Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Matthias Pester

Visualization Tools for 2D and 3D Finite Element Programs – User's Manual –

Preprint SFB393/02-02

Preprint-Reihe des Chemnitzer SFB 393

SFB393/02-02 **January 2002**

Contents

Author:

Matthias Pester
Fakultät für Mathematik
TU Chemnitz, D-09107 Chemnitz `mailto:pester@mathematik.tu-chemnitz.de`

1 Introduction

Visualization of numerical results is a very convenient method to understand and evaluate a solution which has been calculated as a set of millions of numerical values. One of the central research fields of the Chemnitz *Sonderforschungsbereich* (SFB 393: *Numerical Simulation on Massively Parallel Computers*) is the analysis of parallel numerical algorithms for large systems of linear equations arising from differential equations (e.g. in solid and fluid mechanics). Special emphasis is paid to the investigation of preconditioners for finite element systems based on domain decomposition and multilevel techniques on massively parallel computers.

Solving large problems on massively parallel computers (more than 100 processors) makes it more and more impossible to store numerical data from the distributed memory of the parallel computer to the disk for later postprocessing. However, the developer of algorithms is interested in an *on-line* response of his algorithms. Both visual and numerical response of the running program may be evaluated by the user for a decision how to switch or adjust interactively certain parameters that may influence the solution process.

Thus, it is necessary to have an utility for a quick (and rather *dirty* than slow and perfect) interactive visualization directly from the parallel machine to the user's desktop workstation. A high-quality postprocessing (especially in 3D) requires a high-performance graphic workstation. Since, at least up to now, many users have only low-cost machines[1] on their desk, we prefer a quick visualization which is only based on X11 for compatibility with a wide field of workstation models. Another programming interface such as OpenGL should be better, especially for implementing 3D graphics, but it would require higher performance or hardware support.

In [10] we discussed two models for graphical output from a parallel computer, where either the parallel computer or a graphical workstation executes the major part of postprocessing. This manual considers mainly the first kind, i. e. preparing all data in parallel and then transmitting pixel data to the user's workstation. We assume that at least standard X-library calls are available to the parallel program. An approach to use the second model with a 3D visualization tool based on GRAPE ([12]) was implemented and is described in [7]. As another external viewer we can use the IRIS Explorer [17]in the same way. A first version of the X11-based approach for 2D visualization can be found in [9]. Meanwhile this graphic tool has been updated and is also used as a kernel for an X11-based 3D visualization. Therefore, it is appropriate to give a new survey of the current state.

[1]There will be always differences between low-cost and high-performance computers, but on another level after some years.

2 Visualization of 2D Domains

This part of the visualization package is intended to supply a simple graphical interface for 2-dimensional finite element data structures. Furthermore, it is used as the low level interface for certain 3D graphics (see Chapter 3).

This graphics package uses the interface of the X11 library as documented in [8], no further extensions. Hence, it has been portable to any unix-like machine, including special parallel computers as GCPowerPlus. No third-party libraries are required. The look and feel of the user interface was designed to support most of the daily requirements in testing parallel algorithms. Figure 1 shows a set of typical views for one example and most of the menu items that are available in the graphics window.

First, we will describe the programmer's interface.

2.1 Programmer's Interface for 2D Visualization

From the programmer's point of view, the complete visualization including interaction acts as a black-box that is to be initiated by one of the following calls (**Note**, that this call has to be executed locally on each of the processors in a parallel program):

```
call gebgraf  (iDoF, nEL, ne, EL, nNod, Nod, U, W)
call gebmgraf (iDoF, nEL, ne, EL, nNod, Nod, U, Mat, W)
call firegraf (iDoF, nEL, ne, EL, nNod, Nod, U, Xi, W)
```

The meaning of the parameters is given in Table 1.

The program `gebgraf` is used in the case of layer or elasticity simulation in domains with homogeneous material, whereas in the case of multiple materials `gebmgraf` may be used. The `firegraf` call adds some features for the case of flow simulation.

The use of one of those graphic programs requires the programmer to write a little subroutine that is called by the graphic program. This little subroutine (`getdofs`) is intended to define the number of components in the solution vector field `U` and the names of each component to appear in the `DOF`-menu (2.2.2) of the graphics window.

The program returns two parameters:

```
call getdofs (nDoF, Dofs)
```

`nDoF`	:	number of components that can be displayed (see notes below!)
`DoFs`	:	the names of all the components in an array `character*6 DoFs(nDoF)`, i.e. 6 characters for each name (for C programmers: we are using Fortran strings, i.e. by default the strings are not `\0`-terminated)

For convenience, the graphics subroutines make some assumptions on the output of `getdofs`. Thus, if the solution consists of $k > 1$ components per node then `getdofs`

Table 1: Parameters of the 2D graphics subroutines.

```
gebgraf  (iDoF, nEL, ne, EL, nNod, Nod, U, W)
gebmgraf (iDoF, nEL, ne, EL, nNod, Nod, U, Mat, W)
firegraf (iDoF, nEL, ne, EL, nNod, Nod, U, Xi, W)
```

`iDoF`	: preselect one component of the solution vector to be displayed by default; `iDoF=0` means that the field `U` contains no data, and the program will only allow to draw the grid;
`nEL`	: number of elements of the (local) finite element mesh;
`ne`	: number of nodes per element; the program supports 3- and 6-node triangles and 4-, 8- or 9-node quadrilaterals;
`EL`	: integer array (in Fortran: `EL(ne,nEL)`) of the node numbers for each element;
`nNod`	: number of (local) nodes of the grid;
`Nod`	: single precision[2] real array (in Fortran: `Nod(idim,nNod)`) containing x, y-coordinates (and possibly more information) for each node of the mesh; generally, the field dimension `idim` is 3, but might be changed globally by a `common` variable;
`U`	: components of the solution (double precision real), stored as an array of vectors, one vector of length `nNod` for each component (degree of freedom of the problem solved), i.e. in Fortran this field is `U(nNod,nDoF)`, where the number `nDoF` and the names of the various components are obtained from a user-supplied subroutine (named `getdofs`, see below);
`Mat`	: integer array of length `nEL` containing one number (index of material) for each element;
`Xi`	: double precision vector of length `nNod` containing the values of the stream function (used by a program for flow simulation);
`W`	: scratch field of "sufficient" size[3], this is used to store all the computed pixel data, which may be essentially more than the list of nodes.

[2] Single precision is sufficient for visualization, but with respect to adaptive mesh refinement this may be changed to double precision in future versions.

[3] This is buggy since nobody knows before what is sufficient. A new version of the program should use an additional parameter indicating the available size on `W` to avoid crashing by memory lack.

Table 2: For `firegraf` some components have to be arranged in a special order.

`1..nDoF-3`	:	arbitrary components, with velocity (x, y) as fields 1 and 2;
`nDoF-2`	:	pressure, the last field of the array `U`, however, with data values only for the corners of each element (undefined and unused entries for inner points of edges or faces);
`nDoF-1`	:	stream function which is not on `U`, but on `Xi`;
`nDoF`	:	a placeholder to be set in `getdofs`, e.g. as `|U|` or `vector`. This menu entry does not match any single component of the solution, but causes a vectorial (or magnitude) display of the velocity field that is stored in the first two fields of the array `U`.

should return `nDoF=`$k + 1$. Why that? If there is a vector solution available – such as velocity in flow simulation, displacement in elasticity simulation or a gradient of a scalar solution – its components (x- and y-value) are stored as the first two fields in the array `U` and the last number (`nDoF`) is reserved to display the vector solution (arrows or the magnitude of the (x, y)-vector). Thus, `U` has only `nDoF-1` fields of components. If `firegraf` is used, there are some more assumptions as defined in Table 2.

2.2 User's Interface for 2D Visualization

The visualization appears in an extra window together with the simple menu for interaction (Fig. 1). The initial display may be determined by the programmer (`iDoF`, Table 1), but the user may now select any options, parts of the solution and display mode. Furthermore, it is possible to produce a postscript file for high quality printing. We will now give a summary of the various menu options.

At the top of the window there is a menu bar with 6 entries, from left to right these are:

`Draw` pull down menu to select the display mode (2.2.1). The selected mode is performed immediately.

`DoF` pull down menu to select the component of the solution which is to be displayed by the next draw command (2.2.2)

`Color` pull down menu to select one of various predefined color palettes (2.2.3)

`Param` pull down menu to set various parameters for drawing modules (2.2.4)

`Option` pull down menu to set or switch some additional options (2.2.5)

`goon` leave the interaction and return to the program.

2.2.1 The $\boxed{\texttt{Draw}}$ Submenu

Select a display mode.

$\boxed{\texttt{Redraw}}$ Clear the graphics window and repeat the previous drawing, with possibly changed other options (e.g. line color, colormap, scaling, selected component).

$\boxed{\texttt{Bound.}}$ Draw only the boundary of the subdomains. There is an option (2.2.5) to switch between drawing all boundaries of the coarse grid or only the boundaries of the current distribution of subdomains to the processors. The color of the lines is green in the case of Dirichlet boundary conditions and red for Neumann boundary conditions.

Note: We assume the information on boundary conditions to be stored in the node field (`Nod`) as a bit mask indicator in the third entry behind the coordinates x and y. If this information is not stored this display mode has no output.

$\boxed{\texttt{Net-2D}}$ Draw the current mesh. In general, the grid lines are drawn in a standard color (white or black). However, if $\boxed{\texttt{Procs}}$ is selected from the `DoF`-submenu, the grid lines are colored according to the processor number, and for $\boxed{\texttt{Mater}}$ the grid line colors correspond to the material numbers.

$\boxed{\texttt{Vector}}$ Draw small vectors in each grid point. This menu entry replaces the previous one as long as the vector solution is selected from the `DoF`-submenu – the last entry which is supplied by the subroutine `getdofs` (p. 4) in the case of more than 2 components. This menu entry appears alternately with the next one as selected from the `Option`-submenu (p. 12).

$\boxed{\texttt{Tensor}}$ Draw small orthogonal crosses in each grid point corresponding to the eigenvector directions of a tensor solution (if one of the 2 eigenvectors is specified as a vector solution). This menu entry replaces the `Net-2D` one as long as the vector solution is selected from the `DoF`-submenu – the last entry which is supplied by the subroutine `getdofs` (p. 4) in the case of more than 2 components. This menu entry appears alternately with the previous one as selected from the `Option`-submenu (p. 12).

$\boxed{\texttt{Net+U}}$ Show the deformation of a mesh by adding the displacement vectors to the node coordinates and displaying the grid. The line color is the same as for isolines (see below) and may be selected from the `Param`-submenu entry `LinCol`. The scaling for the deformation may be changed by the `Param`-submenu entry `Elast` (p. 11). Note that a large (scaled) deformation may exceed window bounds and will be clipped. It is intentionally not scaled to window size in order to have a real comparison with the undeformed mesh.

$\boxed{\texttt{Net-3D}}$ Draw a 3-dimensional view of the grid with the solution selected from the `DoF`-submenu giving the "height" of the grid points. If this drawing mode is selected there appears a little coordinate system in the upper left corner of the window. Clicking this coordinate system will allow to change the viewpoint by rotating via x-, y- or z-axis (press the corresponding key, and shift-key for backwards). Other keys are available for manipulating the viewpoint: `u,v,w,p,0,1,2,3,*,h` – try it. `ESC` will reset to the previous view, `RETURN` will accept. You must redraw your current solution explicitly to apply the new viewpoint.

`Isolin` Draw isolines of the current component of the solution. The background is not changed (so you can draw different things in the same picture). Before drawing, the number of isolines can be changed in the `Param`-submenu (2.2.4, `LinLvl`). The color of isolines is blue by default and may be changed in the `Param`-submenu (2.2.4, `LinCol`).

