14

Advanced User-Interaction with GUIs in MatLAB®

P. Franciosa, S. Gerbino and S. Patalano
University of Molise, School of Engineering; Termoli
Italy

1. Introduction

The advent of computer graphics and simulation software has strongly influenced the industrial design. Nowadays, when facing out the design of a new product or the re-design of an existing one, it is of interest evaluating different design scenarios, by comparing physical and functional behaviors and product performances. Engineers are aimed to explore many and many "what-if" design scenarios for design optimization.
MatLAB® scientific computing software offers powerful tools and mathematical utilities which can aid engineers in modeling and simulating their own applications, by using friendly graphical user interface (GUI) toolboxes.
Generally speaking, when developing software, toolbox or standalone applications, one may adopt specific programming languages, such as Visual C++®, Visual Basic®, or Java®. For a computer science or information technology engineer it is easy and natural to program in these environments but for other science and engineering researchers all this may be an obstacle since they are not so familiar with those languages and it is usually required a high programming expertise (Perutka, 2010).
In this contest, MatLAB® is a valid solution to develop powerful toolboxes and software by using its high programming language and its utilities (see linear algebra library collections and visualization toolkits, among others).
While most applications can work by just giving inputs and analyzing results in text or graphic format, the use of graphical interface offers many advantages for users who wish to solve complex problems interactively and obtain visual feedbacks.
There are several reasons for using MatLAB® as a GUI development tool (Holland & Marchand, 2002; Scott, 2006). First of all (I), MatLAB® offers a high-level scripting language. This allows researchers to focus on the problem they are trying to solve rather than spending time in developing a GUI architecture based on a low-level language. Second, (II) MatLAB®'s GUI applications can be fully integrated with the wide collection of computational routines. Moreover, (III) GUI applications are not dependent on OS architecture. Since MatLAB®'s code is not compiled, it may be run on any OS supported by MatLAB®. Finally, (IV) Graphic libraries allow to develop friendly GUI applications with powerful user interaction.
The high-level GUI development tool embedded in MatLAB® is called GUIDE (Graphical User Interface Development Environment) and it allows to automatically design the GUI layout and to handle control and object properties.

The scientific literature offers hundreds of valid GUI tools developed to easily solve practical engineering problems. For example, on the MathWorks website, under "file exchange" section, one can find several contributions covering data acquisition and monitoring, data analysis, image processing, mesh/surface visualization, 3D image rendering, FEM applications and so on.
This chapter focuses on two MatLAB®'s GUI applications, developed at University of Molise in collaboration with University of Naples (Italy): SVA-FEA (Statistical Variational Analysis & Finite Element Analysis) and PROMesh (PROcessing Mesh).
The aim is to show how to provide advanced user interaction in several common tasks such as importing data, editing data, controlling FEA runs, visualizing results, and exporting results.

SVA-FEA

SVA-FEA is a graphical tool able to statistically analyze variations occurring into assembly processes of compliant parts. Variations at part level propagate through the assembly due to both assembly sequence and process variability. One key issue to be faced-out when designing a new product is to reduce such a variation. Depending on the complexity of the process (number of assembly phases/stations) or on the physical laws governing the assembly process (see for example, plastic deformation or residual stresses occurring when joining two flanged-parts) manual approaches are often inadequate to give valuable results. In this contest, only a computer tool may help engineers in finding-out the best design setting. The implementation of SVA-FEA was motivated to quickly predict variation occurring into compliant assembly. Many efforts were done to develop a friendly GUI allowing to interactively define input data, assembly process and visualize final results. SVA-FEA is linked, in background mode, to MSC NASTRAN® solver, used to calculate elastic displacements and generalized forces. More details about the SVA-FEA methodology can be found in (Gerbino et *al.*, 2008).

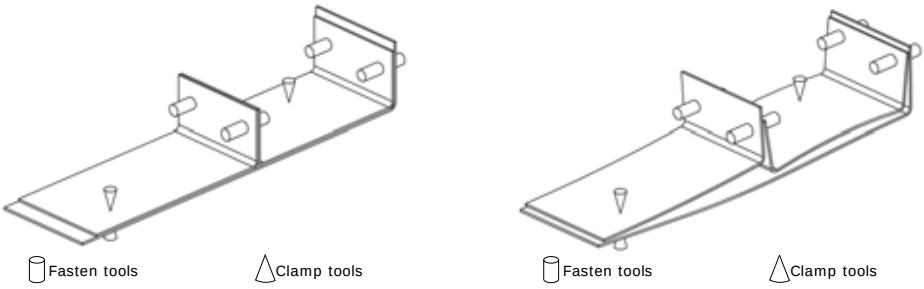
PROMesh

The implementation of PROMesh was originally made within the PUODARSI (Product User-Oriented Development based on Augmented Reality and Interactive Simulation) Italian research project (http://www.kaemart.it/puodarsi), aiming to implement a tool to quickly perform stress-strain analyses and aerodynamic simulations, visualize results and keep them up-to-date while the shape of the object is modified interactively (Bordegoni et *al.*, 2010; Di Gironimo et *al.*, 2009). Starting from this general idea, we developed the PROMesh tool allowing to interactively modify any tessellated model by applying a morphing mesh procedure and to create a 3D closed domain starting from an open shell model. The so-edited geometry can be automatically converted into a suitable FE model, ready to be used for solving a steady fluid dynamic simulation. Comsol Multiphysics® is adopted as solver, working into background mode.
Both SVA-FEA and PROMesh were designed to be friendly as much as possible. Among other things, in order to provide high interaction tools, mouse events (mouse button- down, -up, -move) were programmed to allow fast selection tasks. In particular, mesh data information (see, for example, node coordinates) can be directly accessed just by mouse picking or dragging-dropping within the graphic area.
Some interesting features implemented in those MatLAB®-based applications are related to the possibility to select objects in a graphic window based on the current viewing

orientation. MatLAB®, which partially adopts an OpenGL® graphic engine, provides functionality, similar to that of a camera with zoom lens, which enables to control the viewing of the scene. By properly combining the camera orientation and the cursor-mouse position, mouse selection capabilities can be programmed.
This Chapter is arranged as follows: Section 2 provides an overview of SVA-FEA capabilities, highlighting its data structure and the main user-interaction features. PROMesh is discussed in Section 3. Finally, Section 4 draws final remarks and conclusions.

2. SVA-FEA overview

SVA-FEA provides functionalities to analyze variations occurring into assembly processes of compliant parts. Car body sheet-metal parts, aircraft structural components and plastic injected molded parts are typical elements with some compliance which makes no more applicable the assumption of rigid body when studying 3D tolerance stack-ups.



a. nominal geometry b. variational geometry

Fig. 1. Assembly of compliant parts

Following the classical PCFR cycle (Chang, 1996), with respect to the specific assembly station, parts are firstly positioned onto the fixture frame, then clamped and fastened and, finally, released, reaching the final sub-assembly configuration (Camelio et *al.*, 2004a). When analyzing variations of flexible assembly many factors should be accounted. First of all, the compliance of parts being assembled must be considered. Then, how parts interact each-other need to be investigated. Often, in real industrial applications, assembly processes are made of many sub-stations (Ceglarek et *al.*, 2009). When a part/sub-assembly moves from one station to another one, one should also consider that rigid location errors may add to elastic spring-back deviations. Looking at Fig. 1 one may observe that clamp and fasten tools may deform parts being assembled due to both part errors and clamp/fasten deviations. Understanding how deviations propagate through the assembly process it is of interest especially during the early design stages, when different design scenarios are aimed to be investigated and analyzed. In this contest, simulation environments are welcome as they allow to give valuable results into a reasonable time with no need to made very expensive and time-consuming real prototypes.
Mainly based on these needs, the implementation of SVA-FEA's GUI was motivated in order to analyze many different assembly configurations by varying few input parameters and to quickly view the simulation results.

