in this region, a remarkably stable slope. Stability in stall conditions adds to the benign behaviour of lift, making flight in this region very safe.

Flow visualization reveals the mechanism that allows for a smooth post-stall response. Instead of a single region of reverse flow that grows uncontrolled, multiple smaller zones develop. They are separated by the high velocity wakes of tubercles, which delay their growth and coalescence.

### 2.5. Difficulties encountered

The major impediment encountered was lack of availability of a suitable wind tunnel. The Re at which test had to be conducted required a large chord to be tested. Otherwise, the flow could not be considered incompressible. As a 3D effect was being studied, a reasonable AR had to be maintained. This called for a minimum test section size. The wind tunnel provided by Politecnico di Milano fully satisfied this requirement.

The lack of experience building wing models was also an obstacle to overcome. Finding specialized shops and certain materials took a great effort.
Finally, the transition from CAD (where geometry is ideal) to real parts featuring imperfections required a steep learning curve.
3. Application proposal

Humpback tubercles can be installed by adding rigid pieces to the leading edge as their detrimental effects in cruise conditions are small. Tubercles benefits can also be exploited by installing a lightweight deployment system as proposed next. Furthermore, this system does not increase the onboard computer workload and cruise conditions, this device does not change the aerodynamic performance.

Since tubercles are only necessary when $\alpha$ is high, the low-pressure field on the upper side of the leading edge can be used to deploy the device. In order to do this, a thin membrane is attached to the leading edge in such a way that it can inflate. It must be designed so that the inflated shape is appropriate.

In order to allow air into the cavity under the inflated membrane, an intake is provided. It consists on a tube connecting the inside of the membrane to the lower side of the wing. Thus, when the incidence is large, the intake is subject to a high pressure (close to impact pressure) while a low pressure acts on the outside of the membrane (as it is positioned on the suction area). This pressure difference forces air into the cavity thus deploying the membrane. On the other hand, when $\alpha$ is low the situation is reversed and the membrane deflates.

Deploying the protuberances only at high angles of attack has also positive aspects from a stability point of view. At low incidence, the tubercles modify the moment curve slope making it more instable. Therefore, having them undeployed in cruise improves handling characteristics. Near stall conditions, the effect is the opposite. If the moment curve of the wing with deployable tubercles is interpolated the result is as follows. At cruise conditions, $C_{Mac}$ decreases with $\alpha$ as tubercles are not deployed. When the angle of attack range where tubercles deploy $C_{mca}$ would increase rapidly to meet the moment behaviour of the model with tubercles. Here, global stability would be reduced. At high angles of attack, $C_{Mca}$ would decrease again, as it is the behaviour of the wing with tubercles. To sum up, aircraft stability would be augmented at cruise and at high angles of attack. In the intermediate region, it would be reduced. These considerations are of great importance in aircrafts that, because of their radical configuration or their small size, have stability concerns at low speeds.

A conceptual design is shown in Figure 3.1 and Figure 3.2. The wing shown is that of Furos (see: 10.1 Annex A - Furos Datasheet). The membrane and the tube connecting it to the lower side of the wing can be seen in the cutaway drawing.
For each particular application, the shape of the membrane patch, the material and the tube length and diameter must be defined. Patch shape together with material properties and pressure field determine the geometry of the inflated tubercle. Time of deployment is also affected by the length and diameter of the tube, together with its surface finish. This is because, when tubercles inflates or
deflates, air flows through the tube. The pressure drop along the tube slows down movement of the air.

It is important to define the inflation time. It could happen that at high velocity (in cruise flight, for example) a gust is encountered which suddenly increases $\alpha$. This would cause the tubercle to inflate under high dynamic pressure. If a too small safety margin has been provided the membrane could burst. Using a narrow duct, inflation time can be extended beyond the gust duration. Obviously, this would also affect to the inflation time at low speed. If this solution is chosen, care must be taken when the aircraft is brought to a high $C_L$ to allow sufficient time for inflation.

Another case where the velocity of deployment is important is takeoff. In order to take off with the tubercles inflated there are two choices. If they inflate very fast, the short time elapsed before the high-lift devices deploy could be tolerable. The system could also be designed to deflate very slowly. In this case, inflation by external means might be provided.

Additionally, the pressure drop in the pipe provides a form of energy dissipation, which could help in damping detrimental aeroelastic phenomena. However, damping is only provided for modes that produce significant changes in tubercle volume, low frequency modes are more likely to be damped by this mechanism.

This design can be used in new aircraft designs but it is also very suitable to be installed as a modification of already existing UAVs because no changes in configuration or flight computer are required.
4. FEMPAN

The equilibrium position of a membrane inflated by the aerodynamic pressure of the air flowing around does not have an analytical solution. This is why a programme has been developed to numerically simulate this phenomenon. This programme has been written in FORTRAN 95. It uses at the same time the Panel Method, to calculate the pressure field around the wing and tubercles, and the Finite Element Method (FEM), to calculate the forces and deformation of the inflatable region. The code is called FEMPAN and can also work as a simple Panel Method program. To do this, at the solver menu, “Only Panel Method” has to be selected. The FEMPAN source code is annexed in Annex E – FEMPAN Code Source.

The deformation phenomenon is not expected to be linear because external forces (pressure) changes with membrane inflation and geometry change is large enough to modify stiffness. If a planar membrane is imagined, it is clear that it has no stiffness in its perpendicular direction: it has to change its shape to increase its stiffness. Thus, the typical FEM procedure cannot be used. In other words, calculating at the beginning a stiffness matrix and inverting it to calculate the equilibrium position is not practical.

Alternatively, the method used to find the equilibrium position is to approximate the dynamic evolution from the undeformed configuration to the equilibrium position. Thus, stiffness is recalculated at every time step. From the unbalance between internal and external forces, accelerations are calculated allowing to follow the natural inflation process.

FEMPAN uses a piece of open source code to reorder nodes and make it more efficient. This code, available online is annexed to be used together with FEMPAN in Annex F – Nodes Reordering Code (Open Source).

Next, a general description of the programme and the equations used is presented.

4.1. GiD Presentation

GiD is used as pre- and post-processor for FEMPAN. GiD is a programme developed by International Centre for Numerical Methods in Engineering (CIMNE). It is used by CIMNE codes and can be adapted to work with additional solvers.

GiD has a Computer Assisted Design (CAD) module allowing the creation of different geometries of study. Alternatively, it can import 3D files saved in
popular CAD formats. Once geometry has been introduced into GiD, it can be prepared for the calculations. This includes mesh generation, materials and conditions (such as boundary conditions) application on points, lines or surfaces. Additionally, other numeric information and options can be included.

Figure 4.1 Diagram of the workflow of GiD together with the solver programme. Source: (CIMNE, 2002)

Once the pre-process is concluded, GiD creates a file with all the information needed by the solver and gives it the order to run. Once the simulation is finished, GiD reads the files created analysis code.

The post-process module offers different ways of plotting the results provided by the simulation programme. The complete loop can be seen in Figure 4.1.

In order to adapt GiD to a new solver, problemtypes are created. Problemtypes consist of a folder that includes some files that, once installed in the computer are accessible by GiD. The problemtype files are annexed in Annex D - Problemtype used by FEMPAN. The files in the problemtype folder are:

- The executable code of the solver.
- Other files with information on the kind of conditions, materials and options that have to be used
- A file describing the format to write the input file with all the previously mentioned information and the mesh.

During the programme, no hypothesis is done on the shape of the mesh or the numbering of nodes and elements. Thus, other pre- and post-processors can be used by simply changing the input and output subroutines.
4.2. Panel Method Code

In order to calculate the pressure field around the aerodynamic body where a membrane is installed, ideal flow is assumed. The Re values at which this membranes are expected to fly is around Re=5·10^5. At those Re values, the nature of boundary layer is very difficult to predict, so other techniques including boundary layer simulation would not give an accurate result. Additionally, shear stress is very small compared with pressure differentials and would produce little change in the inflated shape of the membrane. This can be proved imagining a situation where shear stress coefficient is larger. To estimate the shear stress order of magnitude, plane plate case is used.

Let us assume laminar boundary layer, and flight at Re=10^5, local Reynolds number (Re_x) at 0,01·c of the leading edge would be Re_x=10^3. There, laminar shear stress coefficient would be of the order of 0,02. Shear stress is, at least, two orders of magnitude smaller than pressure and can be ignored.

Viscous friction is very important to calculate aerodynamic drag, as it is the main effect at low angle of attack, but it has very little effect on the shape of the inflated membrane.

The Panel Method has been preferred to other CFD techniques as it uses less memory and it can share its mesh with that of the membrane.

4.2.1. General Description and mathematical background

It is well known that ideal incompressible flow can be modelled by enforcing null vorticity and mass conservation. The first condition is assured when velocity is redefined as the gradient of a potential function. Mass conservation at permanent regime is enforced by Equation 4.1.

\[ \nabla^2 \phi = 0 \]

Equation 4.1 Potential equation for ideal incompressible flow

The boundary condition that applies for this equation is zero normal velocity on solid boundaries. Since viscosity effects are not accounted for, boundary layer is not modelled and velocity cannot be set to be zero on the body walls. In lifting bodies, Kutta condition also has to be used. A potential discontinuity appears on the wake caused by circulation.

The Panel Method is based on the fact that the solution to Equation 4.1 is the potential field generated by a doublet distribution on the body surface, the potential field of a freestream and a doublet distribution on the wake summed altogether (Katz, et al., 1991). Additionally, a source distribution can be freely added depending on the boundary condition applied to the flow inside of the solid.
(which is not physical but remains admissible from the mathematical point of view) (Johnson, 1980). This source distribution is used to speed-up convergence as it helps modelling the body thickness.

From an intuitive point of view, the doublet distribution creates a discontinuity on perturbation potential and on the component of the velocity tangent to the surface and source velocity creates a discontinuity in normal velocity.

In FEMPAN, internal flow is enforced to be equal to the freestream. This yields a source distribution on the body wall equal to the normal component of the freestream velocity (Johnson, 1980). If the source intensity is calculated as indicated and the velocity on the inner side is that of the freestream, the velocity at the outer side of the panels becomes parallel to the surface. The doublet and source distributions are approximated by a mesh of panels of constant intensity. Panels can be triangular or quadrilateral.

In order to find out the doublet intensity on each of the panels, the boundary condition is used. Boundary condition is enforced on control points. Each control point provides an equation. Since each panel has a control point in its centre, there is one equation for each degree of freedom. There are two possible ways of enforcing the boundary condition. The first one consists on making the potential equal to that of the freestream in the inside of the body. Alternatively, it can be directly imposed that the perpendicular component of the velocity at the outer side of the control point is zero. In FEMPAN, the second one is used as it makes the extension to a transient panel solver easier.

For each control point, one equation like Equation 4.2 is computed. $A_{ij}$ (called influence coefficient) is the perpendicular velocity caused by the unitary intensity doublet of panel ‘j’ on the control point ‘i’ (if panel ‘j’ is on the trailing edge, $A_{ij}$ includes the effect of the vortices shed). $\mu_j$ is the intensity of the doublet of panel ‘j’, $B_{ij}$ is the perpendicular velocity caused by the unitary intensity source of panel ‘j’ on the control point ‘i’. $\sigma_j$ is the intensity of the source of panel ‘j’ and $\hat{n}_i$ is the normal unitary vector on panel ‘i’.

$$\sum_{j=1}^{N} A_{ij} \mu_j + \sum_{j=1}^{N} B_{ij} \sigma_j + \vec{U}_\infty \cdot \hat{n}_i = 0$$

Equation 4.2 Panel Method equation

Since $\sigma_j$ and $\vec{U}_\infty$ are known, the set of Equation 4.2 becomes the linear system in Equation 4.3.
\[
\begin{bmatrix}
A_{ij}
\end{bmatrix}
\begin{bmatrix}
\mu_j
\end{bmatrix} = \begin{bmatrix}
RHS
\end{bmatrix}
\]

**Equation 4.3 Linear system of the Panel Method**

Once the \( \mu_j \) vector is known, the velocity at any point of the fluid domain can be calculated. Bernoulli equation can be used to calculate the pressure.

In this case, this system has not been used as is but some considerations have been done. First, Equation 4.3 has been divided by \( U_\infty \) and Bernoulli equation by \( \frac{1}{2} \rho U_\infty^2 \) so that the result is the \( C_p \), which is independent of the flight velocity. Secondly, since the linear system is solved by an iterative algorithm that has to perform several matrix-vector products, the system has been transposed as shown in Equation 4.4. Finally, the matrix is conditioned, because the original one is singular.

\[
\begin{bmatrix}
\mu_j
\end{bmatrix}
\begin{bmatrix}
A_{ij}
\end{bmatrix} = \begin{bmatrix}
RHS
\end{bmatrix}
\]

**Equation 4.4 Transposed equations system**

A doublet distribution is always equivalent to a lower order vortex distribution. In constant intensity doublet panels, this means that the effect of the doublet panel is that of a vortex ring, as it is shown in Figure 4.2

![Figure 4.2 A constant intensity doublet panel is equivalent to a vortex ring. Source: (Katz, et al., 1991)](image-url)
Therefore, if the velocity is calculated as the sum of effects of all the panels it has large fluctuations near the surface (it is large near the edge of the panel and smaller near to the centre). This is a consequence of using a mesh with constant doublet intensity panels instead of using a continuous doublet distribution. This has a great effect when calculating the pressure over the body surface.

To avoid this, an equivalent doublet distribution is calculated. The latter is found on nodes. From the nodal doublet values, the doublet intensity can be interpolated within the panels using typical FEM shape functions. This technique assures distribution continuity on the surface. The nodal distribution is enforced to be equal to the original one in the weak form. As it is seen in Equation 4.5 where the test functions \( w_i \) used are the shape functions of each node. The matrix found is the mass matrix used in FEM.

\[
\mu_{\text{elem}} = \sum_{\text{nodes}} N_{\text{node}}(x,y) \mu_{\text{node}} \Rightarrow \int_{\text{mesh}} w_j \mu_{\text{elem}} dS = \sum_{\text{nodes}_\text{mesh}} \int_{\text{nodes}} w_j N_{\text{node}}(x,y) dS \mu_{\text{node}} = M_{ij} \mu_j
\]

Equation 4.5 Strong and weak form of doublet distribution equality

Once the nodal solution is found, it is differenced over the panels using typical FEM techniques (shape functions and isoparametry). As the doublet is a potential discontinuity, its derivative yields the tangential velocity discontinuity. Adding the discontinuity to that caused by the source to the perpendicular velocity and to the freestream velocity a good approximation of the local velocity is found. Pressure is calculated using the non-dimensional form of Bernoulli equation shown in Equation 4.6.

\[
C_p = 1 - \frac{v^2}{U_\infty^2} = 1 - \tilde{v}^2
\]

Equation 4.6 Non-dimensional form of the Bernoulli equation.

### 4.2.2. GiD Interface

When launching FEMPAN from GiD, some conditions and data have to be introduced for the solver to use. Next, this is explained.

Once the geometry is loaded in GiD (it may have been imported or created directly in GiD), conditions are applied. It is recommended to apply conditions to the geometry instead of the mesh in case the latter is changed.

The first condition is the Kutta condition. The line or lines that are part of the trailing edge have to be selected. Then, symmetry can be enforced on the model. This means that only a half of it is meshed and the solver considers the second half. To enforce symmetry, the symmetry condition has to be applied to
some curves that define the symmetry plane. Finally, the airfoil condition can also be selected. Airfoil condition is used to create new reference system, new body axes. Body axes use the chord direction as the X axe and the perpendicular to the symmetry plane as the Z axe. This is useful if geometry is not correctly oriented because the velocity direction is calculated by making two rotations from these axes. If no symmetry condition is used or no airfoil condition is selected, body axes are considered those of GiD.

Because the easiest way of creating wings in GiD is to extrude a curve in the Z direction, body axes in FEMPAN and Z in the span direction.

![Figure 4.3 Body axes used in FEMPAN](image)

Additionally to the conditions, some data have to be introduced. In particular, angle of attack ($\alpha$), sideslip angle ($\beta$) and reference surface are asked. There is also a selection box asking if body axes have to be calculated from the airfoil and symmetry plane or be simply left as they are.

As an output, FEMPAN gives the $C_p$ on the Gauss points of the elements (one for triangular panels and four for quadrilateral ones), velocity vector on panels centres, $C_L$ and $C_D$ on their application point, the shape $a$ of the wake, the intensity of its vortices. FEMPAN also outputs the source and doublet intensities of each panel and the interpolated doublet distribution.

### 4.2.3. Flow chart

The general scheme of the algorithm is very simple and it is represented at Figure 4.4. It starts with reading the input file to store all its information in the memory. Then, it performs some previous calculations and finally doing the proper aerodynamic calculations.

Previous calculations include a reorganization of the mesh where nodes not linked to any element are suppressed and the nodes numeration is changed. This is done to make the mass matrix use the less possible memory, as it is stored in skyline format. Next symmetry plane, chord and body axes are calculated if needed. Memory is allocated to contain all the intermediate
information and the mass matrix is computed. Jacobian matrix for all the elements, their determinants and inverses are also calculated in this part of the programme. Finally, additional nodes are created for the wake.

The panels subroutine starts filling the matrix of the equations and its right-hand-side (RHS). If a symmetric problem is being calculated, doublet intensity of a panel has to be the same to that of its symmetric panel. Then, when the influence coefficients of one panel are calculated, the effect of its symmetric panel is also computed. Additionally, when the influence coefficients matrix and the RHS are being calculated and they had already computed previously, FEMPAN reuses data if possible. For example, the influence coefficient of one panel that does not move on another unmoved panel is recycled.

![Figure 4.4 Panel calculations flow chart](image-url)

Next, the equations system is conditioned. Conditioning is necessary because matrix in this problem is singular as the sum of a row (a column in the transposed system) is always zero. It is because, if all the doublet intensity were
the same, there would be no vortices and no velocity would be induced. Thus, no perpendicular velocity would be induced at any panel and the sum of influence coefficient times the panel intensity would be null. As the panel intensity is the same for all of them in this case, the sum of a row of the matrix is null, which means that its determinant is null. The same conclusion is reached by thinking on doublets as potential discontinuities. A constant potential can be added to the complete fluid domain and the solution is still valid. An additional constraint is enforced. In this case, the sum of the panel intensities has to be null. In terms of the matrix, the new condition consists of a row of ones and a zero at the RHS. In order not to remove any previous equation, this row is summed (multiplied by a factor of the order of magnitude of the row), to every matrix row.

Once the system is conditioned, an iterative algorithm is used to solve the equations. The algorithm used is the bi-conjugate gradient stabilized (Barrett, et al., 1994). Iterative solvers are especially suitable for large equation systems as direct solver computation time increases with $N^3$. Additionally, as it will be seen next, the same system with small changes has to be solved several times. Using the previous solution as initial guess dramatically increases the mean solving velocity.

Before pressures are computed, the geometry of the wake is found. To do this, after the first panel calculation (done with straight wake vortices), the wake is replaced to make vortices parallel to the local air velocity. Otherwise, aerodynamic forces would appear and vortices would end in the equilibrium position. Influence coefficients of panels related to the wake are recalculated and the new linear system is solved. This process is repeated until the first error of the solver is under a certain tolerance.

When the wake geometry is known and the elemental distribution of doublets and sources is known, the pressures are calculated by interpolating the doublet distribution and deriving it.

Finally, the non-dimensional forces are calculated by integrating the $C_p$ field over the surface using Gauss quadrature (one point for triangular panels and 4 for quadrilateral ones).

4.3. FEM Code

The FEM part of the code uses the pressure field to calculate the nodal loads on the membrane and its equilibrium position. Typical stiffness and mass matrices are calculated to solve Equation 4.7.
The reason of solving the problem in this way is the high non-linearity of the problem. In other words, \([K]\) changes with displacement even though material constitutive law is considered linear because big displacements and rotation occur. Therefore, the alternative method of calculating the stiffness matrix and inverting it in order to find the equilibrium solution would have also required an iterative process. Additionally this method allows an easy upgrade: if non-stationary pressures are calculated, it allows studying the dynamics of the phenomenon. Furthermore, as there is no need to invert the stiffness matrix, it does not have to be calculated. Stiffness forces can be calculated calculating strain, then stress and, finally, nodal forces. This avoids extra calculations since strain and stress would be calculated any way.

4.3.1. General Description and mathematical background

The membrane is approximated using two types of 2D elements: three-node triangles (T3) and four-node quadrilaterals (Q4). On them, the weak form of the Second Newton’s Law is used as differential equation to simulate its movement yielding Equation 4.7. The mathematical deduction of that equation follows.

\[
\rho \dddot{\mathbf{x}} = \nabla \cdot \mathbf{\sigma}
\]

Equation 4.8 Newton’s second law for an infinitesimal volume. All terms are divided by the volume

Equation 4.8 describes the behaviour of an infinitesimal volume contained in a continuous medium. \(\mathbf{\sigma}\) is the matrix of surface loads. If a virtual displacement field, which must respect the boundary conditions of the problem (zero displacement at the subjections), is assumed, the work by unity of volume is as displayed in Equation 4.9.

\[
\delta \mathbf{u} \cdot \left( \nabla \cdot \mathbf{\sigma} \right) = \delta \mathbf{u} \cdot \left( \rho \dddot{\mathbf{x}} \right)
\]

Equation 4.9 Virtual work by unity of volume

Integrating Equation 4.9 over the complete volume where it applies Equation 4.10 yields.

\[
\int_{\Omega} \delta \mathbf{u} \cdot \left( \nabla \cdot \mathbf{\sigma} - \rho \dddot{\mathbf{x}} \right) d\Omega = 0
\]

Equation 4.10 Principle of Virtual Work
Integrating it by parts, Equation 4.11 is obtained.

\[
\int_{\text{surface}} \delta \mathbf{u} \cdot (\mathbf{\sigma} \cdot \mathbf{n}) dS - \int_{\text{Volume}} \nabla \delta \mathbf{u} : \mathbf{\sigma} d\Omega - \int_{\text{Volume}} \delta \mathbf{u} \cdot \mathbf{\rho} \mathbf{\ddot{d}} d\Omega = 0
\]

Equation 4.11 Decomposed form of PVW

In the second integral of the Equation 4.11, the gradient of the displacement vector is decomposed in its symmetric and its antisymmetric parts. The symmetric part is the definition of the strain tensor and the antisymmetric part has no effect, as the contract product of it with the stress tensor (which is symmetric) is null. Additionally, the surface integral can be limited to the surface where movement is allowed because, elsewhere, the virtual displacement field is null. Thus, Equation 4.12 is found.

\[
\int_{\text{freesurface}} \delta \mathbf{u} \cdot \mathbf{i} dS - \int_{\text{Volume}} \delta \mathbf{u} \cdot \mathbf{\rho} \mathbf{\ddot{d}} d\Omega = \int_{\text{Volume}} \delta \mathbf{\varepsilon} : \mathbf{\sigma} d\Omega
\]

Equation 4.12 Reordered PVW

One equation is needed for every degree of freedom of the system. Therefore, different virtual displacement fields are used. The virtual displacements used are those caused by the unitary displacement of one node of the system in one of its degrees of freedom. Within the elements, shape functions are used to calculate the local displacement.

Here the element type and the constitutive law are used. For example, for a T3 and a linear medium submitted to a plane state of tensions with no volumetric forces, the set of FEM typical equations displayed in Equation 4.13 are used (Belytschko, et al., 2007).
\[
\begin{align*}
\Phi^{\text{elem}} &= 
\begin{pmatrix}
 u_1^{\text{elem}} \\
 v_1^{\text{elem}} \\
 u_2^{\text{elem}} \\
 v_2^{\text{elem}} \\
 u_3^{\text{elem}} \\
 v_3^{\text{elem}}
\end{pmatrix} \\
[B] &= 
\begin{pmatrix}
 N_{1,x} & 0 & N_{2,x} & 0 & N_{3,x} & 0 \\
 0 & N_{1,y} & 0 & N_{2,y} & 0 & N_{3,y} \\
 N_{1,y} & N_{1,x} & N_{2,y} & N_{2,x} & N_{3,y} & N_{3,x}
\end{pmatrix} \\
\varepsilon^{\text{elem}} &= 
\begin{pmatrix}
 \varepsilon_x^{\text{elem}} \\
 \varepsilon_y^{\text{elem}} \\
 2 \cdot \varepsilon_{xy}^{\text{elem}}
\end{pmatrix} = 
\begin{pmatrix}
 u_x^{\text{elem}} \\
 v_y^{\text{elem}} \\
 u_y^{\text{elem}} + v_x^{\text{elem}}
\end{pmatrix} = [B] \cdot \Phi^{\text{elem}} \\
\sigma^{\text{elem}} &= 
\begin{pmatrix}
 \sigma_x^{\text{elem}} \\
 \sigma_y^{\text{elem}} \\
 \tau_{xy}^{\text{elem}}
\end{pmatrix} = \frac{E_m}{1 - \nu^2} \begin{pmatrix}
 1 & \nu & 0 \\
 \nu & 1 & 0 \\
 0 & 0 & (1 - \nu)/2
\end{pmatrix} \cdot \varepsilon^{\text{elem}} = [D] \cdot \varepsilon^{\text{elem}} = [D][B] \Phi^{\text{elem}} \\
\sigma : \varepsilon &= \begin{pmatrix}
 \delta \varepsilon^{\text{elem}}^T \\
 \sigma^{\text{elem}} \\
 -\delta \Phi^{\text{elem}}^T
\end{pmatrix}^T = \begin{pmatrix}
 \sigma^{\text{elem}} \\
 -\delta \Phi^{\text{elem}}^T
\end{pmatrix}^T = [B]^T[D][B] \Phi^{\text{elem}}
\end{align*}
\]

Equation 4.13 Typical FEM definitions on elements

Integrals in Equation 4.12 are done element by element using Gauss quadrature. This yields Equation 4.14.

\[
\sum_{\text{elem} \in \text{elemsurf}} \int \delta(\Phi^{\text{elem}})^T \begin{pmatrix}
 N_1 \\
 N_2 \\
 N_3
\end{pmatrix} \cdot t dS - \rho \sum_{\text{elem} \in \text{elemvol}} \int \delta(\Phi^{\text{elem}})^T \begin{pmatrix}
 N_1 \\
 N_2 \\
 N_3
\end{pmatrix} (N_1, N_2, N_3) (\Phi^{\text{elem}})^T d\Omega = \\
= \sum_{\text{elem} \in \text{elemvol}} \int \delta(\Phi^{\text{elem}})^T [B]^T [D][B] (\Phi^{\text{elem}})^T d\Omega
\]

Equation 4.14 Elemental form of governing equations

Since virtual nodal displacement is multiplied to all the terms, it can be removed. Additionally, the surface integral will only be computed on the external surface of the body. This is because the sum of internal loads is null, as it is deduced from Newton’s third law.
Equation 4.15 Elemental form of the equations

When the sum for all elements is done, nodal numeration has to be taken into account. In Equation 4.14 and Equation 4.15, matrix indices were those of the node in the element numeration. However, when the sum is done, general node numeration has to be used. The final look of the governing equations is displayed in Equation 4.16.

\[
\sum_{\text{elements external surfaces}} \int \left( \begin{array}{c}
N_1 \\
N_2 \\
N_3 
\end{array} \right) \cdot \mathbf{t} \, dS - \rho \sum_{\text{elements}} \int \left( \begin{array}{ccc}
N_1 N_1 & N_1 N_2 & N_1 N_3 \\
N_2 N_1 & N_2 N_2 & N_2 N_3 \\
N_3 N_1 & N_3 N_2 & N_3 N_3 
\end{array} \right) \mathbf{d} \Omega (\Phi_{\text{elem}}) =
\]

\[
= \sum_{\text{elements}} \int [\mathbf{B}^T \mathbf{D} \mathbf{B}] \mathbf{d} \Omega (\Phi_{\text{elem}})
\]

Equation 4.16 Final equations

As it can be seen, stiffness matrix is not assembled. Since the nodal displacements are known, at each iteration stiffness forces are computed.

Since the equilibrium position is sought, some additional approximation can be done at the transient part. Even though mass matrix is symmetric and non-zero values are very close to the diagonal (which allows to fast iterative solving), it takes a lot of time to calculate acceleration from the sum of forces. To avoid that a diagonal mass matrix is calculated which is much faster to invert. Diagonal terms are calculated as the sum of the terms of its row. This is very reasonable because most of the inertial effects of the movement of one node take place in that node. Therefore, diagonal terms are larger than the rest.

Additionally to these equations, a viscous force is used in order to damp all the vibration modes. This is done using Rayleigh damping (Flores, et al., 2011). This new force is proportional to the nodal velocity and the proportionality matrix is calculated from the mass matrix and the stiffness matrix as expressed in Equation 4.17.
\[
\overrightarrow{F_{\text{damp}}} = \left( \alpha_d [M] + \beta_d[K] \right) \Phi = [C] \dot{\Phi}
\]

**Equation 4.17 Damping force**

The \( \alpha_d \) term, is equivalent to having a viscous fluid through which nodes move. It damps low frequency modes. The \( \beta_d \) term, is equivalent to having some energy dissipation in the material due to its viscosity. It damps high frequency modes. As it will be seen later, the time step used to iteratively find the equilibrium position, is proportional to the period of the highest vibration mode. Therefore, damping high frequency modes (and, therefore augmenting its frequency) has detrimental effects on the computation time, as it requires more iterations.

The damping factor caused by Rayleigh damping terms on modes depend on its frequency as stated in Equation 4.18.

\[
\xi_{\alpha} = \frac{\alpha}{2\omega} \\
\xi_{\beta} = \frac{\beta\omega}{2}
\]

**Equation 4.18 Damping factor of Rayleigh damping terms**

\( \beta_d \) is calculated by the programme. It estimates the highest frequency mode and applies a 0.1 damping factor at it. This is because the highest frequency mode depends upon the size of the smallest element used. It is not practical to estimate it by hand every remesh done. However, \( \alpha_d \) has to be introduced by the user because low frequency modes depend on the geometry instead of the mesh. Overdamping should be avoided as it slows the process, so an initial estimation of the lower natural frequency would yield an optimum \( \alpha_d \).

External loads are calculated using the pressure difference between the outside of the membrane and the inside. Inside pressure is considered equal to that at the node selected as intake.

### 4.3.2. GiD Interface

GiD asks some information for the solver to use related to the FEM. First, membrane elements have to be marked with the movable condition. The intake node also has to be selected. Obviously, it can be done on the geometry instead of the mesh: the movable condition applies to surfaces and the intake one, to one point. When the body surface is meshed, nodes are placed on points detected by GiD. However, those nodes are not necessarily connected to any element. To make sure the intake node is part of the mesh GiD has to be obliged to do it. This
is done at the mesh menu as follows: Mesh > Meshing Criteria > Force points to > Surface-Mesh.

Some other information is asked at the Data menu such as membrane thickness, membrane Young modulus, membrane Poisson ration, dynamic pressure of flight and $\alpha_d$.

As output, the solver gives the displacement vector of each node, the stress and strain matrixes at Gauss points (1 for T3 and 4 for Q4), and nodal forces caused by stiffness and pressure.

### 4.3.3. Flow chart

The FEM part of FEMPAN, starts doing some previous calculations including the computation of nodes that are allowed to move. These nodes are not in contact with any fixed panel, all the panels it has around have the condition movable. Then it verifies that none of those nodes is placed on the trailing edge. Next, the initial position of the node is stored in an array so that the displacement vector can be computed at any time. $[D]$ matrix and $[B']$ matrixes are calculated (jacobians had already been calculated at the panels part of the code). Then the original position of the node in local coordinates is also stored, this eases the calculation of strain as it is done in the XY plane of the element. Next, from the mass matrix calculated for the interpolation, the mass matrix of the dynamic system is calculated. Only the mass of the movable nodes is accounted for.

There are two more subroutines using the FEM. One uses the pressure field provided by the panels part to calculate the external loads on the membrane. The other one uses the nodal displacements and velocities to calculate strain, stress, stiffness forces and damping forces. Every time stresses are calculated, compressive ones are eliminated, as they cannot be there: thin membranes deformate immediately under them. These subroutines are used by the integrator in order to iteratively find the static solution.

### 4.4. Interaction between parts: integrator

As it has been said, the equilibrium position is calculated by iterating an approximation of the dynamic process until the equilibrium is reached. The intermediate process is an approximation because the pressure field calculation does not take into account the movement of parts of the body. Furthermore, pressure field is only updated once the membrane has reached the equilibrium position caused by the previous one. This allows a fast convergence because time used to update pressure is very long compared to the time used in the rest of operations. This approximation has no effect on the result output by the solver.
because only the equilibrium position is plotted. Figure 4.5 displays the general flow chart of the integrator to find the equilibrium position.

\[ \frac{df}{dt}(t) \approx \frac{f(t + \frac{\Delta t}{2}) - f(t - \frac{\Delta t}{2})}{\Delta t} \]

**Equation 4.19 Second order approximation for time derivate**

Equation 4.19 is used to update nodal velocities from known accelerations and to update nodal positions from known velocities. There is one case where it is not used. This is the first calculation of velocity. In that case, the known velocity is known at the same moment as acceleration and then first order approximation of Equation 4.20 is used.

\[ \frac{df}{dt}(t) = \frac{f(t + \Delta t) - f(t)}{\Delta t} \]

**Equation 4.20 First order approximation for the time derivate**

As it can be seen in Figure 4.6, time is discretised in two different series, one where displacement and acceleration are known (top and bottom lines in
Figure 4.6), and another one where velocities are known (middle line). Distance between points of each series is equal to the time step and distance between points of the two series equals a half of the time step.

The integrator starts by calculating the acceleration where displacement and velocity are zero \((i=0)\) using FEM equations. The first velocity is computed using Equation 4.20. Then this velocity is used to calculate the displacement at \((i=1)\). At this point, FEM equations are used to recalculate nodal accelerations. Every time a new position is calculated for nodes, it is checked whether they are at the inside of the wing. At the reference condition, the membrane lies on the wing structure beneath to prevent it from reversely inflate at low angles of attack. This process is repeated until the equilibrium is reached.

To find out if equilibrium has been reached, a reference velocity is calculated. The residual velocity of nodes committed is compared to this reference velocity. The error is defined as the square root of the sum of the square of the nodal velocity components for all the movable nodes. On the other hand, the reference velocity is calculated as the square root of the reference work divided by the mass of the membrane. The reference work is defined as the sum of the dot product of nodal load and nodal displacement for all nodes.

Two remarks have to be done. In order to calculate the damping forces, the velocity used has to be that of the correct moment, not the one calculated half time step before. Otherwise, high frequency modes can get energy from damping instead of dissipating it. Secondly, time step cannot be made arbitrary large. This would cause model instability. The reason is very simple. Let us imagine a mass-spring system, which is out of the equilibrium, and has to be modelled by numerical integration. The acceleration is calculated by Newton’s second law, then velocity after half time step is deduced. Finally, the position after one time step is calculated. If the time step used is too large, spring forces are not recalculated and initial acceleration is supposed to be constant. This makes the...
mass travel to far away and ending up even further to the equilibrium point at the other side. Obviously, after the next time step, it will be even further at the first side and so on: the integrator is unstable. The time step has to be made suitable for the highest frequency mode of the system. The upper bound is presented in Equation 4.21 (Flores, et al., 2011).

\[ \Delta t \leq \frac{2}{\omega_{\text{max}}} \left( \sqrt{1 - \xi^2} - \xi \right) \]

**Equation 4.21 Maximum time step usable**

The highest resonance frequency of the system will be smaller to the highest resonance frequency of elements. This elemental mode frequency is that of the dilatational mode of the smallest element. So a smaller upper bound is shown in Equation 4.22 where \( L_e \) is the characteristic length of the element (the smaller is chosen to be conservative) and \( c_d \) is the velocity of the dilatational wave (Flores, et al., 2011).

\[ \Delta t \leq \frac{L_e}{c_d} \left( \sqrt{1 - \xi^2} - \xi \right) \]

**Equation 4.22 Maximum time step calculated**

For a thin membrane, the velocity of the dilatational wave is as displayed in Equation 4.23 (Flores, et al., 2011).

\[ c_d = \sqrt{\frac{E}{\rho(1 - v^2)}} \]

**Equation 4.23 Dilatational wave velocity**

Membrane density seems to be needed, but it has not influence on the equilibrium position. However, if equations of movement of a damped mass-string system are examined, it will be seen that if non-dimensional time defined as in Equation 4.24 is of great importance. The equilibrium is reached at \( \hat{t} \) which is function of the damping factor.

\[ \hat{t} = t \sqrt{\frac{K}{M}} \]

**Equation 4.24**

Thus, heavy systems take more time to reach the equilibrium but allowed time step is also larger. On the other hand, light systems behave in the opposite manner. In conclusion, the number of time steps to reach the equilibrium is not dependant on the density. Therefore, an arbitrary unity density is used to find the equilibrium position.
4.5. Validation

Once the programme has been explained, its results have to be validated. To validate the proposed code, reference cases that have been calculated by other means are used and results are compared.

4.5.1. Panel Method

FEMPAN is used to calculate the flow around the model used in the wind tunnel. Then lift results will be compared. Drag measured by FEMPAN is only induced one, it does not account for friction drag and therefore they cannot be compared. First, a convergence study is done to find out the number of panels needed to get good quality solutions. To reproduce that case, geometry is introduced in GiD, and symmetry plane is placed 3cm apart of one of the sides, as it was in the wind tunnel tests. The meshes used are structured quadrilaterals on the airfoil and non-structured triangles on the wing tips. Panel density has been augmented on the leading and trailing edges. This kind of mesh is presented in Figure 4.7.

![Figure 4.7 Type of mesh used](image)

The value of $C_L$ at $\alpha=0^\circ$ as a function of the number of divisions of the airfoil converged as it is indicated in Figure 4.8. From the graph, it is considered that, for $N=160$, the solution has converged. With that mesh density, the pressure field obtained and the wake geometry computed are as displayed in Figure 4.9. There the effect of the symmetry plane, placed 3cm off the right wing tip can be observed: The right section of the upper side of the wing is reached by the clear blue region while at the opposite side, it is completely green. However, the effect of spacing is also perceived because of the wake rolling at both sides. The leading edge region is submitted to high pressure.
Once the required density of the mesh is known, the lift is calculated for different $\alpha$ values. It is compared with the tunnel available data in Figure 4.10. Obtained behaviour perfectly matches that of measured lift before stall. However, calculated lift is slightly higher to that measured. Attributing this drift to the low Re used in the wind tunnel is completely reasonable.
At \( \alpha = 15^\circ \), the \( C_p \) has been calculated and it is represented in Figure 4.11. There it can be seen that vortex roll up is more important that at low incidences while the pressure at the leading edge has considerably decreased. The place where the pressure is the lower is at the upper side, just behind the leading edge. Vortices shed at the right wing tip impact the symmetry plane (virtual tunnel wall) and move upwards.

![Figure 4.10 Panel Method result compared with wind tunnel data](image)

4.5.2. FEM

The reference case chosen to validate the use of FEM in the programme is the Henky's problem (Pauletti, et al., 2005). It consists of a circular plane membrane, fixed on its perimeter. The membrane is submitted to a constant
pressure difference that deforms it. The studied case is that with the adimensional load \( \tilde{q} = \frac{\Delta p R}{E_m t} = 0.045 \) and \( \nu = 0.34 \).

First, a convergence study is done to find out what is the number of elements required to have a proper approximation to the solution. The convergence is displayed in Figure 4.12. The solution is considered good when 2000 elements are used. For that number of T3 elements, the deformed shape is that presented in Figure 4.13. Then, the result is compared to those available in the literature in Table 4.1.

Data in Table 4.1 shows how FEMPAN solution is homologous to that of previous algorithms. Hence, the programme is, properly validated.
4.6. Analysis of the model

It is of great interest to see the shapes that can be achieved using this kind of inflatable device. FEMPAN, once it has been validated, is used to calculate the inflated shape of these protuberances. The wing section used is NACA 4415 and the wing aspect ratio is 6.67.

A basic non-dimensional analysis indicates that the governing non-dimensional factor in this case is the non-dimensional dynamic pressure: $\hat{q} = \frac{q \cdot c}{E_m \cdot t}$. If the model is scaled in any way, the non-dimensional displacement and strain will always be the same. In this case the value used is $\hat{q} = 0.3$.

The results obtained are displayed in Figure 4.14 and Figure 4.15. There, it can be seen that tubercles inflate mainly upwards and without penetrating the wing, even though a great part of the lower side of the wing has been allowed to move. The latter is because the membrane is supposed to have, behind a support surface. The effect of span-wise pressure distribution is observed, as the protuberance closer to the wing tip (right of the images) is smaller than those near the symmetry plane (left side of the images).

![Figure 4.14 Bottom view of the inflated tubercles with an α=20º and \(\hat{q} = 0.3\)](image)
The value of $\dot{q}$ has been selected in order to get a reasonable size of the protuberance. For the case of Furos, with a chord of 30cm and flying at a dynamic pressure of 200Pa, the required $E_m t$ is of 200N/m. Using an elastomeric material, with a very low Young modulus ($E_m=0.2\,\text{MPa}$ (Callister, 2005 pp. A-13)), the necessary thickness is $t=1\,\text{mm}$. Materials that are more rigid would need lower thicknesses. The choice of the material should take into account the stability of its properties with temperature, the capability to withstand flight conditions, its need of maintenance, its Young modulus and its capability to be fabricated at the desired thickness.