`Isol-F` Draw contours of the domain, filled with a standard color (white or black) and isolines as in the previous menu point (`Option`→`Isolin`, p. 12).

`Filled` Fill the domain with colors of the current palette corresponding to the current component of the solution. Coloring is done by linear interpolation between the grid points.

`PSopen` Open a new file for postscript output. The file name has to be entered in the console window where the program was started from. After opening this file you have to redraw everything you want to see in your postscript picture. Note, that one postscript file will contain only one image, written as EPS (encapsulated postscript). If you need multiple pictures you will have to close the first file and open a new one. Once a file has been opened the menu entry will be changed to `PSclos`. The postscript file will be closed explicitly by selecting `PSclos` or automatically by selecting `goon` to leave the graphics window interaction. More details on writing and using postscript files are given in section 4.1.

2.2.2 The `DoF` Submenu

The first entries of this submenu are taken from the parameters supplied by the user-defined subroutine `getdofs` (p. 2). As mentioned above the menu list given by `getdofs` will be terminated by an entry interpreting the first two components as vector. Two more entries may be added by the graphics subroutine itself:

`ElSize` Display the (2D-) element in colors corresponding to their element size. This may be useful for adaptive mesh refinement only.

`Mater.` Display materials in different colors. This menu item appears if the program `gebmgraf` was called with an additional vector of material indices.

`Procs` Display subdomains from different processors in different colors. This menu item exists if the program is running on a parallel machine.

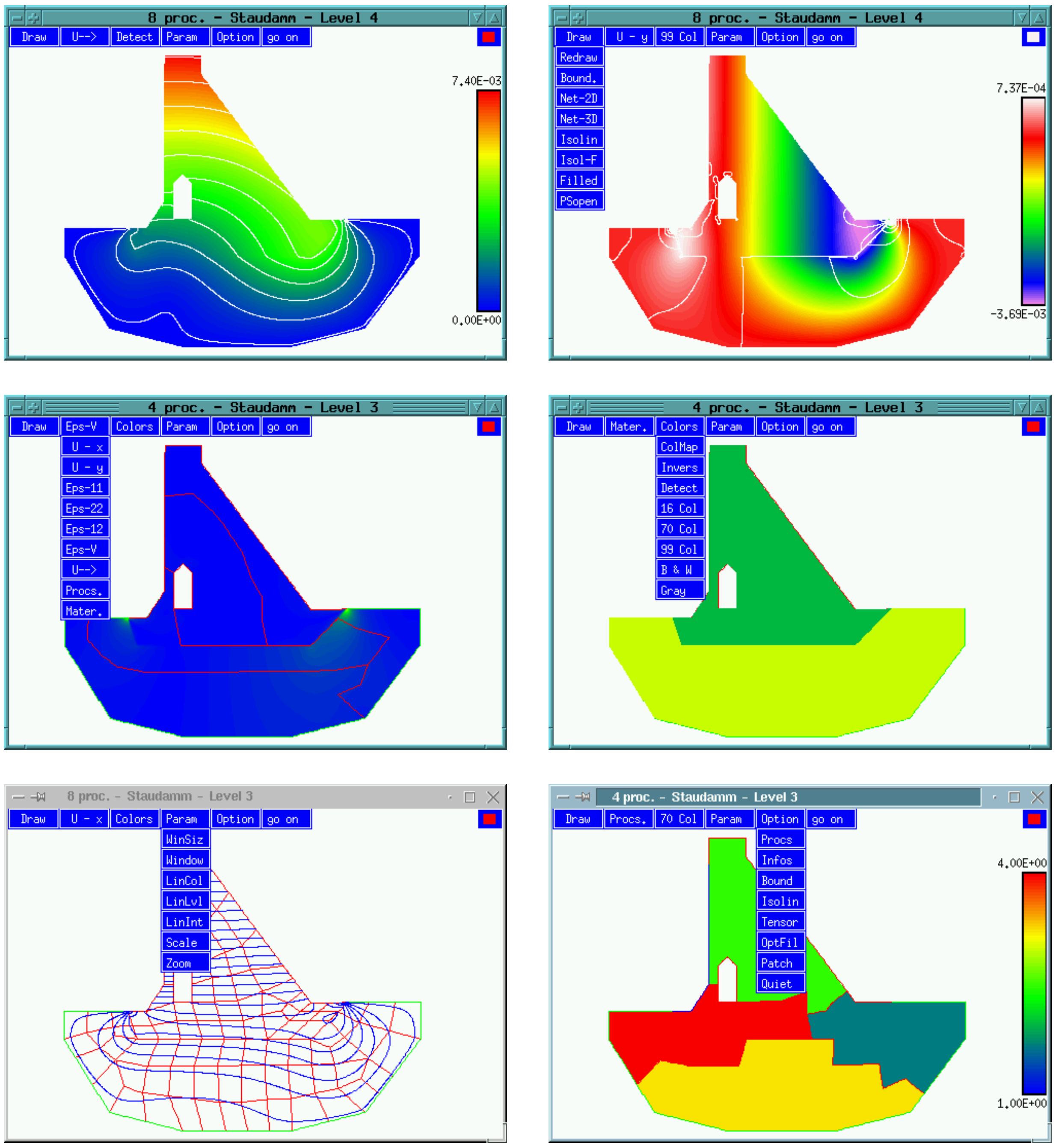


Figure 1: User interface on an X terminal, menus and various views (deformation, stresses, coarse grid, isolines, subdomains, materials, . . .) of an example.

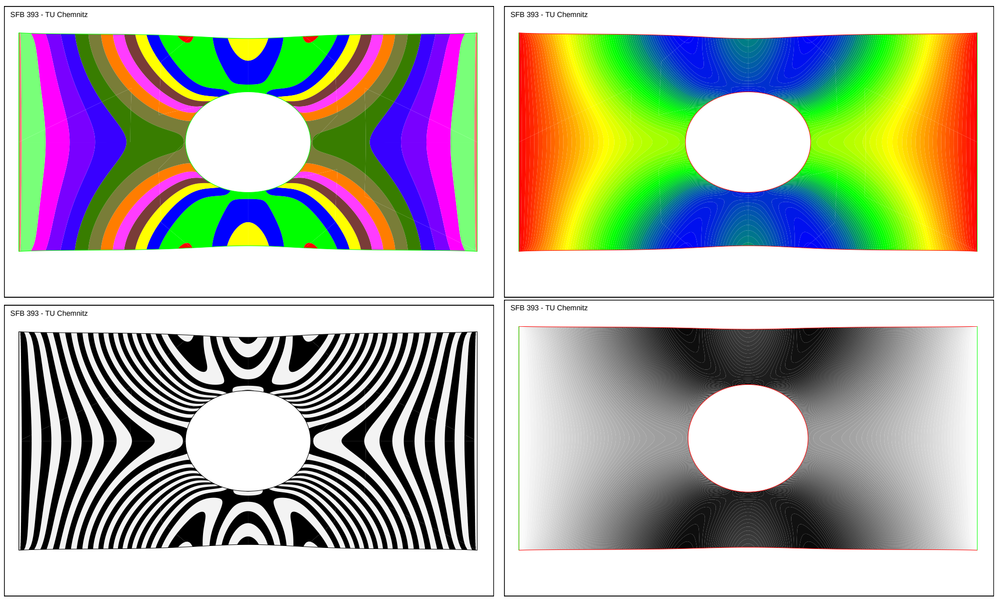


Figure 2: An example displayed using different colormaps.

2.2.3 The `Color` Submenu

Select one of the predefined palettes and switch background or the window's colormap.

`ColMap` Switch between default and private colormap. On pseudo-color screens there may be a maximum of 256 entries to the default colormap which is shared by all applications. Then the program might be unable to allocate enough colors if the colormap entries are occupied by other applications. Clicking this switch will use a private colormap for the graphics window. In this case the colors outside this window will change until the mouse pointer leaves the window – then the colors inside the window will be “wrong”. On true-color screens this switch may have no effect.

`Invers` Switch the colors black and white. By default the background is black and grid lines are white. This switch does not affect the postscript output.

`Detect` Select a default palette which is the same as `[16 Col]` for pseudo-color screens or `[70 Col]` for true-color screens.

`16 Col` An EGA-like palette of 14 colors (black and white are not used for coloring data areas). Useful to show more contrasts. (Fig. 2, top left)

`70 Col` Rainbow palette: blue - green - yellow - red. Useful for a more continuous coloring. (Fig. 2, top right)

`99 Col` Extended rainbow palette: magenta - blue - green - yellow - red - white.

$\boxed{\texttt{B \& W}}$ Black and white coloring, i.e. alternating black and white areas. (Fig. 2, bottom left)

$\boxed{\texttt{Gray}}$ Grayscale with upto 100 levels. The user is prompted for the number of gray levels between black and white. Grayscale is intended to use for printing on non-color printers, since the representation of colors (e.g. of the rainbow palette) might depend on printer drivers. (Fig. 2, bottom right)

This palette is best-suited for lighting effects if this program is used for 3D surfaces (Fig. 3 or Fig. 7). For lighting over a coloured solution refer to Section 4.3.

2.2.4 The $\boxed{\texttt{Param}}$ Submenu

Select and modify some parameters.

$\boxed{\texttt{WinSiz}}$ This will prompt for new width and height of the graphics window (in pixels). Of course, the window may be resized in the usual way using the features of the window manager on your desktop. However, sometimes you might want to define an exact size, e.g. to make a series of snapshots, or `mpeg_encode` needs a multiple of 8 or 16 pixels.

$\boxed{\texttt{Window}}$ Start a textual dialog to manage "virtual windows" within the original graphics window. You may split the window into 2 – 4 virtual windows of equal size (or delete them if no longer needed). The dialog allows either horizontal or vertical arranging of those virtual windows. Each of them can be used for a different display. Figure 3 is a screen shot of a horizontal placement of two virtual windows. Selecting any mode from the `Draw`-menu will always use the active virtual window highlighted by a frame.
Quickly switching between virtual windows is enabled by hotkeys, i.e. pressing the corresponding key $\boxed{1} \cdots \boxed{4}$.

$\boxed{\texttt{LinCol}}$ Select a color from the current palette to be used for drawing isolines. The default color is blue. A new window appears showing an array of all available colors. Click on a color field to select it. The last field shows the complete palette once more. If this is selected, the isoline color will be different corresponding to the value represented by this isoline.

Note: The selected color also affects the appearance of `Net-2D` (2.2.1), if the mesh has 6-point triangles or 8- or 9-point quadrilaterals, i.e. midpoints of edges for quadratic element functions. Then the (inner) lines connecting such midpoints are drawn in the same color that is selected for isolines. If the multi-color field was selected, the inner lines of the grid elements are not drawn.

$\boxed{\texttt{LinLvl}}$ This will ask in the console window for a number that indicates how many levels, i.e. how many isolines should be drawn. The default value is 25.