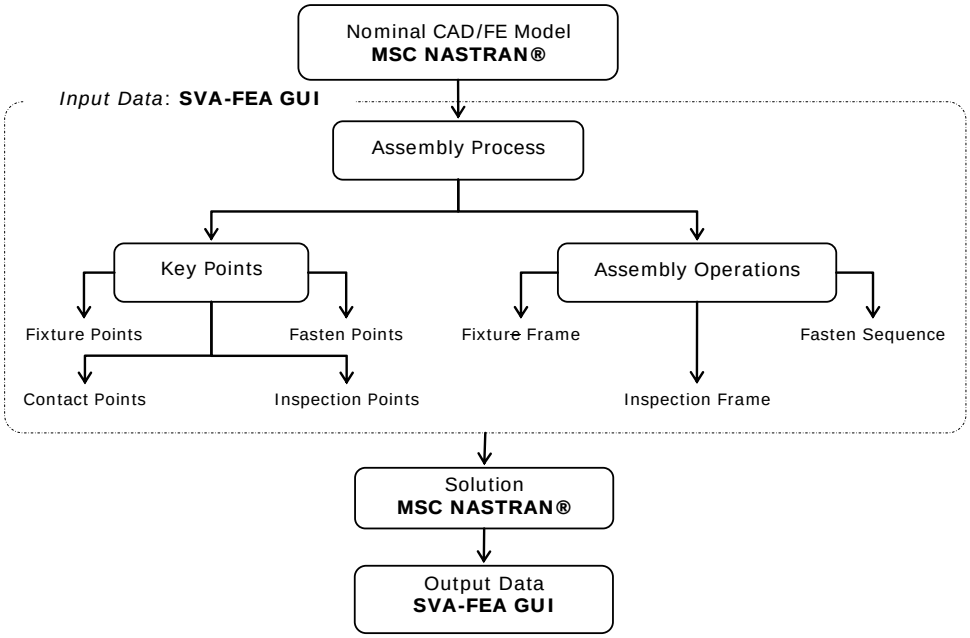


Fig. 2. SVA-FEA architecture (Franciosa, 2010a)

SVA-FEA allows to model both single- and, more in general, multi-station processes. Part deformation is calculated by adopting a FEM approach: forces and elastic displacements are calculated by solving a linear static FE model. The general SVA-FEA architecture (Franciosa, 2010a) is depicted in Fig. 2. Starting from the nominal assembly geometry, imported from a CAD system, the FE model is created and imported, accordingly, into MSC NASTRAN® format. For each sub-station, assembly operations have to be defined. In particular, four sets of Key Points (KPs) are identified: fixture points to model fixture tools; fasten points to model fasten operations; contact points to model the contact between parts to avoid part-to-part penetrations; and, inspection points related to points we want to check-out on the assembly at final stage. For each sub-station, these points must be assigned, accordingly. Moreover, statistical input data are provided in terms of mean and standard deviation. Once input data are correctly assigned, output data, in terms of statistical displacements, are given by solving two consecutive FEA runs.
The whole software architecture is based on MatLAB® environment which drives, in background mode, the MSC NASTRAN® solver. Fig. 3 depicts the SVA-FEA GUI and its main menus.
SVA-FEA's user interface was designed to easily allow: (I) importing mesh data; (II) selecting KPs; (III) defining assembly process; (IV) running FEA analysis; (V) and, viewing simulation results. The GUI layout was developed by using the GUIDE environment.

2.1 SVA-FEA data structure

This Section shows the general structure used to manage input data in SVA-FEA. Input data are managed by using *structure* arrays. The main data structure is depicted into Fig. 4.

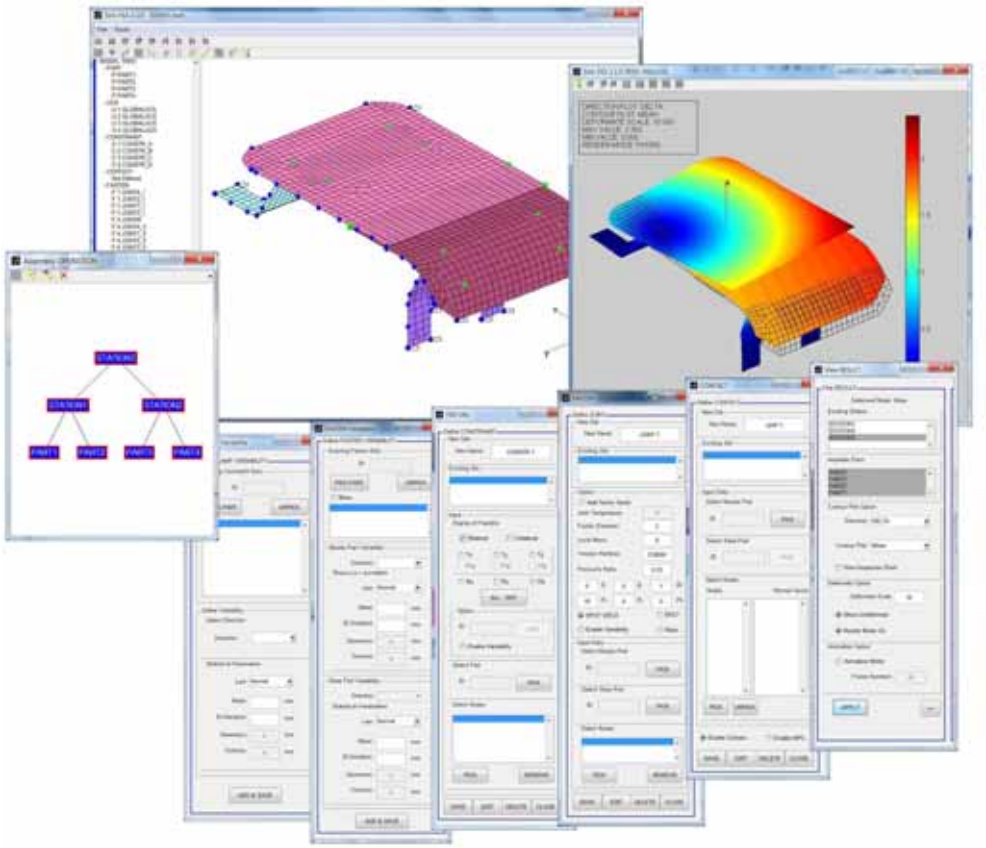


Fig. 3. SVA-FEA user interface

For each part ("part i"), six fields are available:

- MAT: material properties (Young's Modulus and Poisson's ratio) and shell thickness;
- NODE: coordinates of mesh nodes;
- ELEMENT: mesh elements; and,
- FIXTURE/FASTEN/CONTACT: input assignment for fixture, fasten and contact points.

With respect to the latter fields, the following sub-fields are available:

- DoF: list of constrained degrees of freedom (DoF);
- UCS: local coordinate frame definition;
- NodeIDf: node identification for fixture point assignment;
- NodeIDsrc / NodeIDdst: source and destination node identification;
- T: statistical input value in terms of mean and standard deviation.

2.2 SVA-FEA software: handling MSC NASTRAN® input files

The input information needed to do the numerical analysis in MSC NASTRAN® is contained into the ASCII .bdf (Bulk Data Format) file. This file is made of three sub-sections:

- *executive control statement*: includes solver options and diagnosis operations;
- *case control section*: includes sub-case entries and output queries; and,
- *bulk data section*: includes the FE model (nodes, elements and boundary conditions).

The general structure of a .bdf file (SVA-FEA supports the free field format, where data are separated by blanks) is listed below.

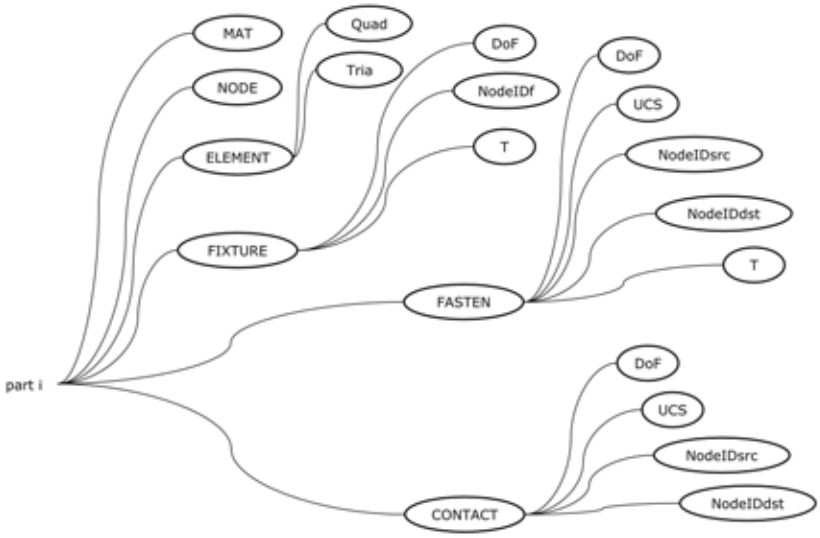
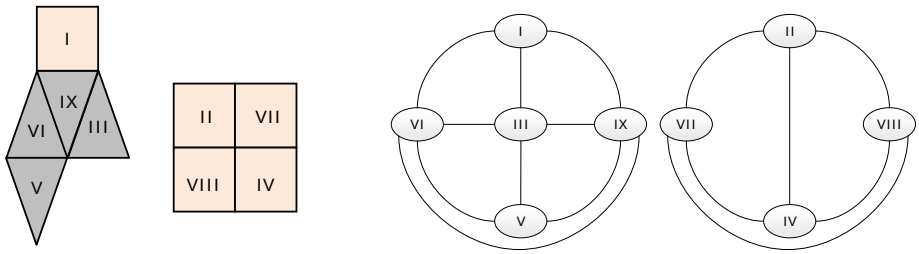


Fig. 4. SVA-FEA data structure (Franciosa, 2010a)

When the number of nodes or elements increases (in real industrial applications, meshes are made of many thousands of nodes) the .bdf file may become very huge and its reading, by using MatLAB® built-in functions, is often not efficient. To overcome this issue we implemented a MEX function (see Annex A.1 on how writing a MEX function) allowing to quickly read and import mesh elements (for example CQUAD4 and CTRIA3 elements by MSC NASTRAN®) and node coordinates. Material properties, geometry constants and node constraint settings will be defined through the SVA-FEA's GUI.