If this solution wants to be implemented, the experimental results obtained in 2 Experimental analysis, should be validated for leading edge tubercles pointing upwards.
5. Technical and economical viability of the proposal

This study gives the main guidelines to design a membrane based high-lift device inspired in humpback whales. However, there are some features cannot be designed generally. They have to be studied for every aircraft where it is installed in function of its geometry and the way it is operated. As it was said at Scope of the study, a study in aeroelastic modes has not been conducted. Therefore, some aeroelastic vibration modes might appear for some configurations. The membrane material selection and its attachment design also have to be done for each wing where it is installed and nothing is said. Installing the membrane requires a slight modification in the wing structure to fit the intake duct and the membrane attachment.

None of those problems to solve seems to be difficult to overcome once the aircraft where the device has to be installed is well known. This is because the membrane installation does not require any modification in the general configuration or the flight computer. Furthermore, for new design aircrafts, the wing design would already take the high-lift device into account, which would make the installation even easier. Technical viability of the proposed solutions is assured.

From the economical point of view, again the gain depends strongly on the platform where it is installed. Operation fees and the final design price are main parameters. The main benefit of the device is the increase of the Maximum Take-Off Weight (MTOW). This would allow an increase on the payload weight the plane is capable to transport, which would allow for an increase of economical revenue per flight. Alternatively, a higher mass of fuel could be loaded and extended range or autonomy operations could be done. This would allow to operate further to the base or to stay more time at the place that wants to be observed.

Let us consider the case of Furos (see: Annex A - Furos Datasheet). There, the operative empty weight (OEW) is of 5910 g. and the MTOW, of 11000g. Let the increase of MTOW obtained with the membrane patches be 5% (conservative compared to Figure 1.8) and the considered flight be a round trip to a point at 100km of the base. At economical cruise speed (70km/h), it would take 2,8h to get there and come back. Considering the observation time, a 3,5h flight is considered, which at 0,5l/h of fuel consumption yields 1,75l of fuel needed at take-off. As 85 oct. benzin is used, the fuel mass at take off would be of 1300g. Thus, the Maximum Payload (MP/L) would be of 3790g with typical configuration and of 4340g with augmented MTOW. This means an MP/L increase of 15%. The
relative increase of P/L grows when longer range flights are done, as more fuel is needed.

To sum up, an approximate MP/L increase of 15% is obtained for Furos with no additional direct cost per flight. The only additional cost is the device design and installation cost (including the additional difficulty in wing structure fabrication) and its maintenance. This cost will be small compared to the design and fabrication cost of the aircraft while its effect on MP/L cannot be neglected.
6. Environmental effects of the life cycle of the proposal

The operation of the design proposed in this study will, with no doubt, have its effects on the environment. As it happened in 5 Technical and economical viability of the proposal, the design process is at a conceptual stage. Therefore, the environmental effects depend upon the choices made in the final design. For example, type of material used, type of aircraft where it flies and the attachment design would have great effect on this analysis.

Even though environmental effects cannot be quantified at this point, they can be mentioned and analysed. Pernicious effects on environment would be those of the membrane material production, patch manufacturing, glue usage and the assembly dismantlement. Those effects should be minimized during the final design process by choosing a recyclable material for the membrane and attaching it by reversible means instead of gluing. As in 5 Technical and economical viability of the proposal, production and dismantlement effects are very small compared to those of the complete aircraft.

On the other hand, the high-lift device operations would have some beneficial effects on the environment. As mentioned in 5 Technical and economical viability of the proposal, they allow to transport a heavier payload on the same aircraft. When a specific payload has to be transported, a smaller aircraft can be used. In other words, it reduces the amount of aircraft weight that has to be transported for a given payload by reducing the MPL/OEW ratio of UAVs. This makes flights for a given payload and mission less fuel consuming and, hence, less pollutant. Depending on the fuel used and the efficiency of the propulsive system and aerodynamics, the consumption reduction would change.

Furthermore, in new UAV designs that include humpback whale based protuberances, this MPL/OEW increase would allow to design smaller aircrafts for a given set of specifications reducing, thus, the overall environmental effect of manufacturing and dismantlement the aircraft.
7. Conclusions

Nowadays, small size aircraft, with capability to fly without a pilot onboard, are spreading widely in the market of terrain visualization. They substitute traditional airplanes, helicopters or satellites in their task of observing the earth. Their use represents an important cost reduction and, for some applications, their proximity to the ground improves the quality of the data retrieved. The technology to make them autonomous and capable of reaching interesting ranges while carrying useful payload is available.

However, this kind of aircrafts, because of their size, fly at a regime where Reynolds numbers are low compared with piloted airplanes. This has some drawbacks from the aerodynamic point of view. The main consequence of low Re flight is that the maximum lift of the wing is reduced, stall appears before. Furthermore, post-stall characteristics are worse at low Re. This is because velocity is too low to have well developed turbulent boundary layer.

Observing humpback whales ventral fins, which swim at approximately the same Re values, a new leading edge shape is proposed for low Re applications. This leading edge improves the post-stall characteristics. The maximum attainable $C_L$ is augmented when some protuberances based on the fins of the whale are installed on the leading edge of a wing. These protuberances have little effect at low angle of attack.

If some thin membrane patches are installed on the leading edge, with a tube that connects it to the lower side of the wing, and it is allowed to inflate by attaching it at its borders, a similar effect can be achieved. These patches would be inflated by the suction present at the leading edge and the overpressure transmitted by the tube from the lower side when high incidences happen. On the other side, at low angle of attack, impact pressure at the leading edge and light suction on the lower side of the wing would empty this device to reduce wet surface and hence friction drag.

The forces that appear on the skin of the wing are mainly caused by the pressure field around it. Shear stress caused by boundary layer is two orders of magnitude smaller even at this low Re values. Thus, Panel Method is appropriate to calculate the loads on a membrane installed as proposed. The membrane can be simulated using Finite Element Method if the stiffness matrix is recalculates as the membrane moves. The high displacements and orientation changes suffered by a thin membrane make its stiffness to change. FEMPANS uses these features to model the behaviour of such membranes. When the proposed high-lift device
is being designed, it can be used to calculate the best attachment shape and materials to obtain the desired inflated shape.

These inflatable protuberances allow increasing the Maximum Take-off Weight while they leave the airplane weight virtually unchanged. Furthermore, mechanical actuators do not need to be added nor flight computer has to be upgraded to any extend.

The proposed device can be included in new UAV designs, but it can also be easily installed as a modification of an existing aircraft. If properly designed, it can have a positive environmental effect as it increases the maximum payload weight to aircraft weight ratio. From an economic point of view, very low costs are involved in operating this kind of devices while it can considerably increase the carried payload.

Experiments conducted at higher Re values than those presented would give interesting information on the applicability of the presented concept at higher Re values. The effect of higher Re is difficult to predict. Smaller tubercles (wavelength and wave amplitude) should also be tested to exploit the observed tendency of properties improvement. When the inflatable concept wants to be used, the benefits of tubercles placed at the upper side have to be experimentally validated.
8. Acknowledgements

In first place, I thank my family for helping me and encouraging my curiosity on science. Roberto Flores, my tutor in this project, has also been greatly helpful by helping me develop some “fire-fighter ideas" without running away.

Additionally, this work would not have been possible without the contribution of many other people. Professore Giuseppe Gibertini, from Politecnico di Milano, allowed the use of the wind tunnel at Bovisa uninterestedly, without which, no experimental chapter would have been done. Once at the Politecnico di Milano, Donato Grassi and Simone Garbaccio exceeded any expectation being extraordinarily active and proposing best and new ways of testing the models. Without them, the whole experiment would have limited to one single graphic. José Luís Perez and Joan Turró mechanized and welded the support system for the wind tunnel: something I had accepted doing without having a clue on how to do it.

Many other people gave me important indications in this long road from the initial idea to the conclusion of this project. Enrique Ortega shared with me some of his expertise in model construction and wind tunnel testing. Piero Dolfi and Alessandro Pancani did their bischerate when they were most needed. The collective “Centre Social Okupat Barrilonia” provided me with a used bike with no charge and a great smile. Trencalós Team offered me their facilities at Escola Superior d’Enginyeries Industrial i Aeronàutica de Terrassa (ETSEIAT) and shared their knowledge on UAVs. CIMNE has also contributed conceding a one year GiD license and members of “Millor que Nou 100% Reutilitzat” were of great help during the construction of the model.

I am very thankful to all the people mentioned above and to everyone who gave “psychological” assistance in those moments when everything seems impossible. Among others, my flatmates were remarkable in this task.
9. Bibliography


10. Annexes

10.1. Annex A - Furos Datasheet

**FUROS – Data Sheet**

**Dimensions**
- Wingspan: 2,280 mm
- Length: 1,730 mm
- Height: 570 mm

**Wing**
- Surface: 68,4 dm²
- Mean aerodynamic chord: 300 mm
- Airfoil: NACA 4415
- Max. Wing loading: 146,2 g/dm²
- Min wing loading: 94,30 g/dm²
- Dihedral: 3°
- RN: 360,000 – 750,000

**Engine**
- Maker: Zencah
- Model: G260PU (25.4 cc)
- Propeller: Zinger 16x6 or 16x8
- Fuel tank: 3 l (85 oct. benzine with 2% oil)

**Weights**
- Empty: 5,910 g
- Min. in flight weight (no fuel): 6,450 g
- Max. Take off weight: 11,000 g
- Payload: 5,080 g (fuel+payload)

**Performances**
- Stall speed (@ 8,000 g): 35 km/h
- Cruise speed: 70 km/h
- Max. Speed: 145 km/h
- Max. Climb rate: 10 m/s
- Fuel consumption (@ 70 km/h): 0.5 l/h
- Endurance (no reserve): 6 h

Furos is a medium range UAV that thanks to its gasoline engine can fly for six hours and cover distances of over 400 km guided by autopilot. The cargo bay has ample space for payload in the central area of the fuselage and the nose has a modular system designed to accommodate thermal cameras. The high power of its engine and its special streamlined design allows to take off and land in very little space from unprepared terrain. It can reach a top speed of 145 km/h and climb to 3,000 m in less than ten minutes. Equipped with a new video link and long range data link can be manually controlled to 20 km or undertake missions on their pre-programmed autopilot.

CATUAV S.L. | Tel. 93 630 35 30 | www.catuav.com | info@catuav.com
10.2. Annex B – Experimental error assessment

It is well known that no experiment can provide the exact value of the physical variable being measured. An error is always present. Since nothing can be done to eliminate this experimental error, it has to be estimated in order to check the reliability of the measurements. Assessment of the error in the wind tunnel data is the scope of this section.

10.2.1. Types of errors

There are two main kinds of errors: the constant or bias error and the random error. Bias error includes those effects that tend to repeat every time the measurement is done and cannot be reduced by repeating several times the experiment and calculating the average value. For example, the error produced by the interference of the balance with the flow is of the bias type. On the other hand, random error is produced by effects that do not repeat and their influence can be easily reduced by averaging results. Electromagnetic noise affecting the balance wiring and model vibration produce random errors.

Depending on the changes made between the measurements some effects can be considered to produce bias or random errors. For example, if every time a set of measurements is taken the model is assembled and disassembled, the wet surface of the support structure changes. The balance interference error includes, then, both bias and random components.

The wind tunnel experiment has not been designed to assess the absolute aerodynamic properties of the models as it is usually done. The interest is on the change of performance caused by the tubercles added to the regular model. This is why the test conditions were kept as constant as possible: the tubercles were glued to the model without detaching the latter from the balance. The distance from the model to the balance, the area of the support mechanism exposed to the wind and the interference of the supporting structure to the flow were kept constant.

As it has been seen, some uncertainties do not alter the final results, as they do not have any effect on the aerodynamic performance change produced by tubercles. Therefore, such uncertainties are not taken into account. In other words, the error considered from now on is the repetition error (the scatter in measurements from different test runs). This error indicates what part of the performance change observed can be attributed to the fact that multiple test runs are performed.
10.2.2. Method

In order to assess the repetition error, a third test run was performed before the tubercles were glued to the model. This gives an idea on the fluctuation of a measured value between two different test runs. Measures were repeated for 13 flight conditions (angle of attack and Re).

It is assumed that the values of force and moment obtained for a given flight condition are normally distributed. For each condition, multiple test would yield an average value ($\bar{X}$) and a standard deviation (S). Those would tend, when the number of test repetitions increases, to the distribution law average ($\mu$) and its standard deviation ($\sigma$) as it is shown in Figure 10.1. In other words, $\bar{X}$ estimates $\mu$ which is the real value of the force or moment measured (except for the bias error) and S estimates $\sigma$.

It is obvious that as the measurement conditions change (different angles of attack or Re values are tested) the parameters of the normal law will change. In particular, $\mu$ will change. However, it can be considered that $\sigma$ is constant: the quality of the results does not change with the measurement condition. This is because, as it has been said in “2.1 Experiment description”, the balance full scale is much larger than the values it is measuring. Furthermore, statistical methods used are quite robust regarding the change in population dispersion. Therefore, if the dispersion of the measurements at different flight conditions changes slightly, results will not be changed appreciably. What will be estimated is the common $\sigma$ expressing how much does a measure change as it is repeated across several test runs (in the same tunnel and without disassembling the balance system).

For each one of the measurement conditions that were repeated, the standard deviation of each sample is calculated obtaining: $S_1, S_2, S_3, ..., S_{13}$. Each of them has been calculated with a sample size of n=2. In order to improve the estimation N=13 measurement pairs are used together.
It is known that, for a sample of size ‘n’ taken from a normal distribution, the variable ‘j’ which equals \((n-1)\) times the unbiased sample variance divided by the true variance of the population, follows a \(\chi^2\) law of \(n-1\) degrees of freedom. This is expressed by Equation 10.1.

\[
j = \frac{(n-1)S^2}{\sigma^2} \approx \chi^2_{n-1}
\]

Equation 10.1

It is also known that, the sum of variables that follow \(\chi^2\) distribution follows in turn a \(\chi^2\) law, as indicated in Equation 10.2.

\[
j_i \approx \chi^2_{\nu} \Rightarrow J = \sum_{i=1}^{N} j_i \approx \chi^2_{\nu}
\]

Equation 10.2

Summing the variables defined previously, Equation 10.3 is found.

\[
J = \sum_{i=1}^{N} j_i = \sum_{i=1}^{N} \frac{(n-1)S^2_i}{\sigma^2} = \frac{(n-1)\sum_{i=1}^{N} S^2_i}{\sigma^2} \approx \chi^2_{N(n-1)}
\]

Equation 10.3

In this case of study, \(n=2\) and \(N=13\). Therefore, Equation 10.3 becomes Equation 10.4.

\[
J = \frac{1}{\sigma^2} \sum_{i=1}^{13} S^2_i \approx \chi^2_{13}
\]

Equation 10.4

From the definition of Student law (Pepió, et al., 2005) in Equation 10.5, the relationship between S and error is calculated in Equation 10.6.

\[
A \approx N(0;1); B \approx \sqrt{\frac{\chi^2_{\nu}}{\nu}} \Rightarrow T = \frac{A}{B} \approx Student_{\nu}
\]

Equation 10.5
In Equation 10.6, $X_i - m_i$ is the error committed. The random distribution of the error is expressed in Equation 10.7:

$$T = \frac{X_i - m_i}{\sigma} \approx \text{Student}_{\nu=N(n-1)}$$

\[
T = \frac{X_i - m_i}{\sqrt{\sum_{i=1}^{N} \left(\frac{(n-1)S_i}{\sigma^2}\right) \frac{1}{N(n-1)}}} \approx \text{Student}_{\nu=N(n-1)}
\]

Equation 10.6

In order to give an idea of the error, an 80% confidence interval is calculated. In order to do this, $T$ is changed by the value that leaves a 10% at each side in the Student law. This size of the confidence interval is chosen because the amount of data used in this analysis is very little and statistical procedures are very conservative in this case.

$$X_i - m_i = T \cdot \sqrt{\frac{\sum_{i=1}^{N} S_i}{N}}$$

Equation 10.7

In this case, Equation 10.8 becomes Equation 10.9 (Pepió, et al., 1999).

$$error = t_{\nu=N(n-1)}^{10\%} \cdot \sqrt{\frac{\sum_{i=1}^{N} S_i}{N}}$$

Equation 10.8

$$error = 1.35 \cdot \sqrt{\frac{\sum_{i=1}^{N} S_i}{N}}$$

Equation 10.9

10.2.3. Results

To obtain the error estimate from Equation 10.9 the $S_i$ values must be calculated from the available data pairs. Nevertheless, one additional step has to be made before. When two measures of the same flight conditions are made, the wind tunnel does not yield exactly the same dynamic pressure. However, since dynamic pressure and aerodynamic forces are measured together, this does not affect the aerodynamic coefficients calculated (apart from the fact that the Re value is also changed, but Re value effects happen at a greater scale). Since results and errors presented in this work will, eventually, be calculated as aerodynamic coefficients the effect of different dynamic pressure has to be
compensated. To do this, the forces and moments of one of the points to be compared are divided by its own dynamic pressure and multiplied by that of the other point. This way, forces are scaled suppressing the effect of the change in ‘q’ from measurement to measurement.

Next, for each of the 13 data pairs and each of the force and moment components, one variance is calculated. These values are added and the sum of variances is obtained for each force and moment component. Finally, Equation 10.9 is used to estimate an error bound for of each one of them. The results obtained are presented at Table 10.1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>0.213</td>
<td>0.466</td>
<td>0.334</td>
<td>0.325</td>
<td>0.0831</td>
<td>0.0167</td>
</tr>
</tbody>
</table>

**Table 10.1 Repetition error for each of the forces and moments**

In order to calculate the error affecting the coefficients, the errors above are divided by the dynamic pressure and the model surface. In the case of moments, they are also divided by the chord. Therefore, errors expressed as coefficients are different for each dynamic pressure used. The errors calculated are shown in Table 10.2.

<table>
<thead>
<tr>
<th></th>
<th>CL</th>
<th>CD</th>
<th>CMac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re=1,2·10⁵</td>
<td>±0,075</td>
<td>±0,035</td>
<td>±0,015</td>
</tr>
<tr>
<td>Re=2,4·10⁵</td>
<td>±0,02</td>
<td>±0,0075</td>
<td>±0,0035</td>
</tr>
<tr>
<td>Re=3,5·10⁵</td>
<td>±0,008</td>
<td>±0,0035</td>
<td>±0,0015</td>
</tr>
</tbody>
</table>

**Table 10.2 Repetition error for each of the coefficients**

10.2.4. Justification

Alternatively, to the method presented above, a more typical error analysis procedure might have been chosen. It would have been done by tracking all the known sources of error and summing up their effects. For example, the balance error measuring forces and moments, the error at measuring the angle of attack, the error measuring dynamic pressure, the error measuring the model surface and chord…

This kind of analysis would not have accounted for unknown effects, while overestimating some of the error sources. For example, the scale used to measure the angle of attack had a resolution of 0.5º but the distance between marks was several millimetres. Thus, the angular error was smaller than the scale resolution. Furthermore, tracking the error of the load cells through the conversion matrix to forces and moments would also have meant considering a greater error than the real one: the main source of error in the load cells is the
drift of the measurement due to the increase of internal temperature caused by the electrical currents flowing through the strain gages. This error has a predictable behaviour: the drift affects all the load cells at the same rate. Since the conversion matrix has been calculated by a least squares method, a large part of this error is suppressed. This is because there are 7 load cells and only 6 force and moment components. Therefore, one degree of freedom is kept at the vector of the load cells currents. In other words, the conversion matrix can allow for a certain level of uniform drift in all the load cells measurements without changing the forces and moments it yields.

The statistical approach has allowed to account for all the errors without overestimating them, and to blend altogether the error in balance measurements and any other source of error.
## 10.3. Annex C – Photographs metadata

Metadata of the photographs taken is presented next:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
<td>f 3.8</td>
</tr>
<tr>
<td>Exposition time</td>
<td>1/8s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>+1</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO400</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>28.0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

**Table 10.3 Metadata of Figure 2.1**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
<td>f 3.8</td>
</tr>
<tr>
<td>Exposition time</td>
<td>1/10s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>+1</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO400</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>28.0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

**Table 10.4 Metadata of Figure 2.2**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
<td>f 3.9</td>
</tr>
<tr>
<td>Exposition time</td>
<td>1/25s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>+1</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO400</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>36.0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

**Table 10.5 Metadata of Figure 2.4**
<table>
<thead>
<tr>
<th>Diaphragm aperture</th>
<th>f 3,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposition time</td>
<td>1/60s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>0</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO100</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>28,0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

Table 10.6 Metadata of Figure 2.5

<table>
<thead>
<tr>
<th>Diaphragm aperture</th>
<th>f 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposition time</td>
<td>1/125s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>0</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO100</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>28,0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

Table 10.7 Metadata of Figure 2.7

<table>
<thead>
<tr>
<th>Diaphragm aperture</th>
<th>f 3,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposition time</td>
<td>1/8s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>+1</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO400</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>28,0mm</td>
</tr>
<tr>
<td>Format</td>
<td>JPEG</td>
</tr>
</tbody>
</table>

Table 10.8 Metadata of Figure 2.1
<table>
<thead>
<tr>
<th>Table 10.9 Metadata of Figure 2.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.10 Metadata of Figure 2.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.11 Metadata of Figure 2.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

Table 10.12 Metadata of Figure 2.12

<table>
<thead>
<tr>
<th>Diaphragm aperture</th>
<th>f 5,6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposition time</td>
<td>1/80s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>0</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO100</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>84,0mm</td>
</tr>
<tr>
<td>Format</td>
<td>Raw</td>
</tr>
</tbody>
</table>

Table 10.13 Metadata of Figure 2.13

<table>
<thead>
<tr>
<th>Diaphragm aperture</th>
<th>f 5,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposition time</td>
<td>1/60s</td>
</tr>
<tr>
<td>Exposition compensation</td>
<td>0</td>
</tr>
<tr>
<td>Sensibility</td>
<td>ISO100</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
<td>62,0mm</td>
</tr>
<tr>
<td>Format</td>
<td>Raw</td>
</tr>
</tbody>
</table>

Table 10.14 Metadata of Figure 2.14
<table>
<thead>
<tr>
<th>Table 10.15 Metadata of Figure 2.28</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diaphragm aperture</strong></td>
</tr>
<tr>
<td><strong>Exposition time</strong></td>
</tr>
<tr>
<td><strong>Exposition compensation</strong></td>
</tr>
<tr>
<td><strong>Sensibility</strong></td>
</tr>
<tr>
<td><strong>Equivalent focal distance for 35mm sensor</strong></td>
</tr>
<tr>
<td><strong>Format</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.16 Metadata of Figure 2.29</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diaphragm aperture</strong></td>
</tr>
<tr>
<td><strong>Exposition time</strong></td>
</tr>
<tr>
<td><strong>Exposition compensation</strong></td>
</tr>
<tr>
<td><strong>Sensibility</strong></td>
</tr>
<tr>
<td><strong>Equivalent focal distance for 35mm sensor</strong></td>
</tr>
<tr>
<td><strong>Format</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.17 Metadata of Figure 2.30</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diaphragm aperture</strong></td>
</tr>
<tr>
<td><strong>Exposition time</strong></td>
</tr>
<tr>
<td><strong>Exposition compensation</strong></td>
</tr>
<tr>
<td><strong>Sensibility</strong></td>
</tr>
<tr>
<td><strong>Equivalent focal distance for 35mm sensor</strong></td>
</tr>
<tr>
<td><strong>Format</strong></td>
</tr>
<tr>
<td>Table 10.18 Metadata of Figure 2.31</td>
</tr>
<tr>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.19 Metadata of Figure 2.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.20 Metadata of Figure 2.33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>Diaphragm aperture</td>
</tr>
<tr>
<td>Exposition time</td>
</tr>
<tr>
<td>Exposition compensation</td>
</tr>
<tr>
<td>Sensibility</td>
</tr>
<tr>
<td>Equivalent focal distance for 35mm sensor</td>
</tr>
<tr>
<td>Format</td>
</tr>
</tbody>
</table>

*Table 10.21 Metadata of Figure 2.34*
10.4. Annex D - Problemtype used by FEMPAN

In order to adapt GiD to FEMPAN, a problemtype has to be installed. This problemtype consists on a series of files that have to be included in a folder called: “FEMPAN.gid”. The executable code of FEMPAN has to be included together with the files listed below:

10.4.1. FEMPAN.bas

This file is used by GiD to write the solver input file. Command beginning with * are processed while those without the symbol are written as they are:

General Data File

Axes:
*GenData(Axes)

Solving type:
*GenData(Solver)

Mass matrix method
*GenData(Mass_matrix_method)

Problem Size
*GenData(Problem_size)

Number of Elements, Number of Nodes, Element Type:
*nelem *npoint *nnode

Load Case
*GenData(Alpha) *GenData(Beta) *GenData(Dynamic_pressure)
*GenData(Reference_surface)

Membrane data
*GenData(Membrane_thickness) *GenData(Membrane_Young_Modulus)
*GenData(Membrane_Poisson_Ratio) *GenData(Membrane_Density)

Damping alpha
*GenData(Damping->Alpha)

Connectivities:
Element Material Node(1) Node(2) Node(3)
*loop elems
*ElemsNum *ElemsConec 0
*end elems

Trailing edge:
*Set Var NKUTTA(int)=0
*Set Cond Kutta *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*Set var NKUTTA=Operation(NKUTTA(int)+1)
*endif
*end

Trailing edge nodes number
*NKUTTA

Trailing edge nodes
*Set Cond Kutta *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*NodesNum
*endif
*end

Symmetry plane:
*Set Var NSYM(int)=0
*Set Cond Symmetry *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*Set var NSYM=Operation(NSYM(int)+1)
*endif
*end
Symmetry plane nodes number
*NSYM
Symmetry plane nodes
*Set Cond Symmetry *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*NodesNum
*endif
*end

Airfoil:
*Set Var NPROF(int)=0
*Set Cond Airfoil *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*Set var NPROF=Operation(NPROF(int)+1)
*endif
*end
Airfoil nodes number
*NPROF
Airfoil nodes
*Set Cond Airfoil *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*NodesNum
*endif
*end

Movable Elems:
*Set Var Nmov(int)=0
*Set Cond Moving_elements *elems
*loop elems *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*Set var Nmov=Operation(Nmov(int)+1)
*endif
*end
Movable elems number
*Nmov
Movable elems
*Set Cond Moving_elements *elems
*loop elems *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*ElemsNum
*endif
*end

Intake:
*Set Var Nintake(int)=0
*Set Cond Intake *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*Set var Nintake=Operation(Nintake(int)+1)
*endif
*end
Intakes number
*Nintake
Intake node
*Set Cond Intake *nodes
*loop nodes *OnlyInCond
*if(strcmp(cond(1),"Yes")==0)
*NodesNum
*endif
*end

Coordinates:
Node X Y Z
*set elems(all)
*loop nodes
*NodesNum *NodesCoord(1,real) *NodesCoord(2,real) *NodesCoord(3,real)
*end nodes

10.4.2. FEMPAN.bas

The bas file is executed by GiD when the input file has already been written. It contains information on what executable has to be run, in what directory it has to do so and what is the input parameter.

@ECHO OFF
rem set basename = %1
rem set directory = %2
rem set ProblemDirectory = %3
del %2\%1.log
del %2\%1.post.res
del %2\%1.post.msh
del %2\%1.err
rem OutputFile: %2\%1.log
rem ErrorFile: %2\%1.err
rem WarningFile: %2\%1.war
%3\FEMPAN.exe %1
10.4.3. FEMPAN.cnd

This file includes the list of conditions to apply to the geometry. For example, the trailing edge line has to be applied the kutta condition.

BOOK: Aerodynamic

CONDITION: Kutta
CONDTYPE: over lines
CONDMESHTYPE: over nodes
QUESTION: Trailing_edge#CB#(Yes, No)
VALUE: Yes
HELP: Select the line of the trailing edge where the Kutta Condition is to be applied.
END CONDITION

CONDITION: Symmetry
CONDTYPE: over lines
CONDMESHTYPE: over nodes
QUESTION: Symmetric#CB#(Yes, No)
VALUE: Yes
HELP: Select a planar curve defining the symmetry plane (only if symmetry condition wants to be used).
END CONDITION

CONDITION: Airfoil
CONDTYPE: over lines
CONDMESHTYPE: over nodes
QUESTION: Airfoil#CB#(Yes, No)
VALUE: Yes
HELP: Select a planar curve defining an airfoil of the wing (Optional).
END CONDITION

BOOK: FEM

CONDITION: Moving_elements
CONDTYPE: over surfaces
CONDMESHTYPE: over elems
QUESTION: Is_movable#CB#(Yes, No)
VALUE: Yes
HELP: Select the surfaces where membrane is to be considered.
END CONDITION

CONDITION: Intake
CONDTYPE: over points
CONDMESHTYPE: over nodes
QUESTION: Intake#CB#(Yes, No)
VALUE: Yes
HELP: Select the point of intake. Only one intake is accepted.
END CONDITION

10.4.4. FEMPAN.prb

This files includes all the data that need to be entered by the user. For example the angle of attack and the membrane Young modulus.
QUESTION: Solver#CB#(Only_Panel_Method, Static_Panel_and_membrane_inflation)
VALUE: Static_Panels_and_membrane_inflation
HELP: Panels Method can be convined with membrane inflation (needs movable condition) as wished.

BOOK: Data

TITLE: Panels_Method

QUESTION: Axes#CB#(CAD, Airfoil)
VALUE: Airfoil
HELP: CAD uses axes from pre-process, Airfoil uses axes generated from geometry (needs airfoil and symmetry conditions)

QUESTION: Alpha
VALUE: 0
HELP: Angle of attack

QUESTION: Beta
VALUE: 0
HELP: Sideslip angle

QUESTION: Reference_surface
VALUE: 1
HELP: Reference surface of the complet model. When symmetry is used, both sides surfaces has to be put.

TITLE: FEM

QUESTION: Mass_matrix_method#CB#(Complete_matrix, Aprox)
VALUE: Complete_matrix

QUESTION: Membrane_thickness
VALUE: 0
HELP: Thickness of the movable part

QUESTION: Membrane_Young_Modulus
VALUE: 0
HELP: Stiffness of the movable part

QUESTION: Membrane_Poisson_Ratio
VALUE: 0
HELP: Poisson Ratio of the movable part

QUESTION: Membrane_Density
VALUE: 0
HELP: Density of the movable part

QUESTION: Dynamic_pressure
VALUE: 0
HELP: Dynamic pressure used to calculate forces on membrane

QUESTION: Damping->Alpha
VALUE: 0
HELP: Damping intensity for low frequency modes

END GENERAL DATA
10.5. Annex E – FEMPAN Code Source

module variables

double precision, allocatable, save :: pos_nodes(:,,:), source(:,),
doublet(:,), panels_matrix(:,,:), rhs(:,), store_rhs(:,)
double precision, allocatable, save :: wake_strings_nodes(:,,:),
wake_intensities(:,), local_axes(:,,:)
double precision, allocatable, save :: mass_mat(:,),
local_coordinates_node(:,,:),
double precision, allocatable, save :: doublet_nodes(:,),
velocity_on_elems(:,,:), cp(:,,:), additional_equation_factor(:,)
double precision, allocatable, save :: loads(:,,:), fem_mass_mat(:,),
store_local_coordinates_node(:,,:), elem_stiff_mat(:,,:),
double precision, allocatable, save :: internal_loads(:,,:),
node_vel(:,,:), node_acc(:,,:), original_pos(:,,:), elem_force(:,,:)
double precision, allocatable, save :: damping_loads(:,,:), strain(:,,:),
stress(:,,:), strain_mat(:,,:), jac(:,,:,:,:), det_jac(:,,:),
inv_jac(:,,:,:,:)
integer, allocatable, save :: to_nodes(:,,:), to_elems(:,,:),
nodes_connections(:,), nodes_kutta(:,), nodes_symmetry(:,)
integer, allocatable, save :: nodes_airfoil(:,), elem_type(:,),
te_elem_to_string(:,,:), mass_mat_pointers(:,)
integer, allocatable, save :: elems_movable(:,), nodes_movable(:,),
inv_nodes_movable(:,), fem_mass_mat_pointers(:,)
integer, allocatable, save :: to_gauss_points(:,,:),
logical, allocatable, save :: node_is_on_symmetry(:,),
node_is_on_kutta(:,), node_is_movable(:,), elem_is_movable(:,)
double precision, save :: axes_vecs(3,3), symmetry_pars(4), velocity(3),
alpha, beta, chord(3), membrane_mass, time_step
double precision, save :: wake_error, long, s_ref, dyn_pressure, thick,
young, poisson, membrane_density
double precision, save :: beta_damping, alpha_damping, aerodyn_force(3),
life_dir(3), force_centre(3), Cl, Cd, intake_pressure, d(3,3),
isopar_nodes(4,2)
double precision, parameter :: pi= 3.14159265
integer, save ::
n elems, n_nodes, max_elem_type, min_elem_type, n_nodes_kutta, n_nodes_symmetry,
added_nodes_symmetry, solver_to_use
integer, save :: n_nodes_airfoil, n_wake_strings, n_wake_segments,
mass_mat_size, n elems movable, n_nodes_movable, intake_node
integer, save :: n movable gauss
logical, save :: symmetry, axes_airfoil, chord_done, complete_mass
character(len=64), save :: name, logfile, err, res, msh, dat, war

end module variables

**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************
**************************************************************************

program body

use variables

implicit none

call project_name()
open(UNIT=6, FILE=logfile, STATUS='REPLACE', ACTION='WRITE',
ACCESS='SEQUENTIAL')
write (6,'(A)') "Running on:

call timestamp(6)
call input()
call previous_calc()
if (solver_to_use .eq. 1) then !Programme may work as a simple panels
method
call panels(.false.)
call output(0.00)
else if (solver_to_use .ne. 1) then
call find_static_solution()
end if
close(6)
close(5)
end program body

**************************************************************************
**************
**************************************************************************
**************
**************************************************************************
**************

subroutine project_name()
use variables, only: name, logfile, err, res, msh, dat, war
implicit none
integer :: j

call get_command_argument(1, name)

!name='alita2'

name=trim(name)
j=len_trim(name)
logfile=name
dat=name
err=name
res=name
msh=name
war=name
if(j .gt. 54) then
open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
write (7,'(A)') "Error en el nombre de proyecto. Máximo número
de caracteres: 54"
close(7)
STOP "Error en el nombre de proyecto"
else
logfile(j+1:j+5)='.log'
dat(j+1:j+5)='.dat'
err(j+1:j+5)='.err'
war(j+1:j+5)='.war'
res(j+1:j+10)='.post.res'
end subroutine project_name

**************************************************************************
****************************

subroutine input()

use variables
implicit none
integer :: i, j, error
character, save :: aux_string*(40)

open(UNIT=8, FILE=dat, STATUS='OLD', ACTION="READ", ACCESS='SEQUENTIAL', IOSTAT=error)

if(error .ne. 0) then !Check that file is well open
  open(UNIT=7, FILE=err, ACTION="WRITE", ACCESS="SEQUENTIAL")
  write(7, '(A)') "Error at data file opening. Error number: ", error
  close(7)
  stop "Error at opening"
end if

write(6, '(A)') "Dat file correctly open"
call jump_line(4,8)
read(8, '(A)') aux_string
if(aux_string .eq. "Airfoil") then
  axes_airfoil=.true.
else if (aux_string .eq. "CAD") then
  axes_airfoil=.false.
else
  open(UNIT=7, FILE=err, ACTION="WRITE", ACCESS="SEQUENTIAL")
  write(7, '(A)') "Error at axes type reading. Not recognized"
  close(7)
  STOP "Error at opening"
end if
call jump_line(1,8)
read(8, '(A)') aux_string
if(aux_string .eq. "Only_Panels_Method") then
  solver_to_use=1
else if (aux_string .eq. "Static_Panels_and_membrane_inflation") then
  solver_to_use=2
else
  open(UNIT=7, FILE=err, ACTION="WRITE", ACCESS="SEQUENTIAL")
  write(7, '(A)') "Error at solver type reading. Not recognized"
  close(7)
  STOP "Error at opening"
end if
call jump_line(1,8)
read(8, *) aux_string
if(aux_string .eq. "Complete_matrix") then
  complete_mass=.true.
else if (aux_string .eq. "Aprox") then
  complete_mass=.false.
else
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "Error at mass method type reading. Not recognized"
  close(7)
  STOP "Error at opening"
end if

if(solver_to_use .eq. 2) complete_mass=.false.

  call jump_line(2,8)
  read(8,*) n elems, n nodes, max elem type ! Number of Elements, Number of Nodes, Element Type

  if(max elem type .ne. 3 .and. max elem type .ne. 4) then !Check that meshing is acceptable
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at data file opening. Element type is not accepted. Use only linear triangle and quadrilateral."
    close(7)
    STOP "Wrong element type"
  end if

  call jump_line(2,8)
  read(8,*) alpha, beta, dyn pressure, s_ref

  if (solver_to_use .ne. 1 .and. dyn pressure .le. 0.) then
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at dynamic pressure. Dynamic pressure must be greater than 0"
    close(7)
    STOP "Wrong dynamic pressure"
  end if

  if (s_ref .le. 0.) then
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at reference surface. Reference surface must be greater than 0"
    close(7)
    STOP "Wrong reference surface"
  end if

  call jump_line(2,8)
  read(8,*) thick, young, poisson, membrane_density

  if (solver_to_use .ne. 1 .and. thick .le. 0.) then
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at membrane thickness. Membrane thickness must be greater than 0"
    close(7)
    STOP "Wrong membrane thickness"
  end if

  if (solver_to_use .ne. 1 .and. young .le. 0.) then
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at Young's modulus. Young's modulus must be greater than 0"
    close(7)
    STOP "Wrong Young's modulus"
  end if
write (7,'(A)') "Error at membrane stiffness. Membrane Young modulus must be greater than 0"
close(7)
STOP "Wrong membrane stiffness"
end if
if (solver_to_use .ne. 1 .and. poisson .le. 0.) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "Error at membrane Poisson ration. Membrane Poisson ration must be greater than 0"
  close(7)
  STOP "Wrong membrane Poisson ration"
end if
if (solver_to_use .ne. 1 .and. membrane_density .le. 0.) then
  open(UNIT=7,FILE=war,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "Membrane density must be greater than 0, it has been set to 1"
  close(7)
  membrane_density = 1.
end if
call jump_line(1,8)

read (8,*) alpha_damping
call jump_line(3,8)
allocate(to_nodes(4,n_elems))
allocate(elem_type(n_elems))

!Reading elements/panels
min_elem_type=4
do j=1, n_elems
  read(8,*) i, to_nodes(:,j)
  if(to_nodes(4,j) .eq. 0) then
    min_elem_type=3
    elem_type(j)=3
  else
    elem_type(j)=4
  end if
end do !i
if(min_elem_type .ne. 3 .and. min_elem_type .ne. 4) then !Check that meshing is acceptable
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "Error at data file opening. Element type is not accepted. Use only linear triangle and quadrilateral."
  close(7)
  STOP "Wrong element type"
end if
!Element type accepted
call jump_line(3,8)
read(8,*) n_nodes_kutta
read(8,*)
allocate(nodes_kutta(2*n_nodes_kutta)) !Space is left for te nodes
duplication

!Reading trailing edge nodes

do j=1, n_nodes_kutta
   read(8,*) nodes_kutta(j)
end do !i

call jump_line(3,8)

read(8,*) n_nodes_symmetry
read(8,*)
allocate(nodes_symmetry(n_nodes)) !Space is left for extra nodes possibly
found later

!Reading symmetry plane nodes

do j=1, n_nodes_symmetry
   read(8,*) nodes_symmetry(j)
end do !i

call jump_line(3,8)

read(8,*) n_nodes_airfoil
call jump_line(1,8)
allocate(nodes_airfoil(n_nodes_airfoil))

!Reading airfoil nodes

do j=1, n_nodes_airfoil
   read(8,*) nodes_airfoil(j)
end do !i

call jump_line(3,8)

read(8,*) n elems movable
call jump_line(1,8)
allocate(elems movable(n elems movable))

!Reading movable elements

do j=1, n elems movable
   read(8,*) elems movable(j)
end do

call jump_line(3,8)

!Reading intake node
read(8,*) intake_node !Number of intakes is temporarily saved in
intake_node
if(intake_node .gt. 1) then
   open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
   write (7,'(A)') "Only one intake point has to be selected in order to
calculate the membrane inflation."
   close(7)
   STOP "Wrong intake number"
else if (intake_node .eq. 0 .and. solver_to_use .ne. 1) then
   open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
One intake node has to be selected. Make sure that there is one point with Intake condition and that it has been forced to the mesh.