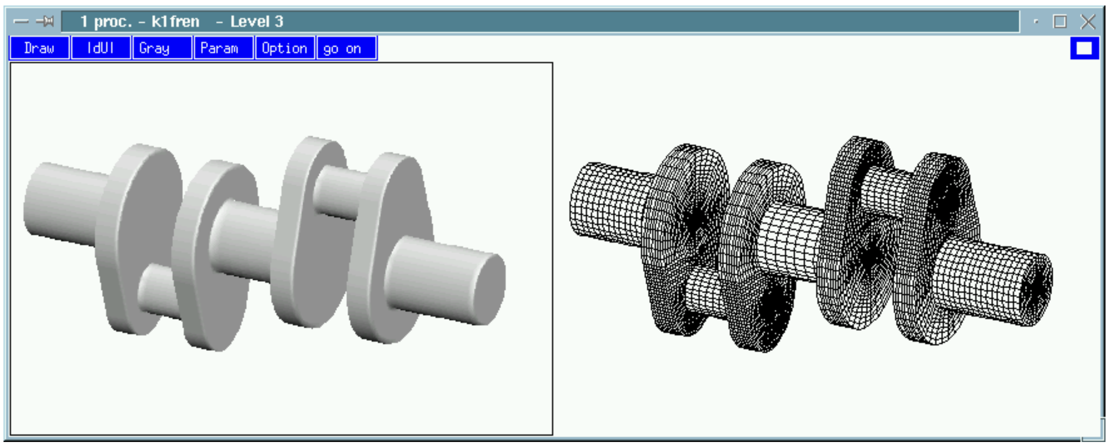


Figure 3: The screenshot shows the graphics window split into two virtual windows with different views of the same (3D) object

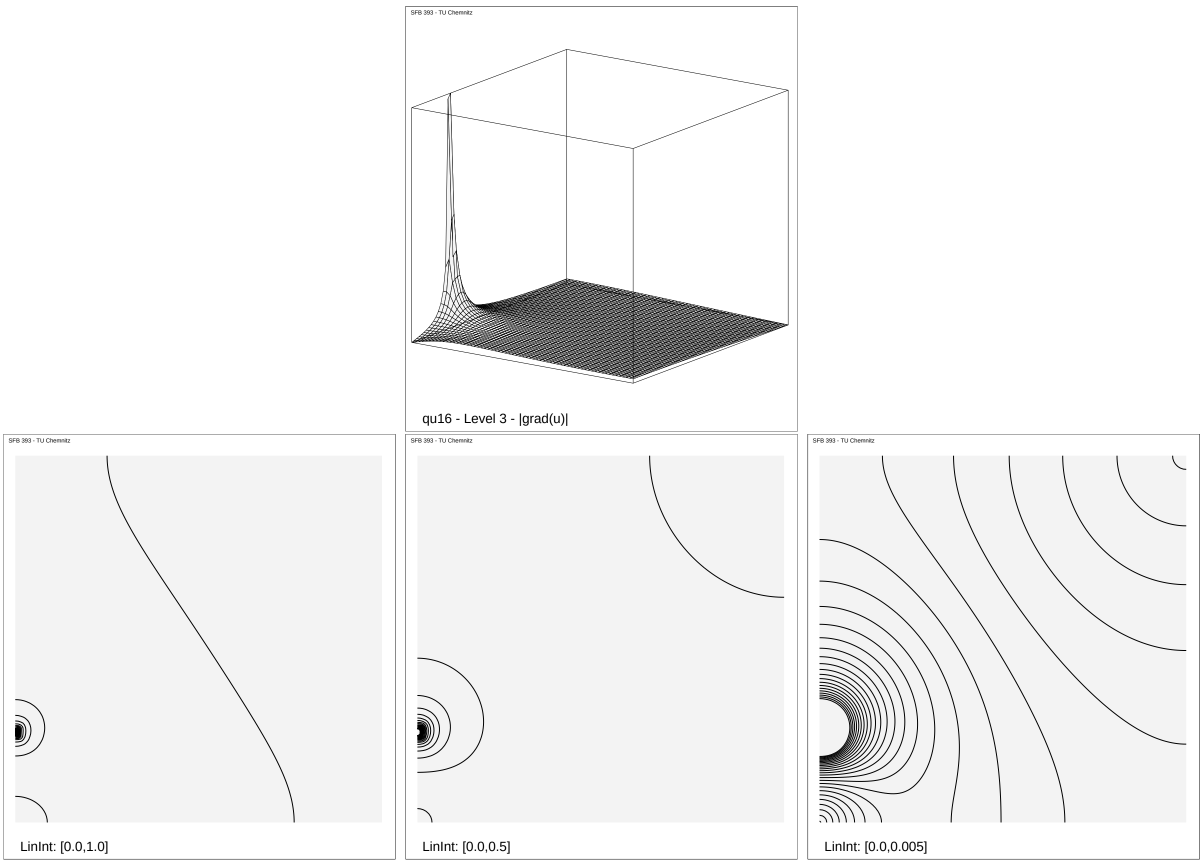


Figure 4: Different selection of **LinInt** for the isolines of a solution u (or $|\text{grad } u|$ in this case) with a local peak, intervals are $0\ldots1$, $0\ldots0.5$, and $0\ldots0.005$.

`LinInt` This will ask for an interval. The default interval from 0 to 1 corresponds to the minimum and maximum of the current component of the solution. Selecting a subinterval will concentrate all the isolines to that part of the component's values.

This may be useful if there is a small peak in the solution which would attract most of the isolines (Figure 4).

`Scale` Define a minimum and maximum value for the currently selected component of the solution. You may enter 0, 0 for minimum and maximum to return to the default behavior, i.e. automatic scaling from the current data values.

This facility is important for the visualization of time-dependent solutions as a series of pictures. It ensures that at different time the same color means the same value.

`Elast` This menu entry yields a textual interaction in the console window. The user may change parameters for the `Net+U` display (deformed grid). You may choose to set a relative or absolute scaling factor for deformation display.
A **relative** factor r means that the maximum displacement of all nodes is scaled to be r % of the maximum expansion of the domain, wih the exception that $r = 100$ means original deformation as supplied by the computational solution.
An **absolute** factor a means that the deformation is displayed a-times the real value, i.e. $a = 1.0$ is the true deformation.
In this menu you may also select any two of the components of the solution (`DoFs`) to be interpreted as horizontal and vertical components of a displacement vector (components 1 and 2 are used by default).

`Zoom` Zoom into the picture. Left-click on a first point in the window. The mouse pointer will change its shape and show a rectangle[4]. Move the mouse to a second point and left-click again to select the rectangle for zooming in. Right-clicking will terminate the interactive zoom selection and switch to the textual mode, where you can enter the (world) coordinates of a center point x, y and a radius r for the zoom area. You can enter $x = y = r = 0$ to leave the zoomed view and return to the default.

Note: The coordinate input mode is available by a hotkey: press `z` on the keyboard (while the graphics window is focused).

2.2.5 The `Option` Submenu

Select additional options for special purposes.

`Procs` You may enter a range of processors (between 0 and `nProc`-1). The next draw will only show the data of those processors.

`Infos` Display a few internal statistics, e.g. the total amount of pixel data that had to be transferred.

[4]Sometimes this rectangle may be badly visible because of low-contrast.

`Bound` Switch between two modes of displaying boundaries (2.2.1). The default is to draw only the boundaries (shapes) of subdomains (1 processor = 1 subdomain). The alternate is to draw all initial bounds, i.e. the coarse grid. (cf note on page 5)

`Isolin` Switch between two modes of displaying isolines, affecting `Isol-F` in the `Draw`-submenu (2.2.1). The major difference is how to draw the domain's background shape before drawing the isolines.

The default is a "quick" mode where the program tries to draw and fill each subdomain as a single polygon.

Since this may be incorrect if a subdomain had a hole inside (it would be filled as the background of the domain, cf Fig. 6), you may switch to a "slow" mode where each small element of the domain is filled one by one.

`Tensor` / `Vector` Switch to the specified drawing mode for vector solutions. This will change the corresponding menu entry in the `Draw`-submenu (2.2.1)

`OptFil` Select an "optimized" way to draw the coloring of the domain (2.2.1, `Filled`). By default, for each small triangle (or quadrilateral) a set of polygons – one for each color – is generated. If all those very small polygons have to be drawn this may take a while. There are implemented three different optimization algorithms (one *worse* than the other). These algorithms are to reduce the number of polygons by connecting them to larger ones. This may considerably reduce the pixel data which has to be transferred (by 30...90 percent). However, this will need a lot of computational time, but at least, this can be done completely parallel on each processor. It is sure that the drawing to the window will be faster if the polygons are optimized before. But it is generally not clear if the total time can be reduced. This depends on the number of nodes in the subdomains, the number of colors and the current data values. The different modes are:

(1) a simple and slow sorting algorithm that produces correct output;

(2) a quick and dirty algorithm that works well in most cases, however, incorrect if the solution in one subdomain contains a "ring" of one color with another color inside; it may happen that the inner field is overwritten by the color of an outer field (Figure 6);

(3) the same as (2) using `XOR`-mode drawing to the screen; this "repairs" the errors mentioned above. (But it is useless for postscript output!)

(0) this selects explicitly **no** optimization and no re-ordering of polygons (normally they are sorted by colors). This is equivalent to the next menu item (`Patch`).

Clicking `OptFil` the first time will ask for the optimization mode; clicking the second time switches off the optimization.

The default case (no optimization) would lead to very large (and hence slowly loadable) postscript files if `PSopen` is active. Therefore, postscript output is always optimized using the last mode selected with `OptFil` or mode `2`, if none was selected before. Thus, to avoid time-consuming optimization and accept large postscript files instead, you must select `OptFil` mode `0` before.

$\boxed{\texttt{Patch}}$ Switch on or off a so-called "patch" mode. In patch mode, all elements are drawn one by one, otherwise all colors are drawn one by one.

This patch mode is helpful to draw 3D surfaces with elements sorted from back to front (the simplest hidden surface method for convex bodies). For this purpose there is also a difference in drawing grids (`Net-2D` or `Net-3D` from the `Draw` menu) – in patch mode, the polygons are filled with the background color on screen or with a light gray in postscript output (see Figure 5).

$\boxed{\texttt{Quiet}}$ Leave the interactive graphics mode immediately and switch to a batch graphics mode. You may decide, if the visualization should be executed or not in future calls to the graphics subroutines. The program repeats up to two different draws (e.g. `[Filled]` for one component and `[IsoLin]` for another one) each time. When multiple virtual windows (2.2.4, `[Window]`, p. 9) are active each of them repeats its current view.

The batch mode can be finished by pressing `RETURN` or any other key while the graphics window has the focus (move the mouse into the window or click on the top bar of the window). Then the next call to `gebgraf` or `firegraf` enables the interaction menu again.

This mode is useful if the program runs in a loop where you want to see the differences in subsequent steps (time-dependency or adaptive meshes). Together with the `xxgrab`-utility (p. 38) you may create a series of image files (in GIF format) in order to produce animations.

2.2.6 Special Effects

Color Bar: There is a small rectangle in the upper right corner of the display area which is used to switch on or off a color bar at the right margin. This bar shows all colors of the current palette, and the lower and upper bounds of the numerical value of the currently displayed component of the solution.

Note that you may fix the lower and upper bounds with `Param`→`Scale` (p. 11)

Soft Interrupt: If a parallel program runs on many processors, the output of the subdomains from each processors is displayed one by one. There is a way to interrupt the output without killing the program: press the `ESC`-key (while the graphics window is focused) or keep the middle mouse button pressed until the cursor changes its shape (a ring of two arrows: ⟲). This interrupt is recognized after the output of the current processor has finished.[5] Thereafter, the output of the remaining processors is ignored – saving a little bit of time.

[5]Therefore, there is no way to interrupt the optimization that is caused by `OptFil`.

Hotkeys: There are some hotkeys for selecting certain menu items by a single key of the keyboard. It is unlikely that they will resist in future versions, so they are generally undocumented. Useful keys may be:

$\boxed{\texttt{z}}$ to enter a "Zoom" area using real world coordinates.

$\boxed{\texttt{1}}\cdots\boxed{\texttt{4}}$ to select one of the previously established "virtual windows" (p. 9).