(a) mesh data (b) graph representation

Fig. 5. Graph representation of mesh data (two domains)

```
$- executive control statement
SOL 101 $- solver type. "101" corresponds to the linear static solver
CEND

$- case control section
SUBCASE 1 $- sub-case entries

$- beginning bulk data section
BEGIN BULK
…
CQUAD4 104 1 122 123 134 133
$- element quad id=104, material id=1, connected nodes: 122,123,134,133
CTRIA3 105 1 150 151 152
$- element tria id=105, material id=1, connected nodes: 150,151,152
…
GRID 150  0.0 0.0 1.0 5
$- node id=150, coordinates [0.0, 0.0, 1.0], local UCS id=5
…
SPC 1 176 1 0.00
$- define single point constraint
…
PWELD 1 3 5.00
…
CWELD 1 1  ALIGN 125 9
$- cweld element to define fasten elements
….
FORCE 1 9 2 10.54 0.00 0.00 1.00
$- define load conditions
…
CORD2R 5  80.00 0.00 4.00 80.00 0.00 5.00 79.00 0.00 4.00
$- define local coordinate UCS
…
ENDDATA
$- ending bulk data file
```

When importing mesh data from .bdf formatted files, no geometric information is available about connected domains. We implemented an automatic procedure allowing to select and store connected domain. Every connected domain corresponds to a part, which can be introduced into the assembly process being simulated.

The general idea to extract connected domains is to consider the imported mesh as a graph in which mesh elements correspond to graph-vertices, while each edge represents an element-to-element connection (see Annex A.2 for a general overview on the main concepts of the Graph Theory used in the application). Looking at Fig. 5, 9 elements (five CQUAD4 and four CTRIA3) define two connected domains: (I, III, V, VI, IX) and (II, IV, VII, VIII). The graph representation depicted into Fig. 5b was obtained considering that two elements are connected if they share one edge or one node. For example, element I is connected to element IX, through one edge, and to elements III and VI, with one node.

Starting from the definition provided into equation (A.2), the adjacency matrix, "A", of the mesh-graph can be easily calculated. Knowing the adjacency matrix ($N_{quad}+N_{tria}$ x $N_{quad}+N_{tria}$ square matrix, where N_{quad} and N_{tria} are, respectively, the number of imported CQUAD4 and CTRIA3 elements), a growing procedure can be applied to detect all connected domains. Below the MatLAB®'s pseudo-code.

```
%- find-out connected domains
function domain=connectedDomain(A)

seed=1;
while true
    idConnect=[];
    visited(seed)=true; %- seed has been already visited

    for i=1:length(seed) %- loop over seed elements
        %- get connected elements by using "A" matrix
        [temp,visited]=getConnected(seed(i),A,visited);
        idConnect=[idConnect,temp];
    end

    %- "local domain"
    tempDomain=[tempDomain,idConnect];

    if ~isempty(idConnect)
        %- update seed counter
        seed=idConnect;
    else
        %- if no connected element is found then save "local domain"
        count=count+1;
        domain{count}=tempDomain; %- new domain counted

        %- look for a new seed (not yet visited)
        seed=getNotVisited(visited);
        tempDomain=seed;
    end

    %- break loop when there is no new seed (all elements were allocated)
    if isempty(seed)
        break
    end
end %- end loop
```

The procedure looks for those elements connected to the initial "seed element". Thus, iteratively, the seed counter is update with the so-connected elements ("idConnect"). When no other connected element is counted, then a connected domain has been selected and it is saved. These connected elements are classified as "visited". The iterative procedure stops when all elements have been marked as visited. Once connected domains are calculated, SVA-FEA updates its data structure: "ELEMENT.Quad", "ELEMENT.Tria" and "NODE" fields are filled, accordingly, with respect to the i-th connected domain (see Fig. 4). The "*patch*" MatLAB® built-in function is used to draw and visualize mesh data.

2.3 SVA-FEA software: selecting mesh nodes

Many efforts were done to make the SVA-FEA's GUI friendly as much as possible. In particular, when fixture or fasten points have to be assigned, the easiest way is just to select, by mouse picking, mesh node from the graphic area.

This task can be accomplished by using MatLAB® graphic tools. Fig. 6 reports the general scheme adopted in MatLAB® for defining a scene (MatLAB® supports both parallel and perspective projections; however, the actual implementation of SVA-FEA supports only the parallel projection).

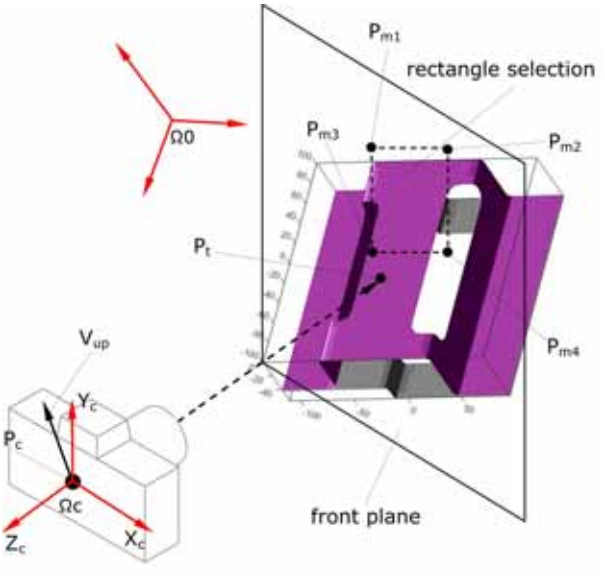


Fig. 6. Camera representation and mouse selection from graphic area

In Fig. 6 "P_t" is the position, in the coordinate frame (Ω_0), of the point the camera points to (*camera target*). "P_c" is the position of the camera frame (Ω_c) with respect to Ω_0 (*camera position*). "V_{up}" defines the rotation (*camera up-vector*) around the camera view axis, "Z_c". "P_{mi}" are the mouse picked points, defined with respect to Ω_0. "P_{mi}" correspond to the intersection between the camera view axis and the front plane (which is parallel to the camera plane X_c-Y_c). By using MatLAB® camera properties, "P_t", "P_c", "V_{up}" and "P_{mi}" can be obtained by:

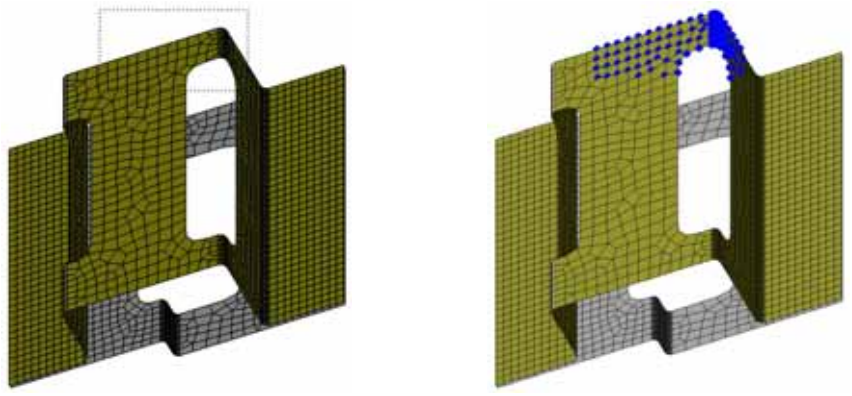
```
Pc=get(gca,'CameraPosition'); %- camera position
Pt=get(gca,'CameraTarget');  %- camera target
Vup=get(gca,'CameraUpVector'); %- camera up-vector
Pmi=get(gca,'CurrentPoint'); %- picked point
```

The aim is to find-out the mesh node nearest to the picked point. To do this, the mesh node and the picked point have to be projected onto the front plane. After calculating the rotation matrix from the frame Ω_0 to Ω_c as into equation (1):

$$Z_c = \frac{P_c - P_t}{\|P_c - P_t\|}, X_c = \frac{V_{up} \wedge Z_c}{\|V_{up} \wedge Z_c\|}, Y_c = \frac{Z_c \wedge X_c}{\|Z_c \wedge X_c\|}$$
$$\downarrow \qquad (1)$$
$$R = \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$$

one can obtain the index ("idSelected") of the nearest mesh node with respect to the picked point. Below the MatLAB® pseudo-code.

```
%- transform mesh-nodes, "Ncoord"
Ncoord=R*Ncoord';
Pmi=R*Pmi';

%- take just (x-y) components
Ncoord= Ncoord(1:2,:);
Pmi=Pmi(1:2);

%- calculate distances
diff=[Ncoord(1,:)-Pmi(1); Ncoord(2,:)-Pmi(2)];
dist=sqrt(sum(diff.^2,1));

%- finally, find-out the index related to the minimum distance
[~,idSelected]=min(dist); %- discard (~) first output
```



(a) drawing rectangle by mouse picking/moving (b) selecting mesh-nodes

Fig. 7. Application of the rectangle-area selection algorithm

The algorithm just described can be extended to allow rectangle-area selection from the graphic area by defining three sub-routines running when picking mouse buttons or moving mouse into the current MatLAB® figure. The procedure can be summarized as follows (see also Fig. 6):

- calculating "P_{m1}" point and "R" matrix when picking down the mouse button ("*WindowButtondownFcn*" callack);
- calculating "P_{m4}" point and draw rectangle-area selection when moving the mouse ("*WindowButtonMotionFcn*" callack); and,
- calculating mesh-nodes inside the rectangle-area selection ("*WindowButtonUpFcn*" callack).