Wrong intake number

Nodes positions reading
allocate(pos_nodes(3,n_nodes+n_nodes_kutta)) !Space is left to additional trailing edge nodes

Reading nodes

do j = 1, n_nodes
   read(*,*) i, pos_nodes(:,j) ! nodos
end do ! i

end subroutine input

**************************************************************************
**************
subroutine jump_line(i,logical_unit)
implicit none
integer, intent(in):: i,logical_unit
integer :: j

do j=1,i
   read(logical_unit,*)
end do

end subroutine jump_line

**************************************************************************
**
subroutine previous_calc()

use variables
implicit none

   call check_nodes_alone()
   call calc_inverse_connectivities()
   call nodes_are_on()
   if (n_nodes_kutta .gt. 0) call analyse_trailing_edge()
   call reorder_nodes_nums()
   call calc_inverse_connectivities()
   call nodes_are_on()
   if(solver_to_use .ne. 1) then
      call calc_movable_nodes()
      call check_movables_on_kutta()
   end if
   call calc_symmetry()
   call calc_chord()
   call calc_axes()
   call velocity_to_axes()
   call allocations ()
call calc_panels_axes(.false.)
call fem_previous()
call fulfill_mass_matrix()
if(solver_to_use .ne. 1) then
    call calc_original_position()
call movable_previous()
call store_local_coordinates()
call calc_fem_mass_mat()
end if
if(n_nodes_kutta .gt. 1) call calc_initial_wake()
call start_output()
end subroutine previous_calc

end subroutine check_nodes_alone

use variables
implicit none
integer :: i, j, k
logical :: correct, found

i=1
do while (i .le. n_nodes)
    correct=.false.
    do while (.not. correct)
        found=.false.
        j=1
        do while (.not. found .and. j .le. n_elems)
            do k=1, elem_type(j)
                if (to_nodes(k,j) .eq. i) found=.true.
            end do
            j=j+1
        end do
        if (found) then
            correct=.true.
        else
            n_nodes=n_nodes-1
            do j=i, n_nodes
                pos_nodes(:,j)=pos_nodes(:,j+1)
            end do
            do j=1, n_elems
                do k=1, elem_type(j)
                    if (to_nodes(k,j) .eq. i) to_nodes(k,j)=to_nodes(k,j)-1
                end do
            end do
            do j=1, n_nodes_kutta
                if (nodes_kutta(j) .gt. i) nodes_kutta(j)=nodes_kutta(j)-1
            end do
            do j=1, n_nodes_symmetry
                if (nodes_symmetry(j) .gt. i) nodes_symmetry(j)=nodes_symmetry(j)-1
            end do
            do j=1, n_nodes_airfoil
                if (nodes_airfoil(j) .gt. i) nodes_airfoil(j)=nodes_airfoil(j)-1
            end do
        end if
    end do
end do
if (intake_node .gt. i) intake_node=intake_node+1
end if
end do
i=i+1
end do

end subroutine check_nodes_alone

**************************************************************************
**************
subroutine analyse_trailing_edge()

use variables
implicit none
integer, allocatable :: elems_flag(:)
integer :: i, j, k, count, nodes_on_te_base(2), walking_nodes(3)
logical :: elem_on_te, keep_going
logical, allocatable :: nodes_inspected(:)
allocate (elems_flag(n_elems))
allocate (nodes_inspected(n_nodes))

elems_flag=0
nodes_inspected(:)=.false.

do i=1, n_nodes_kutta
  do j=1, nodes_connections(nodes_kutta(i))
    elems_flag(to elems(j,nodes_kutta(i)))=1
  end do
end do
!i Elements with one node on the trailing edge have been flagged 1, others 0
keep_going=.true.
do while (keep_going) !All trailing edge sections are being detected separately
  i=1
  do while (nodes_inspected(nodes_kutta(i)) .and. i .lt. n_nodes_kutta)
    i=i+1
  end do
  i=nodes_kutta(i) !i contains a node on te not yet inspected
elem_on_te=.false.
j=0
do while (.not. elem_on te .and. j .lt. nodes_connections(i))
  j=j+1
  count=0
  do k=1, elem_type(to elems(j,i))
    if(node is on_kutta(to_nodes(k,to elems(j,i)))) then
      count=count+1
    end if
  end do
  end do
  if (count .gt.1) elem_on_te=.true.
end do
  j=to elems(j,i)
!j contains one element with two nodes on te, one of which is node 'i'
end do
elems_flag(j)=2

k=0
do i=1, elem_type(j)
   if (node_is_on_kutta(to_nodes(i,j))) then
      k=k+1
      nodes_on_te_base(k)=to_nodes(i,j)
      nodes_inspected(to_nodes(i,j))=.true.
   end if
end do

k=j
walking_nodes(1:2)=nodes_on_te_base
keep_going=.true.
call opposite_node
(walking_nodes(1),walking_nodes(2),walking_nodes(3),k)
do while(keep_going) !Walking nodes walk in one sense on the te
   if (.not. nodes_inspected(walking_nodes(3))) then
      if (.not. node_is_on_kutta(walking_nodes(3))) then
         !New elem has not a side on te
         walking_nodes(1)=walking_nodes(3)
call neighbour_by_two_nodes(walking_nodes(1),walking_nodes(2), k)
         if (k .eq. 0) then
            keep_going=.false.
         else
            call opposite_node
            (walking_nodes(1),walking_nodes(2),walking_nodes(3),k)
         end if
      else !New elem has a side on te
         nodes_inspected(walking_nodes(3))=.true.
         walking_nodes(1)=walking_nodes(2)
         walking_nodes(2)=walking_nodes(3)
call opposite_node
         (walking_nodes(1),walking_nodes(2),walking_nodes(3),k)
      end if
   elseif keep_going=.false.
end if

k=j
walking_nodes(2)=nodes_on_te_base(1)
walking_nodes(1)=nodes_on_te_base(2) !Inverse sense to change the walking sense
keep_going=.true.
call opposite_node
(walking_nodes(1),walking_nodes(2),walking_nodes(3),k)
do while(keep_going) !Walking nodes walk in one sense on the te
   if (.not. nodes_inspected(walking_nodes(3))) then
      if (.not. node_is_on_kutta(walking_nodes(3))) then
         !New elem has not a side on te
         walking_nodes(1)=walking_nodes(3)
call neighbour_by_two_nodes(walking nodes(1),
walking_nodes(2), k)
    if (k .eq. 0) then
        keep_going=.false.
    else
        call opposite_node
    end if
    if (.not. nodes_inspected(walking_nodes(3)))
        elems_flag(k)=2
    end if
    else !New elem has a side on te
        nodes_inspected(walking_nodes(3))=.true.
        walking_nodes(1)=walking_nodes(2)
        walking_nodes(2)=walking_nodes(3)
        call opposite_node
    end if
else
    keep_going=.false.
end if
end do
count=0
do i=1, n_nodes_kutta
    if (nodes_inspected(nodes_kutta(i))) count=count+1
end do
if (count .ge. n_nodes_kutta) then !if all te nodes have been
    inspected
        keep_going=.false.
    else
        keep_going=.true.
    end if
end do

added_nodes_symmetry=0
do i=1, n_nodes_kutta
    pos_nodes(:,n_nodes+i)=pos_nodes(:,nodes_kutta(i))
    nodes_kutta(n_nodes_kutta+i)=n_nodes+i
    if (node_is_on_symmetry(nodes_kutta(i))) then
        added_nodes_symmetry=added_nodes_symmetry+1
        nodes_symmetry(n_nodes_symmetry+added_nodes_symmetry)=n_nodes+i
    end if
    do j=1, nodes_connections(nodes_kutta(i))
        if (elems_flag(to_elems(j,nodes_kutta(i))) .eq. 2) then
            do k=1, elem_type(to_elems(j,nodes_kutta(i)))
                if (to_nodes(k, to elems(j,nodes_kutta(i))) .eq.
                    nodes_kutta(i)) then
                    to_nodes(k, to elems(j,nodes_kutta(i)))=n_nodes+i
                end if
            end do
        end if
    end do
end do

n_nodes_symmetry=n_nodes_symmetry+added_nodes_symmetry
n_nodes=n_nodes+n_nodes_kutta
n_nodes_kutta=2*n_nodes_kutta

deallocate (elems_flag)
deallocate (nodes_inspected)
end subroutine analyse_trailing_edge

**************************************************************************
subroutine opposite_node (first, centre, opposite, element)

use variables
implicit none
integer :: k, first, centre, opposite, element
logical :: found

found=.false.
k=0
do while (.not. found .and. k .lt. elem_type(element))
    k=k+1
    if(to_nodes(k,element) .eq. first) found=.true.
end do
k=k+1
call circular_notation(k,element)
if(to_nodes(k,element) .eq. centre) then
    k=k+1
else
    k=k-3
end if
call circular_notation(k,element)
opposite=to_nodes(k,element)

end subroutine opposite_node

**************************************************************************
subroutine reorder_nodes_nums()

use variables
implicit none
integer, allocatable :: adj(:), adj_row(:), perm(:), perm_inv(:)
integer :: i,j,k, adj_num, adj_max, inv_connect_size
double precision, allocatable :: store_positions(:,:)

adj_max=2*n_nodes*(n_nodes-1)
inv_connect_size=0
allocate (adj(adj_max))
allocation (adj_row(n_nodes+1))
allocation (perm(n_nodes))
allocation (perm_inv(n_nodes))
allocation (store_positions(3,n_nodes))
call adj_set(n_nodes,adj_max,adj_num,adj_row,adj,-1,-1) !Setting adjacency
info up

do i=1, nelems
    do j=1, elem_type(i)
        k=j+1, elem_type(i)
call adj_set(n_nodes,adj_max,adj_num,adj_row,adj,
to_nodes(j,i), to_nodes(k,i))
    end do
end do
end do
call genrcm (n_nodes, adj_num, adj_row, adj, perm)
call perm_inverse3 (n_nodes, perm, perm_inv)
do i=1, n_nodes
   store_positions(:,i)=pos_nodes(:,perm(i))
end do
do i=1, n_nodes
   pos_nodes(:,i)=store_positions(:,i)
end do
do i=1, n_elements
   do j=1, elem_type(i)
      to_nodes(j,i)=perm_inv(to_nodes(j,i))
   end do
end do
do i=1, n_nodes_kutta
   nodes_kutta(i)=perm_inv(nodes_kutta(i))
end do
do i=1, n_nodes_symmetry
   nodes_symmetry(i)=perm_inv(nodes_symmetry(i))
end do
do i=1, n_nodes_airfoil
   nodes_airfoil(i)=perm_inv(nodes_airfoil(i))
end do
if (intake_node .ne. 0) intake_node=perm_inv(intake_node)
deallocate (adj)
deallocate (adj_row)
deallocate (perm)
deallocate (perm_inv)
deallocate (store_positions)
end subroutine reorder_nodes_nums

**************************************************************************
**************
subroutine nodes_are_on()
use variables, only: node_is_on_symmetry, node_is_on_kutta, n_nodes,
   n_nodes_symmetry, &
   n_nodes_kutta, nodes_symmetry, nodes_kutta
implicit none
integer :: i
if (allocated(node_is_on_symmetry)) deallocate(node_is_on_symmetry)
allocation(node_is_on_symmetry(n_nodes))
if (allocated(node_is_on_kutta)) deallocate(node_is_on_kutta)
allocation(node_is_on_kutta(n_nodes))

node_is_on_symmetry=.false.
node_is_on_kutta=.false.
do i=1, n_nodes_symmetry
   node_is_on_symmetry(nodes_symmetry(i))=.true.
end do

do i=1, n_nodes_kutta
    node_is_on_kutta(nodes_kutta(i))=.true.
end do
end subroutine nodes_are_on

**************************************************************************
**************
subroutine calc_movable_nodes()

use variables
implicit none

integer :: i, j

if (allocated(elem_is_movable)) deallocate(elem_is_movable)
allocation(elem_is_movable(n_elems))

elem_is_movable(:)=.false.
do i=1, n_elems_movable
    elem_is_movable(elems_movable(i))=.true.
end do

if (allocated(node_is_movable)) deallocate(node_is_movable)
allocation(node_is_movable(n_nodes))

node_is_movable(:)=.true.
n_nodes_movable=0

do i=1, n_nodes
    do j=1, nodes_connections(i)
        if (.not. elem_is_movable(to_elems(j,i))) node_is_movable(i)=.false.
    end do
    if (node_is_movable(i)) n_nodes_movable=n_nodes_movable+1
end do

if (allocated(nodes_movable)) deallocate(nodes_movable)
allocation(nodes_movable(n_nodes_movable))

if (allocated(inv_nodes_movable)) deallocate(inv_nodes_movable)
allocation(inv_nodes_movable(n_nodes))

inv_nodes_movable(:)=0
j=0

do i=1, n_nodes
    if (node_is_movable(i)) then
        j=j+1
        nodes_movable(j)=i
        inv_nodes_movable(i)=j
    end if
end do

if (n_nodes_movable .eq. 0 .and. solver_to_use .ne. 1) then !Not movable
    nodes available to inflate
    open(UNIT=7,FILE=war,ACTION="WRITE", ACCESS="SEQUENTIAL")
write (7, '(A)') "WARNING: NO movable node was found. Only panels method is used."
write (7, '(A)') "Elements selected as movable must have nodes not attached to the border."
close(7)
solver_to_use=1
else if(n_nodes_movable .ne. 0 .and. solver_to_use .eq. 1) then
!Unnecessary movable nodes
open(UNIT=7, FILE='war', ACTION='WRITE', ACCESS='SEQUENTIAL')
write (7, '(A)') "WARNING: Only panels method has been selected as solver."
write (7, '(A)') "Elements selected as movable will be ignored."
close(7)
end if

end subroutine calc_movable_nodes

*****************************************************************************
**************
subroutine check_movables_on_kutta()

use variables
implicit none

integer :: i

do i=1, n_nodes
  if(node_is_on_kutta(i) .and. node_is_movable(i)) then
    open(UNIT=7, FILE='err', ACTION='WRITE', ACCESS='SEQUENTIAL')
    write (7, '(A)') "Error at problem definition. No point on the trailing edge can be marked as movable."
    write (7, '(A)') "Only rigid trailing edge."
    close(7)
    STOP "Wrong problem definition"
  end if
end do

end subroutine check_movables_on_kutta

*****************************************************************************
**************
subroutine fem_previous()

use variables
implicit none

integer :: i, j, point
double precision :: point_mat(4,2), der_mat(2,4), gauss_point(2)
double precision , parameter :: factor=1./sqrt(3.)

isopar_nodes(1,1)=-1.
isopar_nodes(2,1)= 1.
isopar_nodes(3,1)= 1.
isopar_nodes(4,1)=-1.
isopar_nodes(1,2)=-1.
isopar_nodes(2,2)=-1.
isopar_nodes(3,2)= 1.
isopar_nodes(4,2)= 1. !Contains de position in isoparametric coordinates of nodes
allocate (jac(2,2,4,n_elems)) !One 2x2 Jacobian is stored for each of the gauss points
allocate (inv_jac(2,2,4,n_elems)) !Same for inverse Jacobian
allocate (det_jac(4,n_elems+n_elems_movable)) !One det(Jacobian) is stored for each of the gauss points. Additional ! space is left for undeformed shape det(jac) to calculate stiffness forces

do i=1, n_elems
  if(elem_type(i) .eq. 3)
    jac(1,1,1,i)=local_coordinates_node(1,1,i)-
    local_coordinates_node(1,3,i)
    jac(2,2,1,i)=local_coordinates_node(2,2,i)-
    local_coordinates_node(2,3,i)
    jac(1,2,1,i)=local_coordinates_node(2,1,i)-
    local_coordinates_node(2,3,i)
    jac(2,1,1,i)=local_coordinates_node(1,2,i)-
    local_coordinates_node(1,3,i)
    det_jac(1,i)=jac(1,1,1,i)*jac(2,2,1,i)-jac(1,2,1,i)*jac(2,1,1,i)
    inv_jac(1,1,1,i)=jac(2,2,1,i)
    inv_jac(2,2,1,i)=jac(1,1,1,i)
    inv_jac(1,2,1,i)=-jac(1,2,1,i)
    inv_jac(2,1,1,i)=-jac(2,1,1,i)
    inv_jac(:,:,1,i)=inv_jac(:,:,1,i)/det_jac(1,i)
  else
    do point=1, 4
      point_mat(point,:)=local_coordinates_node(1:2,point,i)
    end do
    do point=1, 4
      gauss_point(:)=factor*isopar_nodes(point,:)
      do j=1, 4
        der_mat(1,j)=0.25*isopar_nodes(j,1)*(1+isopar_nodes(j,2)*gauss_point(2))
        der_mat(2,j)=0.25*isopar_nodes(j,2)*(1+isopar_nodes(j,1)*gauss_point(1))
      end do
      jac(:,:,point,i)=matmul(der_mat,point_mat)
      det_jac(point,i)=jac(1,1,point,i)*jac(2,2,point,i)-jac(1,2,point,i)*jac(2,1,point,i)
      inv_jac(1,1,point,i)=jac(2,2,point,i)
      inv_jac(2,2,point,i)=jac(1,1,point,i)
      inv_jac(1,2,point,i)=-jac(1,2,point,i)
      inv_jac(2,1,point,i)=-jac(2,1,point,i)
      inv_jac(:,:,point,i)=inv_jac(:,:,point,i)/det_jac(point,i)
    end do
  end if
end do
end subroutine fem_previous

***************************************************************************
**************
subroutine update_jacs()

inv_j is inverse of Jacobian

end subroutine update_jacs
use variables
implicit none

integer :: i, j, k, point
double precision :: point_mat(4,2), der_mat(2,4), gauss_point(2)
double precision, parameter :: factor=1./sqrt(3.)

do k=1, n_elems_movable
  i=elems_movable(k)
  if(elem_type(i) .eq. 3) then
    jac(1,1,1,i)=local_coordinates_node(1,1,i)-
    local_coordinates_node(1,3,i)
    jac(2,2,1,i)=local_coordinates_node(2,2,i)-
    local_coordinates_node(2,3,i)
    jac(1,2,1,i)=local_coordinates_node(2,1,i)-
    local_coordinates_node(2,3,i)
    jac(2,1,1,i)=local_coordinates_node(1,2,i)-
    local_coordinates_node(1,3,i)
    det_jac(1,i)=jac(1,1,1,i)*jac(2,2,1,i)-jac(1,2,1,i)*jac(2,1,1,i)
    inv_jac(1,1,1,i)=jac(2,2,1,i)
    inv_jac(2,2,1,i)=jac(1,1,1,i)
    inv_jac(1,2,1,i)=-jac(1,2,1,i)
    inv_jac(2,1,1,i)=-jac(2,1,1,i)
    inv_jac(:,:,1,i)=inv_jac(:,:,1,i)/det_jac(1,i)
  else
    do point=1, 4
      point_mat(point,:)=local_coordinates_node(1:2,point,i)
    end do
    do point=1, 4
      gauss_point(:)=factor*isopar_nodes(point,:)
      do j=1, 4
        der_mat(1,j)=0.25*isopar_nodes(j,1)*(1+isopar_nodes(j,2)*gauss_point(2))
        der_mat(2,j)=0.25*isopar_nodes(j,2)*(1+isopar_nodes(j,1)*gauss_point(1))
      end do
      jac(:,:,point,i)=matmul(der_mat,point_mat)
      det_jac(point,i)=jac(1,1,point,i)*jac(2,2,point,i)-
      jac(1,2,point,i)*jac(2,1,point,i)
      inv_jac(1,1,point,i)=jac(2,2,point,i)
      inv_jac(2,2,point,i)=jac(1,1,point,i)
      inv_jac(1,2,point,i)=-jac(1,2,point,i)
      inv_jac(2,1,point,i)=-jac(2,1,point,i)
      inv_jac(:,:,point,i)=inv_jac(:,:,point,i)/det_jac(point,i)
    end do
  end if
end do
end subroutine update_jacs

!**************************************************************************
**************
subroutine movable_previous()

use variables
implicit none

double precision :: der_mat(2,4), gauss_point(2), aux
double precision, parameter :: factor=1./sqrt(3.)

integer :: i, j, point, gauss_num

d(:,:)=0.
d(1,1)=1.
d(2,2)=d(1,1)
d(1,2)=poisson
d(1,2)=d(2,1)
d(3,3)=0.5*(1-poisson)
aux=young/(1-poisson*poisson)
d(:,)=aux*d(:,):

n_movable_gauss=0
allocate (to_gauss_points(4,n elems_movable))

do i=1, n elems_movable
    if(elem_type(elems_movable(i)) .eq. 3) then
        n_movable_gauss=n_movable_gauss+1
        to_gauss_points(1,i)=n_movable_gauss
    else
        n_movable_gauss=n_movable_gauss+4
        to_gauss_points(2,i)=n_movable_gauss-3
        to_gauss_points(3,i)=n_movable_gauss-2
        to_gauss_points(4,i)=n_movable_gauss
    end if
end do

allocate (strain(3,n_movable_gauss))
allocate (stress(3,n_movable_gauss))
allocate (strain_mat(3,8,n_movable_gauss))
strain_mat=0.

do i=1, n elems_movable
    if(elem_type(elems_movable(i)) .eq. 3) then
        gauss_num=to_gauss_points(1,i)
        der_mat(1,1)=1.
        der_mat(2,1)=0.
        der_mat(1,2)=0.
        der_mat(2,2)=1.
        der_mat(1,3)=-1.
        der_mat(2,3)=-1.
        do j=1, 3
            der_mat(:,j)=matmul(inv_jac(:,:,1,elems_movable(i)),der_mat(:,j))
            strain_mat(1,2*j-1,gauss_num)=der_mat(1,j)
            strain_mat(2,2*j,gauss_num)=der_mat(2,j)
            strain_mat(3,2*j-1,gauss_num)=der_mat(2,j)
            strain_mat(3,2*j,gauss_num)=der_mat(1,j)
        end do
    else
        do point=1, 4
            gauss_num=to_gauss_points(point,i)
        end do
    end if
end do
gauss_point(:,)=factor*isopar_nodes(point,:)
do j=1, 4
der_mat(1,j)=0.25*isopar_nodes(j,1)*(1+isopar_nodes(j,2)*gauss_point(2))
der_mat(2,j)=0.25*isopar_nodes(j,2)*(1+isopar_nodes(j,1)*gauss_point(1))
end do !i
do j=1, 4
der_mat(:,j)=matmul(inv_jac(:,:,point,elems_movable(i)),der_mat(:,j))
strain_mat(1,2*j-1,gauss_num)=der_mat(1,j)
strain_mat(2,2*j,gauss_num)=der_mat(2,j)
strain_mat(3,2*j-1,gauss_num)=der_mat(2,j)
strain_mat(3,2*j,gauss_num)=der_mat(1,j)
end do
end do
end do
end do
end subroutine movable_previous

**************************************************************************
**************
subroutine calc_symmetry()

use variables, only: pos_nodes, nodes_symmetry, node_is_on_symmetry,
symmetry_pars, n_nodes, &
& n_elems, n_nodes_symmetry, symmetry, beta, err,
added_nodes_symmetry
implicit none
double precision :: pars_temp(4), vec1(3), vec2(3), aux, sym_tolerance
integer :: i, j, m
logical :: are_aligned
double precision, allocatable :: node_distance_to_plane(:,),
third_furthest_node_distance()

if(n_nodes_symmetry .eq. 0) then
write(6,'(A)') "No symmetry plane is taken into account"
symmetry=.false.
else if (n_nodes_symmetry .le. 2) then
open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
write (7,'(A)') "Error at symmetry plane, not enough points selected."
Please, select at least 3."
close(7)
STOP "Error at symmetry plane"
else

call all_points_aligned(are_aligned)
if(are_aligned) then
open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
write (7,'(A)') "Error at symmetry plane.All points selected are
on a line. Please, select other points"
close(7)
STOP "Error at symmetry plane"
end if

write(6,'(A)') "Calculating symmetry plane" !Plane equation:
A*x+B*y+C*z+D=0

symmetry=.true.                         !A=symmetry_pars(1),
B=symmetry_pars(2)...

if(abs(beta) .gt. epsilon(beta))
then
    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at symmetry plane. Cannot performe
    symmetric calculations with cross wind."
    write (7,'(A)') "Please, mesh all the model or select beta=0º"
    close(7)
    STOP "Error at symmetry plane"
end if

m=n_nodes_symmetry/2 !Far point is seeked for vec1 direction

vec1=pos_nodes(:,nodes_symmetry(m+1))-pos_nodes(:,nodes_symmetry(1))

symmetry_pars(:)=0.

do i=2,n_nodes_symmetry
    vec2=pos_nodes(:,nodes_symmetry(i))-pos_nodes(:,nodes_symmetry(1))
    call direction_compare(are_aligned,vec1,vec2)
    if(.not. are_aligned) then
        pars_temp(:3)=vec2
        call cross_product(pars_temp(:3),vec1)
        pars_temp(4)=-
        dot_product(pars_temp(:3),pos_nodes(:,nodes_symmetry(i)))
        symmetry_pars(:)=symmetry_pars(:)+pars_temp(:) !Nodes triplets
        that are further from each other (vec1^vec2 is greater) are given more
        importance
    end if
end do

aux=1/sqrt(dot_product(symmetry_pars(1:3),symmetry_pars(1:3)))
symmetry_pars(:)=aux*symmetry_pars(:)

allocate (node_distance_to_plane(n_nodes))
allocate (third_furthest_node_distance(n_elems))

do i=1, n_nodes
    node_distance_to_plane(i)=dot_product(symmetry_pars(1:3),
        pos_nodes(:,i))+symmetry_pars(4)
end do

do i=1, n_elems
    call calc_third_node(i,third_furthest_node_distance(i),
        node_distance_to_plane)
end do

call minimum_abs(third_furthest_node_distance,n_elems, sym_tolerance)

!Normal vector to the plane is wanted to point to nodes sides
j=0
aux=0.
do while(abs(aux) .le. sym_tolerance .and. j .lt. n_nodes)
j=j+1
aux=dot_product(symmetry_pars(1:3),pos_nodes(:,j))+symmetry_pars(4)
end do
if(aux .lt. 0.) then
    symmetry_pars=(-1.)*symmetry_pars
    node_distance_to_plane(:)=-node_distance_to_plane(:)
end if

write(6,'(A)') "Symmetry plane equation:
write(6,'(F5.2,A3,F5.2,A3,F5.2,A3,F5.2,A2)'
symmetry_pars(1),"*X+",symmetry_pars(2),&
&"*Y+",symmetry_pars(3),"*Z+",symmetry_pars(4),"=0"

do i=1, n_nodes
    if(node_is_on_symmetry(i)) then !Node is on symmetry plane points
        if(abs(node_distance_to_plane(i)) .ge. sym_tolerance) then !Node
            is actualy not on the plane
                open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
                write (7,'(A)') "Error at symmetry plane. All points selected
                must be on a planar curve."
                write (7,'(A)') " Please, select only one planar curve"
                close(7)
                STOP "Error at symmetry plane"
        end if
        call project_to_plane(pos_nodes(:,i))
    else !Node is not on the symmetry plane points list
        if(abs(node_distance_to_plane(i)) .lt. sym_tolerance) then !Node
            is actually on the plane
                n_nodes_symmetry=n_nodes_symmetry+1
                nodes_symmetry(n_nodes_symmetry+added_nodes_symmetry)=i
                node_is_on_symmetry(i)=.true.
                call project_to_plane(pos_nodes(:,i))
            else if (node_distance_to_plane(i) .lt. 0) then !Node is at the
                wrong side of the plane
                    open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
                    write (7,'(A)') "Error at symmetry plane. All geometry must be
                    at one side of symmetry plane."
                    write (7,'(A)') "Mesh only one side of the model"
                    close(7)
                    STOP "Error at symmetry plane"
        end if
    end if
end do !i

deallocate (node_distance_to_plane)
deallocate (third_furthest_node_distance)
end if

end subroutine calc_symmetry

**************************************************************************
```fortran
subroutine all_points_aligned(are_aligned)
    use variables
    implicit none
    double precision :: vec1(3), vec2(3)
    integer :: i = 3
    logical, intent(out) :: are_aligned

    vec1(:) = pos_nodes(:, nodes_symmetry(1)) - pos_nodes(:, nodes_symmetry(2))
    call normalize(vec1)

    are_aligned = .true.

    do while (are_aligned .and. i .le. n_nodes_symmetry-added_nodes_symmetry)
        vec2(:) = pos_nodes(:, nodes_symmetry(1)) - pos_nodes(:, nodes_symmetry(i))
        call normalize(vec2)
        call direction_compare(are_aligned, vec1, vec2)
        i = i + 1
    end do

end subroutine all_points_aligned

!**************************************************************************
**
subroutine direction_compare(are_aligned, vec1, vec2)
    implicit none
    double precision, intent(in) :: vec1(3), vec2(3) ! vec1 and vec2 are assumed to be normalized
    double precision :: aux
    logical, intent(out) :: are_aligned

    aux = dot_product(vec1, vec2)
    if (abs(abs(aux) - 1.) .lt. epsilon(aux)) then
        are_aligned = .true.
    else
        are_aligned = .false.
    end if

end subroutine direction_compare

!**************************************************************************
**
subroutine calc_third_node(elem, distance, node_distance_to_plane)
    use variables
    implicit none
    double precision, intent(out) :: distance
    double precision, intent(in) :: node_distance_to_plane(n_nodes)
    integer, intent(in) :: elem
    double precision :: min1, min2, min3
    integer :: i

    min1 = huge(0.); min2 = min1; min3 = min1

    do i = 1, elem_type(elem)
        if (abs(node_distance_to_plane(to_nodes(i, elem))) .le. min1 .and. nodes_connections(to_nodes(i, elem)) .gt. 1) then
            min3 = min2
        else
            min3 = min2
        end if

end do
```

123
min2=min1
min1=abs(node_distance_to_plane(to_nodes(i,elem)))
else if(abs(node_distance_to_plane(to_nodes(i,elem))) .le. min2 .and.
nodes_connections(to_nodes(i,elem)) .gt. 1) then
  min3=min2
min2=abs(node_distance_to_plane(to_nodes(i,elem)))
else if(abs(node_distance_to_plane(to_nodes(i,elem))) .le. min3 .and.
nodes_connections(to_nodes(i,elem)) .gt. 1) then
  min3=abs(node_distance_to_plane(to_nodes(i,elem)))
end if
end do

distance=min3
end subroutine calc_third_node

**************************************************************************
**
subroutine minimum_abs (vector, n, minim)
***
implicit none
integer, intent(in) :: n
double precision, intent (in) :: vector(n)
double precision, intent (out) :: minim
integer i

minim=abs(vector(1))
do i=2, n
  if(abs(vector(i)) .lt. minim) then
    minim=abs(vector(i))
  end if
end do

end subroutine minimum_abs

**************************************************************************
**
subroutine project_to_plane(point)
use variables, only: symmetry_pars
implicit none
double precision , intent(inout) :: point(3)
double precision :: a

  a=dot_product(point, symmetry_pars(1:3))+symmetry_pars(4)
  point=point-a*symmetry_pars(1:3)

end subroutine project_to_plane

**************************************************************************
**
subroutine calc_chord()
use variables, only: n_nodes_airfoil, nodes_airfoil, chord_done, chord,
node_is_on_kutta, &
& pos_nodes, err
implicit none
double precision :: aux_vec(3)
double precision, allocatable :: dist(:)
integer :: i, j, pos

allocate(dist(n_nodes_airfoil))

if(n_nodes_airfoil .gt. 1) then
  chord_done=.true.
  pos=0
  j=1
  write (6,'(A)') "Calculating chord of selectec airfoil."
  do while (pos .eq. 0 .and. j .le. n_nodes_airfoil)
    if(node_is_on_kutta(nodes_airfoil(j))) pos=j
    i=i+1
  end do
  if(pos .eq. 0) then
    !Check that Kutta condition is set at the airfoil
    open (UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
    write (7,'(A)') "Error at conditions. No trailing edge set at airfoil."
    write (7,'(A)') "Airfoil condition is optional, you can use it when bodies"
    close (7)
    STOP "Error at conditions"
  end if
  do j=1, n_nodes_airfoil
    aux_vec=pos_nodes(:,nodes_airfoil(j))-pos_nodes(:,nodes_airfoil(pos))
    dist(j)=dot_product(aux_vec,aux_vec) !module is maximized by maximizing module**2
  end do
  call max_absolute_value_position(dist,j,n_nodes_airfoil)
  chord(:)=pos_nodes(:,nodes_airfoil(j))-pos_nodes(:,nodes_airfoil(pos))
  write (6,'(A14,F8.2,F8.2,F8.2,A1)') "Chord vector (", chord(:), ")"
else
  chord_done=.false.
end if

deallocate(dist)
end subroutine calc_chord

**************************************************************************

subroutine max_absolute_value_position(vec1,i,n)
implicit none
integer, intent(in) :: n
integer, intent(out) :: i
integer :: j
double precision, intent(in) :: vec1(n)
double precision :: maxi=0.

do j=1, n
  if(abs(vec1(j)) .ge. maxi) then
    i=j
    maxi=vec1(j)
  end if
end do
end subroutine max_absolute_value_position

**************************************************************************
**
subroutine calc_axes()

use variables
implicit none
double precision :: aux
integer :: i

if (axes_airfoil .and. symmetry .and. chord_done) then
  axes_vecs(3,:)=-symmetry_pars(1:3) !Normal to symmetry plane pointing towards nodes
  aux=dot_product(axes_vecs(1,:),axes_vecs(3,:))
  axes_vecs(1,:)=axes_vecs(1,:)-aux*axes_vecs(3,:) !Projected on XY plane (symmetry plane)
  call normalize(axes_vecs(1,:))
  axes_vecs(2,:)=axes_vecs(3,:)
  call cross_product(axes_vecs(2,:),axes_vecs(1,:)) !Oy=Oz^Ox
else
  axes_vecs=0.
  do i=1,3
    axes_vecs(i,i)=1.
  end do
end if

if (axes_airfoil .and. .not. symmetry) then
  open(UNIT=7,FILE=war,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "WARNING: Cannot creat axis systemme without symmetry plane. Using CAD axes"
  write (7,'(A)') "Condition symmetry needs 3 or mode nodes."
  close(7)
end if

if(axes_airfoil .and. .not. chord_done) then
  open(UNIT=7,FILE=war,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write (7,'(A)') "WARNING: Cannot creat axis systemme without airfoil curve. Using CAD axes"
  write (7,'(A)') "Condition airfoil needs 2 or mode nodes."
  close(7)
end if

end subroutine calc_axes

**************************************************************************
**************

subroutine normalize (vec)

implicit none
double precision , intent(inout) :: vec(3)
double precision :: aux
aux = dot_product(vec, vec)
aux = sqrt(aux)
aux = 1 / aux
vec(:) = aux * vec(:)

end subroutine normalize

**************************************************************************
**************
subroutine cross_product(vec1, vec2)
implicit none
double precision, intent(inout) :: vec1(3)
double precision, intent(in) :: vec2(3)
double precision :: resul(3)

resul(1) = vec1(2) * vec2(3) - vec1(3) * vec2(2)
resul(2) = vec1(3) * vec2(1) - vec1(1) * vec2(3)
resul(3) = vec1(1) * vec2(2) - vec1(2) * vec2(1)
vec1(:) = resul(:)
end subroutine cross_product

**************************************************************************
**
subroutine velocity_to_axes()
use variables, only: alpha, beta, axes_vecs, velocity, pi
implicit none
double precision :: matrix(3,3), aux

aux = pi / 180. ! Factor from ° to rads PIrad/180°
alpha = alpha * aux
beta = beta * aux

velocity(1) = cos(alpha) * cos(beta)
velocity(2) = sin(alpha) * cos(beta)
velocity(3) = sin(beta) ! Velocity in body axes

matrix = transpose(axes_vecs) ! Orthonormal base inversion
velocity = matmul(matrix, velocity)
write (6, '(A26,F9.5,F9.5,F9.5,A1)') "Velocity in general axes (", velocity, ")
end subroutine velocity_to_axes

**************************************************************************
**
subroutine calc_inverse_connectivities()
use variables, only: to_nodes, to_elems, nodes_connections, n_nodes, n elems, min_elem_type, elem_type, n_nodes_kutta
implicit none
integer, allocatable :: aux(:, :)
integer i, j, k

if (min_elem_type .eq. 3) then
  k = 6 ! 6 elements at each node (expected maximum)
else
    k=4 !4 elements at each node (expected maximum)
end if

if(allocated(nodes_connections)) deallocate(nodes_connections)
allocate(nodes_connections(n_nodes))
if(allocated(to elems)) deallocate(to elems)
allocate(to elems(k,n_nodes))

nodes_connections(:)=0

do i=1,n elems
    do j=1, elem_type(i)
        if(nodes_connections(to_nodes(j,i)) ge k) then !When nodes have
            more elements than expected
                allocate(aux(k,n_nodes))
                aux(:,:)=to elems(:,:)
                deallocate(to elems)
                k=k+1
                allocate(to elems(k,n_nodes))
                to elems(1:k-1,:)=aux(:,:)
                deallocate(aux)
            end if
            nodes_connections(to_nodes(j,i))=nodes_connections(to_nodes(j,i))+1
        end do !j, elem_type
    end do !i, n elems
end subroutine calc_inverse_connectivities

**************************************************************************
**
recursive subroutine circular_notation(i,j)

use variables, only: elem_type
implicit none
integer, intent (inout) :: i
integer, intent (in) :: j

if(i .gt. elem_type(j)) then
    i=i-elem_type(j)
    call circular_notation(i,j)
else if(i .le. 0) then
    i=i+elem_type(j)
    call circular_notation(i,j)
end if

end subroutine circular_notation

**************************************************************************
**
subroutine allocations ()

use variables
implicit none
integer :: i,j,k
allocate (doublet(n_elems))
allocate (panels_matrix(n_elems,n_elems))
allocate (rhs(n_elems))
allocate (source(n_elems))
allocate (local_axes(4,3,n_elems))
allocate (local_coordinates_node(3,4,n_elems))
allocate (doublet_nodes(n_nodes))
allocate (velocity_onelems(3,n_elems))
allocate (cp(4,n_elems))
allocate (additional_equation_factor(n_elems))
if ( solver_to_use .ne. 1) then
allocate (loads(3,n_nodes_movable))
allocate (internal_loads(3,n_nodes_movable))
allocate (damping_loads(3,n_nodes_movable))
allocate (node_vel(3,n_nodes_movable))
allocate (node_acc(3,n_nodes_movable))
allocate (original_pos(3,n_nodes_movable))
allocate (store_local_coordinates_node(3,4,n_elems_movable))
end if
allocate (mass_mat_pointers(n_nodes+1))
mass_mat_pointers(1)=1
mass_mat_pointers(2:n_nodes+1)=0
do i=1,n_nodes
  do j=1,nodes_connections(i)
    do k=1,elem_type(to_elems(j,i))
      if (to_nodes(k,to_elems(j,i)) .gt. mass_mat_pointers(i+1))
        mass_mat_pointers(i+1)=to_nodes(k,to_elems(j,i))
      end do !k
    end do !j
  end do !i
end do !i

mass_mat_pointers(i+1)=mass_mat_pointers(i+1)-(i-1)+mass_mat_pointers(i)
end do
allocate (mass_mat(mass_mat_pointers(n_nodes+1)-1))
allocate (elem_force(3,n_elems))
end subroutine allocations

**************************************************************************
**
subroutine calc_panels_axes(update)
use variables
implicit none
logical, intent(in) :: update
logical :: go
integer :: i,j, panel_nodes(4)
do i=1, n elems
  go=.true.
  if(update) then
    if(.not. elem_is_movable(i)) go=.false.
end if
if (go) then
    call calc_panel_nodes(i, panel_nodes) !If triangle elems are used, it considers them as a quad with 2 coincident nodes. From now, only quad elements need to be considered
    call calc_local_axes(i, panel_nodes)
    do j=1, elem_type(i)
        local_coordinates_node(:,j,i)=pos_nodes(:,to_nodes(j,i))
        call local_coordinates_of_node(local_coordinates_node(:,j,i), i)
        local_coordinates_node(3,j,i)=0. !projected onto the panel
    end do !j
    end if
end do !i
end subroutine calc_panels_axes

**************************************************************************
**
subroutine calc_panel_nodes(i, panel_nodes)

use variables
implicit none
integer, intent(out) :: panel_nodes(4)
integer, intent(in) :: i

panel_nodes(1:3)=to_nodes(1:3,i)
if (elem_type(i) .eq. 3) then
    panel_nodes(4)=to_nodes(3,i) !Quadrilateral element is collapsed to triangle
else
    panel_nodes(4)=to_nodes(4,i)
end if
end subroutine calc_panel_nodes

**************************************************************************
**
subroutine calc_local_axes(elem, panel_nodes)

use variables, only: pos_nodes, to_nodes, n_elems, local_axes, elem_type
implicit none
integer, intent (in) :: panel_nodes(4), elem
integer :: j

local_axes(2,:,elem)=pos_nodes(:,panel_nodes(1))+pos_nodes(:,panel_nodes(2))
 & (pos_nodes(:,panel_nodes(3))+pos_nodes(:,panel_nodes(4)))
!Vector tangent to plane surface

local_axes(3,:,elem)=pos_nodes(:,panel_nodes(1))+pos_nodes(:,panel_nodes(4))
 & (pos_nodes(:,panel_nodes(3))+pos_nodes(:,panel_nodes(2)))
!Another vector tangent to plane surface

call cross_product(local_axes(3,:,elem),local_axes(2,:,elem))
!local_axes(3,:,elem) is now perpendicular to panel oriented as the surface
   call normalize (local_axes(3,:,elem))

local_axes(2,:,elem)=0.75*pos_nodes(:,panel_nodes(1))+0.25*pos_nodes(:,panel_nodes(2))-pos_nodes(:,panel_nodes(3)) !Not expected to be parallel to any side
   local_axes(2,:,elem)=local_axes(2,:,elem)-
   dot_product(local_axes(2,:,elem),local_axes(3,:,elem))*local_axes(3,:,elem)
   call normalize(local_axes(2,:,elem)) !second axe is now the projection on the surface

local_axes(1,:,elem)=local_axes(2,:,elem)
   call cross_product(local_axes(1,:,elem),local_axes(3,:,elem))

local_axes(4,:,elem)=0.
   if (elem_type(elem) .eq. 3) then
      do j=1,3
         local_axes(4,:,elem)=local_axes(4,:,elem)+pos_nodes(:,to_nodes(j,elem))
      end do !j
      local_axes(4,:,elem)=local_axes(4,:,elem)/3. !Mean point
   else
      do j=1,4
         local_axes(4,:,elem)=local_axes(4,:,elem)+pos_nodes(:,to_nodes(j,elem))
      end do !j
      local_axes(4,:,elem)=local_axes(4,:,elem)/4. !Mean point
   end if

end subroutine calc_local_axes

**************************************************************************
**
subroutine local_coordinates_of_node(coordinates, elem)
  use variables, only: local_axes
  implicit none
  double precision , intent (inout) :: coordinates(3)
  integer, intent (in) :: elem

    coordinates=coordinates-local_axes(4,:,elem)
    coordinates=matmul(local_axes(1:3,:,elem),coordinates)

end subroutine local_coordinates_of_node

**************************************************************************
**
subroutine calc_original_position()
  use variables
  integer :: i

    do i=1,n_nodes_movable
       original_pos(:,i)=pos_nodes(:,nodes_movable(i))
    end do
end subroutine calc_original_position