Other "secret" hotkeys are, e.g.,

$\boxed{\texttt{u}}$ draw the 3D profile of the current solution (`Net3D`),

$\boxed{\texttt{U}}$ draw the current solution as colored area (`Filled`),

$\boxed{\texttt{x}}$ draw the first component as (`Net3D`),

$\boxed{\texttt{X}}$ draw the first component as (`Filled`),

$\boxed{\texttt{y}}$ draw the second component as (`Net3D`),

$\boxed{\texttt{Y}}$ draw the second component as (`Filled`),

$\boxed{\texttt{i}}$ specify an interval for isolines (`LinInt`),

$\boxed{\texttt{l}}$ draw isolines for the current solution (`Isol-F`),

$\boxed{\texttt{0}}$ (zero): draw the boundary of the domain (`Bound`),

$\boxed{\texttt{Q}}$ switch to batch graphics mode (`Quiet`),

and some more for special components used in `firegraf` (`O`,`F`,`P`,`R`,`S`,`V`).

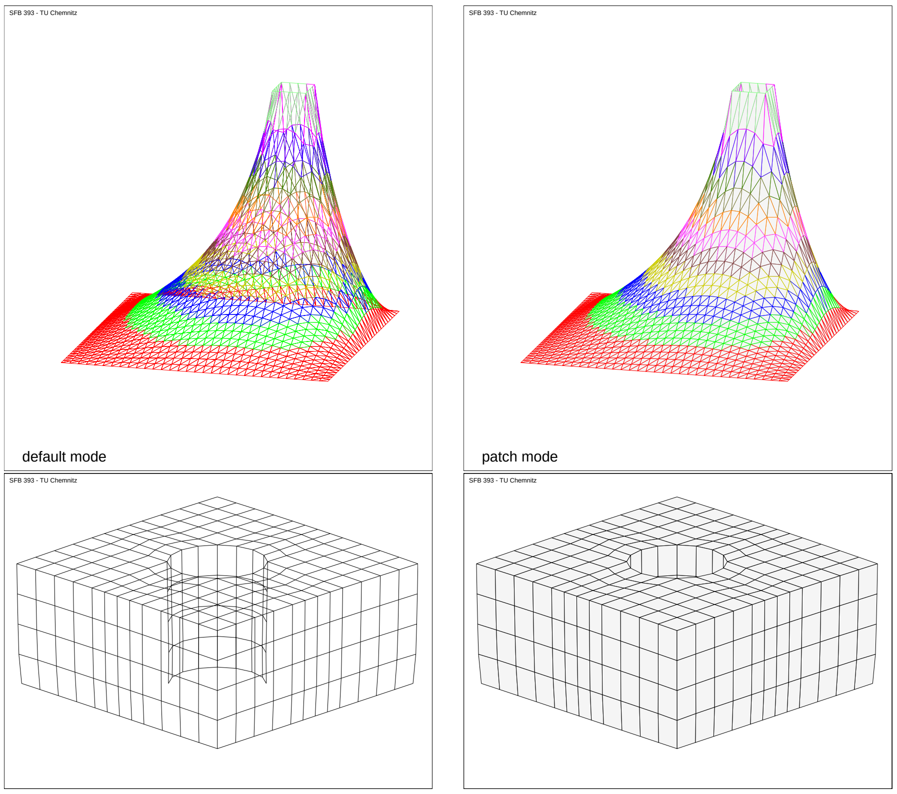


Figure 5: Two examples for drawing in default or patch mode: `Net-3D` for a 2D domain and `Net-2D` for a 3D surface

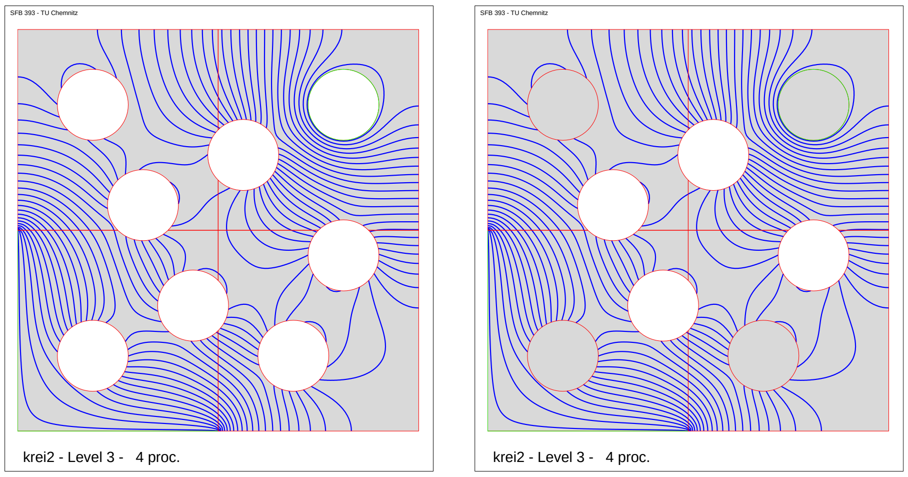


Figure 6: Difference of "`OptFil=1`" (left) and "`OptFil=2`" (right) where, erroneously, inner areas are filled if their boundary is completely inside one processor

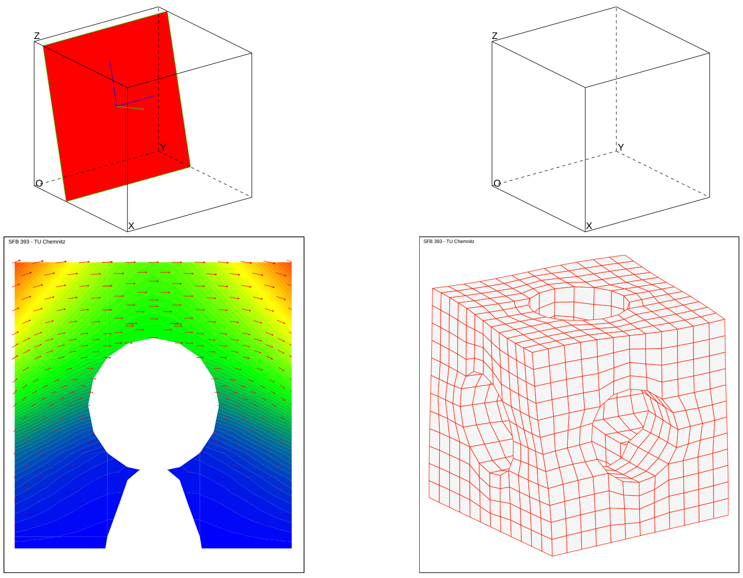


Figure 7: Projection of 3D vectors, left: displacement vectors in a cut plane, right: surface plot with `Net+U` (scaled 2D projection (u_{x_E}, u_{y_E}) of displacement (u_x, u_y, u_z) added to the projection of node coordinates), above: auxiliary display of cut plane and bounding box

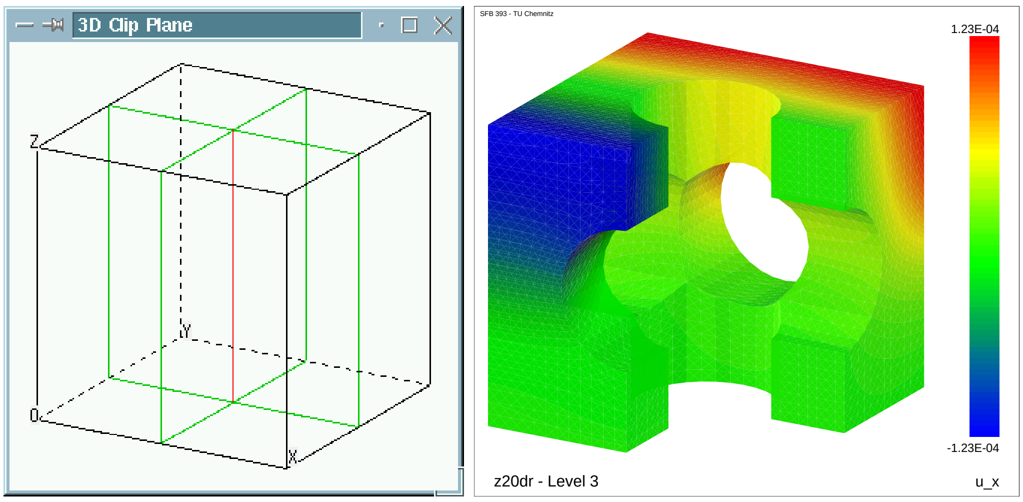


Figure 8: Cut off the intersection of two half spaces

3 Visualization of 3D Domains

3.1 Notes on Finite Element Data Structures

3.1.1 Assembled Matrices

Generally, the Finite Element Method leads to a system matrix $K \in \mathbb{R}^{N\times N}$ which is assembled from the element matrices $K_e \in \mathbb{R}^{r\times r}$

$$K = \sum_{e=1}^{numel} H_e K_e H_e^\top,$$

where N is the number of nodes in the mesh consisting of *numel* finite elements, r is the number of nodes per element (e.g. 4 or 10 for tetrahedra), and $H_e \in \mathbb{R}^{N\times r}$ is a Boolean matrix which rules the mapping of local indices in the element to global indices in the mesh, while $H_e^\top$ extracts the components for element e from a global vector.

For parallel computation using domain decomposition the assembly procedure may be done locally on each of the P processors for the corresponding subdomain $s = 0, \ldots, P-1$, getting P local matrices K_s and again a global mapping

$$K = \sum_{s=0}^{P-1} H_s K_s H_s^\top = \sum_{s=0}^{P-1} H_s \left(\sum_{e=1}^{numel_s} H_e^{(s)} K_e^{(s)} {H_e^{(s)}}^\top \right) H_s^\top .$$

However, K is never computed explicitly. All computation is performed in parallel using the local matrices K_s with only a minimum of communication between the processors for the nodes which are shared on subdomain boundaries ([4, 6, 3]). For example consider the matrix-vector multiplication $y = Kx$. On each processor we have the local part x_s of the global vector x, i.e. $x_s = H_s^\top x$. Then

$$y_s = K_s x_s$$

may be computed without communication, and y_s is the contribution of subdomain s to the global vector $y = \sum_s H_s y_s$. Consequently, the dot product $\langle y, x\rangle$ can be computed as

$$\langle y, x\rangle = \sum_{s=0}^{P-1} \langle y_s, x_s\rangle$$

which requires a simple global sum of local results only.

We denote this as our (classic) FE **data structure (I)**. Each finite element is defined by a list of references (indices) to the nodes belonging to it. The list of elements is stored in an array (parameter `Vol` in 3.2.1). Corresponding to the assembled matrix K or local matrices K_s we have a set of vectors for the degrees of freedom per node. The nodes belonging to edges or faces of subdomain boundaries (or, more general, of the coarse grid) are placed within the list of all nodes as continuous “chains” defined by `Ket1D, Ket2D`. This placement requires a little bit more effort in mesh refinement but yields much less effort in communication ([1]).

3.1.2 Element-by-Element Computation

For adaptive mesh refinement in parallel computation it is very hard to rebalance the computational effort by redistribution of the mesh. If the matrices are assembled as described above it is almost impossible to rebalance without loss of information and restarting the assembly process for the whole mesh.

The better alternative is to keep element data together including the element matrix K_e. In order to enable a quiet simple implementation of the redistribution of elements or clusters of elements among the processors, it is convenient as well to keep nodes and corresponding degrees of freedom together. However, numerical operations have to be executed using the small matrices K_e which is more expensive but may have advantages with respect to cache utilization. The most important advantage is to assemble new matrices only for new elements after an adaptive refinement step.