"P_{m1}" is calculated once picking down the mouse button. "P_{m4}" corresponds to the actual position of the mouse cursor. The MatLAB® pseudo-code is reported below.
The "*patch*" and "*line*" MatLAB® built-in functions are used to draw, respectively, the rectangle-area selection and the selected mesh nodes. Furthermore, the "*inpolygon*" MatLAB®'s command is here adopted to check which nodes are inside the rectangle-area.
As example, in Fig. 7 a mouse selection is drawn into the upper side of figure (Fig. 7a). Then, the selected mesh nodes are marked as blue dots (Fig. 7b).

```
%- inizialize selection phase:
set(fig,'WindowButtondownFcn',{@sClick,Ncoord});

function sClick(Ncoord)

%- built frame
R=[Xc;Yc;Zc];

%- start selection
P1=get(gca,'CurrentPoint'); %- picked point

%- start mouse motion
set(gcf,'WindowButtonMotionFcn',{@moveMouse,P1,R,Ncoord})

function moveMouse(P1,R,Ncoord)

%- actual mouse position
P4 = get(gca,'CurrentPoint');

%- transform into the local frame
P1=R*P1';
P4=R*P4';

%- built rectangle selection
P2=[P1(1) P4(2) 0];
P3=[P4(1) P1(2) 0];

%- go-back into global frame
Vertext=R'*[P1;P2;P4;P3];';

%- draw rectangle
patch('Faces',[1 2 3 4],...
      'Vertices',Vertext');

%-call mouse button-up
set(fig,'WindowButtonUpFcn',{@endClick,R,Ncoord,Vertex});

function endClick(R,Ncoord,Vertex)

inPol=inpolygon(Ncoord(1,:),Ncoord(2,:),Vertex(:,1),Vertex(:,2));

Psel=Ncoord(inPol,:);

line('xdata',Psel(:,1),...
     'ydata',Psel(:,2),...
     'zdata',Psel(:,3))
```

In SVA-FEA, when defining fasten or contact points, mesh-nodes are directly selected from the graphic area (see Fig. 8) by using the rectangle-area selection tool. Once master and slave parts are picked, matched nodes are automatically assigned among parts. After selecting master node, the related matched node is calculated as the nearest one on the slave part. In Fig. 8, matched nodes are marked as circle dots.

2.4 SVA-FEA software: Tree view implementation

Imported connected domains and KPs can be edited and managed thought the MODEL TREE, serving as tree viewer (see Fig. 9). Specific programming languages, such as

Microsoft® Visual Basic or Visual C++, offer dedicated tools to develop tree viewers. In SVA-FEA we implemented the MODEL TREE based on a "*listbox*" control.

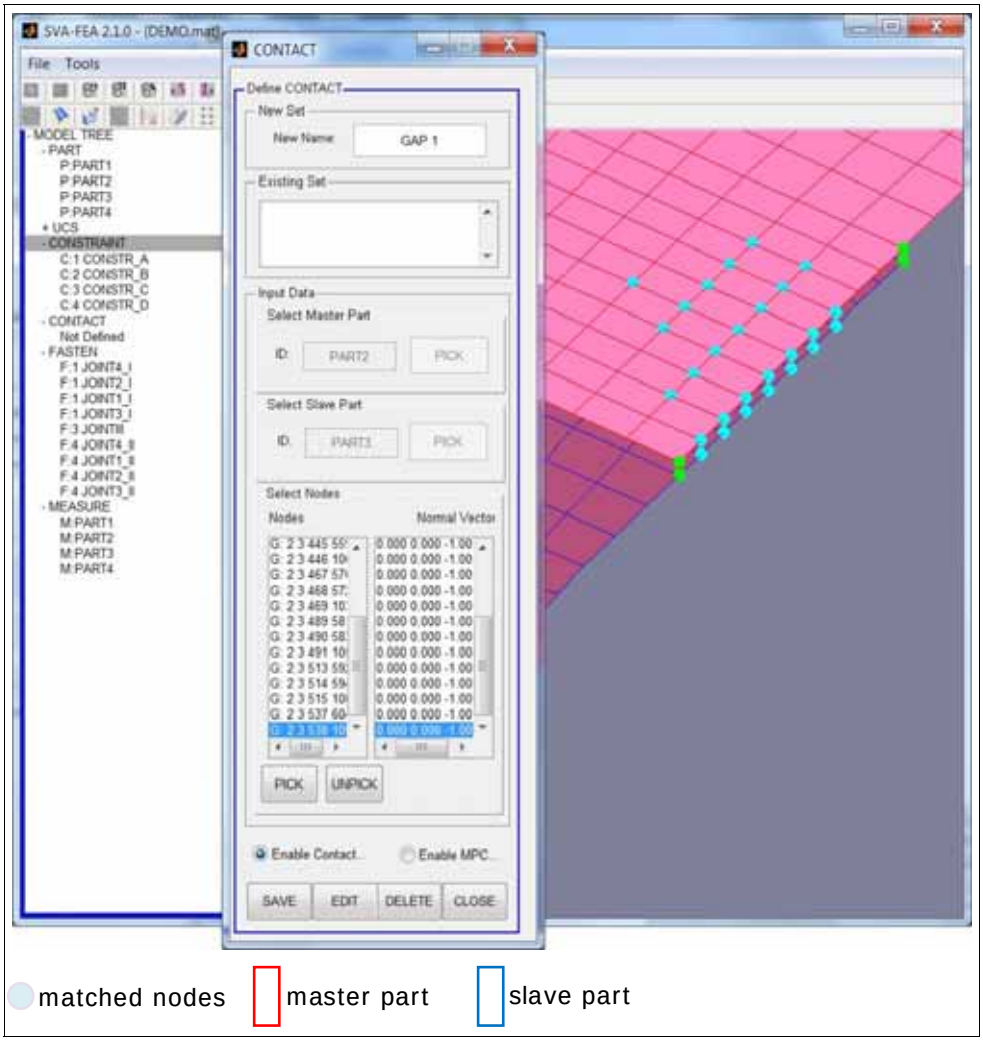
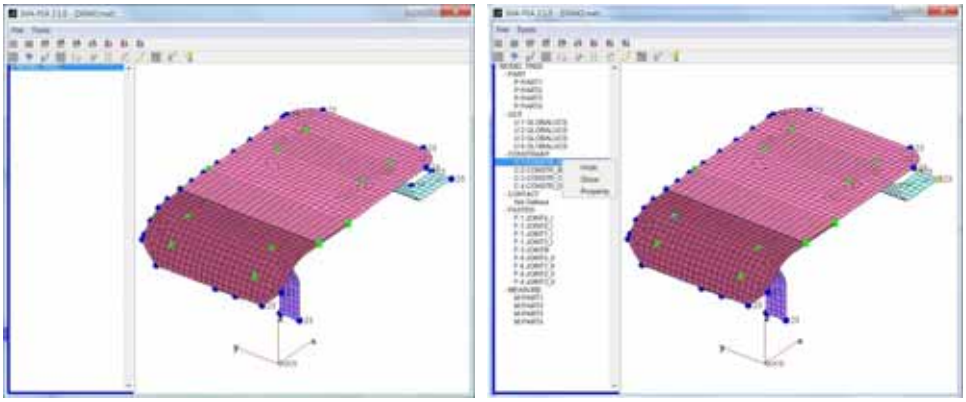


Fig. 8. Defining matched nodes among master and slave parts ("CONTACT" menu)

The "*listbox*" control differentiates its event call-backs depending on the "*SelectionType*" property. For example, if a single mouse click occurs, then the "*SelectionType*" property is automatically set to "*normal*". In presence of a double mouse click, "*SelectionType*" property becomes equal to "*open*".
The example below defines a new figure and a "*listbox*" control. The call-back function named "listCall" is associated to the "*listbox*". Every time clicking on that control, depending on the "*SelectionType*" property, "OPEN" or "NORMAL" strings are written.