**************************************************************************
**
subroutine calc_initial_wake()

use variables, only: pos_nodes, wake_strings_nodes, wake_intensities,
nodes_kutta, to_elems, to_nodes, &
  & elem_type, te_elem_to_string, node_is_on_symmetry,
  & n_elems, n_nodes, &
  & velocity, n_nodes_kutta, n_wake_strings,
n_wake_segments, long, &
  & node_is_on_kutta, nodes_connections
implicit none

double precision :: aux_vec(3), aux
integer :: pos(2), i, j, k, count
integer, allocatable :: te_node_to_string(:)

n_nodes_kutta=n_nodes_kutta/2

if(.not. allocated(te_elem_to_string)) allocate
  (te_elem_to_string(2,n_elems))
  te_elem_to_string=0
if(.not. allocated(te_node_to_string)) allocate (te_node_to_string(n_nodes))
  te_node_to_string=0
n_wake_strings=0

do i=1, n_nodes_kutta
  if(.not. node_is_on_symmetry(nodes_kutta(i))) then
    n_wake_strings=n_wake_strings+1
    te_node_to_string(nodes_kutta(i))=n_wake_strings
    te_node_to_string(nodes_kutta(n_nodes_kutta+i))=n_wake_strings
  end if
end do

!Wake nodes number calculation
aux_vec=pos_nodes(:,to_nodes(1,to_elems(1,nodes_kutta(1))))
aux_vec=aux_vec-pos_nodes(:,to_nodes(2,to_elems(1,nodes_kutta(1))))
long=sqrt(dot_product(aux_vec,aux_vec)) !long is the caracteristic size
  of the mesh at te
n_wake_segments=floor(log(3.*n_nodes_kutta*(0.1)+1)/log(1.1))

!Wake nodes and intensities initialization
if(.not. allocated(wake_intensities)) allocate
  (wake_intensities(n_wake_strings))
  wake_intensities=0.
if(.not. allocated(wake_strings_nodes)) allocate (wake_strings_nodes(3,
  n_wake_segments, n_wake_strings))

  do i=1, n_nodes_kutta
    if(.not. node_is_on_symmetry(nodes_kutta(i))) then
      wake_strings_nodes(:,1,te_node_to_string(nodes_kutta(i)))=pos_nodes(:,nodes_kutta(i))
    end if
  end do

  do j=1, n_wake_segments-1
aux=1.1**(j-1)

\[
do \ i=1, n\_wake\_strings
\]

\[
\text{wake\_strings\_nodes}(::,j+1,i)=\text{wake\_strings\_nodes}(::,j,i)+\text{aux}\times\text{long}\times\text{velocity}(::)
\]

\[
call \ \text{replace\_wake\_node}(i,j)
\]

end do \ i

end do \ j

!Wake string intensity relationship to panels doublet intensity

\[
do \ i=1, n\_elems
\]

\[
\text{count}=0
\]

\[
k=0
\]

\[
do \ \text{while}(\text{count} .lt. 2 .and. k .lt. \text{elem\_type}(i))
\]

\[
k=k+1
\]

\[
\text{if}(\text{node\_is\_on\_kutta}(\text{to\_nodes}(k,i)) \ \text{then}
\]

\[
\text{count}=\text{count}+1
\]

\[
\text{pos}(\text{count})=k
\]

end if

end do

\[
\text{if} \ \ (\text{count} .eq. 2) \ \ \text{then} \ \ !The \ last \ element \ inspected \ has \ two \ nodes \ on
\]

t.e., so it contributes to wake intensity

\[
\text{if}(\text{pos}(1) .eq. 1 .and. \text{pos}(2) .ne. 2) \ \text{call} \ \text{swap}(\text{pos}(1),\text{pos}(2))
\]

\[
\text{if}(\text{.not. node\_is\_on\_symmetry}(\text{to\_nodes}(\text{pos}(1),i)))
\]

\[
\text{te\_elem\_to\_string}(1,i)=\text{te\_node\_to\_string}(\text{to\_nodes}(\text{pos}(1),i))
\]

\[
\text{if}(\text{.not. node\_is\_on\_symmetry}(\text{to\_nodes}(\text{pos}(2),i)))
\]

\[
\text{te\_elem\_to\_string}(2,i)=\text{te\_node\_to\_string}(\text{to\_nodes}(\text{pos}(2),i))
\]

end if

end do

\[
deallocate \ \ (\text{te\_node\_to\_string})
\]

end subroutine \ calc\_initial\_wake

!**************************************************************************

**

subroutine \ swap (a,b)

implicit none

integer, intent(inout) :: a,b

integer :: c

\[
c=a
\]

\[
a=b
\]

\[
b=c
\]

end subroutine \ swap

!**************************************************************************

**

subroutine \ replace\_wake\_node (i,j)

use \ variables

implicit none

double precision :: vertex1(3), vertex2(3), vertex3(3), minim, maxi, dist, output(3), direction(3), aux\_vec(3)

integer, intent(in) :: i,j

integer :: count, k, elem

\[
\text{if} \ (j .gt. 0) \ \text{then}
\]
if(j.eq.1) then
  count=0
  direction=local_axes(4,:,1)-0.05*long*local_axes(3,:,1)-
  wake_strings_nodes(:,2,i)
else
  elem=0
  direction= wake_strings_nodes(:,j,i)-wake_strings_nodes(:,j+1,i)
  dist=sqrt(dot_product(direction,direction))
  maxi=0.
end if
    call normalize (direction)

  do k=1, n_elems
    vertex1=pos_nodes(:,to_nodes(1,k))
    vertex2=pos_nodes(:,to_nodes(2,k))
    vertex3=pos_nodes(:,to_nodes(3,k))
    call through_triangle (vertex1, vertex2, vertex3,
      wake_strings_nodes(:,j+1,i), direction , output)
      if(output(1) .gt. 0. .and. output(2) .ge. 0. .and. output(3) .ge. 0. .and. output(2)+output(3) .le. 1.) then
        if(j.eq.1)then
          count=count+1
        else
          if(output(1) .gt. maxi .and. output(1) .lt. dist) then
            maxi=output(1)
            elem=k
          end if
        end if
      end if
  end do

  if(symmetry .and. j.eq.1) then
    call symmetric_point(vertex1)
    call symmetric_point(vertex2)
    call symmetric_point(vertex3)
    call through_triangle (vertex1, vertex2, vertex3,
      wake_strings_nodes(:,j+1,i), direction , output)
    if(output(1) .gt. 0. .and. output(2) .gt. 0. .and. output(3) .gt. 0. .and. output(2)+output(3) .lt. 1.) count=count+1
  end if

  if (elem_type(k) .eq. 4) then
    vertex1=pos_nodes(:,to_nodes(1,k))
    vertex2=pos_nodes(:,to_nodes(3,k))
    vertex3=pos_nodes(:,to_nodes(4,k))
    call through_triangle (vertex1, vertex2,
      vertex3,wake_strings_nodes(:,j+1,i), direction , output)
    if(output(1) .gt. 0. .and. output(2) .gt. 0. .and. output(3) .gt. 0. .and. output(2)+output(3) .lt. 1.) then
      if(j.eq.1)then
        count=count+1
      else
        if(output(1) .gt. maxi .and. output(1) .lt. dist) then
          maxi=output(1)
          elem=k
        end if
      end if
    end if
  end if
endif
call symmetric_point(vertex1)
call symmetric_point(vertex2)
call symmetric_point(vertex3)
call through_triangle (vertex1, vertex2, vertex3, wake_strings_nodes(:,j+1,i), direction, output)
if(output(1) > 0 .and. output(2) > 0 .and. output(3) > 0 .and. output(2)+output(3) < 1.) &
  count=count+1
end if
end do

dist=dot_product(symmetry_pars(1:3),wake_strings_nodes(:,j+1,i))+symmetry_pars(4)
if (dist < 0)
  wake_strings_nodes(:,j+1,i)=wake_strings_nodes(:,j+1,i)-
dist*symmetry_pars(1:3)
end if

if (j.eq.1 .and. mod(count,2) eq. 1) then !Point is in the body
  k=1, n elems
  aux_vec=wake_strings_nodes(:,2,i)-local_axes(4,:,k)
  dist=sqrt(dot_product(aux_vec,aux_vec))
  if (dist < minim .or. k.eq.1) then
    minim=dist
    elem=k
  end if
end do

wake_strings_nodes(:,2,i)=local_axes(4,:,elem)+0.05*long*local_axes(3,:,elem)
else if (j.ne.1 .and. elem.ne.0) then
  wake_strings_nodes(:,j+1,i)=wake_strings_nodes(:,j+1,i)+&
  & (maxi*dot_product(direction,local_axes(3,:,elem))+0.05*long)*local_axes(3,:,elem)
end if
end if
end subroutine replace_wake_node

**************************************************************************
**
subroutine through_triangle (vertex1, vertex2, vertex3, o, dir, resul)
implicit none
double precision, intent(in) :: vertex1(3), vertex2(3), vertex3(3), o(3),
dir(3)
double precision, intent (out) :: resul(3)
double precision :: e1(3), e2(3), t(3), p(3), q(3), aux

e1=vertex2-vertex1
e2=vertex3-vertex1
t=o-vertex1
p=dir
call cross_product(p,e2)
q=t
call cross_product(q,e1)
aux=dot_product(p,e1)
if (abs(aux) .gt. epsilon(aux)) then
  aux=1/aux
  resul(1)=dot_product(q,e2)
  resul(2)=dot_product(p,t)
  resul(3)=dot_product(q,dir)
  resul=resul*aux
else
  resul(2)=-1. !aux=0, means that panel and ray are parallel. Not intersection condition is placed.
end if

end subroutine through_triangle

!**************************************************************************
**
subroutine start_output()

use variables
implicit none

open(UNIT=5, FILE=res, STATUS='REPLACE', ACTION='WRITE',
ACCESS='SEQUENTIAL')

write(5,'(A)')   'GiD Post Results File 1.0'
write(5,'(A)')   'GaussPoints "Panels Centre" ElemType Quadrilateral "Skin"'
write(5,'(A)')   'Number Of Gauss Points: 1'
write(5,'(A)')   'Natural Coordinates: internal'
write(5,'(A)')   'end gausspoints'
write(5,'(A)')   'GaussPoints "Panels" ElemType Quadrilateral "Skin"'
write(5,'(A)')   'Number Of Gauss Points: 4'
write(5,'(A)')   'Natural Coordinates: internal'
write(5,'(A)')   'end gausspoints'
write(5,'(A)')   'GaussPoints "Force cent" ElemType Linear "Force centre"'
write(5,'(A)')   'Number Of Gauss Points: 1'
write(5,'(A)')   'Natural Coordinates: internal'
write(5,'(A)')   'end gausspoints'
if(n_wake_strings .gt. 1) then !There is at least one wake string
    write(5,'(A)')   'GaussPoints "Wake segment centre" ElemType Linear "Wake"
write(5,'(A)')   'Number Of Gauss Points: 1'
write(5,'(A)')   'Natural Coordinates: internal'
write(5,'(A)')   'end gausspoints'
end if

end subroutine start_output

!**************************************************************************
**

subroutine panels(update)
use variables
implicit none
double precision :: cpoint_coordinates(3), sym_cpoint_coordinates(3), doublet_vel(3), source_vel(3), string_vel(3)
double precision :: aux_vec(3), aux, influence
integer :: i, j, k
logical, intent(in):: update
logical :: skip

if(.not. update) then
allocate (store_rhs(n_elems))
store_rhs(:)=0.
end if

write (6,'(A)') "Calculating influence coefficients."
aux=1/(4*pi)
if (.not. update) then
do i=1, n_elems
rhs(i)= -dot_product(velocity,local_axes(3,:,i))
source(i)=rhs(i)*aux
if(solver_to_use .ne. 1) then
if(.not. elem_is_movable(i)) store_rhs(i)=rhs(i)
end if
end do
else
rhs(:)=store_rhs(:)
do i=1, n_elems
if (elem_is_movable(i)) then
rhs(i)=rhs(i)-dot_product(velocity,local_axes(3,:,i))
source(i)=-dot_product(velocity,local_axes(3,:,i))*aux
end if
end do
end if

do i=1, n_elems !Control point
cpoint_coordinates=local_axes(4,:,i)
if(symmetry) then
sym_cpoint_coordinates=cpoint_coordinates
call symmetric_point(sym_cpoint_coordinates)
end if
do j=1, n_elems !Inducing panel
skip=.false.
if(update) then
if(.not. elem_is_movable(i) .and. .not. elem_is_movable(j))
skip=.true.
end if
if (.not. skip) then
call calc_source_velocity(source_vel, cpoint_coordinates, j)
source_vel=source(j)*source_vel
rhs(i)=rhs(i)-dot_product(source_vel,local_axes(3,:,i))
if(.not. update .and. solver_to_use .ne. 1) then
if(.not. elem_is_movable(i) .and. .not. elem_is_movable(j))
then
store_rhs(i)=store_rhs(i)-dot_product(source_vel,local_axes(3,:,i))
end if
end if
call calc_doublet_velocity(doublet_vel, cpoint_coordinates, j)
panels_matrix(j,i)=dot_product(doublet_vel,local_axes(3,:,i))
if(symmetry) then
 call calc_source_velocity(source_vel, sym_cpoint_coordinates, j)
call symmetric_vector(source_vel)
source_vel=source(j)*source_vel
rhs(i)=rhs(i)-dot_product(source_vel, local_axes(3,:,i))
if(.not. update .and. solver_to_use .ne. 1) then
 if(.not. elem_is_movable(i).and. .not. elem_is_movable(j)) then
 store_rhs(i)=store_rhs(i)-dot_product(source_vel,local_axes(3,:,i))
 end if
end if
call calc_doublet_velocity(doublet_vel, sym_cpoint_coordinates, j)
call symmetric_vector(doublet_vel)
end if
end if
end do !j:Inducing panel

skip=.false.
if(update) then
 if(.not. elem_is_movable(i)) skip=.true.
end if
if (.not. skip) then
 do j=1, n_wake_strings
  call calc_string_velocity(string_vel, cpoint_coordinates, j, 0)
  if(symmetry) then
   call calc_string_velocity(aux_vec, sym_cpoint_coordinates, j, 0)
call symmetric_vector(aux_vec)
   string_vel=string_vel+aux_vec
  end if
   influence=dot_product(local_axes(3,:,i), string_vel)
   do k=1, n_elems
    if(te_elem_to_string(1,k) .eq. j) then
     panels_matrix(k,i)=panels_matrix(k,i)+influence
    end if
    if(te_elem_to_string(2,k) .eq. j) then
     panels_matrix(k,i)=panels_matrix(k,i)-influence
    end if
   end do !k Check all elements
  end do !j:Inducing wake string
 end if
end do

do i=1, nelems
 if(.not. update) then
  additional_equation_factor(i)=0.
 do j=1, n_elems
  if(i.ne.j) then

additional_equation_factor(i)=additional_equation_factor(i)+0.01*abs(panels_matrix(j,i))
end if
end do
end if
do j=1, n_elems
skip=.false.
if(update) then
if(.not. elem_is_movable(i) .and. .not. elem_is_movable(j))
skip=.true.
end if
if (.not. skip) then
panels_matrix(j,i)=panels_matrix(j,i)+additional_equation_factor(i)
end if
end do
if (.not. update) then
do i=1, n_elems
  doublet(i)=rhs(i)/panels_matrix(i,i)
end do
write(6,'(A)') "Solving linear system"
call bic_grad_stab(panels_matrix, rhs, n elems, doublet)
call calc_wake_intensities()
if(n_nodes_kutta .gt. 1) call iterate_wake()
if (.not. update) then
do i=1, n_nodes
  doublet_nodes(i)=doublet(to elems(1,i))
end do
end if
call interpolate_solution()
call calc_pressure()
call calc_aero_coeffs()
end subroutine panels

**************************************************************************

subroutine bic_grad_stab (mat, b, n, sol)
use variables, only: wake_error
implicit none
integer, intent(in) :: n
integer :: i
double precision, intent(in) :: mat (n, n), b(n)
double precision, intent(inout) :: sol(n)
double precision, allocatable :: res(:), old_res(:), v(:), p(:), s(:), t(:)
double precision :: rho, old_rho, alpha, beta, omega, error, error2, divide

end subroutine bic_grad_stab
allocate(res(n))
allocate(old_res(n))
allocate(v(n))
allocate(p(n))
allocate(s(n))
allocate(t(n))

divide=1/sqrt(dot_product(b,b))
res=b - matmul(sol,mat)
error=divide*sqrt(dot_product(res,res))
error2=error
wake_error=error
old_res=res
rho=1.
alpha=1.
omega=1.
v=0.
p=0.
i=0
do while (error .gt. 0.0001)
i=i+1
old_rho=rho
rho=dot_product(old_res,res)
beta=rho*alpha/(old_rho*omega)
p=res+beta*(p-omega*v)
v=matmul(p,mat)
alpha=rho/dot_product(old_res,v)
s=res-alpha*v
t=matmul(s,mat)
omega=dot_product(t,s)/dot_product(t,t)
sol=sol+alpha*p+omega*s
if(mod(i,25) .eq. 0) then
    res= b - matmul(sol,mat)
else
    res=s-omega*t
end if
error=divide*sqrt(dot_product(res,res))
error2=divide*sqrt(dot_product(alpha*p+omega*s,alpha*p+omega*s))
if(error2 .gt. error) error=error2
end do
write(6,'(A, I5)') "Panels solver Iterations: ", i
write(6,'(A, ES20.5E2)') " wake_error: ", wake_error
write(6,'(A, ES20.5E2)') " final_error: ", error
deallocate(res)
deallocate(old_res)
deallocate(v)
deallocate(p)
deallocate(s)
deallocate(t)

end subroutine bic_grad_stab

**************************************************************************
**
subroutine symmetric_point(point)
use variables, only: symmetry_pars
implicit none
**subroutine** symmetric_point

```
double precision, intent(inout) :: point(3)
double precision :: a
```

```
a = dot_product(point, symmetry_pars(1:3)) + symmetry_pars(4)
point = point - 2*a*symmetry_pars(1:3)
```

**endsubroutine** symmetric_point

```
subroutine symmetric_vector(vector)
```

```
use variables, only: symmetry_pars
implicit none
double precision, intent(inout) :: vector(3)
double precision :: aux
```

```
aux = dot_product(vector, symmetry_pars(1:3))
vector = vector - 2.*symmetry_pars(1:3)*aux
```

**end subroutine** symmetric_vector

```
subroutine calc_doublet_velocity( doublet_vel, cpoint_coordinates, elem)
```

```
use variables
implicit none
double precision, intent(out) :: doublet_vel(3)
double precision, intent(in) :: cpoint_coordinates(3)
double precision :: vortex_vel(3)
integer, intent(in) :: elem
integer k, l
```

```
doublet_vel = 0.
do k = 1, elem_type(elem)! side of the panel creating induced velocity
l = k + 1
  call circular_notation(l, elem)
  if (.not. (node_is_on_kutta(to_nodes(k, elem)) .and. node_is_on_kutta(to_nodes(l, elem)))) then ! Zero vortex intensity on trailing edge is imposed
    call calc_vortex_velocity(vortex_vel, cpoint_coordinates, pos_nodes(:, to_nodes(k, elem)), pos_nodes(:, to_nodes(l, elem)))
    doublet_vel = doublet_vel + vortex_vel
  end if
end do ! panel side
end subroutine calc_doublet_velocity
```

```
subroutine calc_string_velocity( string_vel, cpoint_coordinates, string, exception)
```

```
use variables
implicit none
```

```
end subroutine calc_string_velocity
```

141
```fortran
double precision, intent(out) :: string_vel(3)
double precision, intent(in) :: cpoint_coordinates(3)
double precision :: vortex_vel(3)
integer, intent(in) :: string, exception
integer i

string_vel=0.
do i=1, n_wake_segments-1
  if (i .ne. exception) then
    call calc_vortex_velocity(vortex_vel, cpoint_coordinates,
                            wake_strings_nodes(:,i,string), wake_strings_nodes(:,i+1,string))
    string_vel=string_vel+vortex_vel
  end if
end do
end subroutine calc_string_velocity

!***************************************************************************
**
subroutine calc_vortex_velocity(vortex_vel, cpoint_coordinates, node1, node2)

use variables, only: long
implicit none
double precision, intent(out) :: vortex_vel(3)
double precision, intent(in) :: cpoint_coordinates(3), node1(3), node2(3)
double precision :: r0(3), r1(3), r2(3), h

r0=node2-node1
r1=cpoint_coordinates-node1
r2=cpoint_coordinates-node2
vortex_vel=r0
call cross_product(vortex_vel,r1)
h=sqrt(dot_product(vortex_vel,vortex_vel)/dot_product(r0,r0))
if(h .gt. 10*epsilon(h)) then
call normalize(r1)
call normalize(r2)
vortex_vel=vortex_vel/dot_product(vortex_vel,vortex_vel)*(dot_product(r1,r0)-dot_product(r2,r0))
else
  vortex_vel=0.
end if
end subroutine calc_vortex_velocity

!***************************************************************************
**
subroutine calc_source_velocity (source_vel, cpoint_coordinates, elem)

use variables, only: pos_nodes, to_nodes, pi, elem_type, local_axes, local_coordinates_node
implicit none
double precision, intent (in) :: cpoint_coordinates(3)
```
double precision, intent (out) :: source_vel(3)
integer, intent (in) :: elem
double precision :: panel_nodes(3, elem_type(elem)), local_cpoint(3)
double precision :: d1(elem_type(elem)), m(elem_type(elem)),
r(elem_type(elem)), e(elem_type(elem))
double precision :: h(elem_type(elem)), aux, num(elem_type(elem))
integer :: i, j
logical :: on_edge

do i=1, elem_type(elem)
    panel_nodes(:,i)=local_coordinates_node(:,i,elem)
end do
!
local_cpoint=cpoint_coordinates
call local_coordinates_of_node(local_cpoint, elem)
do i=1, elem_type(elem)
j=i+1
call circular_notation(j,elem)
d1(i)=sqrt(dot_product(panel_nodes(1:2,j)-panel_nodes(1:2,i),panel_nodes(1:2,j)-panel_nodes(1:2,i)))
    m(i)=(panel_nodes(2,j)-panel_nodes(2,i))/(panel_nodes(1,j)-panel_nodes(1,i))
    r(i)=sqrt(dot_product(panel_nodes(:,i)-local_cpoint,panel_nodes(:,i)-local_cpoint))
    e(i)=(local_cpoint(1)-panel_nodes(1,i))*(local_cpoint(1)-panel_nodes(1,i))+local_cpoint(3)*local_cpoint(3)
    h(i)=(local_cpoint(1)-panel_nodes(1,i))*(local_cpoint(2)-panel_nodes(2,i))
end do

on_edge=.false.
do i=1, elem_type(elem)
j=i+1
call circular_notation(j,elem)
num(i)=r(i)+r(j)-d1(i)
    if (num(i) .lt. epsilon(num(i))) on_edge=.true.
end do

source_vel=0.
if(.not. on_edge) then
    do i=1, elem_type(elem)
j=i+1
call circular_notation(j,elem)
    aux=log(num(i)/(r(i)+r(j)+d1(i)))/d1(i)
    source_vel(1)=source_vel(1)+(panel_nodes(2,j)-panel_nodes(2,i))*aux
    sourceVel(2)=sourceVel(2)+(panel_nodes(1,i)-panel_nodes(1,j))*aux
    if(abs(local_cpoint(3)) .gt. epsilon(local_cpoint(3))) then
        source vel(3)=source vel(3)*atan((m(i)*e(i)-h(i))/(local_cpoint(3)*r(i)))-atan((m(i)*e(j)-h(j))/(local_cpoint(3)*r(j)))
    else
        source vel(3)=source vel(3)+sign(pi/2,(m(i)*e(i)-h(i)))-sign(pi/2,(m(i)*e(j)-h(j)))
    end if
    end do
sourceVel=-sourceVel

source vel=matmul(transpose(local axes(1:3,:)),source vel)
end if
end subroutine calc_source_velocity

**************************************************************************
**
subroutine iterate_wake()

use variables
implicit none
integer :: i,j,k,m
double precision :: velocities(3), coordinates(3), cpoint_coordinates(3),
doublet_vel(3), string_vel(3)
double precision :: sym_point(3), despl(3), aux_vec(3), mod
double precision :: factor, influence

write(6,'(A)') "Calculating wake geometry iteratively."

m=0
do while (wake_error .gt. 0.01)
m=m+1
write(6,*') m
do j=1, n_wake_segments-1
  do i=1, n_wake_strings
    coordinates=0.5*(wake_strings_nodes(:,j,i)+wake_strings_nodes(:,j+1,i))
    if (symmetry) then
      sym_point=coordinates
      call symmetric_point(sym_point)
    end if
    velocities(:)=velocity(:)
    do k=1, n_elems
      call calc_source_velocity (aux_vec,coordinates,k)
      velocities(:)=velocities(:)+source(k)*aux_vec
      call calc_doublet_velocity (aux_vec,coordinates,k)
      velocities(:)=velocities(:)+doublet(k)*aux_vec
      if (symmetry) then
        call calc_source_velocity (aux_vec,sym_point,k)
        call symmetric_vector(aux_vec)
        velocities(:)=velocities(:)+source(k)*aux_vec
        call calc_doublet_velocity (aux_vec,sym_point,k)
        call symmetric_vector(aux_vec)
        velocities(:)=velocities(:)+doublet(k)*aux_vec
      end if
    end do
  do k=1, n_wake_strings
    if (k.ne.i) then
      call calc_string_velocity(aux_vec, coordinates, k, 0)
      velocities(:)=velocities(:)+wake_intensities(k)*aux_vec
      if(symmetry) then
        call calc_string_velocity(aux_vec, sym_point, k, 0)
        call symmetric_vector(aux_vec)
        velocities(:)=velocities(:)+wake_intensities(k)*aux_vec
      end if
    else
      call calc_string_velocity(aux_vec, coordinates, k, j)
      velocities(:)=velocities(:)+wake_intensities(k)*aux_vec
      if(symmetry) then
        call calc_string_velocity(aux_vec, sym_point, k, j)
        call symmetric_vector(aux_vec)
        velocities(:)=velocities(:)+wake_intensities(k)*aux_vec
      end if
    end if
  end do
end do
end do

!i
end do
!
j
do i=1, n_elems !Control point
  cpoint_coordinates=local_axes(4,:,i)
  if(symmetry) then
    sym_point=cpoint_coordinates
    call symmetric_point(sym_point)
  end if
  do j=1, n_elems !Inducing panel
    if(te_elem_to_string(1,j) .ne. 0 .or. te_elem_to_string(2,j) .ne. 0) then
      !Only if their influence has changed
      call calc_doublet_velocity(doublet_vel, cpoint_coordinates, j)
      panels_matrix(j,i)=dot_product(doublet_vel,local_axes(3,:,i))
      if(symmetry) then
        call calc_doublet_velocity(doublet_vel, sym_point, j)
        call symmetric_vector(doublet_vel)
      end if
      panels_matrix(j,i)=panels_matrix(j,i)+additional_equation_factor(i)
    end if
  end do !j:Inducing panel
end do
!
j:Inducing wake string

!j
end do
!
j
factor=1.1**(j-1)
call normalize(velocities)
aux_vec=wake_strings_nodes(:,j+1,i)

wake_strings_nodes(:,j+1,i)=wake_strings_nodes(:,j,i)+factor*long*velocities(:)
call replace_wake_node(i,j)
despl=wake_strings_nodes(:,j+1,i)-aux_vec
do k=j+2, n_wake_segments
  wake_strings_nodes(:,k,i)=wake_strings_nodes(:,k,i)+despl
end do !k
mod=sqrt(dot_product(despl,despl))
end do !i
end do !j

do j=1, n_wake_strings
  call calc_string_velocity(string_vel, cpoint_coordinates, j, 0)
  if(symmetry) then
    call calc_string_velocity(aux_vec, sym_point, j, 0)
    call symmetric_vector(aux_vec)
  end if
  influence=dot_product(local_axes(3,:,i), string_vel)
do k=1, n_elems
    if(te_elem_to_string(1,k) .eq. j) then
      panels_matrix(k,i)=panels_matrix(k,i)+influence
    end if
    if(te_elem_to_string(2,k) .eq. j) then
      panels_matrix(k,i)=panels_matrix(k,i)-influence
    end if
  end do !k Check all elements
end do !j:Inducing wake string
end do

call bic_grad_stab(panels_matrix, rhs, n_elems, doublet)
call calc_wake_intensities()
end do

end subroutine iterate_wake

**************************************************************************
**
subroutine calc_wake_intensities()

use variables, only: doublet, wake_intensities, n_wake_strings, n_elems,
te_elem_to_string, n_nodes_kutta
implicit none
integer :: i

if (n_nodes_kutta .gt. 1) then
  wake_intensities=0.
  do i=1, n_elems
    if(te_elem_to_string(1,i) .ne. 0)
      wake_intensities(te_elem_to_string(1,i))=wake_intensities(te_elem_to_string
(1,i))+doublet(i)
    if(te_elem_to_string(2,i) .ne. 0)
      wake_intensities(te_elem_to_string(2,i))=wake_intensities(te_elem_to_string
(2,i))-doublet(i)
  end do
end if

end subroutine calc_wake_intensities

**************************************************************************
**
subroutine neighbour_by_two_nodes (node1, node2, elem)

use variables
implicit none
integer :: node1, node2, elem, aux, i
logical :: found

aux=0
found=.false.
do while (.not. found .and. aux .lt. nodes_connections(node1))
  aux=aux+1
  do i=1, elem_type(elem)
    if (to_nodes(i,to_elems(aux,node1)) .eq. node2 .and.
      to_elems(aux,node1) .ne. elem) found=.true.
  end do
end do

aux=to_elems(aux,node1)

if (.not. found) then
  elem=0
else
  elem=aux
end if

end subroutine neighbour_by_two_nodes

**************************************************************************
subroutine interpolate_solution()

use variables
implicit none
double precision, allocatable :: rhs_interpolation(:)
allocate(rhs_interpolation(n_nodes))

call fulfill_rhs(rhs_interpolation)
call skyline_conjgrad(mass_mat, mass_mat_pointers, rhs_interpolation, doublet_nodes, n_nodes, 1D-10)
end subroutine interpolate_solution

**************************************************************************
**
subroutine fulfill_mass_matrix()

use variables
implicit none
integer :: i,j
mass_mat(:)=0.
do i=1, n_nodes
    do j=i, n_nodes !only upper side is calculated as mass matrix is symmetric
        call mass_matrix_term(i,j)
    end do
end do
end subroutine fulfill_mass_matrix

**************************************************************************
**
subroutine mass_matrix_term(i,j)

use variables
implicit none
integer, intent(in) :: i,j
integer :: k, l, elem_node_1, elem_node_2
double precision :: contribution
do k=1, nodes_connections(i) !k is the element in touch with node i
elem_node_2=0
elem_node_1=0
do l=1, elem_type(to elems(k,i))
    if(to nodes(l, to elems(k,i)) .eq. i) elem_node_1=l
    if(to nodes(l, to elems(k,i)) .eq. j) elem_node_2=l
end do
if(elem_node_2 .ne. 0) then ! if the element k has also got node j in it
    if(elem_type(to elems(k,i)) .eq. 3) then
        call tri_elem_contribution(to elems(k,i),elem_node_1,elem_node_2,contribution)
    else
        call quad_elem_contribution(to elems(k,i),elem_node_1,elem_node_2,contribution)
end if
!**************************************************************************
**
subroutine quad_elem_contribution(elem,node1,node2,contribution)

use variables
implicit none

double precision :: gauss_point(2), n1, n2 
double precision , parameter :: factor=1./sqrt(3.)
integer, intent(in) :: elem, node1, node2
double precision , intent(out) :: contribution
integer :: point

contribution=0.
do point=1, 4
    gauss_point(:)=factor*isopar_nodes(point,:)
    n1=0.25*(1+isopar_nodes(node1,1)*gauss_point(1))*(1+isopar_nodes(node1,2)*gauss_point(2))
    if(node2.gt.0) then
        n2=0.25*(1+isopar_nodes(node2,1)*gauss_point(1))*(1+isopar_nodes(node2,2)*gauss_point(2))
        contribution=contribution+n1*n2*det_jac(point,elem)
    else
        contribution=contribution+n1*det_jac(point,elem)
    end if
end do !point

end subroutine quad_elem_contribution

!**************************************************************************
**
subroutine tri_elem_contribution(elem,node_1,node_2,contribution)

use variables
implicit none

double precision :: n(3), gauss_point(2), points(3,2)
integer, intent(in) :: elem, node_1, node_2
double precision , intent(out) :: contribution
integer :: point

contribution=0.
points(1,1)=1./6.
points(2,1)=2./3.
points(3,1)=points(1,1)
points(1,2)=points(1,1)
points(2,2)=points(1,1)
points(3,2)=points(2,1)

do point=1, 3
    gauss_point(:)=points(point,:)
    n(1)=gauss_point(1)
    n(2)=gauss_point(2)
    n(3)=1-gauss_point(1)-gauss_point(2)
    if(node_2 .gt. 0) then
        contribution=contribution+n(node_1)*n(node_2)
    else
        contribution=contribution+n(node_1)
    end if
end do
!
contribution=contribution*det_jac(1,elem)/6.

end subroutine tri Elem_contribution

**************************************************************************
**
subroutine fulfill_rhs(rhs_interpolation)

use variables
implicit none
double precision , intent(out) :: rhs_interpolation(n_nodes)
double precision :: contribution
integer :: i, j, l, elem_node

rhs_interpolation(:)=0.

do i=1, n_nodes
    do j=1, nodes_connections(i)
        do l=1, elem_type(to_elems(j,i))
            if(to_nodes(l,to_elems(j,i)) .eq. i) elem_node=l
        end do
        if(elem_type(to_elems(j,i)) .eq. 3) then
            call triElem_contribution(to_elems(j,i),elem_node, -1,
                contribution)
        else
            call quadElem_contribution(to_elems(j,i),elem_node, -1,
                contribution)
        end if
        rhs_interpolation(i)=rhs_interpolation(i)+doublet(to_elems(j,i))*contribution
    end do
end do

end subroutine fulfill_rhs

**************************************************************************
**
subroutine skyline_conjgrad(mat, pointers, b, sol, n, tolerance)

implicit none
integer, intent(in) :: n, pointers(n+1)
integer :: i
double precision, intent(in) :: mat (pointers(n+1)-1), b(n), tolerance
double precision, intent(inout):: sol(n)
double precision :: alpha, betha, dad, error, divide, error2
double precision, allocatable :: aux1(:), res(:), dir(:)

allocate (aux1(n))
allocate (res(n))
allocate (dir(n))

divide=1/sqrt(dot_product(b,b))
call skyline_product(mat,pointers,sol, n, res)
res=b - res
error=divide*sqrt(dot_product(res,res))
dir=res

i=0
do while(error .gt. tolerance)
i=i+1
call skyline_product(mat,pointers,dir, n, aux1)
dad=dot_product(dir,aux1)
alpha = dot_product(dir,res)/dad
sol= sol+alpha*dir
if(mod(i,25) .eq. 0) then
    call skyline_product(mat,pointers,sol, n, res)
    res=b-res
else
    res=res-alpha*aux1
end if
error=divide*sqrt(dot_product(res,res))
error2=divide*sqrt(dot_product(alpha*dir,alpha*dir))
if(error2 .gt. error) error=error2
betha=dot_product(res,aux1)/dad
dir=res-betha*dir
end do
write(6,'(A, I5)') "skyline Iterations:" , i
write(6,'(A, ES20.5E2)') "skyline_error:" , error

deallocate (aux1)
deallocate (res)
deallocate (dir)

end subroutine skyline_conjgrad

**************************************************************************
**
subroutine skyline_product(mat,pointers,vec, n, resul)
implicit none

integer, intent(in) :: n, pointers(n+1)
double precision , intent(in) :: mat(pointers(n+1)-1), vec(n)
double precision , intent(out) :: resul(n)
integer :: i, j

resul(:)=0.
do i=1, n
    do j=1, i-1
        if(j+pointers(j+1)-1-pointers(j) .ge. i)
            resul(i)=resul(i)+vec(j)*mat(pointers(j)+i-j)
        end do !j
    do j=i, pointers(i+1)-1-pointers(i)+i
        resul(i)=resul(i)+vec(j)*mat(pointers(i)+j-i)
    end do !j
end do !i
end subroutine skyline_product

**************************************************************************
**
subroutine calc_pressure()

use variables
implicit none

double precision :: aux_vel(3), store_vel(3)
integer :: i, n, j

n=0
intake_pressure=0.

do i=1, n_elems
    store_vel(:)=velocity(:)-dot_product(velocity(:),local_axes(3,:,i))*local_axes(3,:,i)
    if(elem_type(i) .eq. 3)
        call calc_tri_doublet_gradient_velocity(aux_vel,i)
        aux_vel=-4*pi*matmul(transpose(local_axes(1:3,:,i)),aux_vel)
        !double precision doublet intensity is -4*pi times calculated one
        velocity_on_elems(:,i)=store_vel(:)+aux_vel
        cp(:,i)=1.
        dot_product(velocity_on_elems(:,i),velocity_on_elems(:,i))
        if((to_nodes(1,i) .eq. intake_node .or. to_nodes(2,i) .eq. intake_node .or. to_nodes(3,i) .eq. intake_node)) then
            intake_pressure=intake_pressure+cp(1,i)
            n=n+1
        end if
    else
        do j=1, 4
            call calc_quad_doublet_gradient_velocity(aux_vel,i,j)
            aux_vel=-4*pi*matmul(transpose(local_axes(1:3,:,i)),aux_vel)
            !double precision doublet intensity is -4*pi times calculated one
            velocity_on_elems(:,i)=store_vel(:)+aux_vel
            cp(j,i)=1.
            dot_product(velocity_on_elems(:,i),velocity_on_elems(:,i))
            if((to_nodes(j,i) .eq. intake_node)) then
                intake_pressure=intake_pressure+cp(j,i)
                n=n+1
            end if
        end do
    end if
end do

if (solver_to_use .ne. 1) intake_pressure=intake_pressure/dble(n)
end subroutine calc_pressure
**************************************************************************
**
subroutine calc_quad_doublet_gradient_velocity(aux_vel,elem,point)
    use variables
    implicit none
    double precision :: der_mat(2,4), evaluation_point(2),
                        doublet_intensity(4)
    integer, intent(in) :: elem, point
    double precision, intent (out) :: aux_vel(3)
    double precision, parameter :: gauss=1./sqrt(3.)
    integer :: i
    do i=1, 4
        doublet_intensity(i)=doublet_nodes(to_nodes(i,elem))
    end do
    evaluation_point(:)=gauss*isopar_nodes(point,:)
    do i=1, 4
        der_mat(1,i)=0.25*isopar_nodes(i,1)*(1+isopar_nodes(i,2)*evaluation_point(2))
        der_mat(2,i)=0.25*isopar_nodes(i,2)*(1+isopar_nodes(i,1)*evaluation_point(1))
    end do
    aux_vel(1:2)=matmul(inv_jac(:,:,point,elem),matmul(der_mat,doublet_intensity))
    aux_vel(3)=0.
end subroutine calc_quad_doublet_gradient_velocity
**************************************************************************
**
subroutine calc_tri_doublet_gradient_velocity(aux_vel,elem)
    use variables
    implicit none
    double precision :: der_mat(2,3), doublet_intensity(3)
    integer, intent(in) :: elem
    double precision, intent (out) :: aux_vel(3)
    integer :: i
    der_mat(1,1)=1.
    der_mat(2,1)=0.
    der_mat(1,2)=0.
    der_mat(2,2)=1.
    der_mat(1,3)=-1.
    der_mat(2,3)=-1.
    do i=1, 3

doublet_intensity(i) = doublet_nodes(to_nodes(i, elem))
end do

aux_vel(1:2) = matmul(inv_jac(:, :, 1, elem), matmul(der_mat, doublet_intensity))
aux_vel(3) = 0.
end subroutine calc_tri_doublet_gradient_velocity

**************************************************************************
**
subroutine calc_aero_coeffs()
use variables
implicit none

integer :: i
double precision :: moment(3), r

aerodyn_force(:) = 0.
moment(:) = 0.