Denoting this as **data structure (II)** we may take advantage of some additional information for our purpose of visualization. Elements are defined by both lists of nodes and lists of faces. Faces are given as lists of edges and edges by nodes. Additional information may be included in the lists, e. g. a material indicator for each element, or a flag for boundary faces.

3.2 Programmer's Interface for 3D Visualization

Since our *quick-and-dirty* visualization for 3D meshes is based on (or reduced to) the previously described user interface for 2D, we have an additional interaction before calling the 2D-graphics module. This interaction may select a view point and/or a clipping plane or other parameters. At the moment, there exist three different programming interfaces for the visualization of 3D FEM data. They correspond to the two kinds of data structures of the previous section and three different user interfaces:

(1) The user may choose between surface and intersection. (cf. 3.3.1)

(2) Only the surface is displayed, but the user may clip away whole elements of a half space and display the surface of the remaining part of the domain. (cf. 3.3.2)

(3) Data transfer to an external high performance 3D graphics program (3.3.3).

The user interfaces will be discussed in Section 3.3. First consider the interfaces at the programmer's side. The following subroutines are available for the appropriate applications:

subroutine	data structrue	user interface	parameter description
`draw3d`	(I)	(1)	(3.2.1)
`x3dgraph`	(II)	(2)	(3.2.2)
`o3dgraph`	(I)	(2)	(3.2.3)
`out3dexpl`	(I,II)	(3)	(3.2.4)

For parallel programs the parameters of those subroutines refer to the local subdomain on each processor.

3.2.1 Subroutine draw3d

```
call draw3d (iDoF, nVol, nNode, Node, Vol, U, NodesInVol,
             W, LngW, Ket1D, Ket2D, Mat)
```

`iDoF`	:	preselect one component of the solution to be displayed by default; `iDoF=0` means that the field `U` contains no data, and the program will only allow to draw the grid;
`nVol`	:	number of elements of the (local) finite element mesh;
`nNode`	:	number of (local) nodes of the grid;
`Node`	:	single precision real array (in Fortran: `Nod(idim,nNode)`) containing x, y, z-coordinates (and possibly more information) for each node of the mesh; generally, the field dimension `idim` is 3, but might be changed globally by a `common` variable;
`Vol`	:	list of finite elements, each defined by `nVol` node numbers, i.e. the array is `Integer Vol(NodesInVol, nVol)`
`U`	:	components of the solution (double precision real), stored as an array of vectors, one vector of length `nNode` for each component (degree of freedom of the problem solved), i.e. in Fortran this field is `U(nNode,nDoF)`, where the number `nDoF` and the names of the various components are obtained from a user-supplied subroutine (named `getdofs`, see below);
`NodesInVol`	:	number of nodes per element; the program supports 4- and 10-node tetrahedra and 8-, 20- or 27-node hexahedra;
`W`	:	scratch field of `LngW` double words, this is used to store temporary arrays and, finally, all the computed pixel data.
`LngW`	:	length of the scratch array in double words
`Ket1D`	:	data structure that defines all nodes belonging to boundary edges
`Ket2D`	:	data structure similar to `Ket1D`, but for inner nodes of boundary faces For the structure of the `Ket1D` and `Ket2D` field see [1] and [3]
`Mat`	:	integer array of length `nEL` containing one integer number (material indicator) per element

The structure of `Ket1D` and `Ket2D` is defined in the source code using

```
include 'include/net3ddat.inc'
include 'include/com_prob.inc'
```

by the following declaration

```
Integer Ket1D(K1DDIM,NanzK1D), Ket2D(K2DDIM,NanzK2D)
```

and `Ket1D(pkzeig,k)`, `Ket2D(pkzeig,j)` are pointers to the first node of a boundary chain, and `Ket1D(pkleng,k)`, `Ket2D(pkleng,j)` define the lengths of those chains ($k = 1, \ldots,$ `NanzK1D`, $j = 1, \ldots,$ `NanzK2D`).

3.2.2 Subroutine x3dgraph

```
call x3dgraph (nDoF, nFaceInVol, nNodeInVol, VolF, VolN,
               mVol, nVol, iMaterial, Face, mFace, nFace,
               iBound, mBound, Edge, mEdge, nEdge,
               Node, mNode, nNode, H, maxH, IER )
```

`nDoF`	:	number of components of the solution (per node)
`mFaceInVol`	:	number of faces per volume element(4 or 6)
`nNodeInVol`	:	number of nodes per volume element(4, 10, 8, 20, or 27)
`VolF`	:	volume elements listed by face numbers[6], `Integer VolF(mVol,nVol)`
`VolN`	:	volume elements listed by node numbers[6], `Integer VolN(mVol,nVol)`
`mVol`	:	leading dimension of the arrays `VolF, VolF`
`nVol`	:	number of volume elements
`iMaterial`	:	index of the material indicator such that `VolF(iMaterial,k)` is a number refering to the material description of volume element `k`.
`Face`	:	faces listed by their bounding edges; obviously the number of edges per face is 3 for tetrahedra (`nFaceInVol=4`) or 4 for hexahedra (`nFaceInVol=6`) `Integer Face(mFace,nFace)`
`mFace`	:	leading dimension of the array `Face`
`nFace`	:	total number of faces
`iBound,mBound`	:	index and bitmask for a flag word such that `IAND(Face(iBound,k),mBound)` is `mBound` if face k belongs to the boundary of the domain and zero for inner faces.
`Edge`	:	edges listed by node numbers; this program expects to find starting and ending point of an edge as the first two entries; further entries (e.g. a middle point) are ignored. `Integer Edge(mEdge,nEdge)`
`mEdge`	:	leading dimension of the array `Edge`
`nEdge`	:	total number of edges
`Node`	:	coordinates of the nodes, `Real Node(mNode,nNode)`
`mNode`	:	leading dimension of the array `Node`
`nNode`	:	total number of nodes
`H`	:	scratch array
`maxH`	:	length of scratch array counted in `integer` words
`IER`	:	error indicator; a nonzero return value indicates an error 1 – no X server found to display 2 – not enough memory on scratch array `H` 3 – probably wrong parameter values

[6] `VolF` and `VolN` may be different entries of a contiguous field `Vol(mVol,nVol)` where `mVol`$\geq$`nFaceInVol+nNodeInVol`.

3.2.3 Subroutine o3dgraph

```
call o3dgraph (nDoF, VolN, nNodeInVol, mVol, nVol,
               Node, mNode, nNode, U, Mat, H, maxH, IER)
```

`nDoF`	:	number of components of the solution (degrees of freedom per node)
`VolN`	:	volume elements listed by node numbers, `Integer VolN(mVol,nVol)`
`nNodeInVol`	:	number of nodes per volume element(4, 10, 8, 20, or 27)
`mVol`	:	leading dimension of the arrays `VolF, VolF`
`nVol`	:	number of volume elements
`Node`	:	coordinates of the nodes, `Real Node(mNode,nNode)`
`mNode`	:	leading dimension of the array `Node`
`nNode`	:	total number of nodes
`U`	:	components of the solution (double precision real), stored as an array of vectors, one vector of length `nNode` for each component of the solution (degree of freedom of the problem solved), i.e. in Fortran this field is `U(nNode,nDoF)`, where the number `nDoF` and the names of the various components are obtained from a user-supplied subroutine (named `getdofs`, see below);
`Mat`	:	field of material indicators (integer numbers) for each volume element`k`.
`H`	:	scratch array
`maxH`	:	length of scratch array counted in `integer` words
`IER`	:	error indicator; a nonzero return value indicates an error 1 – no X server found to display 2 – not enough memory on scratch array `H` 3 – probably wrong parameter values

3.2.4 Subroutine out3dexpl

```
call out3dexpl (nDoF, nVol, mVol, VolN, nNode, mNode,
                Node, U, iUmode, Ket1, Ket2, H, maxH)
```

`nDoF`	: number of components of the solution (degrees of freedom per node)
`nVol`	: number of volume elements
`mVol`	: number of nodes per volume element(4, 10, 8, 20, or 27)
`VolN`	: volume elements listed by node numbers, `Integer VolN(mVol,nVol)`
`nNode`	: total number of nodes
`mNode`	: leading dimension of the array `Node`
`Node`	: coordinates of the nodes, `Real Node(mNode,nNode)`
`U`	: components of the solution (double precision real), stored as an array of vectors, either `nDoF` vectors of length `nNode`, one per component of the solution, or `nNode` vectors of length `nDoF`, one per node.
`iUmode`	: indicates the storage scheme of the field of solutions `iord=1` means: `U(nDoF,nNode)` and `iord=nNode` means: `U(nNode,nDoF)`
`Ket1D`	: boundary edges
`Ket2D`	: boundary faces **(see notes on page 19)**
`H`	: scratch array
`maxH`	: length of scratch array counted in `integer` words

The purpose of this program is to write one of three different data structures to a file or send this data via TCP/IP socket to a remote program. The ASCII file format is very simple. The details of the file structure is given in Section 3.3.3.

Note the special case:

This program written for the first kind of our data structures may handle the second kind with the following modifications in the meaning of parameters:

`mVol`	: is the leading dimension of the `VolN` field, different from the special numbers mentioned above
`iUmode`	: is the number of nodes per Volume
`mNode`	: is larger than 3, since:
`Node`	: contains $x, y, z, u_1, \ldots, u_{nDoF}$ for each node
`U,Ket1D,Ket2D`	: are dummy arguments

The initial indicator for the alternate data structure is the condition `mNode` > 3. In this case, however, the surface mode described below is momentary not supported.

3.2.5 Subroutine getdofs for 3D problems

As in the case of the 2D program, we have a user defined subroutine `getdofs` which returns the names for the different components of the solution. This subroutine has to satisfy all the different interfaces and appears slightly more complicated. Consider an example of this routine (in Table 3) which may be used as a default:
With this default routine and the calling sequence

$$\boxed{\text{user's main program}} \rightarrow \boxed{\text{3D graphics}} \rightarrow \boxed{\text{2D graphics}} \rightarrow \boxed{\texttt{getdofs}}$$

the `DoFs`-submenu may appear in one of the following forms

scalar solution		vectorial solution	
`dU/dxE`	`U`	`u_xE`	`u_x`
`dU/dyE`	`dU/dx`	`u_yE`	`u_y`
`dU/dzE`	`dU/dy`	`u_zE`	`u_z`
`U`	`dU/dz`	`|u|`	`|u|`
`dU/dx`	`|dU|`	`u_x`	`sig_11`
`dU/dy`	`****`	`u_y`	`sig_22`
`dU/dz`	`ElSize`	`u_z`	`sig_33`
`|dU|`	`Procs.`	`sig_11`	`sig_12`
`dU [E]`	`Mater.`	`sig_22`	`sig_23`
`ElSize`		`sig_33`	`sig_13`
`Procs.`		`sig_12`	`|sig|`
`Mater.`		`sig_23`	`****`
		`sig_13`	`ElSize`
		`|sig|`	`Procs.`
		`vector`	`Mater.`
		`ElSize`	
		`Procs.`	
		`Mater.`	

Here we assume that the user's main program has computed the components

- scalar solution U, grad $U = (dU/dx, dU/dy, dU/dz)$ and $|\text{grad } U|$ (i.e. `nDoF=5` for subroutines `x3dgraph`, `o3dgraph` or `out3dexpl`), or
- vectorial solution $u = (u_x, u_y, u_z)$, stresses $\sigma = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{23}, \sigma_{13})$, and $|\sigma| = \sqrt{\sum \sigma_{ij}^2}$ (i.e. `nDoF=10` for those subroutines).