(a) collapsed MODEL TREE (b) expanded MODEL TREE

Fig. 9. MODEL TREE visualization

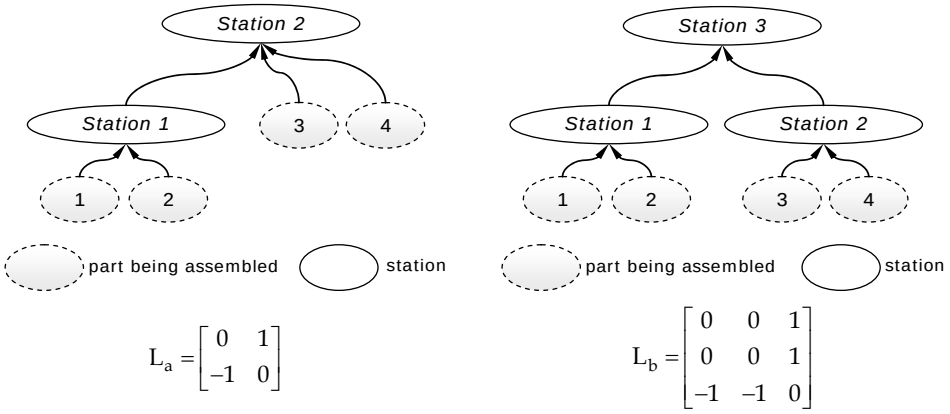
```
function testListbox

%- define a new figure
fig=figure('unit','characters','position',[20 5 160 45]);

%- define a listbox
hList=uicontrol('unit','characters','Style','listbox',...
                'position',[0 0 35 45],...
                'parent',fig,'enable','on');

%- set call-back function
set(hList,'callback',@listCall)

%- call-back function handling
function listCall(src,event)

%- differentiate depending on "SelectionType" property
if strcmp(get(gcf, 'SelectionType'), 'open') %- double-click
     set(src,'string','OPEN','value',1)
elseif strcmp(get(gcf, 'SelectionType'), 'normal') %- single-click
     set(src,'string','NORMAL','value',1)
end
```

Based on this key feature, the MODEL TREE was implemented in SVA-FEA to manage and visualize the modeling history. When importing new part or defining new KPs, the MODEL TREE is automatically updated. As example, Fig. 9 shows the MODEL TREE in which one can browse among part options (PART) and KPs (UCS, CONSTRAINT, CONTACT, FASTEN, MEASURE). Right-mouse-clicking was also programmed to allow a fast editing of the selected item.

2.5 SVA-FEA software: Assembly tree implementation

A crucial aspect to be achieved when performing the variation analysis of compliant parts is the assembly sequence, that is the sequence through which parts or sub-assemblies are put

together. In (Ceglarek, 2009; Camelio et *al.*, 2004b) was reported that the assembly sequence may influence about 60% the final assembly variation. As stated before in this Chapter, for each assembly sub-station, a fixture frame and a fasten tool should be defined, simulating the classical PCFR cycle. As detailed in (Gerbino et *al.*, 2008; Franciosa, 2010a), SVA-FEA calculates the influence that previous sub-stations have on the actual assembly station. Such dependencies can be accounted considering the assembly process as an oriented graph, in which each vertex corresponds to a station, while every edge represents a station-to-station relationship.
Fig. 10 shows two assembly sequences and the related Laplace matrices (see Annex A.2). Knowing those matrices, dependencies among stations are univocally determined. For example, looking at the third column of the "L_b" matrix one can state that "Station 3" depends on "Station 1" and "Station 2". In SVA-FEA we developed the "Assembly OPERATION" tool able to interactively define parts to be assembled and the related station level. Moreover, for each assembly station the related fixture and fasten frame can be defined.



$$L_a = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \qquad L_b = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$$

Fig. 10. Two assembly sequences

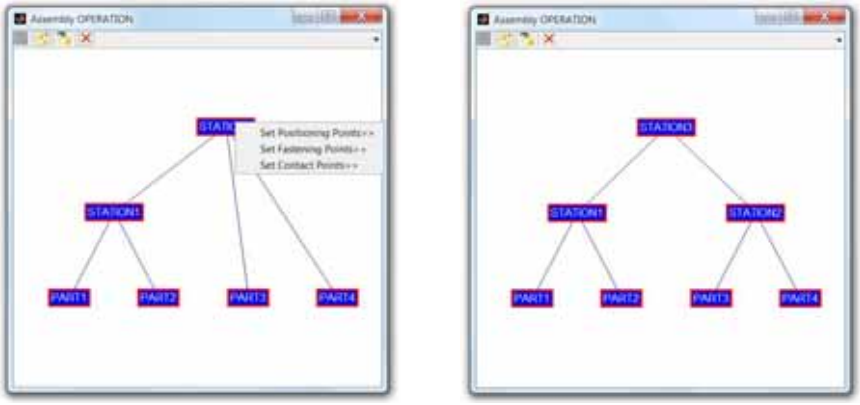


Fig. 11. Assembly OPERATION tool: two different assembly sequences

The "*text*" and "*line*" MatLAB® built-in functions were used to draw, respectively, part and station vertices and the edge links of the assembly graph. Fig. 11 reports the two assembly sequences as seen in SVA-FEA, related to ones depicted into Fig. 10.

2.6 SVA-FEA software: handling MSC NASTRAN output files

Once the assembly sequence is defined and the related KPs are set, accordingly, two consecutive FEA runs are solved, by running MSC NASTRAN®. Input files (in .bdf format) are automatically generated by SVA-FEA, and parsed to the MSC NASTRAN® solver. The following command lines are required to run MSC NASTRAN® from SVA-FEA:

```
%- MSC NASTRAN® path
pathsolve=sprintf('%s %s',cdNastran,filename);

%- run solver
dos(pathsolve);
```

where "cdNastran" is the MSC NASTRAN® installation path, whereas "filename" is the .bdf file to be solved.

FEA results coming from MSC NASTRAN® are stored in two main files: .op2 and .f06. The .op2 file contains post-processing data (see, for example, displacement fields), interpolated by using shape functions. However, the .op2 file has an owner format, not directly accessible or readable by users. On the contrary, the .f06 file is a text file containing node displacements and generalized forces. The general structure of the .f06 file is listed below. "Ti" and "Ri" are the translational and rotational degrees of freedom of the analyzed node, both related to displacements ("DISPLACEMENT VECTOR") and generalized forces ("FORCE OF SINGLE-POINT CONSTRAINT").

```
POINT-ID=241 // node id=241
                    DISPLACEMENT   VECTOR

SUBCASE   TYPE   T1        T2        T3        R1        R2        R3
1          G   -1.36E-05  4.81E-05 -1.38E-02  8.82E-03 -2.04E-03 -1.73E-06
2          G    1.36E-05 -4.81E-05  1.38E-02 -8.82E-03  2.04E-03  1.73E-06
// two sub-cases analyzed

POINT-ID=132
                    FORCE OF SINGLE-POINT CONSTRAINT

SUBCASE    TYPE   T1        T2       T3        R1       R2        R3
1          G     -3.96E-11 5.62E+00 8.94E+01  1.05E+03 -1.0E+02  -5.45E-03
// one sub-case analyzed
```

As described above for the MSC NASTRAN® input file, in order to speed-up the reading phase of the .f06 file, a compiled MEX functions was also here implemented.

2.7 SVA-FEA software: Post-processing simulation results

Simulation results can be easily analyzed and visualized from the POST-PROCESS main GUI (see Fig. 12): deformed or undeformed assembly (or sub-assembly) can be visualized; contour plots of mean or standard deviation values are available. Final results can be exported in Microsoft® EXCEL file, to quickly create graphs and diagrams. The interested reader is referred to (Franciosa et *al.*, 2009, 2010b) where more specific case studies are described and analyzed.

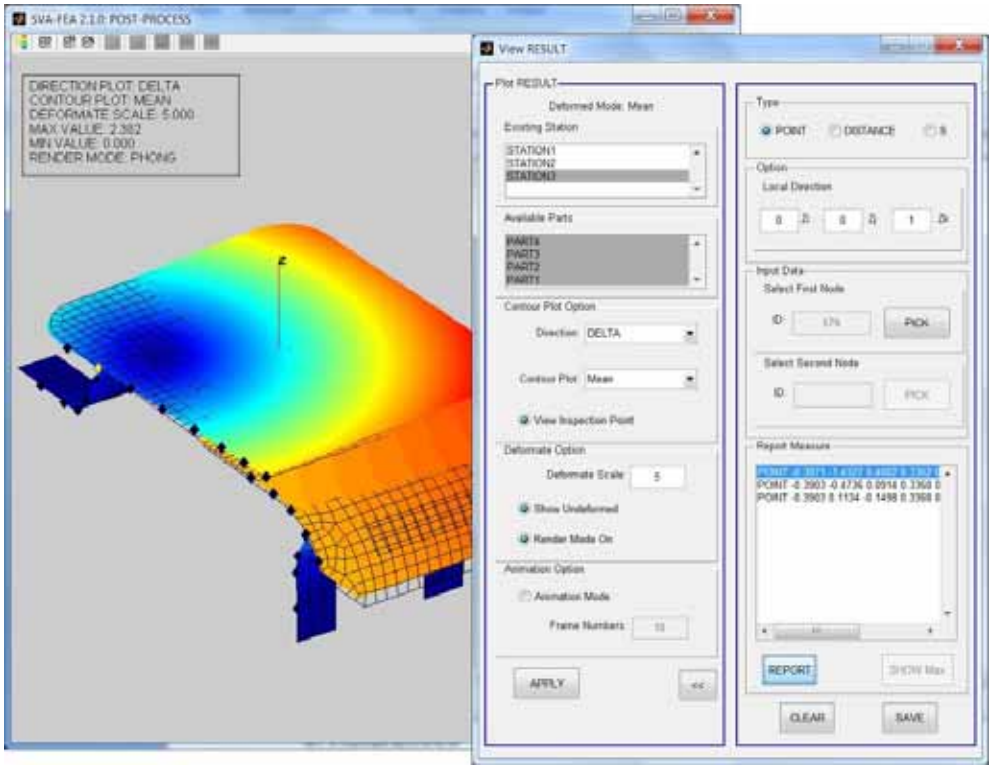
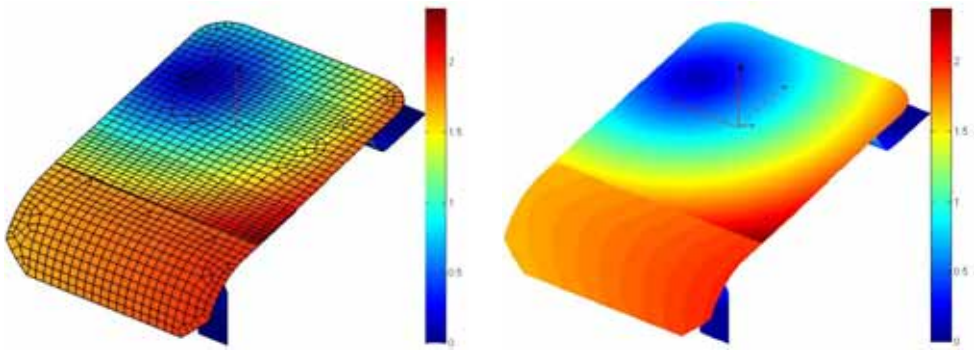