do i = 1, n elems
  if (elem_type(i) .eq. 3) then
    call tri_panel_force(i, moment(:))
  else
    call quad_panel_force(i, moment(:))
  end if
end do

r = 1/s_ref
CD = dot_product(aerodyn_force, velocity)
lift_dir = aerodyn_force - CD * velocity
CL = sqrt(dot_product(lift_dir, lift_dir))
lift_dir = lift_dir / CL
CD = r * CD
CL = r * CL

r = sqrt(dot_product(moment, moment) / dot_product(aerodyn_force, aerodyn_force))
call cross_product(force_centre, moment)
call normalize(force_centre)
force_centre = r * force_centre

aerodyn_force = aerodyn_force / s_ref
end subroutine calc_aero_coeffs
**************************************************************************
**
subroutine tri_panel_force(elem, moment)
use variables
implicit none

integer, intent(in) :: elem
double precision, intent(inout) :: moment(3)
double precision :: det_j, aux_vec(3), force(3), r(3), mom(3)

det_j = 0.5 * det_jac(1, elem) !Panel surface
force(:)=-cp(1,elem)*det_j*local_axes(3,:,elem)
r(:)=local_axes(4,:,elem)
mom(:)=r(:)
call cross_product(mom(:),force(:))
if(symmetry) then
  aux_vec(:)=force(:)
  call symmetric_vector(aux_vec)
  force(:)=force(:)+aux_vec(:)
  call symmetric_vector(r)
  call cross_product(r(:),aux_vec(:)) !Sym(r)^sym(force)
  mom(:)=mom(:)+r(:)
end if
aerodyn_force(:)=aerodyn_force(:)+force(:)
moment(:)=moment(:)+mom(:)

elem_force(:,elem)=force(:)

end subroutine tri_panel_force

**************************************************************************
**
subroutine quad_panel_force(elem, moment)
use variables
implicit none
integer, intent(in) :: elem
integer :: point
double precision, intent (inout) :: moment(3)
double precision :: aux_vec(3), force(3), r(3), mom(3)
double precision, parameter :: factor=1./sqrt(3.)

force(:)=0.
do point=1,4
  force(:)=force(:)-
cp(point,elem)*det_jac(point,elem)*local_axes(3,:,elem)
end do !point
r(:)=local_axes(4,:,elem)
mom(:)=r(:)
call cross_product(mom(:),force(:))
if(symmetry) then
  aux_vec(:)=force(:)
  call symmetric_vector(aux_vec)
  force(:)=force(:)+aux_vec(:)
  call symmetric_vector(r)
  call cross_product(r(:),aux_vec(:)) !Sym(r)^sym(force)
  mom(:)=mom(:)+r(:)
end if
aerodyn_force(:)=aerodyn_force(:)+force(:)
moment(:)=moment(:)+mom(:)

elem_force(:,elem)=force(:)

end subroutine quad_panel_force
subroutine calc_time_step()

use variables
implicit none

double precision :: lon, cd2, 1, aux_vec(3), aux_vec2(3), aux_vec3(3),
area, l1, l2, l3, chi
integer :: i, j, j1

chi=0.05

cd2=young/(membrane_density*(1-poisson*poisson))

lon=sqrt(dot_product(pos_nodes(:,1)-pos_nodes(:,2),pos_nodes(:,1)-
pos_nodes(:,2)))

do i=1, n_elems_movable
  if (elem_type(elems_movable(i)) .eq. 3) then
    aux_vec(:)=pos_nodes(:,to_nodes(2,elems_movable(i)))-
    pos_nodes(:,to_nodes(1,elems_movable(i)))
    aux_vec2(:)=pos_nodes(:,to_nodes(3,elems_movable(i)))-
    pos_nodes(:,to_nodes(1,elems_movable(i)))
    aux_vec3(:)=aux_vec(:)
    call cross_product(aux_vec3,aux_vec2)
    area=sqrt(dot_product(aux_vec3,aux_vec3))
    aux_vec3(:)=aux_vec(:)+aux_vec2(:)
    l1=sqrt(dot_product(aux_vec,aux_vec))
    l2=sqrt(dot_product(aux_vec2,aux_vec2))
    l3=sqrt(dot_product(aux_vec3,aux_vec3))
    l=area/l1
    if(l.lt.lon) lon=l
    l=area/l2
    if(l.lt.lon) lon=l
    l=area/l3
    if(l.lt.lon) lon=l
  else
    do j=1, 4
      j1=j+1
      call circular_notation(j1,elems_movable(i))
      aux_vec=pos_nodes(:,to_nodes(j1,elems_movable(i)))-
      pos_nodes(:,to_nodes(j,elems_movable(i)))
      l=sqrt(dot_product(aux_vec,aux_vec))
      if(l.lt.lon) lon=l
    end do
    aux_vec=pos_nodes(:,to_nodes(1,elems_movable(i)))-
    pos_nodes(:,to_nodes(3,elems_movable(i)))
    l=sqrt(dot_product(aux_vec,aux_vec))
    if(l.lt.lon) lon=l
    aux_vec=pos_nodes(:,to_nodes(2,elems_movable(i)))-
    pos_nodes(:,to_nodes(4,elems_movable(i)))
    l=sqrt(dot_product(aux_vec,aux_vec))
    if(l.lt.lon) lon=l
  end if
end do

beta_damping=lon*chi/sqrt(cd2)
time_step=0.5*(sqrt((lon*lon/cd2)+beta_damping*beta_damping)-beta_damping)

end subroutine calc_time_step

**

subroutine find_static_solution()

use variables

implicit none

integer :: i, j, k, l

double precision :: error, first_error, v_ref

i=1
l=0
first_error=1.
error=1. !Entrance into the loops is assured

write(6,'(A)') "Finding membrane geometry"

do while(first_error .gt. 0.0012)
  l=l+1
  if(i .eq. 1) then
    call panels(.false.) !The first time that panels method is used,
    it is not updating
  else
    call panels(.true.)
  end if
  j=1
  call calc_loads()
  node_vel(:,)=0.
  node_acc(:,)=0.
  call calc_internal_loads()
  if(i.eq.1) then
    call output(0.D0) !Undeformed situation is made available
  else
    call update_jacs()
  end if
  do while(error .gt. 0.001 .or. j .eq. 1)
    call calc_time_step()
    call calc_nodes_new_velocity()!Other steps, centered scheme is used
    Acc(i)=(Vel(i+0.5)-Vel(i-0.5))/time_step
    if(j.eq.1) node_vel(:,)=node_vel(:,)*0.5 !First step, uncentered
    scheme is used
    call calc_nodes_new_position()
    call calc_panels_axes(.true.)
    call calc_internal_loads()
    call calc_loads()

    v_ref=0.
    error=0.
    do k=1, n_nodes_movable
      error=error+dot_product(node_vel(:,k),node_vel(:,k))
    end do
    v_ref=v_ref+dot_product(loads(:,k),pos_nodes(:,nodes_movable(k))-original_pos(:,k))
    error=sqrt(error)
  end do
  call update_jacs()
  call output(0.D0)
end do
v_ref = sqrt(v_ref/membrane_mass)
error = error/(v_ref*sqrt(dble(n_nodes_movable)))
if(j.eq.1) first_error = error
j = j + 1
i = i + 1
end do
write(6,'(A,I4,A,I4)') "Elastic solver iteration number: ", l, "Number of total time steps: ", i
end do

call output(1.0)
end subroutine find_static_solution

**************************************************************************
**
subroutine calc_fem_mass_mat()

use variables
implicit none

integer :: i, j, row, column
membrane_mass=0.
if(complete_mass) then
if(.not. allocated(fem_mass_mat_pointers)) allocate
(fem_mass_mat_pointers(n_nodes_movable+1))
  fem_mass_mat_pointers(1)=1
  fem_mass_mat_pointers(2:n_nodes_movable+1)=0
do i=1, n_nodes_movable
  fem_mass_mat_pointers(i+1)=fem_mass_mat_pointers(i)
  row=nodes_movable(i)
  do j=0, mass_mat_pointers(row+1)-1-mass_mat_pointers(row)
    if(node_is_movable(row+j))
      fem_mass_mat_pointers(i+1)=fem_mass_mat_pointers(i+1)+1
    end do !j
  end do !i
end do !i
if(.not. allocated(fem_mass_mat)) allocate
(fem_mass_mat(fem_mass_mat_pointers(n_nodes_movable+1)-1))
  do i=1, n_nodes_movable
    do j=0, fem_mass_mat_pointers(i+1)-1-fem_mass_mat_pointers(i)
      row=i
column=i+j
      row=nodes_movable(row)
column=nodes_movable(column)
      fem_mass_mat(fem_mass_mat_pointers(i)+j)=mass_mat(mass_mat_pointers(row)+column-row)
    end do !j
  end do !i
else
  allocate (fem_mass_mat(n_nodes_movable))
  fem_mass_mat=0.
  do i=1, n_nodes

157
if(node_is_movable(i)) then
  do j=1, i-1
    if((j+mass_mat_pointers(j)-1)-mass_mat_pointers(j) .ge. i .and. node_is_movable(j)) then
      fem_mass_mat(inv_nodes_movable(i))=fem_mass_mat(inv_nodes_movable(i))+mass_mat(mass_mat_pointers(j)+i-j)
      membrane_mass=membrane_mass+mass_mat(mass_mat_pointers(j)+i-j)
      end if
    end do !j
  do j=i, mass_mat_pointers(i+1)-1-mass_mat_pointers(i)+i
    if(node_is_movable(j)) then
      fem_mass_mat(inv_nodes_movable(i))=fem_mass_mat(inv_nodes_movable(i))+mass_mat(mass_mat_pointers(i)+j-i)
      membrane_mass=membrane_mass+mass_mat(mass_mat_pointers(i)+j-i)
      end if
    end do !j
  end if
end do !i
end if
fem_mass_mat(:)=thick*membrane_density*fem_mass_mat(:)
membrane_mass=membrane_mass*thick*membrane_density
if(.not. complete_mass) then
  do i=1, n_nodes_movable
    fem_mass_mat(i)=1/fem_mass_mat(i)
  end do
end if
end subroutine calc_fem_mass_mat

**************************************************************************
**
subroutine store_local_coordinates()
  use variables
  implicit none
  integer :: i
  do i=1, n elems_movable
    store_local_coordinates_node(:,:,i)=local_coordinates_node(:,:,elems_movable(i))
  end do
end subroutine store_local_coordinates

**************************************************************************
**
subroutine calc_loads()
  use variables
  implicit none
  integer :: i

loads(:,:,)=0.

do i=1, n elems movable
    if (elem_type(elems movable(i)) .eq. 3) then
        call tri elem load(elems movable(i))
    else
        call quad elem load(elems movable(i))
    end if
end do !i

end subroutine calc loads

**************************************************************************
**
subroutine tri elem load(elem)

use variables
implicit none
integer, intent(in) :: elem

integer :: i, node_number_as_movable
double precision :: delta_p, n
delta_p=intake pressure-cp(1,elem)
n=1./3.
do i=1, 3
    if(node_is_movable(to_nodes(i,elem))) then
        node_number_as_movable=inv nodes movable(to_nodes(i,elem))
        loads(:,node_number_as_movable)=loads(:,node_number_as_movable)+
        & dyn pressure*delta_p*n*det jac(1,elem)*(0.5)*local axes(3,:,elem)
    end if
end do

end subroutine tri elem load

**************************************************************************
**
subroutine quad elem load(elem)

use variables
implicit none

double precision :: delta_p, n, iso pos(2)
double precision , parameter :: factor=1./sqrt(3.)
integer, intent(in) :: elem
integer :: i, point, node_number_as_movable

do point=1, 4
    delta_p=intake pressure-cp(point,elem)
    iso pos(:)=factor*isopar nodes(point,:)
do i=1,4
    if(node_is_movable(to_nodes(i,elem))) then
        n=0.25*(1+iso pos(1)*isopar nodes(i,1))*(1+iso pos(2)*isopar nodes(i,2))
        node_number_as_movable=inv nodes movable(to_nodes(i,elem))
        loads(:,node_number_as_movable)=loads(:,node_number_as_movable)+
        & dyn pressure*delta_p*det jac(point,elem)*n*local axes(3,:,elem)
    end if
end do
end subroutine quad_elem_load

!**************************************************************************
**
subroutine calc_nodes_new_velocity()

use variables
implicit none

integer :: i

do i=1, n_nodes_movable

node_acc(:,i)=(loads(:,i)+internal_loads(:,i)+damping_loads(:,i))*fem_mass_mat(i)
!fem_mass_mat contains the inverse of the nodal mass
end do

node_vel(:,i)=node_vel(:,i)+node_acc(:,i)*time_step

end subroutine calc_nodes_new_velocity

end subroutine calc_nodes_new_possition()

subroutine correct_nodes_entrances()

use variables
implicit none

integer :: i

do i=1, n_nodes_movable

pos_nodes(:,nodes_movable(i))=pos_nodes(:,nodes_movable(i))+node_vel(:,i)*time_step
end do

call correct_nodes_entrances()

end subroutine calc_nodes_new_possition

end subroutine correct_nodes_entrances()
integer :: i
logical :: entred

do i=1, n_nodes_movable
   call movable_node_is_inside(i,entred)
   if(entred) then
      node_vel(:,i)=0.
      call place_on_surface(i)
   end if
end do

end subroutine correct_nodes_entrances

subroutine movable_node_is_inside(mov_node, entred)

use variables
implicit none

integer, intent(in) :: mov_node
logical, intent(out) :: entred
double precision :: point1(3), point2(3), direction(3), vertex1(3),
vertex2(3), vertex3(3), output(3)
integer :: i, count, node

point1=pos_nodes(:,nodes_movable(mov_node))
point2=original_pos(:,mov_node)
direction=point1-point2

call normalize(direction) !This vector shold be pointing outwards if the
node has not entred the geometry
count=0

do i=1, n_elems
   node=to_nodes(1,i)
   if(node_is_movable(node)) then
      vertex1(:)=original_pos(:,inv_nodes_movable(node))
   else
      vertex1(:)=pos_nodes(:,node)
   end if
   node=to_nodes(2,i)
   if(node_is_movable(node)) then
      vertex2(:)=original_pos(:,inv_nodes_movable(node))
   else
      vertex2(:)=pos_nodes(:,node)
   end if
   node=to_nodes(3,i)
   if(node_is_movable(node)) then
      vertex3(:)=original_pos(:,inv_nodes_movable(node))
   else
      vertex3(:)=pos_nodes(:,node)
   end if
   call through_triangle (vertex1, vertex2,
vertex3,pos_nodes(:,nodes_movable(mov_node)), direction, output)
   if(output(1) .gt. 0. .and. output(2) .gt. 0. .and. output(3) .gt. 0. .and. output(2)+output(3) .lt. 1.) then
      count=count+1
   end if
   if(elem_type(i) .eq. 4) then
      node=to_nodes(4,i)
if(node_is_movable(node)) then
    vertex2(:)=original_pos(:,inv_nodes_movable(node))
else
    vertex2(:)=pos_nodes(:,node)
end if

call through_triangle (vertex1, vertex2, vertex3, pos_nodes(:,nodes_movable(mov_node)), direction, output)
if(output(1) .gt. 0. .and. output(2) .gt. 0. .and. output(3) .lt. 1.) then
    count=count+1
end if
end if

if(symmetry) then
    vertex1=pos_nodes(:,nodes_symmetry(1))
    vertex2=pos_nodes(:,nodes_symmetry(2))
    do i=3, n_nodes_symmetry
        vertex3=pos_nodes(:,nodes_symmetry(3))
        call through_triangle (vertex1, vertex2, vertex3, pos_nodes(:,nodes_movable(mov_node)), direction, output)
        if(output(1) .gt. 0. .and. output(2) .gt. 0. .and. output(3) .lt. 1.) then
            count=count+1
        end if
    end do
end if
if(mod(count,2) .eq. 0) then
    entred=.false.
else
    entred=.true.
end if

end subroutine movable_node_is_inside

**************************************************************************
**
subroutine place_on_surface(mov_node)

use variables
implicit none

integer, intent(in) :: mov_node
integer :: i, close_elem, node
double precision :: direction(3), vertex1(3), vertex2(3), vertex3(3), vertex4(3), output(3), t, l
t=huge(0.)
close_elem=0

do i=1, n_elems_movable
    node=to_nodes(1,elems_movable(i))
    if(node_is_movable(node)) then
        vertex1(:)=original_pos(:,inv_nodes_movable(node))
    else
        vertex1(:)=pos_nodes(:,node)
    end if
    node=to_nodes(2,elems_movable(i))
    if(node_is_movable(node)) then
        vertex2(:)=original_pos(:,inv_nodes_movable(node))
    else

vertex2(:,)=pos_nodes(:,node)
end if
node=to_nodes(3,elems_movable(i))
if(node_is_movable(node)) then
vertex3(:,)=original_pos(:,inv_nodes_movable(node))
else
vertex3(:,)=pos_nodes(:,node)
end if

if(elem_type(elems_movable(i)) .eq. 3) then
  direction(:,)=vertex1(:,)-vertex2(:,)
call cross_product(direction,vertex3-vertex2)
call normalize(direction)
else
  node=to_nodes(4,elems_movable(i))
  if(node_is_movable(node)) then
    vertex4(:,)=original_pos(:,inv_nodes_movable(node))
  else
    vertex4(:,)=pos_nodes(:,node)
  end if
  direction(:,)=0.5*(vertex1+vertex2-vertex3-vertex4)
call cross_product(direction, 0.5*(vertex1+vertex4-vertex2-vertex3))
  call normalize(direction)
end if

call through_triangle (vertex1, vertex2, vertex3,pos_nodes(:,nodes_movable(mov_node)), direction, output)
  if(output(2) .gt. 0. .and. output(3) .gt. 0. .and. output(2)+output(3) .lt. 1.) then
    if(abs(output(1)) .lt. abs(t)) then
      t=output(1)
close_elem=i
    end if
  end if
endif
if(elem_type(elems_movable(i)) .eq. 4) then
  call through_triangle (vertex1, vertex4, vertex3,pos_nodes(:,nodes_movable(mov_node)), direction, output)
else
  output(2)+output(3) .lt. 1.) then
    if(abs(output(1)) .lt. abs(t)) then
      t=output(1)
close_elem=i
    end if
  end if
endif
end do

if(close_elem .ne. 0) then
  node=to_nodes(1,elems_movable(close_elem))
  if(node_is_movable(node)) then
    vertex1(:,)=original_pos(:,inv_nodes_movable(node))
  else
    vertex1(:,)=pos_nodes(:,node)
  end if
  node=to_nodes(2,elems_movable(close_elem))
  if(node_is_movable(node)) then
    vertex2(:,)=original_pos(:,inv_nodes_movable(node))
  else
    vertex2(:,)=pos_nodes(:,node)
end if
node=to_nodes(3,elems_movable(close_elem))
if(node_is_movable(node)) then
    vertex3(:)=original_pos(:,inv_nodes_movable(node))
else
    vertex3(:)=pos_nodes(:,node)
end if

if(elem_type(elems_movable(close_elem)) .eq. 3) then
direction(:)=vertex1(:)-vertex2(:)
call cross_product(direction, vertex3-vertex2)
call normalize(direction)
else
    node=to_nodes(4,elems_movable(close_elem))
    if(node_is_movable(node)) then
        vertex4(:)=original_pos(:,inv_nodes_movable(node))
    else
        vertex4(:)=pos_nodes(:,node)
    end if
    direction(:)=0.5*(vertex1+vertex2-vertex3-vertex4)
call cross_product(direction, 0.5*(vertex1+vertex4-vertex2-vertex3))
call normalize(direction)
end if

pos_nodes(:,nodes_movable(mov_node))=pos_nodes(:,nodes_movable(mov_node))+t *direction
else
    t=dot_product(pos_nodes(:,nodes_movable(mov_node))-original_pos(:,1), &
    & pos_nodes(:,nodes_movable(mov_node))-original_pos(:,1))
    close_elem=1
    do i=2, n_nodes_movable
        l=dot_product(pos_nodes(:,nodes_movable(mov_node))-original_pos(:,i), &
        & pos_nodes(:,nodes_movable(mov_node))-original_pos(:,i))
        if(l .lt. t) then
            t=l
close_elem=i
        end if
    end do
    pos_nodes(:,nodes_movable(mov_node))=original_pos(:,close_elem)
end if

end subroutine place_on_surface

**************************************************************************
**
subroutine calc_internal_loads()

use variables
implicit none

double precision :: displ_vec(8), stiff_force_vec(8), stiff_aux_vec(3)
double precision :: velocity_vec(8), velocity_force_vec(8),
velocity_aux_vec(3)
double precision :: damping_store (n_nodes_movable)
double precision :: strain_rate(3), stress_rate(3)
integer :: i, j, nodes_num, node_mov
internal_loads(:,:,)=0.
damping_loads(:,:,)=0.
stiff_aux_vec(3)=0.

do i=1, n_elems_movable
    nodes_num= elem_type(elems_movable(i))
    do j=1, nodes_num
        if(node_is_movable(to_nodes(j,elems_movable(i)))) then
            node_mov=inv_nodes_movable(to_nodes(j,elems_movable(i)));
            displ_vec(2*j-1:2*j)=local_coordinates_node(1:2,j,elems_movable(i))-
            store_local_coordinates_node(1:2,j,i) !In-plane components for displacement
            in local axes is taken into account
            velocity_aux_vec(:,:,)=matmul(local_axes(:,:,1:3,elems_movable(i)),node_vel(:,node_mov)+0.5*time_step*node_acc(:,node_mov)) !Velocity is transferred to
            local axes. Velocity at moment of loads calculation is used
            velocity_vec(2*j-1:2*j)=velocity_aux_vec(1:2) !in-plane components
            are taken into account
        else
            displ_vec(2*j-1:2*j)=0.
            velocity_vec(2*j-1:2*j)=0.
        end if
    end do
end do

strain(:,to_gauss_points(1,i))=matmul(strain_mat(:,1:2*nodes_num,to_gauss_points(1,i)),displ_vec(1:2*nodes_num))
stress(:,to_gauss_points(1,i))=matmul(d(:,:,),strain(:,to_gauss_points(1,i)))
call
check_compression(stress(:,to_gauss_points(1,i)),strain(:,to_gauss_points(1,i)))

strain_rate(:,:,)=matmul(strain_mat(:,:,1:2*nodes_num,to_gauss_points(1,i)),velocity_vec(1:2*nodes_num))
stress_rate(:,:,)=matmul(d(:,:,),strain_rate(:,:,))
if(nodes_num .eq. 3) then
    stiff_force_vec(1:2*nodes_num)= &
    & 0.5*det_jac(1,n_elems+i)*matmul(transpose(strain_mat(:,:,1:2*nodes_num,to_gauss_points(1,i))), &
    & stress(:,to_gauss_points(1,i)))
    & & stress_rate(:,:,)
    velocity_force_vec(1:2*nodes_num)= &
    & 0.5*det_jac(1,n_elems+i)*matmul(transpose(strain_mat(:,:,1:2*nodes_num,to_gauss_points(1,i))), &
    & stress(:,to_gauss_points(1,i)))
    & & stress_rate(:,:,)
else
    stiff_force_vec(1:2*nodes_num)= &
    & det_jac(1,n_elems+i)*matmul(transpose(strain_mat(:,:,1:2*nodes_num,to_gauss_points(1,i))), &
    & stress(:,to_gauss_points(1,i)))
    & & stress_rate(:,:,)
    velocity_force_vec(1:2*nodes_num)= &
    & det_jac(1,n_elems+i)*matmul(transpose(strain_mat(:,:,1:2*nodes_num,to_gauss_points(1,i))),)
    & & stress_rate(:,:,)
end if

165
strain(:,to_gauss_points(j,i)) = matmul(strain_mat(:,1:2*nodes_num,to_gauss_points(j,i)), displ_vec(1:2*nodes_num))

stress(:,to_gauss_points(j,i)) = matmul(d(:,,:), strain(:,to_gauss_points(j,i)) )

  call check_compression(stress(:,to_gauss_points(j,i)), strain(:,to_gauss_points(j,i)))

  stiff_force_vec(1:2*nodes_num) = stiff_force_vec(1:2*nodes_num) +
  & det_jac(j,n_elems+i)*matmul(transpose(strain_mat(:,1:2*nodes_num,to_gauss_points(j,i))),
  & stress(:,to_gauss_points(j,i)))

strain_rate(:) = matmul(strain_mat(:,1:2*nodes_num,to_gauss_points(j,i)), velocity_vec(1:2*nodes_num))

stress_rate(:) = matmul(d(:,,:), strain_rate(:))

velocity_force_vec(1:2*nodes_num) = velocity_force_vec(1:2*nodes_num) +
  & det_jac(j,n_elems+i)*matmul(transpose(strain_mat(:,1:2*nodes_num,to_gauss_points(j,i))),
  & stress_rate(:))

end do

end if

stiff_force_vec(1:2*nodes_num) = stiff_force_vec(1:2*nodes_num)*thick

velocity_force_vec(1:2*nodes_num) = velocity_force_vec(1:2*nodes_num)*thick*beta_damping

velocity_aux_vec(3) = 0.

  do j=1, nodes_num
    if(node_is_movable(to_nodes(j,elems_movable(i)))) then
      node_mov = inv_nodes_movable(to_nodes(j,elems_movable(i)))
      stiff_aux_vec(1:2) = stiff_force_vec(2*j-1:2*j)
      internal_loads(:,node_mov) = internal_loads(:,node_mov) -
      matmul(transpose(local_axes(1:3,:,elems_movable(i))), stiff_aux_vec)
      velocity_aux_vec(1:2) = velocity_force_vec(2*j-1:2*j)
      damping Loads(:,node_mov) = damping loads(:,node_mov) -
      matmul(transpose(local_axes(1:3,:,elems_movable(i))), velocity_aux_vec)
    end if
  end do

end do

if(complete_mass) then
  call
  skyline_product(fem_mass_mat,fem_mass_mat_pointers, node_vel(1,:)+0.5*time_step*node_acc(1,:), &
  & n_nodes_movable, damping_store)
  damping loads(1,:) = damping loads(1,:) - alpha_damping*damping_store

  call
  skyline_product(fem_mass_mat,fem_mass_mat_pointers, node_vel(2,:)+0.5*time_step*node_acc(2,:), &
  & n_nodes_movable, damping_store)
  damping loads(2,:) = damping loads(2,:) - alpha_damping*damping_store
call 
skyline_product(fem_mass_mat,fem_mass_mat_pointers,node_vel(3,:),+0.5*time_step*node_acc(3,:), & 
& n_nodes_movable, damping_store) 
damping_loads(3,:) = damping_loads(3,:) - alpha_damping*damping_store
else 
  do i=1, n_nodes_movable 
     damping_loads(:,i) = damping_loads(:,i) - alpha_damping/fem_mass_mat(i)*(node_vel(:,i)+0.5*time_step*node_acc(:,i)) 
  end do 
end if 
end subroutine calc_internal_loads 
**************************************************************************
**
subroutine check_compression(stress, strain)
use variables, only: young 
implicit none 
double precision, intent(inout) :: stress(3) 
double precision, intent(in) :: strain(3) 
double precision :: angle, centre, radius, centre_strain, radius_strain

centre=0.5*(stress(1)+stress(2)) !Centre of the Mohr circle 
radius=0.5*sqrt((stress(1)-stress(2))*(stress(1)-stress(2)))+4*stress(3)*stress(3)
if(centre-radius .lt. 0) then 
  One principal stress is compressive 
  if(centre+radius .lt. 0) then 
    Both principal stresses are null 
  stress(:)=0. 
  else 
    centre_strain=0.5*(strain(1)+strain(2)) !Centre of the Mohr circle 
    radius_strain=0.5*sqrt((strain(1)-strain(2))*(strain(1)-strain(2)))+4*strain(3)*strain(3)
    angle=atan2(stress(3),stress(1)-centre) !Angle of rotation of the reference system 
    centre=0.5*(young*centre_strain+radius_strain) !Mohr circle is redefine to be tangent to sigma=0 and have the other stress = epsilonI*E (without poisson terme) 
    radius=centre 
    stress(3)=radius*sin(angle) !Tensions at the original reference 
    system of the new circle 
    stress(1)=centre+radius*cos(angle) 
    stress(2)=2*centre-stress(1) 
  end if 
end if 
end subroutine check_compression 
**************************************************************************
**
subroutine output(i)
use variables 
implicit none 
double precision :: i 
call mesh_output() 
call result_output(i) 
end subroutine output
**

subroutine mesh_output()

use variables
integer :: i, j, k, l, last_node
logical :: normal

open(UNIT=4, FILE=msh, STATUS='REPLACE', ACTION='WRITE',
ACCESS='SEQUENTIAL')

write(4,'(A)') 'MESH "Skin" dimension 3 Elemtype Quadrilateral Nnode 4'
write(4,'(A)') 'Coordinates'
write(4,'(A)') '# Node number, node x, node y, node z'
do i=1, n_nodes
  normal=.true.
  if(solver_to_use .ne. 1) then
    if(node_is_movable(i)) normal=.false.
  end if
  if(.not. normal) then
    write(4, '(I5.1,ES20.7,ES20.7,ES20.7)') i,
    original_pos(:,inv_nodes_movable(i))
  else
    write(4, '(I5.1,ES20.7,ES20.7,ES20.7)') i, pos_nodes(:,i)
  end if
end do
write(4,'(A)') 'End coordinates'
write(4,'(A)') 'Elements'
write(4,'(A)') '# Element number, node 1, node 2, node 3, node 4'
do i=1, n_elems
  if(elem_type(i) .eq. 4) then
    write(4, '(I5, I7, I7, I7, I7)') i, to_nodes(:,i)
  else
    write(4, '(I5, I7, I7, I7, I7)') i, to_nodes(elem_type(i),i),
    to_nodes(elem_type(i),i)
  end if
end do
write(4, '(A)') 'End elements'

write(4,'(A)') 'MESH "Wake" dimension 3 Elemtype Linear Nnode 2'
write(4,'(A)') 'Coordinates'
write(4,'(A)') '# Node number, node x, node y, node z'
k=n_nodes
do i=1, n_wake_strings
  do j=1, n_wake_segments !2
    write(4, '(I5.1,ES20.7,ES20.7,ES20.7)') k, wake_strings_nodes(:,j,i)
  end do
  k=k+1
end do
last_node=k
write(4,'(A)') 'End coordinates'
write(4,'(A)') 'Elements'
write(4,'(A)') '# Element number, node 1, node 2'
k=0
l=n_nodes
do i=1, n_wake_strings
  do j=1, n_wake_segments-1 !1
    k=k+1
    l=l+1
    write(4,'(I8, I9, I9)') k, l, l+1
  end do
  l=l+1
end do
write (4,'(A)') 'End elements'
write(4,'(A)') 'MESH "Force centre" dimension 3 Elemtype  Linear Nnode 2'
write(4,'(A)') 'Coordinates'
write(4,'(A)') '# Node number, node x, node y, node z'
write(4,'(I5.1,ES20.7,ES20.7,ES20.7)') last_node+1, force_centre(:)
write(4,'(I5.1,ES20.7,ES20.7,ES20.7)') last_node+2, force_centre(:)
write (4,'(A)') 'End coordinates'
write (4,'(A)') 'Elements'
write (4,'(A)') '# Element number, node 1, node 2'
write(4,'(I8, I9, I9)') 1, last_node+1, last_node+2
write(4,'(I8, I9, I9)') 2, last_node+1, last_node+2
write (4,'(A)') 'End elements'
close(4)
end subroutine mesh_output

**************************************************************************
**
subroutine result_output(m)
use variables
implicit none
integer :: i, j, k, l
double precision :: m, matrix(3,3)
write(5,'(A, ES10.4, A)') 'Result "Source Intensity" "Time Step'',m,' Scalar OnGaussPoints "Panels Centre"'
write(5,'(A)') 'Values'
do i = 1, n elems
  write(5,'(I8, ES20.7)') i, source(i)
end do ! i
write(5,'(A)') 'End Values'
write(5,'(A, ES10.4, A)') 'Result "Doublet Intensity (Elemental Solution)" "Time Step'',m, &
  & Scalar OnGaussPoints "Panels Centre"'
write(5,'(A)') 'Values'
do i = 1, n elems
  write(5,'(I8, ES20.7)') i, -4*pi*doublet(i)
end do ! i
write(5,'(A)') 'End Values'
write(5,*)

169
write(5,'(A, ES10.4, A)') 'Result "Doublet Intensity (Nodal Solution)"
           "Time Step",m,' Scalar OnNodes'
write(5,'(A)') 'Values'
           do i = 1, n_nodes
write(5,'(I8, ES20.7)') i, -4*pi*doublet_nodes(i)
           end do
write(5,'(A)') 'End Values'
write(5,*)

if (solver_to_use .ne. 1) then
write(5,'(A, ES10.4, A)') 'Result "Pressure Nodal Forces" "Time
Step"',m,' Vector OnNodes'
write(5,'(A)') 'Values'
           do i=1, n_nodes_movable
write(5,'((I5.1,ES20.7,ES20.7,ES20.7,ES20.7)') nodes_movable(i),
loads(:,i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Stiffness Nodal Forces" "Time
Step"',m,' Vector OnNodes'
write(5,'(A)') 'Values'
           do i=1, n_nodes_movable
write(5,'((I5.1,ES20.7,ES20.7,ES20.7,ES20.7)') nodes_movable(i),
internal_loads(:,i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Displacement Vector" "Time
Step"',m,' Vector OnNodes'
write(5,'(A)') 'Values'
           do i=1,n_nodes_movable
write(5,'((I8,ES20.7,ES20.7,ES20.7)') nodes_movable(i),
pos_nodes(:,nodes_movable(i))-original_pos(:,i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Air Velocity" "Time Step"',m,
Vector OnGaussPoints "Panels Centre"
write(5,'(A)') 'Values'
           do i=1, n_elems
write(5,'(I8,ES20.7,ES20.7,ES20.7)') i, velocity_on elems(:,i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Cp" "Time Step"',m,' Scalar
OnGaussPoints "Panels"
write(5,'(A)') 'Values'
           do i=1, n elems
write(5,'(I8,ES20.7)') i, cp(1,i)
write(5,'(ES20.7)') cp(2,i)
write(5,'(ES20.7)') cp(3,i)
write(5,'(ES20.7)') cp(4,i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
end if
write(5,'(A, ES10.4, A)') 'Result "Wake Forces" "Time Step",m,' Vector
OnGaussPoints "Panels"
write(5,'(A)') 'Values'
           do i=1, n_arrays
write(5,'(I8,ES20.7,ES20.7,ES20.7)') i, arrays(i)
           end do
write(5,'(A)') 'End Values'
write(5,*)
if(n_wake_strings .gt. 1) then
write(5,'(A, ES10.4, A)') 'Result "Wake Intensity" "Time Step"',m,'
Scalar OnGaussPoints "Wake segment centre"'
write(5,'(A)') 'Values'
k=0
l=0
do i=1, n_wake_strings 
do j=1, n_wake_segments-1 !1 
k=k+1
l=l+1
write(5,'(I8, ES20.7)') k, 4*pi*wake_intensities(i)
end do 
l=l+1
end do
write(5,'(A)') 'End Values'
end if
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Total Non-dimensional Force" "Time Step"',m,'
Vector OnGaussPoints "Force cent"'
write(5,'(A)') 'Values'
write(5,'(I8,ES20.7,ES20.7,ES20.7)') 1, aerodyn_force
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "Aerodynamic Coefficients" "Time Step"',m,'
Vector OnGaussPoints "Force cent"'
write(5,'(A)') 'Values'
write(5,'(I8,ES20.7,ES20.7,ES20.7)') 1, CL*lift_dir
write(5,'(I8,ES20.7,ES20.7,ES20.7)') 2, CD*velocity
write(5,'(A)') 'End Values'
write(5,*)
if(m.gt.0) then 
write(5,'(A, ES10.4, A)') 'Result "strain" "Time Step"',m,'
Matrix OnGaussPoints "Panels"'
write(5,'(A)') 'Values'
do i=1, n elems_movable 
matrix(1,1)=strain(1,to_gauss_points(1,i))
matrix(2,2)=strain(2,to_gauss_points(1,i))
matrix(3,3)=-poisson*(matrix(1,1)+matrix(2,2))
matrix(1,2)=0.5*strain(3,to_gauss_points(1,i))
matrix(2,1)=matrix(1,2)
matrix(1,3)=0.;matrix(2,3)=0.;matrix(3,1)=0.;matrix(3,2)=0.
matrix(:, :)=matmul(local_axes(1:3,:,elems_movable(i))),matmul(matrix,transpose(local_axes(1:3,:,elems_movable(i)))))
elems_movable(i), matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
if (elem_type(elems_movable(i)).eq. 3) then 
matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3),

matrix(1,3)
else
  do j=2, 4
    matrix(1,1)=strain(1,to_gauss_points(j,i))
    matrix(2,2)=strain(2,to_gauss_points(j,i))
    matrix(3,3)=-poisson*(matrix(1,1)+matrix(2,2))
    matrix(1,2)=0.5*strain(3,to_gauss_points(j,i))
    matrix(2,1)=matrix(1,2)
    matrix(1,3)=0.;matrix(2,3)=0.;matrix(3,1)=0.;matrix(3,2)=0.
  end do
end if
end do

write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)') matrix(1,1), &
& matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3),
matrix(1,3)
end do
write(5,'(A)') 'End Values'
write(5,*)
write(5,'(A, ES10.4, A)') 'Result "stress" "Time Step",m,' Matrix
  OnGaussPoints "Panels"'
write(5,'(A)') 'Values'
do i=1, n elems movable
  matrix(1,1)=stress(1,to_gauss_points(1,i))
  matrix(2,2)=stress(2,to_gauss_points(1,i))
  matrix(3,3)=0.
  matrix(1,2)=stress(3,to_gauss_points(1,i))
  matrix(2,1)=matrix(1,2)
  matrix(1,3)=0.;matrix(2,3)=0.;matrix(3,1)=0.;matrix(3,2)=0.
  matrix(:, :) = matmul(local axes(1:3,:), elems movable(i)),
  matmul(matrix, transpose(local axes(1:3,:), elems movable(i))))
  'OnGaussPoints "Panels"'
  write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)')
  matrix(1,1), &
  & matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
  write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)')
  matrix(1,1), &
  & matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
  write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)')
  matrix(1,1), &
  & matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
  if (elem type(elems movable(i) .eq. 3) then
    'OnGaussPoints "Panels"'
    write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)')
    matrix(1,1), &
    & matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
    write(5,'(ES20.7, ES20.7, ES20.7, ES20.7, ES20.7)')
    matrix(1,1), &
    & matrix(2,2), matrix(3,3), matrix(1,2), matrix(2,3), matrix(1,3)
    else
      do j=2, 4
        matrix(1,1)=stress(1,to_gauss_points(j,i))
        matrix(2,2)=stress(2,to_gauss_points(j,i))
        matrix(3,3)=0.
        matrix(1,2)=stress(3,to_gauss_points(j,i))
        matrix(2,1)=matrix(1,2)
        matrix(1,3)=0.;matrix(2,3)=0.;matrix(3,1)=0.;matrix(3,2)=0.
matrix(:, :) = matmul(local_axes(1:3, :, elems_movable(i)), matmul(matrix, transpose(local_axes(1:3, :, elems_movable(i)))))
matrix(1, 1), &
& matrix(2, 2), matrix(3, 3), matrix(1, 2), 0., 0.
end do
end if
end do
write(5, '(A)')  'End Values'
write(5, *)
end if
end subroutine result_output
function adj_bandwidth ( node_num, adj_num, adj_row, adj )

!**************************************************************************
***80
!
!! ADJ_BANDWIDTH computes the bandwidth of an adjacency matrix.
!!
! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
! Modified:
!!
!! 11 March 2005
!!
! Author:
!!
!! John Burkardt
!!
! Author:
!!
!! Original FORTRAN77 version by Alan George, Joseph Liu.
!! FORTRAN90 version by John Burkardt.
!!
! Reference:
!!
!! Alan George, Joseph Liu,
!! Computer Solution of Large Sparse Positive Definite Systems,
!! Prentice Hall, 1981.
!!
! Parameters:
!!
!! Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
!! Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.
!! Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1).  Information about
!! row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.
!! Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.
!! For each row, it contains the column indices of the nonzero entries.
!! Output, integer ( kind = 4 ) ADJ_BANDWIDTH, the bandwidth of the
!! adjacency
!! matrix.
!!
implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_bandwidth
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) band_hi
integer ( kind = 4 ) band_lo
integer ( kind = 4 ) col
integer ( kind = 4 ) i
integer ( kind = 4 ) j

band_lo = 0
band_hi = 0

do i = 1, node_num
   do j = adj_row(i), adj_row(i+1) - 1
      col = adj(j)
      band_lo = max ( band_lo, i - col )
      band_hi = max ( band_hi, col - i )
   end do
end do

adj_bandwidth = band_lo + 1 + band_hi

return
end

function adj_contains_ij ( node_num, adj_num, adj_row, adj, i, j )

**************************************************************************
!! ADJ_CONTAINS_IJ determines if (I,J) is in an adjacency structure.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 23 October 2003
!! Author:
!! John Burkardt
!! Parameters:
!! Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
!! Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.
!! Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1).  Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.
!! Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.
!! Input, integer ( kind = 4 ) I, J, the two nodes, for which we want to know whether I is adjacent to J.
!! Output, logical ADJ_CONTAINS_IJ, is TRUE if I = J, or the adjacency structure contains the information that I is adjacent to J.
!! implicit none
integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
logical adj_contains_ij
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) k
integer ( kind = 4 ) khi
integer ( kind = 4 ) klo

! Symmetric entries are not stored.
!
if ( i == j ) then
  adj_contains_ij = .true.
  return
end if
!
! Illegal I, J entries.
!
if ( node_num < i ) then
  adj_contains_ij = .false.
  return
else if ( i < 1 ) then
  adj_contains_ij = .false.
  return
else if ( node_num < j ) then
  adj_contains_ij = .false.
  return
else if ( j < 1 ) then
  adj_contains_ij = .false.
  return
end if
!
! Search the adjacency entries already stored for row I,
! to see if J has already been stored.
!
klo = adj_row(i)
khi = adj_row(i+1)-1

  do k = klo, khi

    if ( adj(k) == j ) then
      adj_contains_ij = .true.
      return
    end if
  end do

end subroutine adj_insert_ij ( node_num, adj_max, adj_num, adj_row, adj, i, j )

!**************************************************************************
!
!! ADJ_INSERT_IJ inserts (I,J) into an adjacency structure.
!! Licensing:
!!  This code is distributed under the GNU LGPL license.
!! Modified:
!!  02 January 2007
!! Author:
!!  John Burkardt
!! Parameters:
!!  Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
!!  Input, integer ( kind = 4 ) ADJ_MAX, the maximum number of adjacency
!!    entries.
!!  Input/output, integer ( kind = 4 ) ADJ_NUM, the number of adjacency
!!    entries.
!!  Input/output, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1).  Information
!!    about
!!    row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.
!!  Input/output, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency
!!    structure.
!!  Input, integer ( kind = 4 ) I, J, the two nodes which are adjacent.
!
use variables, only:err
implicit none

integer ( kind = 4 ) adj_max
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_max)
integer ( kind = 4 ) adj_num
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) j_spot
integer ( kind = 4 ) k
!
! A new adjacency entry must be made.
! Check that we're not exceeding the storage allocation for ADJ.
!
if ( adj_max < adj_num + 1 ) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write ( 7, '(a)' ) ' '
  write ( 7, '(a)' ) 'ADJ_INSERT_IJ - Fatal error!'
  write ( 7, '(a)' ) ' All available storage has been used.'
  write ( 7, '(a)' ) ' No more information can be stored!'
  write ( 7, '(a)' ) ' This error occurred for '
  write ( 7, '(a,i8)' ) ' Row I = ', i
  write ( 7, '(a,i8)' ) ' Column J = ', j
close(7)
stop "Problem at nodes reordering"
end if

! The action is going to occur between ADJ_ROW(I) and ADJ_ROW(I+1)-1:
!
j_spot = adj_row(i)
do k = adj_row(i), adj_row(i+1) - 1

if ( adj(k) == j ) then
    return
else if ( adj(k) < j ) then
    j_spot = k + 1
else
    exit
end if
end do

adj(j_spot+1:adj_num+1) = adj(j_spot:adj_num)
adj(j_spot) = j

adj_row(i+1:node_num+1) = adj_row(i+1:node_num+1) + 1
adj_num = adj_num + 1

return
end

function adj_perm_bandwidth ( node_num, adj_num, adj_row, adj, perm, perm_inv )

**************************************************************************
!! ADJ_PERM_BANDWIDTH computes the bandwidth of a permuted adjacency matrix.
!!
!! Discussion:
!!    The matrix is defined by the adjacency information and a permutation.
!!    The routine also computes the bandwidth and the size of the envelope.
!!
!! Licensing:
!!    This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!    11 March 2005
!!
!! Author:
!!    John Burkardt
!!
!! Reference:
!!    Alan George, Joseph Liu,
!!    Computer Solution of Large Sparse Positive Definite Systems,
Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.

Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.

Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure. For each row, it contains the column indices of the nonzero entries.

Input, integer ( kind = 4 ) PERM(NODE_NUM), PERM_INV(NODE_NUM), the permutation and inverse permutation.

Output, integer ( kind = 4 ) ADJ_PERM_BANDWIDTH, the bandwidth of the permuted adjacency matrix.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_perm_bandwidth
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) band_hi
integer ( kind = 4 ) band_lo
integer ( kind = 4 ) col
integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) perm(node_num)
integer ( kind = 4 ) perm_inv(node_num)

band_lo = 0
band_hi = 0

do i = 1, node_num
    do j = adj_row(perm(i)), adj_row(perm(i)+1) - 1
        col = perm_inv(adj(j))
        band_lo = max ( band_lo, i - col )
        band_hi = max ( band_hi, col - i )
    end do
end do

adj_perm_bandwidth = band_lo + 1 + band_hi

return
end subroutine adj_perm_show ( node_num, adj_num, adj_row, adj, perm, perm_inv )

**************************************************************************
***80
ADJ_PERM_SHOW displays a symbolic picture of a permuted adjacency matrix.

Discussion:

The matrix is defined by the adjacency information and a permutation.
The routine also computes the bandwidth and the size of the envelope.
If no permutation has been done, you must set PERM(I) = PERM_INV(I) = I
before calling this routine.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

28 October 2003

Author:

John Burkardt

Reference:

Alan George, Joseph Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice Hall, 1981.

Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.
Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.
Input, integer ( kind = 4 ) ADJ_NUM, the adjacency structure.
For each row, it contains the column indices of the nonzero entries.
Input, integer ( kind = 4 ) PERM(NODE_NUM), PERM_INV(NODE_NUM), the permutation and inverse permutation.

implicit none

integer ( kind = 4 ), parameter :: n_max = 100

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
character band(n_max)
integer ( kind = 4 ) band_lo
integer ( kind = 4 ) col
integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) k
integer ( kind = 4 ) nonzero_num
integer ( kind = 4 ) perm(node_num)
integer ( kind = 4 ) perm_inv(node_num)

band_lo = 0
nonzero_num = 0

if ( n_max < node_num ) then
    write (*, '(a)') '
    write (*, '(a)') 'ADJ_PERM_SHOW - Fatal error!
    write (*, '(a)') ' NODE_NUM is too large!
    write (*, '(a,i8)') ' Maximum legal value is ', n_max
    write (*, '(a,i8)') ' Your input value was ', node_num
    stop
end if

write (*, '(a)') '
write (*, '(a)') ' Nonzero structure of matrix:
write (*, '(a)') '

do i = 1, node_num
    do k = 1, node_num
        band(k) = '.'
    end do
    band(i) = 'D'
    do j = adj_row(perm(i)), adj_row(perm(i)+1) - 1
        col = perm_inv(adj(j))
        if ( col < i ) then
            nonzero_num = nonzero_num + 1
        end if
        band_lo = max ( band_lo, i - col )
        if ( col /= i ) then
            band(col) = 'X'
        end if
    end do
    write ( *, '(2x,i8,1x,100a1)' ) i, band(1:node_num)
end do

write ( *, '(a)') '
write ( *, '(a,i8)' ) ' Lower bandwidth = ', band_lo
write ( *, '(a,i8,a)' ) ' Lower envelope contains ', &
    nonzero_num, ' nonzeros.'
return
end subroutine adj_print ( node_num, adj_num, adj_row, adj, title )

**************************************************************************
***80
ADJ_PRINT prints adjacency information.

Discussion:

The list has the form:

<table>
<thead>
<tr>
<th>Row</th>
<th>Nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 5 9</td>
</tr>
<tr>
<td>2</td>
<td>7 8 9 15 78 79 81 86 91 99</td>
</tr>
<tr>
<td>3</td>
<td>48 49 53</td>
</tr>
</tbody>
</table>

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

18 December 2002

Author:

John Burkardt

Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.
Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1), organizes the adjacency entries into rows. The entries for row I are in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1.
Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure, which contains, for each row, the column indices of the nonzero entries.
Input, character ( len = * ) TITLE, a title to be printed.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
character ( len = * ) title

call adj_print_some ( node_num, 1, node_num, adj_num, adj_row, adj, title )

return
end

subroutine adj_print_some ( node_num, node_lo, node_hi, adj_num, adj_row, adj, title )
** ADJ_PRINT_SOME prints some adjacency information.

Discussion:

The list has the form:

Row  Nonzeros
1  2  5  9
2  7  8  9  15  78  79  81  86  91  99
     100 103
3  48  49  53

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

18 December 2002

Author:

John Burkardt

Parameters:

Input, integer (kind = 4) NODE_NUM, the number of nodes.

Input, integer (kind = 4) NODE_LO, NODE_HI, the first and last nodes
for which the adjacency information is to be printed.

Input, integer (kind = 4) ADJ_NUM, the number of adjacency entries.

Input, integer (kind = 4) ADJ_ROW(NODE_NUM+1), organizes the adjacency
entries into rows. The entries for row I are in entries ADJ_ROW(I)
through ADJ_ROW(I+1)-1.

Input, integer (kind = 4) ADJ(ADJ_NUM), the adjacency structure, which
contains, for each row, the column indices of the nonzero entries.

Input, character (len = *) TITLE, a title to be printed.

** implicit none

integer (kind = 4) adj_num
integer (kind = 4) node_num

integer (kind = 4) adj(adj_num)
integer (kind = 4) adj_row(node_num+1)
integer (kind = 4) i
integer (kind = 4) jhi
integer (kind = 4) jlo
integer (kind = 4) jmax
integer ( kind = 4 ) jmin
integer ( kind = 4 ) node_hi
integer ( kind = 4 ) node_lo
character ( len = * ) title

if ( 0 < len_trim ( title ) ) then
  write ( *, '(a)' ) ' ' trim ( title )
end if

write ( *, '(a)' ) ' Sparse adjacency structure:'
write ( *, '(a)' ) ' Number of nodes       = ', node_num
write ( *, '(a)' ) ' Number of adjacencies = ', adj_num
write ( *, '(a)' ) ' Node Min Max       Nonzeros '
write ( *, '(a)' ) ' '

do i = node_lo, node_hi
  jmin = adj_row(i)
  jmax = adj_row(i+1) - 1
  if ( jmax < jmin ) then
    write ( *, '(2x,3i4)' ) i, jmin, jmax
  else
    do jlo = jmin, jmax, 5
      jhi = min ( jlo + 4, jmax )
      if ( jlo == jmin ) then
        write ( *, '(2x,3i4,3x,5i8)' ) i, jmin, jmax, adj(jlo:jhi)
      else
        write ( *, '(2x,12x,3x,5i8)' ) adj(jlo:jhi)
      end if
    end do
  end if
end do

return
end subroutine adj_set ( node_num, adj_max, adj_num, adj_row, adj, irow, jcol )

!**************************************************************************
!! ADJ_SET sets up the adjacency information.
!!
!! Discussion:
!!
!! The routine records the locations of each nonzero element,
!! one at a time.
!!
The first call for a given problem should be with IROW or ICOL negative. This is a signal indicating the data structure should be initialized.

Then, for each case in which A(IROW,JCOL) is nonzero, or in which IROW is adjacent to JCOL, call this routine once to record that fact.

Diagonal entries are not to be stored.

The matrix is assumed to be symmetric, so setting I adjacent to J will also set J adjacent to I.

Repeated calls with the same values of IROW and JCOL do not actually hurt. No extra storage will be allocated.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

23 October 2003

Author:

John Burkardt

Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

Input, integer ( kind = 4 ) ADJ_MAX, the maximum dimension of the adjacency array.

Input/output, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.

Input/output, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.

Input/output, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.

Input, integer ( kind = 4 ) IROW, JCOL, the row and column indices of a nonzero entry of the matrix.

use variables, only: err
implicit none

integer ( kind = 4 ) adj_max
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_max)
logical adj_contains_ij
integer ( kind = 4 ) adj_num
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) irow
integer ( kind = 4 ) jcol
!
! Negative IROW or JCOL indicates the data structure should be initialized.
!
if ( irow < 0 .or. jcol < 0 ) then
 !
  write ( *, '(a)' ) ' '  
  write ( *, '(a)' ) 'ADJ_SET - Note:'
  write ( *, '(a)' ) ' Initializing adjacency information.'
  write ( *, '(a,i8)' ) ' Number of nodes NODE_NUM = ', node_num
  write ( *, '(a,i8)' ) ' Maximum adjacency ADJ_MAX = ', adj_max

  adj_num = 0
  adj_row(1:node_num+1) = 1
  adj(1:adj_max) = 0

  return
end if
!
! Diagonal entries are not stored.
!
if ( irow == jcol ) then
  return
end if
!
!
if ( node_num < irow ) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write ( 7, '(a)' ) ' '  
  write ( 7, '(a)' ) 'ADJ_SET - Fatal error!'
  write ( 7, '(a)' ) ' NODE_NUM < IROW.'
  write ( 7, '(a,i8)' ) ' IROW = ', irow
  write ( 7, '(a,i8)' ) ' NODE_NUM = ', node_num
  close(7)
  stop "Problem at nodes reordering"
else if ( irow < 1 ) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write ( 7, '(a)' ) ' '  
  write ( 7, '(a)' ) 'ADJ_SET - Fatal error!'
  write ( 7, '(a)' ) ' IROW < 1.'
  write ( 7, '(a,i8)' ) ' IROW = ', irow
  close(7)
  stop "Problem at nodes reordering"
else if ( node_num < jcol ) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write ( 7, '(a)' ) ' '  
  write ( 7, '(a)' ) 'ADJ_SET - Fatal error!'
  write ( 7, '(a)' ) ' NODE_NUM < JCOL.'
  write ( 7, '(a,i8)' ) ' JCOL = ', jcol
  write ( 7, '(a,i8)' ) ' NODE_NUM = ', node_num
  close(7)
  stop "Problem at nodes reordering"
else if ( jcol < 1 ) then
  open(UNIT=7,FILE=err,ACTION="WRITE", ACCESS="SEQUENTIAL")
  write ( 7, '(a)' ) ' '  
  write ( 7, '(a)' ) 'ADJ_SET - Fatal error!'
  write ( 7, '(a)' ) ' JCOL < 1.'
  write ( 7, '(a,i8)' ) ' JCOL = ', jcol
  close(7)
  stop "Problem at nodes reordering"
if (.not. &
  adj_contains_ij ( node_num, adj_num, adj_row, adj, irow, jcol ) ) then
  call adj_insert_ij ( node_num, adj_max, adj_num, adj_row, adj, irow,
  jcol )
end if
if (.not. &
  adj_contains_ij ( node_num, adj_num, adj_row, adj, jcol, irow ) ) then
  call adj_insert_ij ( node_num, adj_max, adj_num, adj_row, adj, jcol,
  irow )
end if
return
end

subroutine adj_show ( node_num, adj_num, adj_row, adj )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
**80
!! ADJ_SHOW displays a symbolic picture of an adjacency matrix.
!!
!! Discussion:
!!    The matrix is defined by the adjacency information and a permutation.
!!    The routine also computes the bandwidth and the size of the envelope.
!!
!! Licensing:
!!    This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!    11 March 2005
!!
!! Author:
!!    John Burkardt
!!
!! Reference:
!!    Alan George, Joseph Liu,
!!    Computer Solution of Large Sparse Positive Definite Systems,
!!    Prentice Hall, 1981.
!!
!! Parameters:
!!    Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
!!    Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.
!!    Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.
!!    Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.
!! For each row, it contains the column indices of the nonzero entries.
implicit none

integer ( kind = 4 ), parameter :: n_max = 100

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
character band(n_max)
integer ( kind = 4 ) band_lo
integer ( kind = 4 ) col
integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) k
integer ( kind = 4 ) nonzero_num

band_lo = 0
nonzero_num = 0

if ( n_max < node_num ) then
  write ( *, '(a)' ) 'ADJ_SHOW - Fatal error!'
  write ( *, '(a)' ) '  NODE_NUM is too large!'
  write ( *, '(a,i8)' ) '  Maximum legal value is ', n_max
  write ( *, '(a,i8)' ) '  Your input value was   ', node_num
  stop
end if

write ( *, '(a)' ) '  Nonzero structure of matrix:'
write ( *, '(a)' ) '  D

do i = 1, node_num
  do k = 1, node_num
    band(k) = '.'
  end do
  band(i) = 'D'

  do j = adj_row(i), adj_row(i+1) - 1
    col = adj(j)
    if ( col < i ) then
      nonzero_num = nonzero_num + 1
    end if
    band_lo = max ( band_lo, i-col )
    band(col) = 'X'
  end do

  write ( *, '(2x,i8,1x,100a1)' ) i, band(1:node_num)
end do

write ( *, '(a)' ) ''
write ( *, '(a,i8)' ) '  Lower bandwidth = ', band_lo
write ( *, '(a,i8,a)' ) '  Lower envelope contains ', &
    nonzero_num, ' nonzeros.'
return
end

subroutine degree ( root, adj_num, adj_row, adj, mask, deg, iccsze, ls, &
    node_num )
Output, integer ( kind = 4 ) LS(NODE_NUM), stores in entries 1 through ICCSIZE the nodes in the connected component, starting with ROOT, and proceeding by levels.

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) deg(node_num)
integer ( kind = 4 ) i
integer ( kind = 4 ) iccsze
integer ( kind = 4 ) ideg
integer ( kind = 4 ) j
integer ( kind = 4 ) jstop
integer ( kind = 4 ) jstrt
integer ( kind = 4 ) lbegin
integer ( kind = 4 ) ls(node_num)
integer ( kind = 4 ) lvlend
integer ( kind = 4 ) lvsize
integer ( kind = 4 ) mask(node_num)
integer ( kind = 4 ) nbr
integer ( kind = 4 ) node
integer ( kind = 4 ) root

! The sign of ADJ_ROW(I) is used to indicate if node I has been considered.
ls(1) = root
adj_row(root) = -adj_row(root)
lvlend = 0
iccsze = 1

! LBEGIN is the pointer to the beginning of the current level, and
! LVLEND points to the end of this level.
do
lbegin = lvlend + 1
lvlend = iccsze

! Find the degrees of nodes in the current level,
! and at the same time, generate the next level.
do i = lbegin, lvlend
node = ls(i)
jstrt = -adj_row(node)
jstop = abs ( adj_row(node+1) ) - 1
ideg = 0

do j = jstrt, jstop
nbr = adj(j)
if ( mask(nbr) /= 0 )
    ideg = ideg + 1
if ( 0 <= adj_row(nbr) )
    adj_row(nbr) = -adj_row(nbr)
    iccsze = iccsze + 1
    ls(iccsze) = nbr
end if
end if
end do
deg(node) = ideg
end do
!
! Compute the current level width.
!
lvsize = iccsze - lvlend
!
! If the current level width is nonzero, generate another level.
!
if ( lvsize == 0 )
    exit
end if
end do
!
! Reset ADJ_ROW to its correct sign and return.
!
do i = 1, iccsze
    node = ls(i)
    adj_row(node) = -adj_row(node)
end do
return
end subroutine genrcm ( node_num, adj_num, adj_row, adj, perm )

**************************************************************************
***80
!! GENRCM finds the reverse Cuthill-McKee ordering for a general graph.
!!
!! Discussion:
!!   For each connected component in the graph, the routine obtains
!!   an ordering by calling RCM.
!!
!! Licensing:
!!   This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!   04 January 2003
!!
!! Author:
Original FORTRAN77 version by Alan George, Joseph Liu.
FORTRAN90 version by John Burkardt

Reference:

Alan George, Joseph Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice Hall, 1981.

Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.

Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row $i$ is stored in entries ADJ_ROW($i$) through ADJ_ROW($i+1$)-1 of ADJ.

Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.
For each row, it contains the column indices of the nonzero entries.

Output, integer ( kind = 4 ) PERM(NODE_NUM), the RCM ordering.

Local Parameters:

Local, integer LEVEL_ROW(NODE_NUM+1), the index vector for a level structure. The level structure is stored in the currently unused spaces in the permutation vector PERM.

Local, integer MASK(NODE_NUM), marks variables that have been numbered.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) iccsze
integer ( kind = 4 ) mask(node_num)
integer ( kind = 4 ) level_num
integer ( kind = 4 ) level_row(node_num+1)
integer ( kind = 4 ) num
integer ( kind = 4 ) perm(node_num)
integer ( kind = 4 ) root

mask(1:node_num) = 1
num = 1

do i = 1, node_num

! For each masked connected component...
!
  if ( mask(i) /= 0 ) then

    root = i

192
Find a pseudo-peripheral node ROOT. The level structure found by ROOT_FIND is stored starting at PERM(NUM).

```fortran
    call root_find ( root, adj_num, adj_row, adj, mask, level_num, &
                     level_row, perm(num), node_num )
```

RCM orders the component using ROOT as the starting node.

```fortran
    call rcm ( root, adj_num, adj_row, adj, mask, perm(num), iccsze, &
               node_num )
```

```fortran
    num = num + iccsze
```

We can stop once every node is in one of the connected components.

```fortran
    if ( node_num < num ) then
        return
    end if
```

```fortran
end do
```

```fortran
return
end subroutine graph_01_adj ( node_num, adj_num, adj_row, adj )
```

**************************************************************************
!! GRAPH_01_ADJ returns the adjacency vector for graph 1.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

22 October 2003

Author:

John Burkardt

Reference:

Alan George, Joseph Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice Hall, 1981.

Parameters:

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacencies.
Output, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1), node pointers into ADJ.
Output, integer (kind = 4) ADJ(ADJ_NUM), the adjacency information.

implicit none

integer (kind = 4) adj_num
integer (kind = 4) node_num
integer (kind = 4) adj(adj_num)
integer (kind = 4) adj_row(node_num+1)

adj(1:adj_num) = (/ &
 4, 6, &
 3, 5, 7, 10, &
 2, 4, 5, &
 1, 3, 6, 9, &
 2, 3, 7, &
 1, 4, 7, 8, &
 2, 5, 6, 8, &
 6, 7, &
 4, &
 2 /)

adj_row(1:node_num+1) = (/ 1, 3, 7, 10, 14, 17, 21, 25, 27, 28, 29 /)

return
end

subroutine graph_01_size ( node_num, adj_num )

**************************************************************************
!!! GRAPH_01_ADJ_NUM returns the number of adjacencies for graph 1.
!!!
!!! Licensing:
!!!
!!! This code is distributed under the GNU LGPL license.
!!!
!!! Modified:
!!!
!!! 22 October 2003
!!!
!!! Author:
!!!
!!! John Burkardt
!!!
!!! Reference:
!!!
!!! Alan George, Joseph Liu,
!!! Computer Solution of Large Sparse Positive Definite Systems,
!!! Prentice Hall, 1981.
!!!
!!! Parameters:
!!!
!!! Output, integer (kind = 4) NODE_NUM, the number of items that can
!!! be adjacent.
!!!
!!! Output, integer (kind = 4) ADJ_NUM, the number of adjacencies.
!!! implicit none
integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

node_num = 10
adj_num = 28

return
der
end subroutine i4_swap ( i, j )

!**************************************************************************
***80
!! I4_SWAP swaps two I4's.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 30 November 1998
!! Author:
!! John Burkardt
!! Parameters:
!! Input/output, integer ( kind = 4 ) I, J. On output, the values of I
!! and J have been interchanged.
!!
implicit none

integer ( kind = 4 ) i
integer ( kind = 4 ) j
integer ( kind = 4 ) k

k = i
i = j
j = k

return
der
end function i4_uniform ( a, b, seed )

!**************************************************************************
***80
!! I4_UNIFORM returns a scaled pseudorandom I4.
!! Discussion:
!! An I4 is an integer ( kind = 4 ) value.
!! The pseudorandom number will be scaled to be uniformly distributed
!! between A and B.
!! Licensing:
This code is distributed under the GNU LGPL license.

Modified:
12 November 2006

Author:
John Burkardt

Reference:
Paul Bratley, Bennett Fox, Linus Schrage,
A Guide to Simulation,
Springer Verlag, pages 201-202, 1983.
Pierre L’Ecuyer,
Random Number Generation,
in Handbook of Simulation,
edited by Jerry Banks,
Bennett Fox,
Algorithm 647:
Implementation and Relative Efficiency of Quasirandom
Sequence Generators,
ACM Transactions on Mathematical Software,
Volume 12, Number 4, pages 362-376, 1986.
Peter Lewis, Allen Goodman, James Miller
A Pseudo-Random Number Generator for the System/360,
IBM Systems Journal,
Volume 8, pages 136-143, 1969.

Parameters:

Input, integer ( kind = 4 ) A, B, the limits of the interval.

Input/output, integer ( kind = 4 ) SEED, the "seed" value,
which should NOT be 0. On output, SEED has been updated.

Output, integer ( kind = 4 ) I4_UNIFORM, a number between
A and B.

implicit none

integer ( kind = 4 ) a
integer ( kind = 4 ) b
integer ( kind = 4 ) i4_uniform
integer ( kind = 4 ) k
real ( kind = 4 ) r
integer ( kind = 4 ) seed
integer ( kind = 4 ) value

if ( seed == 0 ) then
  write ( *, *(a) ) ' '
  write ( *, *(a) ) 'I4_UNIFORM - Fatal error!'
  write ( *, *(a) ) ' Input value of SEED = 0.'
  stop
end if

k = seed / 127773

seed = 16807 * (seed - k * 127773) - k * 2836

if (seed < 0) then
    seed = seed + 2147483647
end if

r = real(seed, kind = 4) * 4.656612875E-10

! Scale R to lie between A-0.5 and B+0.5.
!
! r = (1.0E00 - r) * ( real(min(a, b), kind = 4) - 0.5E00 ) &
! + r * ( real(max(a, b), kind = 4) + 0.5E00 )
!
! Use rounding to convert R to an integer between A and B.
!
value = nint(r, kind = 4)

value = max(value, min(a, b))
value = min(value, max(a, b))
i4_uniform = value

return
end subroutine i4_uniform

end subroutine i4col_compare (m, n, a, i, j, isgn)

**************************************************************************
***
!! I4COL_COMPARE compares columns I and J of an I4COL.
!!
!! Example:
!!
!! Input:
!!
!!   M = 3, N = 4, I = 2, J = 4
!!
!!   A =
!!       1 2 3 4
!!       5 6 7 8
!!       9 10 11 12
!!
!! Output:
!!
!!   ISGN = -1
!!
!! Licensing:
!!
!!   This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!!   30 June 2000
!!
!! Author:
John Burkardt

Parameters:

Input, integer ( kind = 4 ) M, N, the number of rows and columns.

Input, integer ( kind = 4 ) A(M,N), an array of N columns of vectors of length M.

Input, integer ( kind = 4 ) I, J, the columns to be compared.
I and J must be between 1 and N.

Output, integer ( kind = 4 ) ISGN, the results of the comparison:
-1, column I < column J,
0, column I = column J,
+1, column J < column I.

implicit none

integer ( kind = 4 ) m
integer ( kind = 4 ) n

integer ( kind = 4 ) a(m,n)
integer ( kind = 4 ) i
integer ( kind = 4 ) isgn
integer ( kind = 4 ) j
integer ( kind = 4 ) k

! Check.

if ( i < 1 .or. n < i ) then
  write ( *, '(a)' ) 'I4COL_COMPARE - Fatal error!'
  write ( *, '(a)' ) '  Column index I is out of bounds.'
  stop
end if

if ( j < 1 .or. n < j ) then
  write ( *, '(a)' ) 'I4COL_COMPARE - Fatal error!'
  write ( *, '(a)' ) '  Column index J is out of bounds.'
  stop
end if

isgn = 0

if ( i == j ) then
  return
end if

k = 1

do while ( k <= m )

  if ( a(k,i) < a(k,j) ) then
    isgn = -1
    return
  else if ( a(k,j) < a(k,i) ) then
    isgn = +1
    return
  end if

  k = k + 1
end do
end if
k = k + 1
end do
return
end subroutine i4col_sort_a ( m, n, a )

**************************************************************************
!! I4COL_SORT_A ascending sorts an I4COL.
!!
!! In lexicographic order, the statement "X < Y", applied to two real
!! vectors X and Y of length M, means that there is some index I, with
!! 1 <= I <= M, with the property that
!! X(J) = Y(J) for J < I,
!! and
!! X(I) < Y(I).
!! In other words, the first time they differ, X is smaller.
!!
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!! 25 September 2001
!!
!! Author:
!! John Burkardt
!!
!! Parameters:
!! Input, integer ( kind = 4 ) M, the number of rows of A, and the length
!! of
!! a vector of data.
!! Input, integer ( kind = 4 ) N, the number of columns of A.
!! Input/output, integer ( kind = 4 ) A(M,N).
!! On input, the array of N columns of M-vectors.
!! On output, the columns of A have been sorted in ascending
!! lexicographic order.
!!
implicit none

integer ( kind = 4 ) m
integer ( kind = 4 ) n

integer ( kind = 4 ) a(m,n)
integer ( kind = 4 ) i
integer ( kind = 4 ) indx
integer ( kind = 4 ) isgn
integer ( kind = 4 ) j

if ( m <= 0 ) then
  return
end if

if ( n <= 1 ) then
  return
end if
!
! Initialize.
!
i = 0
indx = 0
isgn = 0
j = 0
!
! Call the external heap sorter.
!
do
  call sort_heap_external ( n, indx, i, j, isgn )
!
! Interchange the I and J objects.
!
  if ( 0 < indx ) then
    call i4col_swap ( m, n, a, i, j )
!
! Compare the I and J objects.
!
  else if ( indx < 0 ) then
    call i4col_compare ( m, n, a, i, j, isgn )
  else if ( indx == 0 ) then
    exit
  end if
end do
return
end subroutine i4col_swap ( m, n, a, i, j )

**************************************************************************
***80
*
!! I4COL_SWAP swaps columns I and J of an I4COL.
!!
!! Example:
!!
!! Input:
!!
!! M = 3, N = 4, I = 2, J = 4
!!
!! A = (
Output:

\[
A = \begin{pmatrix}
1 & 4 & 3 & 2 \\
5 & 8 & 7 & 6 \\
9 & 12 & 11 & 10 \\
\end{pmatrix}
\]

Licensing:

This code is distributed under the GNU LGPL license.

Author:

John Burkardt

Parameters:

Input, integer (kind = 4) M, N, the number of rows and columns in the array.

Input/output, integer (kind = 4) A(M,N), an array of N columns of length M.

Input, integer (kind = 4) I, J, the columns to be swapped.

```
implicit none

integer (kind = 4) m
integer (kind = 4) n
integer (kind = 4) a(m,n)
integer (kind = 4) col(m)
integer (kind = 4) i
integer (kind = 4) j

if (i < 1 .or. n < i .or. j < 1 .or. n < j) then
    write (*, '(a)') 'I4COL_SWAP - Fatal error!
    write (*, '(a)') 'I or J is out of bounds.'
    write (*, '(a,i8)') 'I = ', i
    write (*, '(a,i8)') 'J = ', j
write (*, '(a,i8)') 'N = ', n
    stop
end if

if (i == j) then
    return
end if

col(1:m) = a(1:m,i)
```
a(1:m,i) = a(1:m,j)
a(1:m,j) = col(1:m)

return
end

subroutine i4mat_print_some ( m, n, a, ilo, jlo, ihi, jhi, title )

!**************************************************************************
!!! I4MAT_PRINT_SOME prints some of an I4MAT.
!! Licensing:
!!    This code is distributed under the GNU LGPL license.
!! Modified:
!!    04 November 2003
!! Author:
!!    John Burkardt
!! Parameters:
!!!    Input, integer ( kind = 4 ) M, N, the number of rows and columns.
!!!    Input, integer ( kind = 4 ) A(M,N), an M by N matrix to be printed.
!!!    Input, integer ( kind = 4 ) ILO, JLO, the first row and column to print.
!!!    Input, integer ( kind = 4 ) IHI, JHI, the last row and column to print.
!!!    Input, character ( len = * ) TITLE, a title.

implicit none

integer ( kind = 4 ), parameter :: incx = 10
integer ( kind = 4 ) m
integer ( kind = 4 ) n

integer ( kind = 4 ) a(m,n)
character ( len = 7 ) ctemp(incx)
integer ( kind = 4 ) i
integer ( kind = 4 ) i2hi
integer ( kind = 4 ) i2lo
integer ( kind = 4 ) ihi
integer ( kind = 4 ) ilo
integer ( kind = 4 ) inc
integer ( kind = 4 ) j
integer ( kind = 4 ) j2
integer ( kind = 4 ) j2hi
integer ( kind = 4 ) j2lo
integer ( kind = 4 ) jhi
integer ( kind = 4 ) jlo
character ( len = * ) title
write ( *, '(a)' ) '  Row'
write ( *, '(a)' ) '  Row'

i2lo = max ( ilo, 1 )
i2hi = min ( ihi, m )
do i = i2lo, i2hi
  do j2 = 1, inc
    j = j2lo - 1 + j2
    write ( ctemp(j2), '(i7)' ) a(i,j)
  end do
  write ( *, '(i5,1x,10a7)' ) i, ( ctemp(j), j = 1, inc )
end do
end do
write ( *, '(a)' ) '  Row'
return
end subroutine i4mat_transpose_print ( m, n, a, title )

!**************************************************************************
!! I4MAT_TRANSPOSE_PRINT prints an I4MAT, transposed.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 28 December 2004
Author:

John Burkardt

Parameters:

Input, integer ( kind = 4 ) M, N, the number of rows and columns.

Input, integer ( kind = 4 ) A(M,N), an M by N matrix to be printed.

Input, character ( len = * ) TITLE, a title.

implicit none

integer ( kind = 4 ) m
integer ( kind = 4 ) n
integer ( kind = 4 ) a(m,n)
character ( len = * ) title

call i4mat_transpose_print_some ( m, n, a, 1, 1, m, n, title )

return
end subroutine i4mat_transpose_print_some ( m, n, a, ilo, jlo, ihi, jhi, title )

**************************************************************************
!! I4MAT_TRANSPOSE_PRINT_SOME prints some of the transpose of an I4MAT.

This code is distributed under the GNU LGPL license.

Modified:

09 February 2005

Author:

John Burkardt

Parameters:

Input, integer ( kind = 4 ) M, N, the number of rows and columns.

Input, integer ( kind = 4 ) A(M,N), an M by N matrix to be printed.

Input, integer ( kind = 4 ) ILO, JLO, the first row and column to print.

Input, integer ( kind = 4 ) IHI, JHI, the last row and column to print.

Input, character ( len = * ) TITLE, a title.

implicit none

204
integer ( kind = 4 ), parameter :: incx = 10
integer ( kind = 4 ) m
integer ( kind = 4 ) n

integer ( kind = 4 ) a(m,n)
character ( len = 7 ) ctemp(incx)
integer ( kind = 4 ) i
integer ( kind = 4 ) i2
integer ( kind = 4 ) i2hi
integer ( kind = 4 ) i2lo
integer ( kind = 4 ) ihi
integer ( kind = 4 ) ilo
integer ( kind = 4 ) inc
integer ( kind = 4 ) j
integer ( kind = 4 ) j2hi
integer ( kind = 4 ) j2lo
integer ( kind = 4 ) jhi
integer ( kind = 4 ) jlo
character ( len = * ) title

write ( *, '(a)' ) ' '
write ( *, '(a)' ) trim ( title )

do i2lo = max ( ilo, 1 ), min ( ihi, m ), incx
  i2hi = i2lo + incx - 1
  i2hi = min ( i2hi, m )
  i2hi = min ( i2hi, ihi )
  inc = i2hi + 1 - i2lo
  write ( *, '(a)' ) ' '
  do i = i2lo, i2hi
    i2 = i + 1 - i2lo
    write ( ctemp(i2), '(i7)') i
  end do
  write ( *, '('' Row '',10a7)' ) ctemp(1:inc)
end do

write ( *, '(a)' ) ' Row'
write ( *, '(a)' ) ' Col'
write ( *, '(a)' ) '

j2lo = max ( jlo, 1 )
j2hi = min ( jhi, n )

do j = j2lo, j2hi
  do i2 = 1, inc
    i = i2lo - 1 + i2
    write ( ctemp(i2), '(i7)' ) a(i,j)
  end do
  write ( *, '(i5,1x,10a7)' ) j, ( ctemp(i), i = 1, inc )
end do
write ( *, '(a)' )

return
end

subroutine i4vec_heap_d ( n, a )

!**************************************************************************
!! I4VEC_HEAP_D reorders an I4VEC into an descending heap.
!!
!! Discussion:
!!
!! An I4VEC is a vector of integer values.
!!
!! A descending heap is an array A with the property that, for every index J,
!! A(J) >= A(2*J) and A(J) >= A(2*J+1), (as long as the indices
!! 2*J and 2*J+1 are legal).
!!
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!! 15 April 1999
!!
!! Author:
!!
!! John Burkardt
!!
!! Reference:
!!
!! Albert Nijenhuis, Herbert Wilf,
!! Combinatorial Algorithms,
!! Academic Press, 1978, second edition,
!!
!! Parameters:
!!
!! Input, integer ( kind = 4 ) N, the size of the input array.
!!
!! Input/output, integer ( kind = 4 ) A(N).
!! On input, an unsorted array.
!! On output, the array has been reordered into a heap.
!!
implicit none

integer ( kind = 4 ) n
integer ( kind = 4 ) a(n)
integer ( kind = 4 ) i
integer ( kind = 4 ) ifree
integer ( kind = 4 ) key
integer ( kind = 4 ) m

! Only nodes N/2 down to 1 can be "parent" nodes.
! do i = n/2, 1, -1
! Copy the value out of the parent node.
! Position IFREE is now "open".
key = a(i)
ifree = i

do
! Positions 2*IFREE and 2*IFREE + 1 are the descendants of position
! IFREE. (One or both may not exist because they exceed N.)
m = 2 * ifree
! Does the first position exist?
if ( n < m ) then
exit
end if
! Does the second position exist?
if ( m + 1 <= n ) then
! If both positions exist, take the larger of the two values,
! and update M if necessary.
if ( a(m) < a(m+1) ) then
m = m + 1
end if
end if
! If the large descendant is larger than KEY, move it up,
! and update IFREE, the location of the free position, and
! consider the descendants of THIS position.
if ( a(m) <= key ) then
exit
end if
a(ifree) = a(m)
ifree = m
end do
! Once there is no more shifting to do, KEY moves into the free spot IFREE.
a(ifree) = key
subroutine i4vec_indicator ( n, a )

!**************************************************************************
***80
!
!! I4VEC_INDICATOR sets an I4VEC to the vector A(I)=I.
!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!! 09 November 2000
!!
!! Author:
!!
!! John Burkardt
!!
!! Parameters:
!!
!! Input, integer ( kind = 4 ) N, the number of elements of A.
!!
!! Output, integer ( kind = 4 ) A(N), the array to be initialized.
!!
implicit none

integer ( kind = 4 ) n

integer ( kind = 4 ) a(n)
integer ( kind = 4 ) i

do i = 1, n
    a(i) = i
end do

return
end

subroutine i4vec_print ( n, a, title )

!**************************************************************************
***80
!
!! I4VEC_PRINT prints an I4VEC.
!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!! 28 November 2000
!!
!! Author:
!!
!! John Burkardt
! Parameters:
! 
!   Input, integer ( kind = 4 ) N, the number of components of the vector.
! 
!   Input, integer ( kind = 4 ) A(N), the vector to be printed.
! 
!   Input, character ( len = * ) TITLE, a title to be printed first.
!   TITLE may be blank.
! 
implicit none

integer ( kind = 4 ) n

integer ( kind = 4 ) a(n)
integer ( kind = 4 ) b
integer ( kind = 4 ) i
character ( len = * ) title

if ( 0 < len_trim ( title ) ) then
  write ( *, '(a)' ) ' ' 
  write ( *, '(a)' ) trim ( title )
end if

big = maxval ( abs ( a(1:n) ) )

write ( *, '(a)' ) '
if ( big < 1000 ) then
  do i = 1, n
    write ( *, '(2x,i8,2x,i4)' ) i, a(i)
  end do
else if ( big < 100000 ) then
  do i = 1, n
    write ( *, '(2x,i8,2x,i7)' ) i, a(i)
  end do
else
  do i = 1, n
    write ( *, '(2x,i8,2x,i12)' ) i, a(i)
  end do
end if

return
end subroutine i4vec_reverse ( n, a )

**************************************************************************
!! I4VEC_REVERSE reverses the elements of an I4VEC.
!! Example:
!! Input:
!! N = 5,
!! A = ( 11, 12, 13, 14, 15 ).
!! Output:
!! A = ( 15, 14, 13, 12, 11 ).
Licensing:

This code is distributed under the GNU LGPL license.

Modified:

26 July 1999

Author:

John Burkardt

Parameters:

Input, integer (kind = 4) N, the number of entries in the array.
Input/output, integer (kind = 4) A(N), the array to be reversed.

implicit none

integer (kind = 4) n
integer (kind = 4) a(n)
integer (kind = 4) i

do i = 1, n/2
   call i4_swap ( a(i), a(n+1-i) )
end do

return
end

subroutine i4vec_sort_heap_a ( n, a )

**************************************************************************
!! I4VEC_SORT_HEAP_A ascending sorts an I4VEC using heap sort.
!!
!! Discussion:
!!
!! An I4VEC is a vector of integer values.
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!! 15 April 1999
!!
!! Author:
!!
!! John Burkardt
!!
!! Reference:
!!
!! Albert Nijenhuis, Herbert Wilf,
!! Combinatorial Algorithms,
!! Academic Press, 1978, second edition,
Parameters:

Input, integer ( kind = 4 ) N, the number of entries in the array.
Input/output, integer ( kind = 4 ) A(N).
On input, the array to be sorted;
On output, the array has been sorted.

implicit none

integer ( kind = 4 ) n
integer ( kind = 4 ) a(n)
integer ( kind = 4 ) n1

if ( n <= 1 ) then
  return
end if

1: Put A into descending heap form.