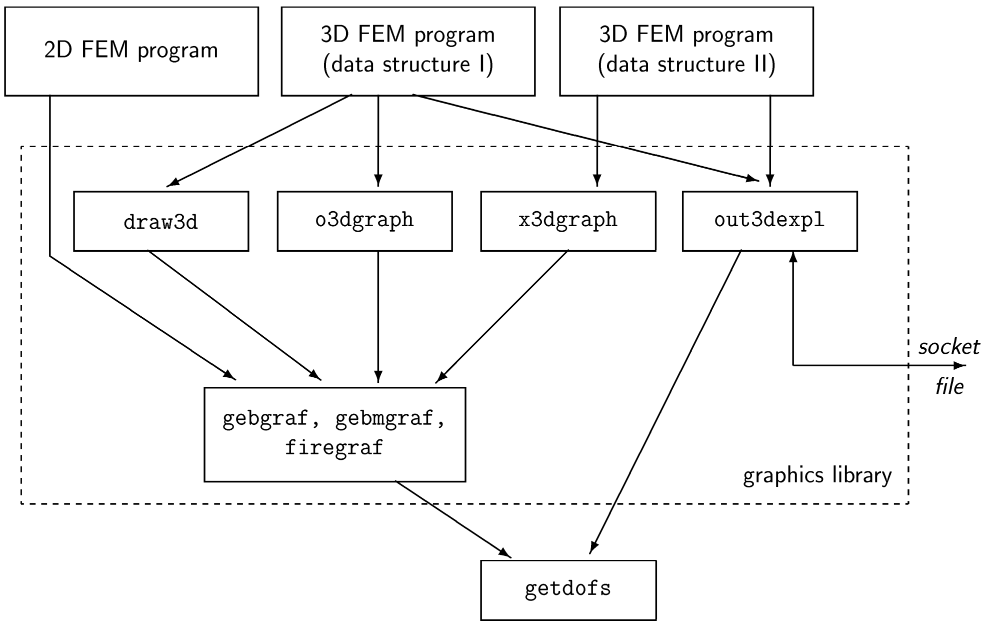
The other components are added by the 3D graphics subroutine automatically, depending on the interface being used. The component names with a trailing letter '`E`' refer to transformed (rotated) vector coordinates with respect to the current view point, e.g. (u_{x_E}, u_{y_E}) is the displacement in the screen plane or the projection of the 3D vector

Table 3: Subroutine **getdofs** (default example)

```
SUBROUTINE GETDOFS (nDoFs,DoFs)
integer nDoFs
character*6 DoFs(*)
include 'include/net3ddat.inc'
include 'include/Graf3D.inc'
nDoFs = NDF                ! NDF is input from COMMON 'net3ddat.inc'
i0=0
if (NDF .EQ. 1) then       ! scalar solution
  if (act_3d2d) then       ! act_3d2d and act_o3d are set in 'Graf3D.inc'
    DoFs(1)="dU/dxE"       ! vectors transformed to screen coordinates
    DoFs(2)="dU/dyE"
    DoFs(3)="dU/dzE"
    i0=3
  endif
  DoFs(1+i0)=" U"
  DoFs(2+i0)="dU/dx"       ! gradient was computed additionally
  DoFs(3+i0)="dU/dy"
  DoFs(4+i0)="dU/dz"
  DoFs(5+i0)=" |dU|"
  if (act_3d2d) then
    DoFs(6+i0)="dU [E]"    ! projection of vectors to screen plane
  else
    DoFs(6+i0)=" **** "    ! useless for 3D
  endif
  nDoFs=6+i0
else                       ! vectorial solution
  DoFs(4)=" |u|"
  if (act_3d2d) then
    DoFs(1)=" u_xE"        ! vectors transformed to screen coord.
    DoFs(2)=" u_yE"
    DoFs(3)=" u_zE"
    i0=4
    nDoFs=nDoFs+3
  endif
  DoFs(1+i0)=" u_x"        ! vectors in original coordinates
  DoFs(2+i0)=" u_y"
  DoFs(3+i0)=" u_z"
  if ((.not. act_3d2d) .or. act_o3d) i0=1
  DoFs(4+i0)="sig_11"      ! stress tensor values
  DoFs(5+i0)="sig_22"
  DoFs(6+i0)="sig_33"
  DoFs(7+i0)="sig_12"
  DoFs(8+i0)="sig_23"
  DoFs(9+i0)="sig_13"
  DoFs(10+i0)="|sig|"
  nDoFs=10+i0
endif
if (NDF .GE. 3 .AND. (act_3d2d .OR. act_o3d) ) then
  nDoFs = nDoFs+1
  if (act_o3d) then
    DoFs(nDoFs) = " **** "   ! useless for 3D
  else
    DoFs(nDoFs) = "vector"   ! first 2 components as 2D vector
  endif
endif
END
```

(u_x, u_y, u_z) to the screen (Figure 7). Thus, in the 3D case the 2D operation `Net+U` makes sense only for such a vector mapping.

Finally, we have a short schematic view on the calling hierarchy of the programs:


2D FEM program
3D FEM program (data structure I)
3D FEM program (data structure II)
draw3d
o3dgraph
x3dgraph
out3dexpl
socket
file
gebgraf, gebmgraf, firegraf
graphics library
getdofs

3.3 User's Interface for 3D Visualization

3.3.1 First User Interface for 3D: Surface or Intersection

This was our first implementation of a 3D visualization *frame* around the 2D visualization program based on the classic FEM data structure (I) (page 17). If the programmer decided to call the subroutine `draw3d` (3.2.1), the user will be prompted to select among some options for mapping the 3D data to the 2D screen. Consider the following example (Table 4).

In line `1` we select `'s'` to get a surface plot. Next in line `11` we could select to invoke one of the actions

`x,y,z`	– change the specified component of the normal vector
`X,Y,Z`	– rotate around the specified axis (angle is requested)
`d`	– for $d > 0$ you will get a perspective projection
`M`	– rotate the view coordinate system by mouse dragging
`L`	– switch on/off some lighting effects for a more natural view
`s`	– specify a scaling factor for adding computed elastic deformations
`P`	– write a postscript file from the auxiliary window (coordinate system)
`v,F`	– create a sequence of views (either rotating or scaling)
`ENTER`	– finish input, call the 2D graphics program `gebgraf`

Table 4: Example 1 of a user's dialog

```
 1  surface / cutting plane / quit ([s]/p/q) s
 2  Specify a normal vector from the plane towards the viewer,
 3  and a positive distance for perspective view:
 4
 5  current view           3D bounding box
 6   nx=     1.000         x:    -40.00 ..    165.00
 7   ny=    -1.000         y:    -29.97 ..     29.97
 8   nz=     0.500         z:    -50.00 ..     50.00
 9    d=     0.000
10
11  Select component !  ({x,y,z,d,M,L}; {[P]S-3D-Box,mo[v]ie}; {ENTER})
12
13  distance from origin=  0.  (=parallel projection)
14  normal vector screen -> eye:
15   0.666666687 -0.666666687  0.333333343
16  2d x-axis:  0.707106769  0.707106769  0.
17  2d y-axis: -0.235702261  0.235702261  0.942809045
18
19  grafische Darstellung ? - [j]/n :
20
21  Select new view point ? (ENTER/n) or cut (p)lane :p
22  Select component !  ({x,y,z,d}; {[P]S-3D-Box,mo[v]ie}; {ENTER})
23
24  distance from origin=  0.
25  normal vector screen -> eye:
26   0.  0.  1.
27  2d x-axis:  1.  0.  0.
28  2d y-axis:  0.  1.  0.
29
30  grafische Darstellung ? - [j]/n :
```

Any change of parameters is displayed in the auxiliary graphics window showing the bounding box as a cuboid in the current view coordinate system.

In the example we did not change anything. Finally, pressing the `RETURN` key causes the program to compute a 2D FEM data structure representing the projected 3D surface and calls `gebgraf`.

In line `19` we have to confirm that the result may be displayed in the graphics window. Now we have the same interaction as described for 2D problems in Section 2.2. Finishing this 2D visualization (clicking on `goon`) returns to our simple 3D dialog having the alternatives shown in line `21`. Typing `'p'` (for *plane*, or `'e'` for *Ebene*) switches to the cutting plane mode. The auxiliary window shows the bounding box from a default view point and the current position of the cutting plane (upper left picture in Figure 7). The next line (`22`) is similar to line11 described above. Here we may specify the normal vector of the cutting plane and its distance from the origin. Note that the normal vector is **signed** and points to the viewer. In the auxiliary display the *front side* of the plane will appear red and the *back side* is blue. So one may verify the local coordinates within the cutting plane.

3.3.2 Second User Interface for 3D – Surface and Clipping Planes

This implementation is based on the extended finite element data structure (II) (page 18). It appears if the programmer decided to call either `x3dgraph` (3.2.2) or `o3dgraph` (3.2.3). The prompting is similar to that previously described (but not exactly the same). Therefore, consider one more example (Table 5).

First (at Line 10) we may select among several options to modify the 3D view.

`x,y,z`	– change the specified component of the normal vector (view vector)
`X,Y,Z`	– rotate around the specified axis (angle is requested)
`d`	– for $d > 0$ you will get a perspective projection
`M`	– rotate the view coordinate system by mouse dragging
`L`	– switch on/off some lighting effects for a more natural view
`C`	– define or modify clipping planes
`R`	– reset to the default view
`P`	– write a postscript file from the auxiliary window (coordinate system)
`?`	– display a short help message
`ENTER`	– finish input, call the 2D graphics program `gebgraf`

In the example we define two clipping planes (lines 16 and 24) and choose to cut off the intersection of the two half spaces above those two planes (line 33, Figure 8). The input of `'L'` (line 35) switches on the lighting effects of the program yielding a more realistic 3D impression of the pictures written to postscript files (not on the screen display).

3.3.3 Third User Interface for 3D – File or Socket

Refering to an example[7] demonstrating the visualization of FEM data by means of the IRIS Explorer [17] on Silicon Graphics workstations, we implemented an appropriate simple external file data structure as an interface to any external visualization program.

A header defines the type of data ("2"=faces, "3"=tetrahedra, "4"=hexahedra) and the three counters: number of elements, number of nodes, number of data values per node.

The data structures are either

- volume structure: a list of elements (tetrahedra or hexahedra) by node numbers, or
- face structure: a list of all faces (triangles or quadrangles) as 3D polygons by node numbers, or
- surface: a reduced list of faces (only the surface polygons of the 3D domain),

each followed by a list of nodes (k, x_k, y_k, z_k) and a list of data values for each node $(k, f_{1k}, \ldots, f_{mk})$.