Fig. 12. POST-PROCESS tool

Contour plot utilities were implemented by using the MatLAB®'s "*patch*" function. This graphic object allows to visualize mesh data and to specify a color for each mesh-element or mesh-node. MatLAB® supports three shaders: flat, Gourand and Phong. The flat shader produces a uniform lighting across faces (that is, mesh-elements) of the object. Gourand and Phong algorithms, instead, calculate the mesh-node normals and interpolates linearly across the faces (Lengyel, 2003).

The code below can be adopted to generate a contour plot visualization (the related results are depicted into Fig. 13). "ELEMENT" contains the mesh-element connections (CQUAD4 and CTRIA3 elements imported from the input .bdf file). "cData" is a matrix containing contour data to be plotted (for example, the displacement field along the Z axis direction).

```
%- define a contour plot visualization (mesh edge shown)
patch('Faces',ELEMENT,'Vertices',Ncoord,... %- define mesh plotting
     'LineStyle','-','EdgeColor','k',... %- show "black" edge
     'FaceVertexCData',cData,'FaceColor','inter',... %- define contour data
     'FaceLighting','phong','EdgeLighting','phong') %- define shader

%- define a contour plot visualization (mesh edge hidden)
patch('Faces',ELEMENT,'Vertices',Ncoord,... %- define mesh plotting
     'LineStyle','none',... %- hide edge
     'FaceVertexCData',cData,'FaceColor','inter',... %- define contour
data
     'FaceLighting','phong','EdgeLighting','phong') %- define shader
```



(a) mesh-edge shown (b) mesh-edge hidden

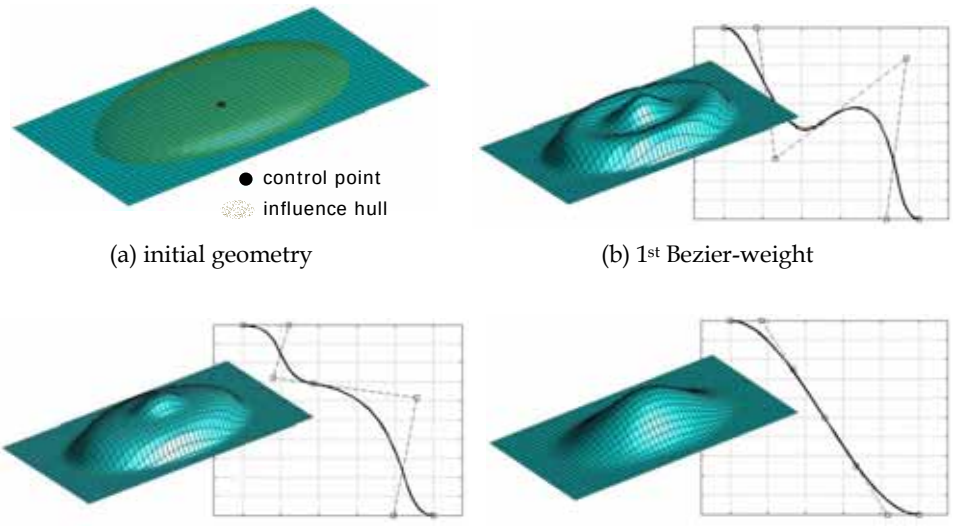
Fig. 13. Contour plot visualization

3. PROMesh overview

Re-designing real parts or, more generally, industrial products, involves different tasks. First of all, geometry shape must be digitalized in order to get a first geometrical model, which can be handled and edited. This model typically comes out as 3D tessellated model usually made of several thousands of nodes and triangles. Then, it needs to be processed in a CAE environment to simulate and understand its performances (in terms, for example, of structural or aerodynamic behavior). If the preliminary CAE results do not match design intents, the initial geometry has to be modified. The re-design loop is iterated until reaching a good balance among functional and esthetic requirements. Obviously, all this may become very time consuming and tedious when many geometry configurations need to be investigated. In this contest, interactive and automatic tools are welcome, since they may drive the designers to quickly test different design scenarios.

Within the PUODARSI Italian research project, PROMesh tool was implemented to quickly perform a fluid-dynamic simulation on any tessellated geometry object, after modifying it interactively (for example, by mouse drag-and-drop). PROMesh adopts Comsol Multiphysics® to solve Navier-Stokes equations, governing the fluid-dynamic phenomenon. Comsol Multiphysics® offers a powerful API interface allowing, among other things, to save and manipulate its data structure within the MatLAB® workspace and to extract and visualize simulation data.

Generally speaking, PROMesh allows: (I) loading any tessellated model, made of triangle or quadrilateral elements; (II) editing the imported geometry; (III) exporting that geometry to Comsol Multiphysics®; and solving a steady fluid dynamic simulation in automatic way. In particular, external flows around free shape objects are simulated. Further details about numerical algorithms can be found in (Di Gironimo et *al.*, 2009).
The PROMesh's software architecture borrowed several algorithms from SVA-FEA, for example, the mesh-node selection algorithm - Section 2.3. A new feature was implemented to allow user to interactively modify the geometry shape by the mouse control. To do this, a *Morphing Mesh Procedure* (MMP) was implemented.



(a) initial geometry (b) 1st Bezier-weight

c. 2nd Bezier-weight d. 3rd Bezier-weight

Fig. 14. Generation of three different morphed geometries based on Bezier-weight functions

3.1 PROMesh software: MMP and user interaction

Morphing mesh is a well known technique used in computer graphic applications as a powerful tool for free-shape modeling and designing. The numerical procedure implemented in PROMesh may be summarized as follows (see (Franciosa & Gerbino, 2009) for more details). User defines a set of control points on the model, by picking them on the graphical interface. Then, the relative influence hull is assigned for each point. Control points directly influence final shape of the deformed object, and this shape can be fine-tuned by adjusting the influence hull's radius and/or the position of each control point. The influence hull defines the 3D region within which any mesh-node is influenced by the related control point. Based on this general idea, one can write:

$$\begin{aligned} &\Delta N_j = f\left(d_{i,j}\right)\cdot M \\ &\forall j = 1,\ldots,N_{node} \\ &\forall i = 1,\ldots,r \end{aligned} \tag{2}$$

where "ΔN_j" is the displacement of the j-th mesh-node, calculated once the morphing matrix, "M", and the weight function, "$f(d_j)$", are known. "r" is the number of control points. As demonstrated in (Franciosa & Gerbino, 2009), "M" matrix can be easily calculated from the control point coordinates. Moreover, the weight function is equal to 1 when the mesh-node "N_j" is coincident with the i-th control point and tends toward zero for points "N_j" whose distance from the i-th control point is greater than zero.