! call i4vec_heap_d ( n, a )

2: Sort A.

! The largest object in the heap is in A(1).
! Move it to position A(N).
! call i4_swap ( a(1), a(n) )
! Consider the diminished heap of size N1.
! do n1 = n - 1, 2, -1
! Restore the heap structure of A(1) through A(N1).
! call i4vec_heap_d ( n1, a )
! Take the largest object from A(1) and move it to A(N1).
! call i4_swap ( a(1), a(n1) )

end do

return

end subroutine level_set ( root, adj_num, adj_row, adj, mask, level_num, &
level_row, level, node_num )

***80
!! LEVEL_SET generates the connected level structure rooted at a given
!! node.
!! Discussion:
!! Only nodes for which MASK is nonzero will be considered.
The root node chosen by the user is assigned level 1, and masked. All (unmasked) nodes reachable from a node in level 1 are assigned level 2 and masked. The process continues until there are no unmasked nodes adjacent to any node in the current level. The number of levels may vary between 2 and NODE_NUM.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

28 October 2003

Author:

Original FORTRAN77 version by Alan George, Joseph Liu.
FORTRAN90 version by John Burkardt

Reference:

Alan George, Joseph Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice Hall, 1981.

Parameters:

Input, integer ( kind = 4 ) ROOT, the node at which the level structure is to be rooted.

Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.

Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.

Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure. For each row, it contains the column indices of the nonzero entries.

Input/output, integer ( kind = 4 ) MASK(NODE_NUM). On input, only nodes with nonzero MASK are to be processed. On output, those nodes which were included in the level set have MASK set to 1.

Output, integer ( kind = 4 ) LEVEL_NUM, the number of levels in the level structure. ROOT is in level 1. The neighbors of ROOT are in level 2, and so on.

Output, integer ( kind = 4 ) LEVEL_ROW(NODE_NUM+1), LEVEL(NODE_NUM), the rooted level structure.

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) iccsze
integer ( kind = 4 ) j
integer ( kind = 4 ) jstop
integer ( kind = 4 ) jstrt
integer ( kind = 4 ) lbegin
integer ( kind = 4 ) level_num
integer ( kind = 4 ) level_row(node_num+1)
integer ( kind = 4 ) level(node_num)
integer ( kind = 4 ) lvlend
integer ( kind = 4 ) lvsize
integer ( kind = 4 ) mask(node_num)
integer ( kind = 4 ) nbr
integer ( kind = 4 ) node
integer ( kind = 4 ) root

mask(root) = 0
level(1) = root
level_num = 0
lvlend = 0
iccsze = 1

! LBEGIN is the pointer to the beginning of the current level, and
! LVLEND points to the end of this level.
!
! do

  lbegin = lvlend + 1
  lvlend = iccsze
  level_num = level_num + 1
  level_row(level_num) = lbegin

! Generate the next level by finding all the masked neighbors of nodes
! in the current level.
!
! do i = lbegin, lvlend

  node = level(i)
  jstrt = adj_row(node)
  jstop = adj_row(node+1) - 1

  do j = jstrt, jstop

    nbr = adj(j)

    if ( mask(nbr) /= 0 ) then
      iccsze = iccsze + 1
      level(iccsze) = nbr
      mask(nbr) = 0
    end if

  end do

end do

! Compute the current level width (the number of nodes encountered.)
! If it is positive, generate the next level.
lvsize = iccsze - lvlend

if ( lvsize <= 0 ) then
  exit
end if

end do

level_row(level_num+1) = lvlend + 1

! Reset MASK to 1 for the nodes in the level structure.
!
mask(level(1:iccsze)) = 1

return
end

subroutine level_set_print ( node_num, level_num, level_row, level )

!**************************************************************************
***80
!! LEVEL_SET_PRINT prints level set information.
!!
! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
! Modified:
!!
!! 26 October 2003
!!
! Author:
!!
!! John Burkardt
!!
! Parameters:
!!
!! Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
!!
!! Input, integer ( kind = 4 ) LEVEL_NUM, the number of levels.
!!
!! Input, integer ( kind = 4 ) LEVEL_ROW(LEVEL_NUM+1), organizes the
!! entries
!! of LEVEL.  The entries for level I are in entries LEVEL_ROW(I)
!! through LEVEL_ROW(I+1)-1.
!!
!! Input, integer ( kind = 4 ) LEVEL(NODE_NUM), is simply a list of the
!! nodes in an order induced by the levels.
!!
implicit none

integer ( kind = 4 ) level_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) level(node_num)
integer ( kind = 4 ) level_row(level_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) jhi
integer ( kind = 4 ) jlo
integer ( kind = 4 ) jmax
integer ( kind = 4 ) jmin

write ( *, '(a)' ) 'LEVEL_SET_PRINT'
write ( *, '(a,i8)' ) 'The number of nodes is ', node_num
write ( *, '(a,i8)' ) 'The number of levels is ', level_num
write ( *, '(a)' ) 'Level Min Max Nonzeros '

do i = 1, level_num
   jmin = level_row(i)
   jmax = level_row(i+1) - 1
   if ( jmax < jmin ) then
      write ( *, '(2x,3i4,6x,10i8)' ) i, jmin, jmax
   else
      do jlo = jmin, jmax, 5
         jhi = min ( jlo + 4, jmax )
         if ( jlo == jmin ) then
            write ( *, '(2x,3i4,3x,5i8)' ) i, jmin, jmax, level(jlo:jhi)
         else
            write ( *, '(2x,12x,3x,5i8)' ) level(jlo:jhi)
         end if
      end do
   end if
end do
return
end subroutine perm_check ( n, p, ierr )

***80
!! PERM_CHECK checks that a vector represents a permutation.
!! Discussion:
!! The routine verifies that each of the integers from 1
!! to N occurs among the N entries of the permutation.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 01 February 2001
Author: John Burkardt

Parameters:

Input, integer ( kind = 4 ) N, the number of entries.
Input, integer ( kind = 4 ) P(N), the array to check.
Output, integer ( kind = 4 ) IERROR, error flag.
0, the array represents a permutation.
nonzero, the array does not represent a permutation. The smallest missing value is equal to IERROR.

implicit none

integer ( kind = 4 ) n
integer ( kind = 4 ) ierror
integer ( kind = 4 ) ifind
integer ( kind = 4 ) iseek
integer ( kind = 4 ) p(n)

ierror = 0

do iseek = 1, n
    ierror = iseek
    do ifind = 1, n
        if ( p(ifind) == iseek ) then
            ierror = 0
            exit
        end if
    end do
    if ( ierror /= 0 ) then
        return
    end if
end do

return
end subroutine perm_inverse3 ( n, perm, perm_inv )

**************************************************************************
***80
!! PERM_INVERSE3 produces the inverse of a given permutation.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 28 October 2003
Author: John Burkardt

Parameters:

Input, integer ( kind = 4 ) N, the number of items permuted.
Input, integer ( kind = 4 ) PERM(N), a permutation.
Output, integer ( kind = 4 ) PERM_INV(N), the inverse permutation.

```fortran
implicit none
integer ( kind = 4 ) n
integer ( kind = 4 ) i
integer ( kind = 4 ) perm(n)
integer ( kind = 4 ) perm_inv(n)
do i = 1, n
   perm_inv(perm(i)) = i
end do
return
end subroutine perm_uniform ( n, seed, p )
```

PERM_UNIFORM selects a random permutation of N objects.

Discussion:
The routine assumes the objects are labeled 1, 2, ... N.

Licensing:
This code is distributed under the GNU LGPL license.

Modified:
12 May 2002

Author:
Original FORTRAN77 version by Albert Nijenhuis, Herbert Wilf.
FORTRAN90 version by John Burkardt

Reference:
Albert Nijenhuis, Herbert Wilf,
Combinatorial Algorithms,
Academic Press, 1978, second edition,

Parameters:
Input, integer ( kind = 4 ) N, the number of objects to be permuted.

Input/output, integer ( kind = 4 ) SEED, a seed for the random number generator.

Output, integer ( kind = 4 ) P(N), a permutation of ( 1, 2, ..., N ), in standard index form.

implicit none

integer ( kind = 4 ) n
integer ( kind = 4 ) i
integer ( kind = 4 ) i4_uniform
integer ( kind = 4 ) j
integer ( kind = 4 ) p(n)
integer ( kind = 4 ) seed

call i4vec_indicator ( n, p )

do i = 1, n
   j = i4_uniform ( i, n, seed )
   call i4_swap ( p(i), p(j) )
end do

return
end

subroutine r8vec_permute ( n, a, p )

!**************************************************************************
!! R8VEC_PERMUTE permutes an R8VEC in place.
!!
!! Discussion:
!!
!! This routine permutes an array of real "objects", but the same logic can be used to permute an array of objects of any arithmetic type, or an array of objects of any complexity. The only temporary storage required is enough to store a single object. The number of data movements made is N + the number of cycles of order 2 or more, which is never more than N + N/2.
!!
!! Example:
!!
!! Input:
!!
!!  N = 5
!!  P = ( 2, 4, 5, 1, 3 )
!!  A = ( 1.0, 2.0, 3.0, 4.0, 5.0 )
!!       (11.0, 22.0, 33.0, 44.0, 55.0 )
!!
!! Output:
!!
!!  A = ( 2.0, 4.0, 5.0, 1.0, 3.0 )
!!      (22.0, 44.0, 55.0, 11.0, 33.0 ).
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
Parameters:

- Input, integer (kind = 4) N, the number of objects.
- Input/output, real (kind = 8) A(2,N), the array to be permuted.
- Input, integer (kind = 4) P(N), the permutation. P(I) = J means that the I-th element of the output array should be the J-th element of the input array. P must be a legal permutation of the integers from 1 to N, otherwise the algorithm will fail catastrophically.

```plaintext
! implicit none

integer (kind = 4) n
integer (kind = 4), parameter :: ndim = 2
real (kind = 8) a(ndim,n)
real (kind = 8) a_temp(ndim)
integer (kind = 4) ierror
integer (kind = 4) iget
integer (kind = 4) iput
integer (kind = 4) istart
integer (kind = 4) p(n)

call perm_check ( n, p, ierror )

if ( ierror /= 0 ) then
  write (*, *'(a)')
  write (*, *'(a)') 'R8VEC_PERMUTE - Fatal error!'
  write (*, *'(a)') ' The input array does not represent'
  write (*, *'(a)') ' a proper permutation. In particular, the'
  write (*, *'(a)') ' array is missing the value ', ierror
  stop
end if

! Search for the next element of the permutation that has not been used.
!
do istart = 1, n

  if ( p(istart) < 0 ) then
    cycle
  else if ( p(istart) == istart ) then
    p(istart) = -p(istart)
    cycle
  else
```
a_temp(1:ndim) = a(1:ndim,istart)
iget = istart
!
! Copy the new value into the vacated entry.
!
do

iput = iget
iget = p(iget)
p(iput) = -p(iput)

if ( iget < 1 .or. n < iget ) then
  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) 'R82VEC_PERMUTE - Fatal error!'
  write ( *, '(a)' ) ' A permutation index is out of range.'
  write ( *, '(a,i8,a,i8)' ) ' P(', iput, ') = ', iget
  stop
end if

if ( iget == istart ) then
  a(1:ndim,iput) = a_temp(1:ndim)
  exit
end if

a(1:ndim,iput) = a(1:ndim,iget)
end do
end if
end do
!
! Restore the signs of the entries.
!
p(1:n) = -p(1:n)

return
end subroutine r8mat_print_some ( m, n, a, ilo, jlo, ihi, jhi, title )
!  Input, integer ( kind = 4 ) M, N, the number of rows and columns.
!  Input, real ( kind = 8 ) A(M,N), an M by N matrix to be printed.
!  Input, integer ( kind = 4 ) ILO, JLO, the first row and column to print.
!  Input, integer ( kind = 4 ) IHI, JHI, the last row and column to print.
!  Input, character ( len = * ) TITLE, an optional title.

implicit none

integer ( kind = 4 ), parameter :: incx = 5
integer ( kind = 4 ) m
integer ( kind = 4 ) n
real ( kind = 8 ) a(m,n)
character ( len = 14 ) ctemp(incx)
logical d_is_int
integer ( kind = 4 ) i
integer ( kind = 4 ) i2hi
integer ( kind = 4 ) i2lo
integer ( kind = 4 ) ihi
integer ( kind = 4 ) ilo
integer ( kind = 4 ) inc
integer ( kind = 4 ) j
integer ( kind = 4 ) j2
integer ( kind = 4 ) j2hi
integer ( kind = 4 ) j2lo
integer ( kind = 4 ) jhi
integer ( kind = 4 ) jlo
character ( len = * ) title

if ( 0 < len_trim ( title ) ) then
  write ( *, '(a)' ) '  '
  write ( *, '(a)' ) trim ( title )
end if

do j2lo = max ( jlo, 1 ), min ( jhi, n ), incx
  j2hi = j2lo + incx - 1
  j2hi = min ( j2hi, n )
  j2hi = min ( j2hi, jhi )
  inc = j2hi + 1 - j2lo
  write ( *, '(a)' ) '  '
  do j = j2lo, j2hi
    j2 = j + 1 - j2lo
    write ( ctemp(j2), '(i7,7x)') j
  end do
  write ( *, '('' Col   '',5a14)' ) ctemp(1:inc)
  write ( *, '(a)' ) '  Row'
  write ( *, '(a)' ) '  '
  i2lo = max ( ilo, 1 )
i2hi = min ( ihi, m )

do i = i2lo, i2hi
    do j2 = 1, inc
        j = j2lo - 1 + j2
        write ( ctemp(j2), '(g14.6)' ) a(i,j)
    end do
    write ( *, '(i5,1x,5a14)' ) i, ( ctemp(j), j = 1, inc )
end do
end do

write ( *, '(a)' ) ' '
return
end subroutine r8mat_transpose_print_some ( m, n, a, ilo, jlo, ihi, jhi, title )
integer ( kind = 4 ) n

real ( kind = 8 ) a(m,n)
character ( len = 14 ) ctemp(incx)
integer ( kind = 4 ) i
integer ( kind = 4 ) i2
integer ( kind = 4 ) i2hi
integer ( kind = 4 ) i2lo
integer ( kind = 4 ) hi
integer ( kind = 4 ) ilo
integer ( kind = 4 ) inc
integer ( kind = 4 ) j
integer ( kind = 4 ) j2hi
integer ( kind = 4 ) j2lo
integer ( kind = 4 ) jhi
integer ( kind = 4 ) jlo
character ( len = * ) title

if ( 0 < len_trim ( title ) ) then
  write ( * , '(a)' ) ' ' 
  write ( * , '(a)' ) trim ( title ) 
end if

do i2lo = max ( ilo, 1 ), min ( ihi, m ), incx
  i2hi = i2lo + incx - 1
  i2hi = min ( i2hi, m )
  i2hi = min ( i2hi, ihi )
  inc = i2hi + 1 - i2lo
  write ( * , '(a)' ) ' '
  do i = i2lo, i2hi
     i2 = i + 1 - i2lo
     write ( ctemp(i2), '(i7,7x)') i 
  end do
  write ( * , '('' Row   '',5a14)' ) ctemp(1:inc)
  write ( * , '(a)' ) ' Col'
  write ( * , '(a)' ) ' ' 
  j2lo = max ( jlo, 1 )
  j2hi = min ( jhi, n )
  do j = j2lo, j2hi
    do i2 = 1, inc
      i = i2lo - 1 + i2
      write ( ctemp(i2), '(g14.6)' ) a(i,j) 
    end do
    write ( * , '(i5,1x,5a14)' ) j, ( ctemp(i), i = 1, inc )
  end do
  write ( * , '(a)' ) ' '
end do

write ( * , '(a)' ) ' '

223
return
end

subroutine rcm ( root, adj_num, adj_row, adj, mask, perm, iccsze, node_num )

**************************************************************************
!!! RCM renumbers a connected component by the reverse Cuthill McKee
algorithm.
!! Discussion:
!! The connected component is specified by a node ROOT and a mask.
The numbering starts at the root node.
!! An outline of the algorithm is as follows:
!! \( X(1) = \text{ROOT} \).
!! for ( \( I = 1 \) to \( N-1 \))
!! \begin{align*}
!! & \text{Find all unlabeled neighbors of } X(I), \\
!! & \text{assign them the next available labels, in order of increasing degree.}
!! \end{align*}
!! When done, reverse the ordering.
!! Licensing:
!! This code is distributed under the GNU LGPL license.
!! Modified:
!! 02 January 2007
!! Author:
!! Original FORTRAN77 version by Alan George, Joseph Liu.
!! FORTRAN90 version by John Burkardt
!! Reference:
!! Alan George, Joseph Liu,
!! Computer Solution of Large Sparse Positive Definite Systems,
!! Prentice Hall, 1981.
!! Parameters:
!! Input, integer ( kind = 4 ) \( \text{ROOT} \), the node that defines the connected
!! component. It is used as the starting point for the RCM ordering.
!! Input, integer ( kind = 4 ) \( \text{ADJ}._\text{NUM} \), the number of adjacency entries.
!! Input, integer ( kind = 4 ) \( \text{ADJ}._\text{ROW}(\text{NODE}\_\text{NUM}+1) \). Information about
!! row \( I \) is stored in entries \( \text{ADJ}._\text{ROW}(I) \) through \( \text{ADJ}._\text{ROW}(I+1)-1 \) of \( \text{ADJ} \).
!! Input, integer ( kind = 4 ) \( \text{ADJ}(\text{ADJ}\_\text{NUM}) \), the adjacency structure.
!! For each row, it contains the column indices of the nonzero entries.
Input/output, integer (kind = 4) MASK(NODE_NUM), a mask for the
nodes. Only those nodes with nonzero input mask values are considered by the
routine. The nodes numbered by RCM will have their mask values
set to zero.

Output, integer (kind = 4) PERM(NODE_NUM), the RCM ordering.

Output, integer (kind = 4) ICCSZE, the size of the connected
component that has been numbered.

Input, integer (kind = 4) NODE_NUM, the number of nodes.

Local Parameters:

Workspace, integer DEG(NODE_NUM), a temporary vector used to hold
the degree of the nodes in the section graph specified by mask and
root.

implicit none

integer (kind = 4) adj_num
integer (kind = 4) node_num

integer (kind = 4) adj(adj_num)
integer (kind = 4) adj_row(node_num+1)
integer (kind = 4) deg(node_num)
integer (kind = 4) fnbr
integer (kind = 4) i
integer (kind = 4) iccsze
integer (kind = 4) j
integer (kind = 4) jstop
integer (kind = 4) jstrt
integer (kind = 4) k
integer (kind = 4) l
integer (kind = 4) lbegin
integer (kind = 4) lnbr
integer (kind = 4) lperm
integer (kind = 4) lvlend
integer (kind = 4) mask(node_num)
integer (kind = 4) nbr
integer (kind = 4) node
integer (kind = 4) perm(node_num)
integer (kind = 4) root

Find the degrees of the nodes in the component specified by MASK and
ROOT.

call degree (root, adj_num, adj_row, adj, mask, deg, iccsze, perm,
node_num)

mask(root) = 0

if (iccsze <= 1) then
  return
end if

lvlend = 0
lnbr = 1
! LBEGIN and LVLEND point to the beginning and
! the end of the current level respectively.
! do while ( lvlend < lnbr )
    lbegin = lvlend + 1
    lvlend = lnbr
    do i = lbegin, lvlend
    ! For each node in the current level...
    ! node = perm(i)
    jstrt = adj_row(node)
    jstop = adj_row(node+1) - 1
    ! Find the unnumbered neighbors of NODE.
    ! FNBR and LNBR point to the first and last neighbors
    ! of the current node in PERM.
    fnbr = lnbr + 1
    do j = jstrt, jstop
        nbr = adj(j)
        if ( mask(nbr) /= 0 ) then
            lnbr = lnbr + 1
            mask(nbr) = 0
            perm(lnbr) = nbr
        end if
    end do
    ! If no neighbors, skip to next node in this level.
    if ( lnbr <= fnbr ) then
        cycle
    end if
    ! Sort the neighbors of NODE in increasing order by degree.
    ! Linear insertion is used.
    k = fnbr
    do while ( k < lnbr )
        l = k
        k = k + 1
        nbr = perm(k)
        do while ( fnbr < l )
            lperm = perm(l)
            if ( deg(lperm) <= deg(nbr) ) then
                exit
            end if
        end do
perm(l+1) = lperm
l = l - 1
end do
perm(l+1) = nbr
end do
end do

! We now have the Cuthill-McKee ordering. Reverse it.
!
call i4vec_reverse ( iccsze, perm )
return
end subroutine root_find ( root, adj_num, adj_row, adj, mask, level_num, &
level_row, level, node_num )

**************************************************************************
!! ROOT_FIND finds a pseudo-peripheral node.
!!
!! Discussion:
!!
!! The diameter of a graph is the maximum distance (number of edges) between
!! any two nodes of the graph.
!!
!! The eccentricity of a node is the maximum distance between that node
!! and any other node of the graph.
!!
!! A peripheral node is a node whose eccentricity equals the diameter of
!! the graph.
!!
!! A pseudo-peripheral node is an approximation to a peripheral node; it
!! may be a peripheral node, but all we know is that we tried our best.
!!
!! The routine is given a graph, and seeks pseudo-peripheral nodes, using
!! a modified version of the scheme of Gibbs, Poole and Stockmeyer. It
determines such a node for the section subgraph specified by MASK and
ROOT.
!!
!! The routine also determines the level structure associated with the
given pseudo-peripheral node; that is, how far each node is from the
pseudo-peripheral node. The level structure is returned as a list of nodes
LS, and pointers to the beginning of the list of nodes that are at a
distance of 0, 1, 2, ..., NODE_NUM-1 from the pseudo-peripheral node.
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
28 October 2003

Author:

Original FORTRAN77 version by Alan George, Joseph Liu.
FORTRAN90 version by John Burkardt

Reference:

Alan George, Joseph Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice Hall, 1981.

Norman Gibbs, William Poole, Paul Stockmeyer,
An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix,
SIAM Journal on Numerical Analysis,

Norman Gibbs,
Algorithm 509: A Hybrid Profile Reduction Algorithm,
ACM Transactions on Mathematical Software,

Parameters:

Input/output, integer ( kind = 4 ) ROOT. On input, ROOT is a node in the
component of the graph for which a pseudo-peripheral node is
sought. On output, ROOT is the pseudo-peripheral node obtained.

Input, integer ( kind = 4 ) ADJ_NUM, the number of adjacency entries.

Input, integer ( kind = 4 ) ADJ_ROW(NODE_NUM+1). Information about
row I is stored in entries ADJ_ROW(I) through ADJ_ROW(I+1)-1 of ADJ.

Input, integer ( kind = 4 ) ADJ(ADJ_NUM), the adjacency structure.
For each row, it contains the column indices of the nonzero entries.

Input, integer ( kind = 4 ) MASK(NODE_NUM), specifies a section
subgraph.
Nodes for which MASK is zero are ignored by FNROOT.

Output, integer ( kind = 4 ) LEVEL_NUM, is the number of levels in the
level structure rooted at the node ROOT.

Output, integer ( kind = 4 ) LEVEL_ROW(NODE_NUM+1), LEVEL(NODE_NUM),
the
level structure array pair containing the level structure found.

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_row(node_num+1)
integer ( kind = 4 ) iccsze
integer ( kind = 4 ) j
integer ( kind = 4 ) jstrt
integer ( kind = 4 ) k
integer ( kind = 4 ) kstop
integer ( kind = 4 ) kstrt
integer ( kind = 4 ) level(node_num)
integer ( kind = 4 ) level_num
integer ( kind = 4 ) level_num2
integer ( kind = 4 ) level_row(node_num+1)
integer ( kind = 4 ) mask(node_num)
integer ( kind = 4 ) mindeg
integer ( kind = 4 ) nabor
integer ( kind = 4 ) ndeg
integer ( kind = 4 ) node
integer ( kind = 4 ) root

! Determine the level structure rooted at ROOT.
! call level_set ( root, adj_num, adj_row, adj, mask, level_num, &
!   level_row, level, node_num )
!
! Count the number of nodes in this level structure.
!
! iccsze = level_row(level_num+1) - 1
!
! Extreme case:
! A complete graph has a level set of only a single level.
! Every node is equally good (or bad).
!
if ( level_num == 1 ) then
  return
end if
!
! Extreme case:
! A "line graph" 0--0--0--0--0 has every node in its only level.
! By chance, we've stumbled on the ideal root.
!
if ( level_num == iccsze ) then
  return
end if
!
! Pick any node from the last level that has minimum degree
! as the starting point to generate a new level set.
!
do
  mindeg = iccsze
  jstrt = level_row(level_num)
  root = level(jstrt)
  if ( jstrt < iccsze ) then
    do j = jstrt, iccsze
      node = level(j)
      ndeg = 0
      kstrt = adj_row(node)
      kstop = adj_row(node+1) - 1
      !
    end do
  end if

  !
end do
do  k = kstrt, kstop
    nabor = adj(k)
    if ( 0 < mask(nabor) ) then
        ndeg = ndeg + 1
    end if
end do

if ( ndeg < mindeg ) then
    root = node
    mindeg = ndeg
end if

end do

end if

! Generate the rooted level structure associated with this node.
!
call level_set ( root, adj_num, adj_row, adj, mask, level_num2, & level_row, level, node_num )
!
! If the number of levels did not increase, accept the new ROOT.
!
if ( level_num2 <= level_num ) then
    exit
end if

level_num = level_num2
!
! In the unlikely case that ROOT is one endpoint of a line graph, 
! we can exit now.
!
if ( iccsze <= level_num ) then
    exit
end if

end do

return
end subroutine sort_heap_external ( n, indx, i, j, isgn )

**************************************************************************
***80
!! SORT_HEAP_EXTERNAL externally sorts a list of items into ascending order.
!!
!! Discussion:
!!
!! The actual list of data is not passed to the routine. Hence this
!! routine may be used to sort integers, reals, numbers, names,
!! dates, shoe sizes, and so on. After each call, the routine asks
!! the user to compare or interchange two items, until a special
!! return value signals that the sorting is completed.
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
| Modified: |
| 05 February 2004 |
| Author: |
| Original FORTRAN77 version by Albert Nijenhuis, Herbert Wilf. |
| FORTRAN90 version by John Burkardt |
| Reference: |
| Albert Nijenhuis, Herbert Wilf, |
| Combinatorial Algorithms, |
| Academic Press, 1978, second edition, |
| Parameters: |
| Input, integer ( kind = 4 ) N, the number of items to be sorted. |
| Input/output, integer ( kind = 4 ) INDX, the main communication signal. |
| The user must set INDX to 0 before the first call. |
| Thereafter, the user should not change the value of INDX until the sorting is done. |
| On return, if INDX is |
| greater than 0, |
| * interchange items I and J; |
| * call again. |
| less than 0, |
| * compare items I and J; |
| * set ISGN = -1 if I < J, ISGN = +1 if J < I; |
| * call again. |
| equal to 0, the sorting is done. |
| Output, integer ( kind = 4 ) I, J, the indices of two items. |
| On return with INDX positive, elements I and J should be interchanged. |
| On return with INDX negative, elements I and J should be compared, and the result reported in ISGN on the next call. |
| Input, integer ( kind = 4 ) ISGN, results of comparison of elements I and J. (Used only when the previous call returned INDX less than 0). |
| ISGN <= 0 means I is less than or equal to J; |
| 0 <= ISGN means I is greater than or equal to J. |
| implicit none |

```fortran
integer ( kind = 4 ) i
integer ( kind = 4 ), save :: i_save = 0
integer ( kind = 4 ) indx
integer ( kind = 4 ) isgn
integer ( kind = 4 ) j
integer ( kind = 4 ), save :: j_save = 0
```
integer ( kind = 4 ), save :: k = 0
integer ( kind = 4 ), save :: k1 = 0
integer ( kind = 4 ) n
integer ( kind = 4 ), save :: n1 = 0
!
!  INDX = 0: This is the first call.
!
if ( indx == 0 ) then
  i_save = 0
  j_save = 0
  k = n / 2
  k1 = k
  n1 = n
!
!  INDX < 0: The user is returning the results of a comparison.
!
else if ( indx < 0 ) then
  if ( indx == -2 ) then
    if ( isgn < 0 ) then
      i_save = i_save + 1
    end if
    j_save = k1
    k1 = i_save
    indx = -1
    i = i_save
    j = j_save
    return
  end if
  if ( 0 < isgn ) then
    indx = 2
    i = i_save
    j = j_save
    return
  end if
  if ( k <= 1 ) then
    if ( n1 == 1 ) then
      i_save = 0
      j_save = 0
      indx = 0
    else
      i_save = n1
      n1 = n1 - 1
      j_save = 1
      indx = 1
    end if
    i = i_save
    j = j_save
    return
  end if
end if
k = k - 1
k1 = k

! 0 < INDX, the user was asked to make an interchange.
! else if ( indx == 1 ) then
  k1 = k
end if

do
  i_save = 2 * k1
  if ( i_save == n1 ) then
    j_save = k1
    k1 = i_save
    indx = -1
    i = i_save
    j = j_save
    return
  else if ( i_save <= n1 ) then
    j_save = i_save + 1
    indx = -2
    i = i_save
    j = j_save
    return
  end if
  if ( k <= 1 ) then
    exit
  end if
  k = k - 1
  k1 = k
end do

if ( n1 == 1 ) then
  i_save = 0
  j_save = 0
  indx = 0
  i = i_save
  j = j_save
else
  i_save = n1
  n1 = n1 - 1
  j_save = 1
  indx = 1
  i = i_save
  j = j_save
end if

return
end

subroutine timestamp ( logical_unit )
!! TIMESTAMP prints the current YMDHMS date as a time stamp.
!!
!! Example:
!!
!! 31 May 2001   9:45:54.872 AM
!!
!! Licensing:
!!
!! This code is distributed under the GNU LGPL license.
!!
!! Modified:
!!
!! 06 August 2005
!!
!! Author:
!!
!! John Burkardt
!!
!! Parameters:
!!
!! None
!! implicit none

integer, intent(in) :: logical_unit
character ( len = 8 ) ampm
integer ( kind = 4 ) d
integer ( kind = 4 ) h
integer ( kind = 4 ) m
integer ( kind = 4 ) mm
character ( len = 9 ), parameter, dimension(12) :: month = (/ &
    'January ', 'February ', 'March    ', 'April    ', &
    'May      ', 'June     ', 'July     ', 'August   ', &
    'September', 'October  ', 'November ', 'December ' /)
integer ( kind = 4 ) n
integer ( kind = 4 ) s
integer ( kind = 4 ) values(8)
integer ( kind = 4 ) y

call date_and_time ( values = values )

y = values(1)
d = values(2)
h = values(5)
n = values(6)
s = values(7)
mm = values(8)

if ( h < 12 ) then
    ampm = 'AM'
else if ( h == 12 ) then
    if ( n == 0 .and. s == 0 ) then
        ampm = 'Noon'
    else
        ampm = 'PM'
    end if
else
    h = h - 12

end if
if ( h < 12 ) then
  ampm = 'PM'
else if ( h == 12 ) then
  if ( n == 0 .and. s == 0 ) then
    ampm = 'Midnight'
  else
    ampm = 'AM'
  end if
end if

write ( logical_unit, '
( i2,1x,a,1x,i4,2x,i2,a1,i2.2,a1,i2.2,a1,i3.3,1x,a )

') &
  d, trim ( month(m) ), y, h, ':', n, ':', s, '.', mm, trim ( ampm )

return
end subroutine triangulation_neighbor_triangles ( triangle_order, triangle_num, &
  triangle_node, triangle_neighbor )
The output information in TRIANGLE_NEIGHBOR:

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Neighboring Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1 -1 2</td>
</tr>
<tr>
<td>2</td>
<td>1 4 3</td>
</tr>
<tr>
<td>3</td>
<td>2 5 7</td>
</tr>
<tr>
<td>4</td>
<td>2 -1 8</td>
</tr>
<tr>
<td>5</td>
<td>3 8 6</td>
</tr>
<tr>
<td>6</td>
<td>5 9 7</td>
</tr>
<tr>
<td>7</td>
<td>3 6 -1</td>
</tr>
<tr>
<td>8</td>
<td>5 4 10</td>
</tr>
<tr>
<td>9</td>
<td>6 10 12</td>
</tr>
<tr>
<td>10</td>
<td>9 8 11</td>
</tr>
<tr>
<td>11</td>
<td>12 10 14</td>
</tr>
<tr>
<td>12</td>
<td>9 11 13</td>
</tr>
<tr>
<td>13</td>
<td>-1 12 16</td>
</tr>
<tr>
<td>14</td>
<td>11 -1 15</td>
</tr>
<tr>
<td>15</td>
<td>16 14 -1</td>
</tr>
<tr>
<td>16</td>
<td>13 15 -1</td>
</tr>
</tbody>
</table>

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

14 February 2006

Author:

John Burkardt

Parameters:

Input, integer ( kind = 4 ) TRIANGLE_ORDER, the order of the triangles.
Input, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.
! Input, integer ( kind = 4 )
TRIANGLE_NODE(TRIANGLE_ORDER,TRIANGLE_NUM),
! the nodes that make up each triangle.
!
! Output, integer ( kind = 4 ) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), the
three
! triangles that are direct neighbors of a given triangle.
! TRIANGLE_NEIGHBOR(1,I) is the index of the triangle which touches side
1,
! defined by nodes 2 and 3, and so on. TRIANGLE_NEIGHBOR(1,I) is
negative
! if there is no neighbor on that side. In this case, that side of the
! triangle lies on the boundary of the triangulation.
!
implicit none

integer ( kind = 4 ) triangle_num
integer ( kind = 4 ) triangle_order

integer ( kind = 4 ) col(4,3*triangle_num)
integer ( kind = 4 ) i
integer ( kind = 4 ) icol
integer ( kind = 4 ) j
integer ( kind = 4 ) k
integer ( kind = 4 ) side1
integer ( kind = 4 ) side2
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) tri
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)
integer ( kind = 4 ) tri1
integer ( kind = 4 ) tri2

! Step 1.
! From the list of nodes for triangle T, of the form: (I,J,K)
! construct the three neighbor relations:
! (I,J,3,T) or (J,I,3,T),
! (J,K,1,T) or (K,J,1,T),
! (K,I,2,T) or (I,K,2,T)
!
! where we choose (I,J,1,T) if I < J, or else (J,I,1,T)
!
do tri = 1, triangle_num

  i = triangle_node(1,tri)
  j = triangle_node(2,tri)
  k = triangle_node(3,tri)

  if ( i < j ) then
    col(1:4,3*(tri-1)+1) = (/ i, j, 3, tri /)
  else
    col(1:4,3*(tri-1)+1) = (/ j, i, 3, tri /)
  end if

  if ( j < k ) then
    col(1:4,3*(tri-1)+2) = (/ j, k, 1, tri /)
  else
    col(1:4,3*(tri-1)+2) = (/ k, j, 1, tri /)
  end if
if ( k < i ) then
  col(1:4,3*(tri-1)+3) = (/ k, i, 2, tri /)
else
  col(1:4,3*(tri-1)+3) = (/ i, k, 2, tri /)
end if

end do

! Step 2. Perform an ascending dictionary sort on the neighbor relations.
! We only intend to sort on rows 1 and 2; the routine we call here
! sorts on rows 1 through 4 but that won't hurt us.
!
! What we need is to find cases where two triangles share an edge.
! Say they share an edge defined by the nodes I and J. Then there are
! two columns of COL that start out ( I, J, ?, ? ). By sorting COL,
! we make sure that these two columns occur consecutively. That will
! make it easy to notice that the triangles are neighbors.
!
call i4col_sort_a ( 4, 3*triangle_num, col )
!
! Step 3. Neighboring triangles show up as consecutive columns with
! identical first two entries. Whenever you spot this happening,
! make the appropriate entries in TRIANGLE_NEIGHBOR.
!
triangle_neighbor(1:3,1:triangle_num) = -1
icol = 1
!!
do
  if ( 3 * triangle_num <= icol ) then
    exit
  end if
  if ( col(1,icol) /= col(1,icol+1) .or. col(2,icol) /= col(2,icol+1) )
  then
    icol = icol + 1
    cycle
  end if
  side1 = col(3,icol)
  tri1 = col(4,icol)
  side2 = col(3,icol+1)
  tri2 = col(4,icol+1)
  triangle_neighbor(side1,tri1) = tri2
  triangle_neighbor(side2,tri2) = tri1
  icol = icol + 2
end do
!!
return
!!
end subroutine triangulation_order3_adj_count ( node_num, triangle_num, &
triangle_node, triangle_neighbor, adj_num, adj_col )

**************************************************************************
***80
TRIANGULATION_ORDER3_ADJ_COUNT counts adjacencies in a triangulation.

Discussion:

This routine is called to count the adjacencies, so that the appropriate amount of memory can be set aside for storage when the adjacency structure is created.

The triangulation is assumed to involve 3-node triangles.

Two nodes are "adjacent" if they are both nodes in some triangle. Also, a node is considered to be adjacent to itself.

Diagram:

```
  3
 s \|
i \| d \|
e \| \ side 2
 3 \|
1------2
```

The local node numbering

```
21-22-23-24-25
  \| \| \| \| \|
16-17-18-19-20
  \| \| \| \| \|
11-12-13-14-15
  \| \| \| \| \|
 6---7---8---9---10
  \| \| \| \| \|
 1---2---3---4---5
```

A sample grid.

Below, we have a chart that summarizes the adjacency relationships in the sample grid. On the left, we list the node, and its neighbors, with an asterisk to indicate the adjacency of the node to itself (in some cases, you want to count this self adjacency and in some you don't). On the right, we list the number of adjacencies to lower-indexed nodes, to the node itself, to higher-indexed nodes, the total number of adjacencies for this node, and the location of the first and last entries required to list this set of adjacencies in a single list of all the adjacencies.

```
  N  Adjacencies  Below  Self Above  Total  First  Last
  -- -- -- -- -- -- -- -- -- -- -- ---
```

239
| 1: * 2 6 | 0 1 2 3 1 3 |
| 2: * 3 6 7 | 1 1 3 5 4 8 |
| 3: * 4 7 8 | 1 1 3 5 9 13 |
| 4: * 5 8 9 | 1 1 3 5 14 18 |
| 5: * 9 10 | 1 1 2 4 19 22 |
| 6: 1 2 * 7 11 | 2 1 2 5 23 27 |
| 7: 2 3 * 8 11 12 | 3 1 3 7 28 34 |
| 8: * 3 4 7 9 * 11 12 | 3 1 3 7 35 41 |
| 9: * 4 5 8 10 * 13 14 | 3 1 3 7 42 48 |
| 10: 5 9 * 14 15 | 2 1 2 5 49 53 |
| 11: 6 7 * 12 16 | 2 1 2 5 54 58 |
| 12: 7 8 11 * 13 15 17 | 3 1 3 7 59 65 |
| 13: * 9 12 14 * 17 18 | 3 1 3 7 66 72 |
| 14: * 10 13 * 15 18 19 | 3 1 3 7 73 79 |
| 15: 10 14 * 19 20 | 2 1 2 5 80 84 |
| 16: 11 12 * 17 21 | 2 1 2 5 85 89 |
| 17: 12 13 16 * 18 21 22 | 3 1 3 7 90 96 |
| 18: 13 14 17 * 19 22 23 | 3 1 3 7 97 103 |
| 19: * 14 15 18 * 20 23 24 | 3 1 3 7 104 110 |
| 20: * 15 19 24 25 | 2 1 2 5 111 115 |
| 21: * 16 17 22 | 2 1 1 4 116 119 |
| 22: 17 18 21 * 23 | 3 1 1 5 120 124 |
| 23: * 18 19 22 24 | 3 1 1 5 125 129 |
| 24: 19 20 23 * 25 | 3 1 1 5 130 134 |
| 25: 20 24 * | 2 1 0 3 135 137 |
| -- -- -- -- -- -- -- | -- -- -- -- -- -- 138 -- |

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 August 2006

Author:

John Burkardt

Parameters

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.
Input, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.
Input, integer ( kind = 4 ) TRIANGLE_NODE(3,TRIANGLE_NUM), lists the nodes
that make up each triangle, in counterclockwise order.
Input, integer ( kind = 4 ) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), for each
side of a triangle, lists the neighboring triangle, or -1 if there is
no neighbor.
Output, integer ( kind = 4 ) ADJ_NUM, the number of adjacencies.
Output, integer ( kind = 4 ) ADJ_COL(NODE_NUM+1). Information about
column J is stored in entries ADJ_COL(J) through ADJ_COL(J+1)-1 of
ADJ.
implicit none

integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num
integer ( kind = 4 ), parameter :: triangle_order = 3

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) adj_col(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) n1
integer ( kind = 4 ) n2
integer ( kind = 4 ) n3
integer ( kind = 4 ) triangle
integer ( kind = 4 ) triangle2
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)

adj_num = 0
!
! Set every node to be adjacent to itself.
!
adj_col(1:node_num) = 1
!
! Examine each triangle.
!
do triangle = 1, triangle_num

   n1 = triangle_node(1,triangle)
   n2 = triangle_node(2,triangle)
   n3 = triangle_node(3,triangle)
!
! Add edge (1,2) if this is the first occurrence,
! that is, if the edge (1,2) is on a boundary (TRIANGLE2 <= 0)
! or if this triangle is the first of the pair in which the edge
! occurs (TRIANGLE < TRIANGLE2).
!
   triangle2 = triangle_neighbor(1,triangle)
   if ( triangle2 < 0 .or. triangle < triangle2 ) then
      adj_col(n1) = adj_col(n1) + 1
      adj_col(n2) = adj_col(n2) + 1
   end if
!
! Add edge (2,3).
!
   triangle2 = triangle_neighbor(2,triangle)
   if ( triangle2 < 0 .or. triangle < triangle2 ) then
      adj_col(n2) = adj_col(n2) + 1
      adj_col(n3) = adj_col(n3) + 1
   end if
!
! Add edge (3,1).
!
   triangle2 = triangle_neighbor(3,triangle)
   if ( triangle2 < 0 .or. triangle < triangle2 ) then
      adj_col(n1) = adj_col(n1) + 1
      adj_col(n3) = adj_col(n3) + 1
   end if
end do
!
! We used ADJ_COL to count the number of entries in each column.
! Convert it to pointers into the ADJ array.
!
adj_col(2:node_num+1) = adj_col(1:node_num)

adj_col(1) = 1
do i = 2, node_num + 1
   adj_col(i) = adj_col(i-1) + adj_col(i)
end do

adj_num = adj_col(node_num+1) - 1

return
end

subroutine triangulation_order3_adj_set ( node_num, triangle_num, &
   triangle_node, triangle_neighbor, adj_num, adj_col, adj )

**************************************************************************
!! TRIANGULATION_ORDER3_ADJ_SET sets adjacencies in a triangulation.
!!
!! This routine is called to count the adjacencies, so that the
!! appropriate amount of memory can be set aside for storage when
!! the adjacency structure is created.
!!
!! The triangulation is assumed to involve 3-node triangles.
!!
!! Two nodes are "adjacent" if they are both nodes in some triangle.
!! Also, a node is considered to be adjacent to itself.
!!
!! This routine can be used to create the compressed column storage
!! for a linear triangle finite element discretization of
!! Poisson's equation in two dimensions.
!!
!! Diagram:
!!
!!    3
!!     |
!!     i   d
!!     |   |
!!     e \  \ side 2
!!      \ \  
!!       \ \  
!!         \ 
!!           \ 1------2
!!             \ side 1
!!
!! The local node numbering
!!
!!
!!   21-22-23-24-25
!!   | | | | | |
!!   | | | | | |
Below, we have a chart that summarizes the adjacency relationships in the sample grid. On the left, we list the node, and its neighbors, with an asterisk to indicate the adjacency of the node to itself (in some cases, you want to count this self adjacency and in some you don't). On the right, we list the number of adjancencies to lower-indexed nodes, to the node itself, to higher-indexed nodes, the total number of adjacencies for this node, and the location of the first and last entries required to list this set of adjacencies in a single list of all the adjacencies.