[7]by Loris Renggli, Swiss Federal Institute of Technology, Lausanne (Switzerland), 1993

Table 5: Example 2 of a user's dialog

```
enter components of the view vector directed from screen plane to viewer,
specify a positive distance for a perspective view:

current state :
 nx=     1.000       x:      0.00 ..      3.00
 ny=    -1.000       y:      0.00 ..      3.00
 nz=     0.500       z:      0.00 ..      3.00
  d=     0.000

Select:  ({x,y,z,d,M,C,L,R}; {{P}S-3D-Box}; {{H}elp}; {{N}o/{Q}uit})c
**** Define clipping planes  ****
planes so far:  0
"n" = new plane : n
normal vector <x,y,z>
(pointing into the half space to be cut off, <0,0,0> = remove this plane)
and distance from origin (x,y,z,d):1 0 0 1.5
**** Define clipping planes  ****
planes so far:  1
No.     normal vector      distance
 1    1.0000 0.0000 0.0000    1.500
"n" = new plane or index of plane to be modified : n
normal vector <x,y,z>
(pointing into the half space to be cut off, <0,0,0> = remove this plane)
and distance from origin (x,y,z,d):0 -1 0 -1.5
**** Define clipping planes  ****
planes so far:  2
No.     normal vector      distance
 1    1.0000 0.0000 0.0000    1.500
 2    0.0000-1.0000 0.0000   -1.500
"n" = new plane or index of plane to be modified :
Each plane defines a half space.
You may choose to cut off the intersection or the union of those half spaces.
([I]/U): i

Select:  ({x,y,z,d,M,C,L,R}; {{P}S-3D-Box}; {{H}elp}; {{N}o/{Q}uit})L
lighting effects switched on.

enter components of the view vector directed from screen plane to viewer,
specify a positive distance for a perspective view:

current state :
 nx=     1.000       x:      0.00 ..      3.00
 ny=    -1.000       y:      0.00 ..      3.00
 nz=     0.500       z:      0.00 ..      3.00
  d=     0.000

Select:  ({x,y,z,d,M,C,L,R}; {{P}S-3D-Box}; {{H}elp}; {{N}o/{Q}uit})
  1888 boundary faces visible.
grafische Darstellung ? - [j]/n :
```

Table 6: Contents of a data file for external postprocessing

line(s)	contents
1	identification string: `PFEMread 1.0`
2	type information (an integer value: 1, 2, 3, or 4), where 1 = 2D polygons 2 = 3D polygons (faces, surface plot) 3 = tetrahedra (4 vertices each) 4 = hexahedra (8 vertices each)
3	`np, nv, nd` (3 integers), where: `np` = number of polygons or polyhedra `nv` = number of vertices `nd` = number of data values per vertex
4...`np`+3	first data block containing polygons or polyhedra, 1 per line: polygons: `i n` $\mathtt{v}_1$ ...$\mathtt{v}_n$ (n=3 or 4) tetrahedra: `i` $\mathtt{v}_1$ ...$\mathtt{v}_4$ tetrahedra: `i` $\mathtt{v}_1$ ...$\mathtt{v}_8$ where `i`=index of polygon/polyhedron, $\mathtt{v}_j$=indices of vertices
`np+4` ... `np+nv+3`	second data block containing vertices, 1 per line: `i x y z` (index and 3D-coordinates, for type 1 only `i x y`)
`np+nv+4` ... `np+2·nv+3`	third data block containing data values related to vertices: `i` $\mathtt{f}_1$ ... $\mathtt{f}_{nd}$

The general format of the file is given in Table 6. As an example consider simply a cube consisting of only one hexahedral element. The program `out3dexpl` will offer to select which type of data file should be written: the complete hexahedra (tetrahedra) mesh, otherwise all faces as 3D polygons, or only the polygons of the surface. So we may obtain one of the following data files:

Type 4 (hexahedra)

```
FERead 1.0
  4
         1         8         5
1 1 2 3 4 5 6 7 8
1  0.000000E+00  0.000000E+00  0.000000E+00
2  0.100000E+02  0.000000E+00  0.000000E+00
3  0.100000E+02  0.100000E+02  0.000000E+00
4  0.000000E+00  0.100000E+02  0.000000E+00
5  0.000000E+00  0.000000E+00  0.100000E+02
6  0.100000E+02  0.000000E+00  0.100000E+02
7  0.100000E+02  0.100000E+02  0.100000E+02
8  0.000000E+00  0.100000E+02  0.100000E+02
1  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
2  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
3  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
```

```
4  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
5  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
6  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
7  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
8  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
```

The header is followed by a single line describing the (only) hexahedron by its node numbers, 8 lines with node coordinates and further 8 lines with data values for each node.

Type 2 (3D polygons)

```
FERead 1.0
  2
         6         8         5
1 4 1 2 3 4
2 4 5 6 7 8
3 4 1 2 6 5
4 4 2 3 7 6
5 4 3 4 8 7
6 4 4 1 5 8
1  0.000000E+00  0.000000E+00  0.000000E+00
2  0.100000E+02  0.000000E+00  0.000000E+00
3  0.100000E+02  0.100000E+02  0.000000E+00
4  0.000000E+00  0.100000E+02  0.000000E+00
5  0.000000E+00  0.000000E+00  0.100000E+02
6  0.100000E+02  0.000000E+00  0.100000E+02
7  0.100000E+02  0.100000E+02  0.100000E+02
8  0.000000E+00  0.100000E+02  0.100000E+02
1  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
2  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
3  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
4  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
5  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
6  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
7  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
8  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
```

The data columns of the last section are the components of the computational solution, here: $u, \dfrac{\partial u}{\partial x}, \dfrac{\partial u}{\partial y}, \dfrac{\partial u}{\partial z}, |\mathrm{grad}\, u|$.

For a tetrahedral mesh of the same cube we obtain similar files which differ only in the header and the first data section (since vertices are the same), e.g.

Type 3 (tetrahedra)

```
FERead 1.0
  3
         6         8         5
```

```
1 1 2 3 4
2 2 1 5 4
3 1 6 5 4
4 7 1 3 4
5 7 1 6 4
6 8 6 7 4
1  0.100000E+02  0.000000E+00  0.000000E+00
2  0.100000E+02  0.100000E+02  0.000000E+00
3  0.000000E+00  0.100000E+02  0.000000E+00
4  0.000000E+00  0.100000E+02  0.100000E+02
5  0.100000E+02  0.100000E+02  0.100000E+02
6  0.100000E+02  0.000000E+00  0.100000E+02
7  0.000000E+00  0.000000E+00  0.000000E+00
8  0.000000E+00  0.000000E+00  0.100000E+02
1  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
2  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
3  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
4  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
5  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
6  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
7  0.00000E+00  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
8  0.10000E+01  0.00000E+00  0.00000E+00  0.10000E+00  0.10000E+00
```

Such data files may be postprocessed by any separate tool, possibly after reformatting the file structure or adapting an input module. As an example we used the IRIS Explorer [17] which includes a lot of postprocessing modules to be placed, connected and controlled on a graphical desktop. Figure 9 shows an example where the module placed in the upper left corner of the map editor is used to read a data stream (similar to the contents of the data file) directly from the FEM program via internet socket.

The subroutine `out3dexpl` will use the socket data stream if the user specifies the single character `"#"` as filename for data output. In this case the program will function as server, i.e. waiting for requests from any client to get data. The subroutine `out3dexpl` returns to normal computation mode after the client's "request" to continue. In the following `PFEM` denotes the parallel FEM server functionality which is realized by the subroutine `out3dexpl` as described in Section 3.2.4.

The user interface for the Explorer visualization is determined by various modules. There are two separate modules written for the IRIS Explorer to work as client with the PFEM server.

PFEMread: Establish a connection to PFEM (host address of the server is input for this module). There are several buttons (Fig. 10) where the user may select one of three modes to request the 3D FEM mesh (polygons of the surface only, polygons of all faces, or volume data of tetrahedrons or hexahedrons) or "`go on`" to close the server connection and finish the server mode of `out3dexpl`. The *shrink* and *explode* options are related to the subdomains to visualize the domain decomposition on the parallel computer. For shrinking of single elements there is a standard module available from the Explorer module library.

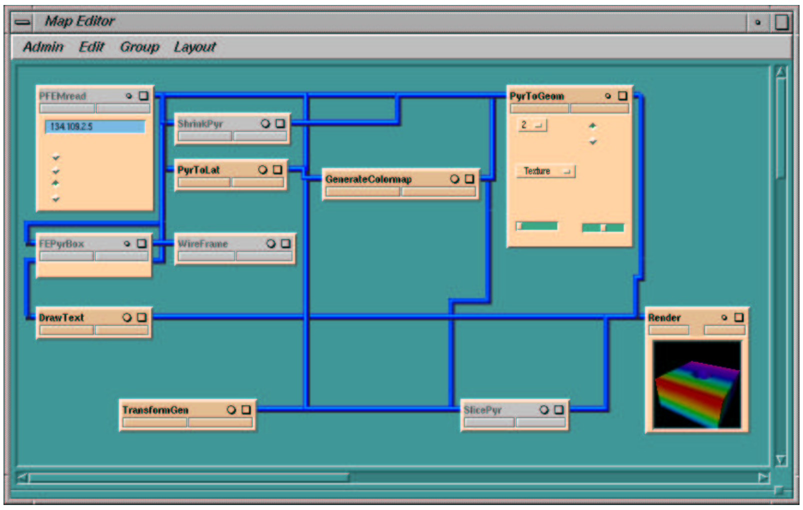


Figure 9: The Map Editor of the IRIS Explorer

PFEMdata: This module presumes the module PFEMread to have already received the mesh data as well as the number and names of the data components available on PFEM. In the map editor the corresponding output and input links of PFEMread and PFEMdata must be connected. The user may choose to receive one particular component of the solution or all components at once.

Another module may be connected for data arising from elasticity problems:

PFEMelast: Suitable for a solution where the first three components are x-, y- and z-components of a displacement to be added to the coordinates of the grid points. The user may vary a scaling factor to visualize the displacement. This module does not connect to PFEM.

The internal protocol for this client-server dialog is not necessary to be described here. A short description is given as Appendix.

An unsolved problem is the handling of large FEM data structures. Explorer modules run out of memory very soon. Thus, the performance of our machine (SGI O^2) is sufficient for 50,000 . . . 100,000 grid points only.

Another example for postprocessing from the file data is a small tool `fem_ogl` [11]. This program reads data of such a file and displays grid and solution in different ways based on OpenGL (Figure 12). In [7, 10] we discussed another method for interactively connecting a FEM program and an external visualization program based on the Graphical Programming Environment GRAPE [12].

Summarizing, we have to state that up to now several external viewers are well suited for a nearly perfect 3D rendering, but for small problem sizes only. Their difficulty is the missing support of parallelism. Our hand-coded X11 based visualization interface is still