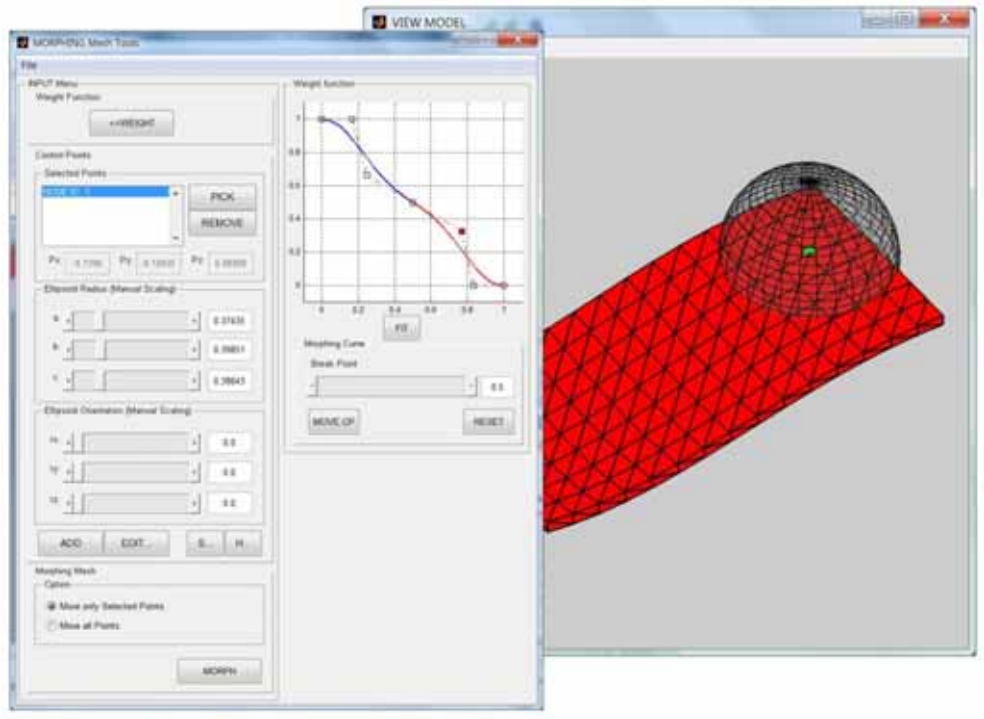
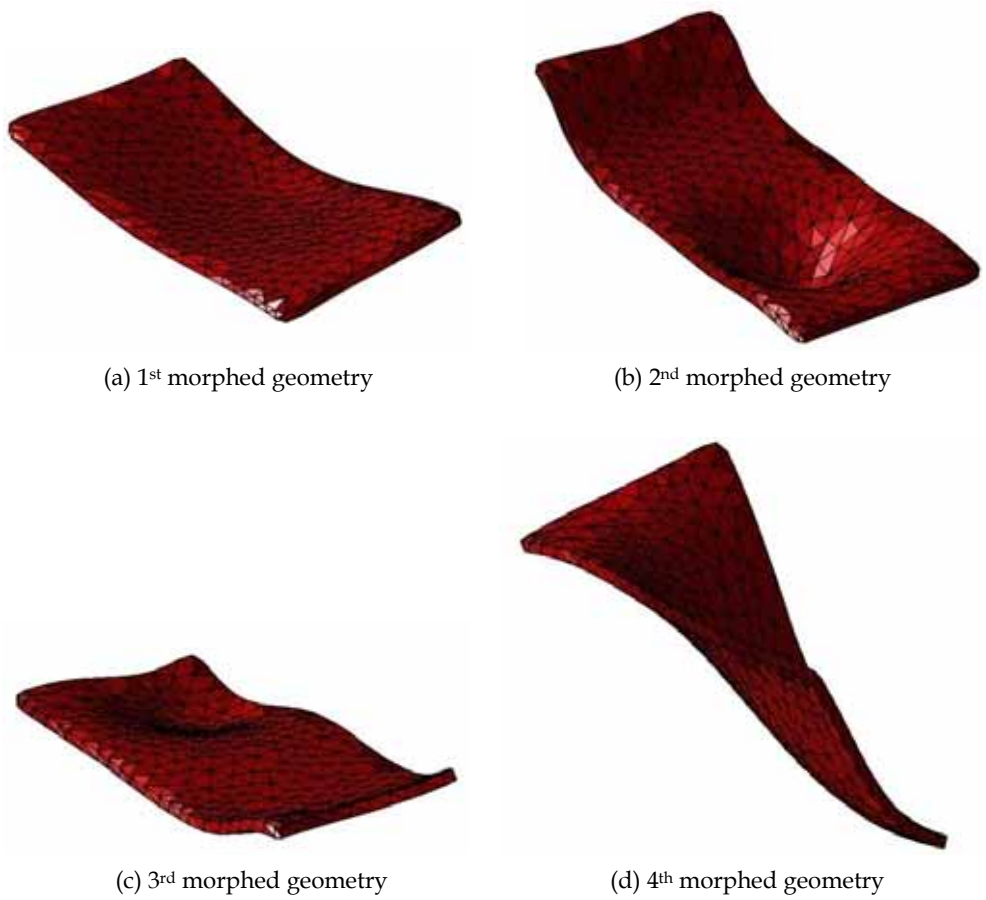


Fig. 15. PROMesh user interface: morphing mesh module

In PROMesh the weight functions was assumed as a piecewise Bezier curve, which can be modified by acting on its control polygon. As example, Fig. 14 shows the application of the MMP on an initial flat geometry. Once defined one control point and its influence hull (PROMesh supports only ellipsoid domains), three different geometries were generated, by varying the Bezier's shape. Since the morphing matrix depends on the control point coordinates, the geometry can be morphed with a mouse control in the graphical area. Partially based on the mouse selection algorithm (see Section 2.3), the interactive morphing procedure can be summarized as follows:

- calculate the selected control point "P_{ci}" and the camera rotation matrix, "R", when picking down the mouse button ("*WindowButtondownFcn*" callack);
- calculate the actual position of the control point "$P_{ci,act}$" point and apply the MMP, when moving the mouse ("*WindowButtonMotionFcn*" callack); and,
- end the procedure when releasing the mouse button ("*WindowButtonUpFcn*" callack).

Fig. 15 depicts the morphing mesh tool embedded in PROMesh. After picking some control points, the related influence hulls can be manually tuned by varying their sizes and their orientations ("*slider*" controls). Then, the weigh function can be edited by moving the control polygon of the Bezier curve, and the geometry changes in real time.
Figure 16 shows four morphed geometries obtained thought the high user-interaction offered by PROMesh.
Source files of PROMesh are available on:
http://www.mathworks.com/matlabcentral/fileexchange/authors/38957.



(a) 1st morphed geometry (b) 2nd morphed geometry

(c) 3rd morphed geometry (d) 4th morphed geometry

Fig. 16. Application of the morphing mesh procedure

4. Conclusions and final remarks

The Chapter focused on two MatLAB®'s GUI applications: SVA-FEA® and PROMesh®. SVA-FEA® is a graphical tool developed to do statistical tolerance analysis of compliant assembly. It allows to manage imported mesh data, define assembly key points and specify

the assembly sequence. On the other side, PROMesh® offers dedicated algorithms to handle tessellated data geometry. The implementation of such tools was characterized by the inner needs to have friendly GUIs, allowing an advanced user-interaction. In particular, the following goals were achieved: (I) selection of mesh nodes through mouse clicking or mouse area-selection; (II) tree view and assembly tree implementations, based on graphs; (III) implementation of compiled MEX functions to speed-up huge calculations, involving the reading and writing tasks of formatted ASCII files. Furthermore, PROMesh® was oriented to allow user to interactively select mesh nodes and "morph" the mesh geometry.
The experiences made in developing these computer tools demonstrates that it is possible to provide advanced user-interaction without a specific skill in computer science.

Annex A: Methods and tools

A.1 Handling large data set in MatLAB®

When large data sets are allocated and accessed within loops, MatLAB® is not too much efficient. This is especially true when managing formatted text files rather than binary files. One manner to optimize and speed-up MatLAB® accessing data is by using compiled source codes, written in FORTRAN or C++ language (Kernighan & Dennis, 1978). This Annex describes how to write and compile a MEX function, written in C++ language, for MatLAB®.
Assume to create an array, A, whose entries are all integers from 1 to 10^8 (obviously, such as array may be easily defined as “A=1:1e8”; this example wants to show, instead, how loops are not so efficient into MatLAB®). From MatLAB® script one can write:

```
%- define array size
N=1e8;

%- initialize array
A=zeros(1,N);

%- start loop
for i=1:N
    A(i)=i; %-allocate “integer” value
end
```

On a Win 7 64bit, 8GB RAM, 2 i7 quad-core processors machine the run-time is 0.9441 s. The same array will be now generated by using a MEX compiled function.
The source code for a MEX file consists of two main distinct parts:

- *computational routine*: it contains the code performing the needed computations; and,
- *gateway routine*: it is the main function which links with MatLAB®.