<table>
<thead>
<tr>
<th>N</th>
<th>Adjacencies</th>
<th>Below</th>
<th>Self</th>
<th>Above</th>
<th>Total</th>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>* 2 6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1 * 3 6 7</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2 * 4 7 8</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>3 * 5 8 9</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>4 * 9 10</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>1 2 * 7 11</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>2 3 6 * 8 11 12</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>28</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>3 4 7 * 9 12 13</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>35</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>4 5 8 * 10 13 14</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>42</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>5 9 * 14 15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>49</td>
<td>53</td>
</tr>
<tr>
<td>11</td>
<td>6 7 * 12 16</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>54</td>
<td>58</td>
</tr>
<tr>
<td>12</td>
<td>7 8 11 * 13 16 17</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>59</td>
<td>65</td>
</tr>
<tr>
<td>13</td>
<td>8 9 12 * 14 17 18</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>66</td>
<td>72</td>
</tr>
<tr>
<td>14</td>
<td>9 10 13 * 15 18 19</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>73</td>
<td>79</td>
</tr>
<tr>
<td>15</td>
<td>10 14 * 19 20</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>80</td>
<td>84</td>
</tr>
<tr>
<td>16</td>
<td>11 12 * 17 21</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>85</td>
<td>89</td>
</tr>
<tr>
<td>17</td>
<td>12 13 16 * 18 21 22</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>90</td>
<td>96</td>
</tr>
<tr>
<td>18</td>
<td>13 14 17 * 19 22 23</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>97</td>
<td>103</td>
</tr>
<tr>
<td>19</td>
<td>14 15 18 * 20 23 24</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>104</td>
<td>110</td>
</tr>
<tr>
<td>20</td>
<td>15 19 * 24 25</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>111</td>
<td>115</td>
</tr>
<tr>
<td>21</td>
<td>16 17 * 22</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>116</td>
<td>119</td>
</tr>
<tr>
<td>22</td>
<td>17 18 21 * 23</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>120</td>
<td>124</td>
</tr>
<tr>
<td>23</td>
<td>18 19 22 * 24</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>125</td>
<td>129</td>
</tr>
<tr>
<td>24</td>
<td>19 20 23 * 25</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>130</td>
<td>134</td>
</tr>
<tr>
<td>25</td>
<td>20 24 *</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>135</td>
<td>137</td>
</tr>
</tbody>
</table>

Licensing:

This code is distributed under the GNU LGPL license.

Modified:
implicit none

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num
integer ( kind = 4 ), parameter :: triangle_order = 3

integer ( kind = 4 ) adj(adj_num)
integer ( kind = 4 ) adj_copy(node_num)
integer ( kind = 4 ) adj_col(node_num+1)
integer ( kind = 4 ) k1
integer ( kind = 4 ) k2
integer ( kind = 4 ) n1
integer ( kind = 4 ) n2
integer ( kind = 4 ) n3
integer ( kind = 4 ) node
integer ( kind = 4 ) triangle
integer ( kind = 4 ) triangle2
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)

adj(1:adj_num) = -1
adj_copy(1:node_num) = adj_col(1:node_num)

! Set every node to be adjacent to itself.
!
do node = 1, node_num
    adj(adj_copy(node)) = node
    adj_copy(node) = adj_copy(node) + 1
end do


Examine each triangle.

do triangle = 1, triangle_num

    n1 = triangle_node(1,triangle)
    n2 = triangle_node(2,triangle)
    n3 = triangle_node(3,triangle)

    triangle2 = triangle_neighbor(1,triangle)

    if ( triangle2 < 0 .or. triangle < triangle2 ) then
        adj(adj_copy(n1)) = n2
        adj_copy(n1) = adj_copy(n1) + 1
        adj(adj_copy(n2)) = n1
        adj_copy(n2) = adj_copy(n2) + 1
    end if

    triangle2 = triangle_neighbor(2,triangle)

    if ( triangle2 < 0 .or. triangle < triangle2 ) then
        adj(adj_copy(n2)) = n3
        adj_copy(n2) = adj_copy(n2) + 1
        adj(adj_copy(n3)) = n2
        adj_copy(n3) = adj_copy(n3) + 1
    end if

    triangle2 = triangle_neighbor(3,triangle)

    if ( triangle2 < 0 .or. triangle < triangle2 ) then
        adj(adj_copy(n1)) = n3
        adj_copy(n1) = adj_copy(n1) + 1
        adj(adj_copy(n3)) = n1
        adj_copy(n3) = adj_copy(n3) + 1
    end if

end do

Ascending sort the entries for each node.

do node = 1, node_num

    k1 = adj_col(node)
    k2 = adj_col(node+1) - 1
    call i4vec_sort_heap_a ( k2+1-k1, adj(k1:k2) )
end do

return
end subroutine triangulation_order3_example2 ( node_num, triangle_num, node_xy, & triangle_node, triangle_neighbor )
TRIANGULATION_ORDER3_EXAMPLE2 returns an example triangulation.

Discussion:

This triangulation is actually a Delaunay triangulation.

The appropriate input values of NODE_NUM and TRIANGLE_NUM can be determined by calling TRIANGULATION_ORDER3_EXAMPLE2_SIZE first.

Diagram:

```
  21-22-23-24-25
  |   |   |   |   |
  16-17-18-19-20
  |   |   |   |   |
  11-12-13-14-15
  |   |   |   |   |
  6--7--8--9--10
  |   |   |   |   |
  1--2--3--4--5
```

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

03 January 2007

Author:

John Burkardt

Parameters

- Input, integer (kind = 4) NODE_NUM, the number of nodes.
- Input, integer (kind = 4) TRIANGLE_NUM, the number of triangles.
- Output, real (kind = 8) NODE_XY(2,NODE_NUM), the coordinates of the nodes.
- Output, integer (kind = 4) TRIANGLE_NODE(3,TRIANGLE_NUM), lists the nodes that make up each triangle, in counterclockwise order.
- Output, integer (kind = 4) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), for each side of a triangle, lists the neighboring triangle, or -1 if there is no neighbor.

```
integer (kind = 4), parameter :: dim_num = 2
```
integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num
integer ( kind = 4 ), parameter :: triangle_order = 3
real ( kind = 8 ) node_xy(dim_num,node_num)
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)

node_xy = reshape ( (/ &
  0.0D+00, 0.0D+00, &
  1.0D+00, 0.0D+00, &
  2.0D+00, 0.0D+00, &
  3.0D+00, 0.0D+00, &
  4.0D+00, 0.0D+00, &
  0.0D+00, 1.0D+00, &
  1.0D+00, 1.0D+00, &
  2.0D+00, 1.0D+00, &
  3.0D+00, 1.0D+00, &
  4.0D+00, 1.0D+00, &
  0.0D+00, 2.0D+00, &
  1.0D+00, 2.0D+00, &
  2.0D+00, 2.0D+00, &
  3.0D+00, 2.0D+00, &
  4.0D+00, 2.0D+00, &
  0.0D+00, 3.0D+00, &
  1.0D+00, 3.0D+00, &
  2.0D+00, 3.0D+00, &
  3.0D+00, 3.0D+00, &
  4.0D+00, 3.0D+00, &
  0.0D+00, 4.0D+00, &
  1.0D+00, 4.0D+00, &
  2.0D+00, 4.0D+00, &
  3.0D+00, 4.0D+00, &
  4.0D+00, 4.0D+00 & &
/) , (/ dim_num, node_num /))

triangle_node(1:triangle_order,1:triangle_num) = reshape ( (/ &
  1, 2, 6, &
  7, 6, 2, &
  2, 3, 7, &
  8, 7, 3, &
  3, 4, 8, &
  9, 8, 4, &
  4, 5, 9, &
 10, 9, 5, &
 6, 7, 11, &
12, 11, 7, &
 7, 8, 12, &
13, 12, 8, &
 8, 9, 13, &
14, 13, 9, &
 9, 10, 14, &
15, 14, 10, &
11, 12, 16, &
17, 16, 12, &
12, 13, 17, &
18, 17, 13, &
13, 14, 18, &
19, 18, 14, &
14, 15, 19, &
/) , (/ triangle_order, triangle_num /))
20, 19, 15, &
16, 17, 21, &
22, 21, 17, &
17, 18, 22, &
23, 22, 18, &
18, 19, 23, &
24, 23, 19, &
19, 20, 24, &
25, 24, 20 /), (/ triangle_order, triangle_num /) )

triangle_neighbor(1:3,1:triangle_num) = reshape ( (/ &
-1, 2, -1, &
9, 1, 3, &
-1, 4, 2, &
11, 3, 5, &
-1, 6, 4, &
13, 5, 7, &
-1, 8, 6, &
15, 7, -1, &
2, 10, -1, &
17, 9, 11, &
4, 12, 10, &
19, 11, 13, &
6, 14, 12, &
21, 13, 15, &
8, 16, 14, &
23, 15, -1, &
10, 18, -1, &
25, 17, 19, &
12, 20, 18, &
27, 19, 21, &
14, 22, 20, &
29, 21, 23, &
16, 24, 22, &
31, 23, -1, &
18, 26, -1, &
-1, 25, 27, &
20, 28, 26, &
-1, 27, 29, &
22, 30, 28, &
-1, 29, 31, &
24, 32, 30, &
-1, 31, -1 /), (/ 3, triangle_num /) )

return
end

subroutine triangulation_order3_example2_size ( node_num, triangle_num, &
hole_num )

!**************************************************************************
***80
!
!! TRIANGULATION_ORDER3_EXAMPLE2_SIZE returns the size of an example.
!!
!! Diagram:
!!
!! 21-22-23-24-25
!!    |   |   |   |
!! 16-17-18-19-20
Licensing:

This code is distributed under the GNU LGPL license.

Modified:

03 January 2007

Author:

John Burkardt

Parameters

Output, integer ( kind = 4 ) NODE_NUM, the number of nodes.
Output, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.
Output, integer ( kind = 4 ) HOLE_NUM, the number of holes.

implicit none

integer ( kind = 4 ) hole_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num

node_num = 25
triangle_num = 32
hole_num = 0

return
end

subroutine triangulation_order6_adj_count ( node_num, triangle_num, &
triangle_node, triangle_neighbor, adj_num, adj_col )

**************************************************************************

***80
!! TRIANGULATION_ORDER6_ADJ_COUNT counts adjacencies in a triangulation.
!!
Discussion:

This routine is called to count the adjacencies, so that the
appropriate amount of memory can be set aside for storage when
the adjacency structure is created.

The triangulation is assumed to involve 6-node triangles.

Two nodes are "adjacent" if they are both nodes in some triangle.
Also, a node is considered to be adjacent to itself.
Diagram:

```
  3
 s |\
 i | \|
 d | \|
 e 6 5  side 2
 3 | \|
 1----4----2
 side 1

The local node numbering

21-22-23-24-25
 | \   | \   |
 | | 16 17 18 19 20
 | | \| 11-12-13-14-15
 | \  \  \  \\
 6 7 8 9 10
 | | \   | \\
 | | | 1--2--3--4--5
 |

A sample grid.

Below, we have a chart that lists the nodes adjacent to each node, with an asterisk to indicate the adjacency of the node to itself (in some cases, you want to count this self adjacency and in some you don't).

<table>
<thead>
<tr>
<th>N</th>
<th>Adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>* 2 3 6 7 11</td>
</tr>
<tr>
<td>2:</td>
<td>1 * 3 6 7 11</td>
</tr>
<tr>
<td>3:</td>
<td>1 2 * 4 5 6 7 8 9 11 12 13</td>
</tr>
<tr>
<td>4:</td>
<td>3 * 5 8 9 13</td>
</tr>
<tr>
<td>5:</td>
<td>3 4 * 8 9 10 13 14 15</td>
</tr>
<tr>
<td>6:</td>
<td>1 2 3 * 7 11</td>
</tr>
<tr>
<td>7:</td>
<td>1 2 3 6 * 8 11 12 13</td>
</tr>
<tr>
<td>8:</td>
<td>3 4 5 7 * 9 11 12 13</td>
</tr>
<tr>
<td>9:</td>
<td>3 4 5 8 * 10 13 14 15</td>
</tr>
<tr>
<td>10:</td>
<td>5 9 * 13 14 15</td>
</tr>
<tr>
<td>11:</td>
<td>1 2 3 6 7 8 * 12 13 16 17 21</td>
</tr>
<tr>
<td>12:</td>
<td>3 7 8 11 * 13 16 17 21</td>
</tr>
<tr>
<td>13:</td>
<td>3 4 5 7 8 9 10 11 12 * 14 15 16 17 18 19 21 22 23</td>
</tr>
<tr>
<td>14:</td>
<td>5 9 10 13 * 15 18 19 23</td>
</tr>
<tr>
<td>15:</td>
<td>5 9 10 13 14 * 18 19 20 23 24 25</td>
</tr>
<tr>
<td>16:</td>
<td>11 12 13 * 17 21</td>
</tr>
<tr>
<td>17:</td>
<td>11 12 13 16 * 18 21 22 23</td>
</tr>
</tbody>
</table>
```
Below, we list the number of adjacencies to lower-indexed nodes, to the node itself, to higher-indexed nodes, the total number of adjacencies for this node, and the location of the first and last entries required to list this set of adjacencies in a single list of all the adjacencies.

<table>
<thead>
<tr>
<th>N</th>
<th>Below</th>
<th>Self</th>
<th>Above</th>
<th>Total</th>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>31</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>40</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>46</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>55</td>
<td>63</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>62</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>73</td>
<td>78</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>79</td>
<td>90</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>91</td>
<td>99</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>19</td>
<td>100</td>
<td>118</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>119</td>
<td>127</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>128</td>
<td>139</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>140</td>
<td>145</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>146</td>
<td>154</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>155</td>
<td>163</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>164</td>
<td>172</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>173</td>
<td>178</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>179</td>
<td>187</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>188</td>
<td>193</td>
</tr>
<tr>
<td>23</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>194</td>
<td>205</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>206</td>
<td>211</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>212</td>
<td>217</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>218</td>
<td>---</td>
</tr>
</tbody>
</table>

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 August 2006

Author:

John Burkardt

Parameters:

Input, integer (kind = 4) NODE_NUM, the number of nodes.
Input, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.

Input, integer ( kind = 4 ) TRIANGLE_NODE(6,TRIANGLE_NUM), lists the nodes
that make up each triangle. The first three nodes are the vertices,
in counterclockwise order. The fourth value is the midside
node between nodes 1 and 2; the fifth and sixth values are
the other midside nodes in the logical order.

Input, integer ( kind = 4 ) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), for each
side of a triangle, lists the neighboring triangle, or -1 if there is
no neighbor.

Output, integer ( kind = 4 ) ADJ_NUM, the number of adjacencies.

Output, integer ( kind = 4 ) ADJ_COL(NODE_NUM+1). Information about
column J is stored in entries ADJ_COL(J) through ADJ_COL(J+1)-1 of
ADJ.

implicit none

integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num
integer ( kind = 4 ), parameter :: triangle_order = 6

integer ( kind = 4 ) adj_num
integer ( kind = 4 ) adj_col(node_num+1)
integer ( kind = 4 ) i
integer ( kind = 4 ) n1
integer ( kind = 4 ) n2
integer ( kind = 4 ) n3
integer ( kind = 4 ) n4
integer ( kind = 4 ) n5
integer ( kind = 4 ) n6
integer ( kind = 4 ) triangle
integer ( kind = 4 ) triangle2
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)

adj_num = 0

! Set every node to be adjacent to itself.

adj_col(1:node_num) = 1

! Examine each triangle.

do triangle = 1, triangle_num

n1 = triangle_node(1,triangle)
n2 = triangle_node(2,triangle)
n3 = triangle_node(3,triangle)
n4 = triangle_node(4,triangle)
n5 = triangle_node(5,triangle)
n6 = triangle_node(6,triangle)

! For sure, we add the adjacencies:

43 / (34)
adj_col(n3) = adj_col(n3) + 1
adj_col(n4) = adj_col(n4) + 1
adj_col(n1) = adj_col(n1) + 1
adj_col(n5) = adj_col(n5) + 1
adj_col(n4) = adj_col(n4) + 1
adj_col(n5) = adj_col(n5) + 1
adj_col(n2) = adj_col(n2) + 1
adj_col(n6) = adj_col(n6) + 1
adj_col(n4) = adj_col(n4) + 1
adj_col(n6) = adj_col(n6) + 1
adj_col(n5) = adj_col(n5) + 1
adj_col(n6) = adj_col(n6) + 1

! Add edges (1,2), (1,4), (2,4) if this is the first occurrence,
! that is, if the edge (1,4,2) is on a boundary (TRIANGLE2 <= 0)
! or if this triangle is the first of the pair in which the edge
! occurs (TRIANGLE < TRIANGLE2).
!
! Maybe add
! 21 / 12
! 41 / 14
! 42 / 24

triangle2 = triangle_neighbor(1,triangle)

if ( triangle2 < 0 .or. triangle < triangle2 ) then
    adj_col(n1) = adj_col(n1) + 1
    adj_col(n2) = adj_col(n2) + 1
    adj_col(n1) = adj_col(n1) + 1
    adj_col(n4) = adj_col(n4) + 1
    adj_col(n2) = adj_col(n2) + 1
    adj_col(n4) = adj_col(n4) + 1
end if
!
! Maybe add
! 32 / 23
! 52 / 25
! 53 / 35

triangle2 = triangle_neighbor(2,triangle)

if ( triangle2 < 0 .or. triangle < triangle2 ) then
    adj_col(n2) = adj_col(n2) + 1
    adj_col(n3) = adj_col(n3) + 1
    adj_col(n2) = adj_col(n2) + 1
    adj_col(n5) = adj_col(n5) + 1
    adj_col(n3) = adj_col(n3) + 1
    adj_col(n5) = adj_col(n5) + 1
end if
!
! Maybe add
! 31 / 13
! 61 / 16
! 63 / 36
triangle2 = triangle_neighbor(3, triangle)

if (triangle2 ≤ 0 .or. triangle < triangle2) then
  adj_col(n1) = adj_col(n1) + 1
  adj_col(n3) = adj_col(n3) + 1
  adj_col(n1) = adj_col(n1) + 1
  adj_col(n6) = adj_col(n6) + 1
  adj_col(n3) = adj_col(n3) + 1
  adj_col(n6) = adj_col(n6) + 1
end if

end do

adj_col(2:node_num+1) = adj_col(1:node_num)
adj_col(1) = 1
do i = 2, node_num + 1
  adj_col(i) = adj_col(i-1) + adj_col(i)
end do
adj_num = adj_col(node_num+1) - 1
return
end subroutine triangulation_order6_adj_set ( node_num, triangle_num, 
triangle_node, triangle_neighbor, adj_num, adj_col, adj )

**************************************************************************
***80
!! TRIANGULATION_ORDER6_ADJ_SET sets adjacencies in a triangulation.
!!
!!  Discussion:
!!  
!!    This routine is called to count the adjacencies, so that the
!!    appropriate amount of memory can be set aside for storage when
!!    the adjacency structure is created.
!!
!!    The triangulation is assumed to involve 6-node triangles.
!!
!!    Two nodes are "adjacent" if they are both nodes in some triangle.
!!    Also, a node is considered to be adjacent to itself.
!!
!!    This routine can be used to create the compressed column storage
!!    for a quadratic triangle finite element discretization of
!!    Poisson’s equation in two dimensions.
!!
!!  Diagram:
!!  
!!    3
!!    s |\   
!!    i | \   
!!    d | \   
!!    e 6 5 side 2
!!    | \   
!!    3 | 

254
The local node numbering

21-22-23-24-25
16 17 18 19 20
11-12-13-14-15
6 7 8 9 10
1--2--3--4--5

A sample grid.

Below, we have a chart that lists the nodes adjacent to each node, with an asterisk to indicate the adjacency of the node to itself (in some cases, you want to count this self adjacency and in some you don't).

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c} \hline N & Adjacencies \\
1 & * 2 3 6 7 11 \\
2 & 1 * 3 6 7 11 \\
3 & 1 2 * 4 5 6 7 8 9 11 12 13 \\
4 & 3 * 5 8 9 13 \\
5 & 3 4 * 8 9 10 13 14 15 \\
6 & 1 2 3 * 7 11 \\
7 & 1 2 3 6 * 8 11 12 13 \\
8 & 3 4 5 7 * 9 11 12 13 \\
9 & 3 4 5 8 * 10 13 14 15 \\
10 & 5 9 * 13 14 15 \\
11 & 1 2 3 6 7 8 * 12 13 16 17 21 \\
12 & 3 7 8 11 * 13 16 17 21 \\
13 & 3 4 5 7 8 9 10 11 12 * 14 15 16 17 18 19 21 22 23 \\
14 & 5 9 10 13 * 15 18 19 23 \\
15 & 5 9 10 13 14 * 18 19 20 23 24 25 \\
16 & 11 12 13 * 17 21 \\
17 & 11 12 13 16 * 18 21 22 23 \\
18 & 13 14 15 17 * 19 21 22 23 \\
19 & 13 14 15 18 * 20 23 24 25 \\
20 & 15 19 * 23 24 25 \\
21 & 11 12 13 16 17 18 * 22 23 \\
22 & 13 17 18 21 * 23 \\
23 & 13 14 15 17 18 19 20 21 22 * 24 25 \\
24 & 15 19 20 23 * 25 \\
25 & 15 19 20 23 24 * \\
\hline \end{array} \]

Below, we list the number of adjancencies to lower-indexed nodes, to
the node itself, to higher-indexed nodes, the total number of
adjacencies for this node, and the location of the first and last
entries required to list this set of adjacencies in a single list
of all the adjacencies.

<table>
<thead>
<tr>
<th>N</th>
<th>Below</th>
<th>Self</th>
<th>Above</th>
<th>Total</th>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>---</td>
<td>0</td>
</tr>
<tr>
<td>1:</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>3:</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>4:</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>5:</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>31</td>
<td>39</td>
</tr>
<tr>
<td>6:</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>40</td>
<td>45</td>
</tr>
<tr>
<td>7:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>46</td>
<td>54</td>
</tr>
<tr>
<td>8:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>55</td>
<td>63</td>
</tr>
<tr>
<td>9:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>62</td>
<td>72</td>
</tr>
<tr>
<td>10:</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>73</td>
<td>78</td>
</tr>
<tr>
<td>11:</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>79</td>
<td>90</td>
</tr>
<tr>
<td>12:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>91</td>
<td>99</td>
</tr>
<tr>
<td>13:</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>19</td>
<td>100</td>
<td>118</td>
</tr>
<tr>
<td>14:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>119</td>
<td>127</td>
</tr>
<tr>
<td>15:</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>128</td>
<td>139</td>
</tr>
<tr>
<td>16:</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>140</td>
<td>145</td>
</tr>
<tr>
<td>17:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>146</td>
<td>154</td>
</tr>
<tr>
<td>18:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>155</td>
<td>163</td>
</tr>
<tr>
<td>19:</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>164</td>
<td>172</td>
</tr>
<tr>
<td>20:</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>173</td>
<td>178</td>
</tr>
<tr>
<td>21:</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>179</td>
<td>187</td>
</tr>
<tr>
<td>22:</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>188</td>
<td>193</td>
</tr>
<tr>
<td>23:</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>194</td>
<td>205</td>
</tr>
<tr>
<td>24:</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>206</td>
<td>211</td>
</tr>
<tr>
<td>25:</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>212</td>
<td>217</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>218</td>
<td>--</td>
</tr>
</tbody>
</table>

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 August 2006

Author:

John Burkardt

Parameters

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

Input, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.

Input, integer ( kind = 4 ) TRIANGLE_NODE(6,TRIANGLE_NUM), lists the
nodes that make up each triangle. The first three nodes are the vertices,
in counterclockwise order. The fourth value is the midside
node between nodes 1 and 2; the fifth and sixth values are
the other midside nodes in the logical order.
Input, integer (kind = 4) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), for each side of a triangle, lists the neighboring triangle, or -1 if there is no neighbor.

Input, integer (kind = 4) ADJ_NUM, the number of adjacencies.

Input, integer (kind = 4) ADJ_COL(NODE_NUM+1). Information about column J is stored in entries ADJ_COL(J) through ADJ_COL(J+1)-1 of ADJ.

Output, integer (kind = 4) ADJ(ADJ_NUM), the adjacency information.

implicit none

integer (kind = 4) adj_num
integer (kind = 4) node_num
integer (kind = 4) triangle_num
integer (kind = 4), parameter :: triangle_order = 6

integer (kind = 4) adj(adj_num)
integer (kind = 4) adj_copy(node_num)
integer (kind = 4) adj_col(node_num+1)
integer (kind = 4) k1
integer (kind = 4) k2
integer (kind = 4) n1
integer (kind = 4) n2
integer (kind = 4) n3
integer (kind = 4) n4
integer (kind = 4) n5
integer (kind = 4) n6
integer (kind = 4) node
integer (kind = 4) triangle
integer (kind = 4) triangle2
integer (kind = 4) triangle_neighbor(3,triangle_num)
integer (kind = 4) triangle_node(triangle_order,triangle_num)

adj(1:adj_num) = -1
adj_copy(1:node_num) = adj_col(1:node_num)

! Set every node to be adjacent to itself.
!
do node = 1, node_num
   adj(adj_copy(node)) = node
   adj_copy(node) = adj_copy(node) + 1
end do
!
! Examine each triangle.
!
do triangle = 1, triangle_num

   n1 = triangle_node(1,triangle)
   n2 = triangle_node(2,triangle)
   n3 = triangle_node(3,triangle)
   n4 = triangle_node(4,triangle)
   n5 = triangle_node(5,triangle)
   n6 = triangle_node(6,triangle)
!
! For sure, we add the adjacencies:
! 43 / (34)
adj(adj_copy(n3)) = n4
adj_copy(n3) = adj_copy(n3) + 1
adj(adj_copy(n4)) = n3
adj_copy(n4) = adj_copy(n4) + 1
adj(adj_copy(n1)) = n5
adj_copy(n1) = adj_copy(n1) + 1
adj(adj_copy(n5)) = n1
adj_copy(n5) = adj_copy(n5) + 1
adj(adj_copy(n4)) = n5
adj_copy(n4) = adj_copy(n4) + 1
adj(adj_copy(n5)) = n4
adj_copy(n5) = adj_copy(n5) + 1
adj(adj_copy(n1)) = n4
adj_copy(n1) = adj_copy(n1) + 1
adj(adj_copy(n4)) = n1
adj_copy(n4) = adj_copy(n4) + 1
adj(adj_copy(n2)) = n6
adj_copy(n2) = adj_copy(n2) + 1
adj(adj_copy(n6)) = n2
adj_copy(n6) = adj_copy(n6) + 1
adj(adj_copy(n4)) = n6
adj_copy(n4) = adj_copy(n4) + 1
adj(adj_copy(n6)) = n4
adj_copy(n6) = adj_copy(n6) + 1
adj(adj_copy(n5)) = n6
adj_copy(n5) = adj_copy(n5) + 1
adj(adj_copy(n6)) = n5
adj_copy(n6) = adj_copy(n6) + 1

triangle2 = triangle_neighbor(1,triangle)

if ( triangle2 < 0 .or. triangle < triangle2 ) then
    adj(adj_copy(n1)) = n2
    adj_copy(n1) = adj_copy(n1) + 1
    adj(adj_copy(n2)) = n1
    adj_copy(n2) = adj_copy(n2) + 1
    adj(adj_copy(n1)) = n4
    adj_copy(n1) = adj_copy(n1) + 1
    adj(adj_copy(n4)) = n1
    adj_copy(n4) = adj_copy(n4) + 1
    adj(adj_copy(n2)) = n4
    adj_copy(n2) = adj_copy(n2) + 1
    adj(adj_copy(n4)) = n2
end if
adj_copy(n4) = adj_copy(n4) + 1
end if

if ( triangle2 < 0 .or. triangle < triangle2 ) then
  adj(adj_copy(n2)) = n3
  adj_copy(n2) = adj_copy(n2) + 1
  adj(adj_copy(n3)) = n2
  adj_copy(n3) = adj_copy(n3) + 1
  adj(adj_copy(n2)) = n5
  adj_copy(n2) = adj_copy(n2) + 1
  adj(adj_copy(n5)) = n2
  adj_copy(n5) = adj_copy(n5) + 1
  adj(adj_copy(n3)) = n5
  adj_copy(n3) = adj_copy(n3) + 1
  adj(adj_copy(n5)) = n3
  adj_copy(n5) = adj_copy(n5) + 1
end if

triangle2 = triangle_neighbor(2,triangle)

if ( triangle2 < 0 .or. triangle < triangle2 ) then
  adj(adj_copy(n1)) = n3
  adj_copy(n1) = adj_copy(n1) + 1
  adj(adj_copy(n3)) = n1
  adj_copy(n3) = adj_copy(n3) + 1
  adj(adj_copy(n1)) = n6
  adj_copy(n1) = adj_copy(n1) + 1
  adj(adj_copy(n6)) = n1
  adj_copy(n6) = adj_copy(n6) + 1
  adj(adj_copy(n3)) = n6
  adj_copy(n3) = adj_copy(n3) + 1
  adj(adj_copy(n6)) = n3
  adj_copy(n6) = adj_copy(n6) + 1
end if

end do

! Ascending sort the entries for each node.
!
!
!
!
!
!
!
subroutine triangulation_order6_example2 ( node_num, triangle_num, node_xy, &
    triangle_node, triangle_neighbor )

**************************************************************************
!! TRIANGULATION_ORDER6_EXAMPLE2 returns an example triangulation.
!!
!! This triangulation is actually a Delaunay triangulation.
!!
!! The appropriate input values of NODE_NUM and TRIANGLE_NUM can be
!! determined by calling TRIANGULATION_ORDER6_EXAMPLE2_SIZE first.
!!
!! Diagram:
!!
!! 21-22-23-24-25
!!   |
!!   |
!! 16 17 18 19 20
!!   |
!!   |
!! 11-12-13-14-15
!!   |
!!   |
!!  6 7 8 9 10
!!   |
!!   |
!!  1--2--3--4--5
!!

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

03 January 2007

Author:

John Burkardt

Parameters

Input, integer ( kind = 4 ) NODE_NUM, the number of nodes.

Input, integer ( kind = 4 ) TRIANGLE_NUM, the number of triangles.

Output, real ( kind = 8 ) NODE_XY(2,NODE_NUM), the coordinates of
the nodes.

Output, integer ( kind = 4 ) TRIANGLE_NODE(6,TRIANGLE_NUM), lists the
nodes that make up each triangle. The first three nodes are the
vertices, in counterclockwise order. The fourth value is the midside
node between nodes 1 and 2; the fifth and sixth values are the
other midside nodes in the logical order.
Output, integer ( kind = 4 ) TRIANGLE_NEIGHBOR(3,TRIANGLE_NUM), for each side of a triangle, lists the neighboring triangle, or -1 if there is no neighbor.

implicit none

integer ( kind = 4 ), parameter :: dim_num = 2
integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num
integer ( kind = 4 ), parameter :: triangle_order = 6
real ( kind = 8 ) node_xy(dim_num,node_num)
integer ( kind = 4 ) triangle_neighbor(3,triangle_num)
integer ( kind = 4 ) triangle_node(triangle_order,triangle_num)

node_xy = reshape ( (/ &
  0.0D+00, 0.0D+00, &
  1.0D+00, 0.0D+00, &
  2.0D+00, 0.0D+00, &
  3.0D+00, 0.0D+00, &
  4.0D+00, 0.0D+00, &
  0.0D+00, 1.0D+00, &
  1.0D+00, 1.0D+00, &
  2.0D+00, 1.0D+00, &
  3.0D+00, 1.0D+00, &
  4.0D+00, 1.0D+00, &
  0.0D+00, 2.0D+00, &
  1.0D+00, 2.0D+00, &
  2.0D+00, 2.0D+00, &
  3.0D+00, 2.0D+00, &
  4.0D+00, 2.0D+00, &
  0.0D+00, 3.0D+00, &
  1.0D+00, 3.0D+00, &
  2.0D+00, 3.0D+00, &
  3.0D+00, 3.0D+00, &
  4.0D+00, 3.0D+00, &
  0.0D+00, 4.0D+00, &
  1.0D+00, 4.0D+00, &
  2.0D+00, 4.0D+00, &
  3.0D+00, 4.0D+00, &
  4.0D+00, 4.0D+00 & &
  0.0D+00, 1.0D+00, &
  1.0D+00, 1.0D+00, &
  2.0D+00, 1.0D+00, &
  3.0D+00, 1.0D+00, &
  4.0D+00, 1.0D+00, &
  0.0D+00, 2.0D+00, &
  1.0D+00, 2.0D+00, &
  2.0D+00, 2.0D+00, &
  3.0D+00, 2.0D+00, &
  4.0D+00, 2.0D+00, &
  0.0D+00, 3.0D+00, &
  1.0D+00, 3.0D+00, &
  2.0D+00, 3.0D+00, &
  3.0D+00, 3.0D+00, &
  4.0D+00, 3.0D+00, &
  0.0D+00, 4.0D+00, &
  1.0D+00, 4.0D+00, &
  2.0D+00, 4.0D+00, &
  3.0D+00, 4.0D+00, &
  4.0D+00, 4.0D+00 & &
  0.0D+00, 1.0D+00, &
  1.0D+00, 1.0D+00, &
  2.0D+00, 1.0D+00, &
  3.0D+00, 1.0D+00, &
  4.0D+00, 1.0D+00, &
  0.0D+00, 2.0D+00, &
  1.0D+00, 2.0D+00, &
  2.0D+00, 2.0D+00, &
  3.0D+00, 2.0D+00, &
  4.0D+00, 2.0D+00, &
  0.0D+00, 3.0D+00, &
  1.0D+00, 3.0D+00, &
  2.0D+00, 3.0D+00, &
  3.0D+00, 3.0D+00, &
  4.0D+00, 3.0D+00, &
  0.0D+00, 4.0D+00, &
  1.0D+00, 4.0D+00, &
  2.0D+00, 4.0D+00, &
  3.0D+00, 4.0D+00, &
  4.0D+00, 4.0D+00 & &
  0.0D+00, 1.0D+00, &
  1.0D+00, 1.0D+00, &
  2.0D+00, 1.0D+00, &
  3.0D+00, 1.0D+00, &
  4.0D+00, 1.0D+00, &
  0.0D+00, 2.0D+00, &
  1.0D+00, 2.0D+00, &
  2.0D+00, 2.0D+00, &
  3.0D+00, 2.0D+00, &
  4.0D+00, 2.0D+00, &
  0.0D+00, 3.0D+00, &
  1.0D+00, 3.0D+00, &
  2.0D+00, 3.0D+00, &
  3.0D+00, 3.0D+00, &
  4.0D+00, 3.0D+00, &
  0.0D+00, 4.0D+00, &
  1.0D+00, 4.0D+00, &
  2.0D+00, 4.0D+00, &
  3.0D+00, 4.0D+00, &
  4.0D+00, 4.0D+00 & &
//, (/ dim_num, node_num /) )

triangle_node(1:triangle_order,1:triangle_num) = reshape ( (/* &
  1, 3, 11, 2, 7, 6, &
  13, 11, 3, 12, 7, 8, &
  3, 5, 13, 4, 9, 8, &
  15, 13, 5, 14, 9, 10, &
  11, 13, 21, 12, 17, 16, &
  23, 21, 13, 22, 17, 18, &
  13, 15, 23, 14, 19, 18, &
  25, 23, 15, 24, 19, 20 /), (/* triangle_order, triangle_num /) )

triangle_neighbor(1:3,1:triangle_num) = reshape ( (/* &
  -1, 2, -1, &
  5, 1, 3, &
  -1, 4, 2, &
  7, 3, -1, &
  2, 6, -1, &
//, (/ triangle_order, triangle_num /) )

261
```
integer ( kind = 4 ) hole_num
integer ( kind = 4 ) node_num
integer ( kind = 4 ) triangle_num

node_num = 25
triangle_num = 8
hole_num = 0

return
end
```
AERODYNAMIC ASSESSMENT OF HUMPBACK WHALE VENTRAL FIN SHAPES

Author: Damià Rita Espada

BUDJET

Director: Roberto Maurice Flores Le Roux

September 2011
Budget

The budget of this project consists of the expenditures listed below.

<table>
<thead>
<tr>
<th>Units</th>
<th>Unitary price</th>
<th>Price</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rib Wood</td>
<td>1</td>
<td>4,50 €</td>
<td>4,50 €</td>
</tr>
<tr>
<td>Rib Mechanization</td>
<td>1</td>
<td>35,00 €</td>
<td>35,00 €</td>
</tr>
<tr>
<td>Polyethylene (125x60x5 cm)</td>
<td>1</td>
<td>11,35 €</td>
<td>11,35 €</td>
</tr>
<tr>
<td>4-9cm profile polyethylene cut</td>
<td>2</td>
<td>5,10 €</td>
<td>10,20 €</td>
</tr>
<tr>
<td>100mm transversal polyethylene cut</td>
<td>5</td>
<td>1,00 €</td>
<td>5,00 €</td>
</tr>
<tr>
<td>Polyethylene glue</td>
<td>3</td>
<td>5,05 €</td>
<td>15,15 €</td>
</tr>
<tr>
<td>Filleted Bar M-10</td>
<td>1</td>
<td>2,50 €</td>
<td>2,50 €</td>
</tr>
<tr>
<td>Steel bar 12x3</td>
<td>4</td>
<td>2,80 €</td>
<td>11,20 €</td>
</tr>
<tr>
<td>Wooden bar 18x27x2400</td>
<td>1</td>
<td>4,30 €</td>
<td>4,30 €</td>
</tr>
</tbody>
</table>
| Bike parts | 1           | 0,00 € | 0,00 €                           | Courtesy CSO Barrilonia  
| M10 nuts | 1           | 2,95 € | 2,95 €                           |
| Balsa wood bar 40x40 | 1           | 12,75 € | 12,75 €                           |
| Balsa wood 1mm | 8           | 2,05 € | 16,40 €                           |
| Sand papers | 9           | 0,50 € | 4,50 €                           |
| Model Support | 1           | 0,00 € | 0,00 €                           | Courtesy Mr. José Luis and Joan Turró  
| Epoxy Resine | 1           | 7,78 € | 7,78 €                           |
| Box Cutter | 1           | 4,02 € | 4,02 €                           |
| Milano Journey | 1           | 80,00 € | 80,00 €                           |
| Work hours | 1000         | 10,00 € | 10,000,00 €                       |
| GiD license | 1           | 0,00 € | 0,00 €                           | Courtesy CIMNE  
| Sum |               | 10,227,60 € |                                    |

Thus, the total cost of the project is 10,227,60€.