The general form of a source MEX file is shown below:

```
// include mex header
#include "mex.h"

// COMPUTATIONAL ROUTINE SECTION
void userfnc#1(…)
{
```

```
    //… routine code…
}

double userfnc#2(…)
{
//… routine code…
}

// GATEWAY ROUTINE SECTION
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
// nlhs = number of output items
// nrhs = number of input items
// plhs = output pointers
// prhs = input pointers

{
// get pointer from MatLAB® input
A = mxGetPr(plhs[…]);

// create MatLAB® variable
  plhs[…] = mxCreateDoubleMatrix(…);

// allocate output variable
userfnc(…);
}
```

The name of the gateway function is always “mexFunction”. This function parses all MatLAB® inputs into pointer variables (“mxGetPr”) and create the output MatLAB® variables (“mxCreateDoubleMatrix”). Assuming "testmex.c" is the source code file, the following line should be written to compile it from MatLAB® (for 64bit MatLAB® distributions the Microsoft® Visual C++ compiler is suggested by MathWorks; how to install the Microsoft® Visual C++ for MatLAB® can be found in (Baker, 2009)).

```
%- link and compile mex source code
mex testmex.c
```

Therefore, one can now write a MEX file which creates the A array. The source C++ code of the "testmex.c" file is something like this:

```
#include "mex.h"

// computational routine
void allocateArray(double A[], int nr)
{
int i; // locale variable
for (i=0; i<nr; i++){
          A[i]=i+1; // "fill" array
                    }
}
// gateway routine
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
```

```
    double *A; // pointer to “int” type
    int nr; // define “int” variable

    // get “int” value from the input pointer
    nr = (int)*mxGetPr(prhs0]);

    // allocate MatLAB® double matrix
    plhs[0] = mxCreateDoubleMatrix(nr,1, mxREAL);

    // get the pointer to the output
    A = mxGetPr(plhs[0]);

   // allocate the output array by using the computational routine
    allocateArray(A,nr);
 }
```

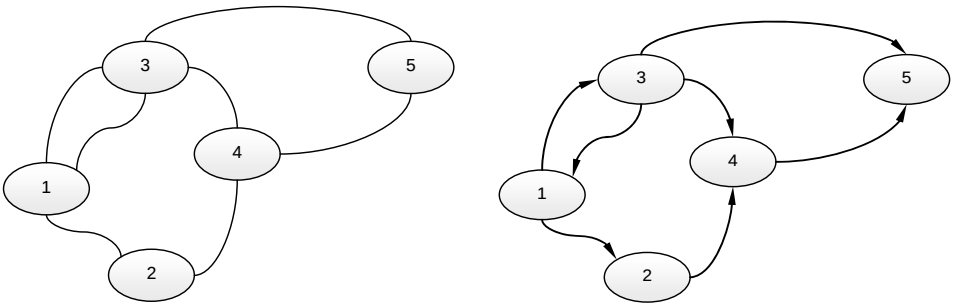
The compiled function may be easily called from MatLAB®, typing:

```
%- use compiled function
A=testmex(N);
```

The elapsed run-time is now 0.2893 s, that is, about 70% faster than the previous MatLAB® script. This way is particularly useful when managing large data sets, or nested loops are required.

A.2 Adjacency matrix and Laplace matrix

A graph "G" is usually defined by means of the vector list of vertices, "V", and the edge matrix, "E" (Berge, 2001; Deo, 2004).



(a) not-oriented graph (b) oriented graph

Fig. A.1. Graph representation

"V" is a vector of integer ranging from 1 to N_v, where N_v is the total number of vertices. "E" is an N_ex2 matrix, in which the i-th row has the indices of vertices connected by that edge (N_e is the number of edges). Let (i, j) be the couple of entries on the i-th row. For not-oriented graphs (see Fig. A.1a) it is (i, j)=(j, i), whereas (i, j)≠(j, i) for oriented graphs (see Fig. A.1b). Moreover, the same couple of vertices may be connected with more than one edge (in Fig. A.1, vertices 1 and 3 are connected with 2 edges).

$$E_{NO} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 3 & 1 \\ 2 & 4 \\ 3 & 4 \\ 3 & 5 \\ 4 & 5 \end{bmatrix} \equiv \begin{bmatrix} 2 & 1 \\ 3 & 1 \\ 1 & 3 \\ 4 & 2 \\ 4 & 3 \\ 5 & 3 \\ 5 & 4 \end{bmatrix}, \qquad E_{O} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 3 & 1 \\ 2 & 4 \\ 3 & 4 \\ 3 & 5 \\ 4 & 5 \end{bmatrix} \tag{A.1}$$

Looking at Fig. A.1, edge matrices, E_{NO} and E_O, for not-oriented and oriented graphs, respectively, are stated into equation (A.1). A useful representation of graphs, based on the edge matrix, may be achieved with the adjacency matrix, "A".

$$A(i,j) = \begin{cases} \sum_{k=1}^{Ne} edge_k, i \neq j \\ 0, otherwise \end{cases} \quad \forall i,j = 1,2,...,N_v \tag{A.2}$$

It is a symmetric square N_vxN_v matrix and defined as in equation (A.2). The entry (i, j) in "A" counts all edges connecting the vertex V_i to V_j. For example, looking at Fig. A.1a, the adjacency matrix becomes as into equation (A.3).

$$A = \begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{A.3}$$

For oriented graph it may be useful to preserve the sign of the edge. For this purpose, the Laplace or Kirchoff matrix, "L" can be introduced. It is a square N_vxN_v matrix and defined as in equation (A.4).

$$L(i,j) = \begin{cases} +\sum_{k=1}^{Ne} edge_k, i \neq j \text{ and } V_i \text{ directed to } V_j \\ -\sum_{k=1}^{Ne} edge_k, i \neq j \text{ and } V_j \text{ directed to } V_i \\ 0, \text{ otherwise} \end{cases} \quad \forall i,j = 1,2,...,N_v \tag{A.4}$$

The entry (i, j) in "L" counts all edges directed from the vertex V_i to V_j. For example, looking at Fig. A.1b, it has:

$$L = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & -1 & -1 & 0 \end{bmatrix} \tag{A.5}$$

5. References

Baker, L. (2009). *Microsoft 32/64-bit Visual C++ 2008 Express Support Files*, available online from http://www.mathworks.com/matlabcentral/fileexchange/22689-microsoft-3264-bit-visual-c-2008-express-support-files

Berge, C. (2001). *The Theory of Graph*, Dover Publications, ISBN-10: 9780486419756

Bordegoni M., Ferrise F., Ambrogio M., Caruso F., Bruno F. (2010). Data exchange and multi-layered architecture for a collaborative design process in virtual environments. *Journal on Interactive Design and Manufacturing*, Vol. 4, pp. 137-138.

Camelio, J. A., Hu, S. J., Ceglarek, D. (2004a). Modeling Variation Propagation in Multi-Station Assembly Systems with Compliant Parts, *ASME Journal of Mechanical Design*, Vol. 125, pp. 673-681

Camelio, J. A., Hu, S. J., Ceglarek, D. (2004b). Impact of Fixture Design on Sheet Metal Assembly Variation, *Journal of Manufacturing Systems*, Vol. 23, pp. 182-193

Ceglarek, D., Huang, W., Zhou, S., Ding, Y., Kumar, R., Zhou, Y. (2009). Time-Based Competition in Multistage Manufacturing: Stream-of-Variation Analysis (SOVA) Methodology - Review, *Journal of Flexible Manufacturing Systems*, Vol. 16, pp. 11-44

Chang, M., Gossard, D. C. (1996). Modeling the Assembly of Compliant, non-Ideal Parts, *Computer-Aided Design*, Vol. 29, pp. 701-708

Deo, N. (2004). *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall of India, ISBN-10: 0133634736

Di Gironimo, G., Franciosa, P., Gerbino, S. (2009). An RE-CAE Methodology for Re-Designing Free Shape Objects Interactively, *Int. Journal on Interactive Design and Manufacturing*, DOI 10.1007/s12008-009-0082-8

Franciosa, P., Gerbino, S. (2009). Handling Tessellated Free Shape Objects with a Morphing Mesh Procedure in Comsol Multiphysics®, In: *Proc. of COMSOL Conference'09*, Milano (Italy), October 14-16, 2009

Franciosa, P., Gerbino, S., Patalano, S. (2009). Variation Analysis of Compliant Assemblies: A Comparative Study of a Single-Station Assembly, *Journal Anales de Ingenieria Grafica*, N. 20, pp. 57-64

Franciosa, P. (2010a). *Modeling and Simulation of Variational Rigid and Compliant Assembly for Tolerance Analysis*, PhD Dissertation, University of Naples, Federico II, School of Engineering-Italy, available online from http://www.fedoa.unina.it

Franciosa, P., Gerbino, S., Patalano, S. (2010b). Variation Analysis of Compliant Assemblies: A Comparative Study of a Multi-Station Assembly, *Journal Anales de Ingenieria Grafica*, N. 21, pp. 45-52

Gerbino, S., Patalano, S., Franciosa, P. (2008). Statistical Variation Analysis of Multi-Station Compliant Assemblies based on Sensitivity Matrix, *Int. Journal Computer Applications in Technology*, Vol. 33, 1, pp. 12-23

Holland, T. O., Marchand, P. (2002). *Graphics and GUIs with Matlab*, Chapman and Hall/CRC, 3rd edition, ISBN-10: 1584883200

Kernighan, B., Dennis, M. R. (1978). *The C Programming Language*, Englewood Cliffs, Prentice Hall, ISBN 0-13-110163-3

Lengyel, E. (2003). *Mathematics for 3D Game Programming and Computer Graphics*, Charles River Media, ISBN-10: 1584500379

Perutka, K. (2010). Tips and Tricks for Programming in Matlab, In: *Matlab - Modeling Programming and Simulation*, Edited by Emilson Pereira Leite, pp. 2-16, available online from http://www.intechopen.com/books

Scott, T. Smith (2006). *Matlab Advanced GUI Development*, Dog Ear Publishing, ISBN-10: 1598581813