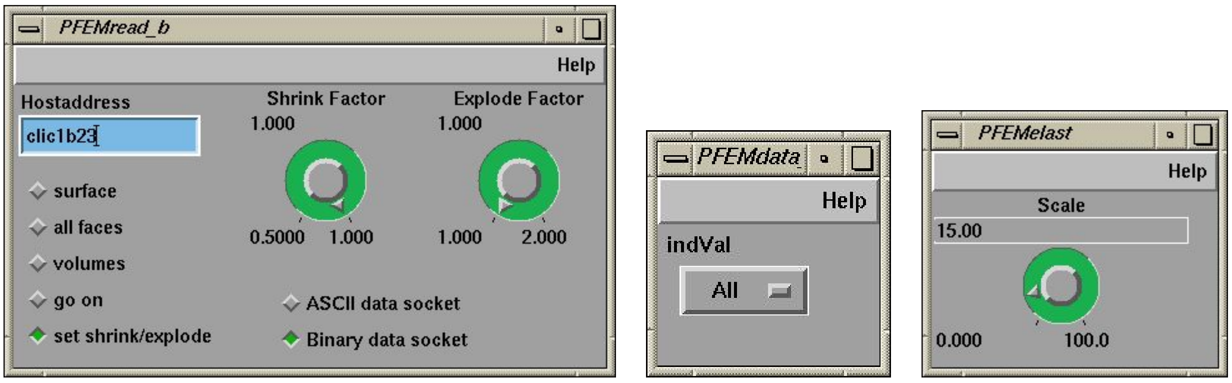


Figure 10: Explorer module interfaces `PFEMread`, `PFEMdata` and `PFEMelast`

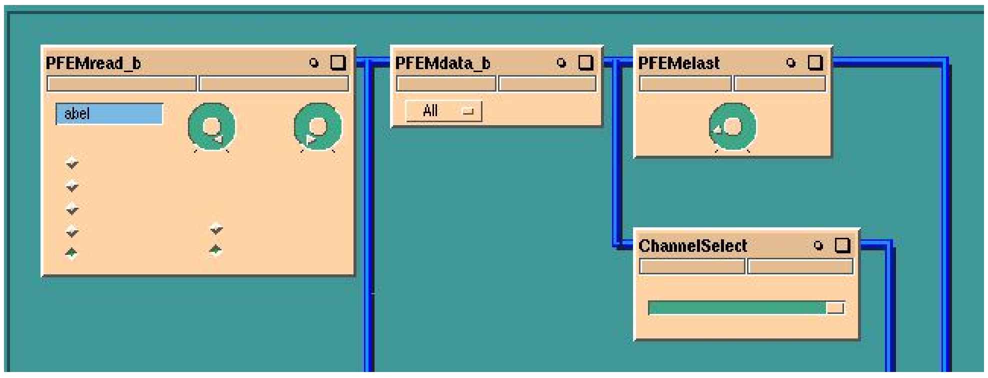


Figure 11: PFEM modules connected in the map editor of Explorer

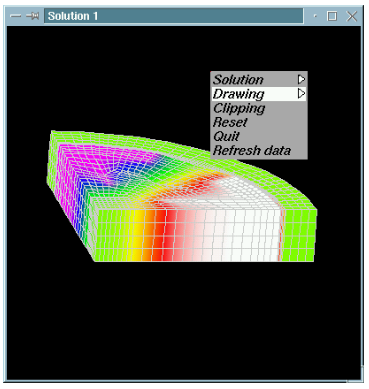


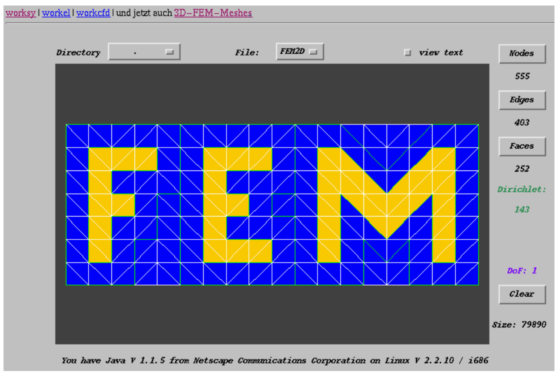
Figure 12: A small tool for OpenGL based visualization

the only way to show results of a massively parallel computation within a reasonable time (still large enough in relation to the computational time for the numerical solution), and without saving masses of data before knowing the worthiness of saving them.

3.4 Additonal Tools: Java Applets

There are two Java applets available. They can be used to visualize the objects described by the original 2D or 3D mesh files:

- http://www-usercgi.tu-chemnitz.de/~pester/meshes/shownets.cgi
  for 2D mesh files (Fig. 13), shows nodes, edges, faces, and source text of data files.



You have Java V 1.1.5 from Netscape Communications Corporation on Linux V 2.2.10 / i686

Figure 13: Snapshot of the 2D mesh viewer Java applet

- http://www-usercgi.tu-chemnitz.de/~pester/meshes/showstd.cgi
  for 3D mesh files (Fig. 14) with many features including cut planes to view inside.

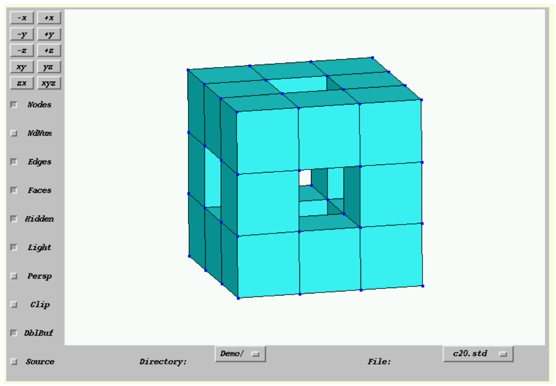


Figure 14: Snapshot of the 3D mesh viewer Java applet

4 Special Features

4.1 Postscript Output

If you need a better quality than obtained by a screen snapshot you should produce a postscript file. This is supported by the graphics program with the menu item `PSopen` (2.2.1, p. 6). The procedure is very simple but may take a long time for a high level of mesh refinement.

How to write a postscript file:

1. If necessary select optimization level 1 or 0 (the default is 2, cf `OptFil`, p. 12)
2. Select `Draw`→`PSopen`.
3. Enter a name for the output file (`.ps` is appended if not typed in). Be sure to have write permission. The program will refuse to overwrite an existing file.

   (You may press the `Return`-key without typing a name to return to the graphics menu without opening a postscript file.)
4. Everything you draw to the screen from now on will also be written to the postscript file, i.e. as vector graphic, and with a higher resolution for the coordinates (about 30000×30000 dots instead of a few hundred pixels on the screen). Writing to the file takes a little bit more time than only drawing to the screen. Preparing the output, however, may take much more time because of a slow optimization to reduce the size of the output file and speed-up later loading times for the postscript file (cf notes on `OptFil`, p. 12).
5. Select `Draw`→`PSclos`.
   (This is implicitly done if you select `[goon]` from the top menu).

How to use the postscript file:
The postscript file is written in EPS format (Encapsulated Postscript). Thus, it is easy to embed into LaTeX, e.g.:

```
\usepackage{graphicx}
...
\begin{center}
\includegraphics[width=0.6\textwidth]{mysolution}
\end{center}
```

An important row within the postscript file is that defining the bounding box. It looks somewhat like this:

```
%%BoundingBox:   30 150 500 660
```

The `\includegraphics` (or `\epsfig`) command may need this bounding box to check the space to be reserved for the figure. All the other data of the postscript file is only needed by `dvips`.

Thus, you can substantially improve the performance of the LaTeX compile step, if the bounding box definition is placed at the beginning of the file. However, the program may calculate the bounding box only at the end of all output, so you will (if you want) have to change the file manually[8], i.e. replace the line

```
%%BoundingBox:   (atend)
```

at the top of the file by the correct line from the bottom of the file.

Some remarks on PDF:

The `\includegraphics` command as specified above will search for `mysolution.eps` or `mysolution.ps` if it is invoked by the command `latex`, it will search for `mysolution.pdf` (or `*.bmp, *.png, *.tif, *.jpg, *.mps`) if you compile the LaTeX file by `pdflatex`.

The postscript files written by our graphics program can be converted to PDF (Portable Document Format) easily using the command

```
epstopdf mysolution.ps
```

Simple modifications in the postscript file:

It might happen that you want to make little changes in the postscript file such as changing a comment, removing the frame border, changing line thickness or colors. Therefore, let's have a short excurse to the *user interface* within the postscript file:

At the top of the file you will find a section like this

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Switch visibility of bounding box:
/Border { true } def
% Switch visibility of caption text:
/Capt { true } def
% Text and text height [pt] for caption:
/CaptText (qu16 - Level 3 -   4 proc.) def
/CaptSize 16 def
% Width for grid lines:
/wGrd { 0.1 SC div w } def
% Width and colors for boundary lines:
/wBnd { 0.5 SC div w } def
```

[8]I wrote a small shell script `BBtop` for this purpose.

```
/cBnD { 0 1 0 sc } def  % Dirichlet
/cBnN { 1 0 0 sc } def  % Neumann
% Width for isolines:
/wIso { 1.2 SC div w } def
% patch background color:
/cpat { 0.96 sg } def
% enable clipping (for Net+U or Zoom):
/Clip { true } def
% lighting effects (shadows)
/Shfact { 1.0 } def  % range 0.0 (bright) ... 1.0 (dark)
/Shsq  { true } def  % use a quadratic scale
%%%%%%%%%% END OF PARAMETER SECTION %%%%%%%%%%
```

This section is what you can modify without being an expert in postscript writing. The numerical and Boolean values in braces may be changed, other text should stay unmodified. In detail, this means in the following lines:

`/Border` **:** Change `true` to `false`, if you want no border line around the picture. The border line has just the size of the bounding box.

`/Capt` **:** Change `true` to `false` if no text is to be displayed within the picture.

`/CaptText` **:** The caption text which is displayed at the bottom of the picture (only if `Capt=true`). By default this is the name of the data file (the user's coarse grid file), the number of refinement steps (levels), and the number of processors that were used for computation. You may change only the text inside the brackets `( ... )`.

`/CaptSize` **:** The text height which is specified in points (pt). Note, however, that this may be a relative value if the figure is scaled by `\includegraphics[width=...]{...}` or `\includegraphics[height=...]{...}`.

`/wGrd`, `/wBnd`, `/wIso` **:** Line width specified in points (pt) for the different kinds of lines in a picture: grid lines, boundary lines, isolines. The line width is also scaled by `\includegraphics[...]{...}`.

`/cBnD`, `/cBnN`, `/cpat` **:** Define some special colors for boundary lines with Dirichlet boundary conditions (green by default), or with Neumann boundary conditions (red by default), or the background color for the patch mode (p. 13).

The color definition is either in RGB mode (e.g. "`0 1 0 sc`" means *green*, and "`0.54 0.27 0.08 sc`" means *saddle brown*), or in gray scale, where 0 is *black* and 1 is *white* (e.g. "`0.96 sg`" is a light gray as in Figure 5).

`/Clip` **:** `false` will disable clipping of `Net+U` display at domain bounds.

`/Shfact`, `/Shsq` **:** You may modify those numbers to find the best view of a 3D picture with light and shadow.

4.2 Video Sequences

Although computer performance is growing to “ infinity”, fluid dynamics simulation cannot be done in real-time. On current high-performance parallel computers a few seconds have to be spent to calculate one time-step of $10^{-1} \dots 10^{-3}$ s. However, a single picture of any solution at a certain time-step or a few such pictures at different time-steps on a sheet of paper are worth nothing compared with an animated video sequence that shows the time-dependent behavior of the simulated process. Thus, one should be able to save pictures of the solution for all time-steps. Since this may be a few hundred or thousand single pictures, it must be able to run automatically for a while (i.e. without manual interaction).

Our graphics tools are supporting this purpose with the help of a little external tool `xxgrab`[9]. The general situation is that you may use three computers in the following way. Of course, any two or all three may be physically the same machine, but you might find it better to distribute the work to remote computers.

Computer A: This may be the front-end of a high performance parallel computer or the interacting processor “0” of your parallel virtual machine or whatever you use for computation.

Computer B: This may be any compute server where you can run the program `xxgrab`. This program will (only in certain intervals) need a little bit of CPU time when it grabs a new image from your screen to save it to disk. Grabbing takes less than one second, but saving may take a few seconds more. But while saving an image the grabbed window may be already used to display the next image. That’s why it can be efficient to use different computers for displaying and saving the image.

Computer C: Your desktop computer running the X-server. Here you have telnet sessions to any other computers, and you can view the graphical output of the parallel program running on computer A.

Both A and B must have access to your X-server on host C (environment variable `DISPLAY`).

Any action of `xxgrab` is invoked by a signal from the program on the compute server (A). Therefore, a small package of subroutines is available to be used in your (parallel) program. If the program is running as a real parallel application, only one processor has to call those subroutines:

`call grabinit(iwin,ierr)`
: Open a socket connection from computer A to computer B where the program `xxgrab` must already run at this time (`xxgrab` acts as server to which `grabinit` may connect). The parameter `iwin` is the “handle” of the graphics window managed by your X-server (see [5]). If B and C are different computers you will be prompted for the hostname (or internet address) of computer B at this point.

[9]written by Michael Seibt

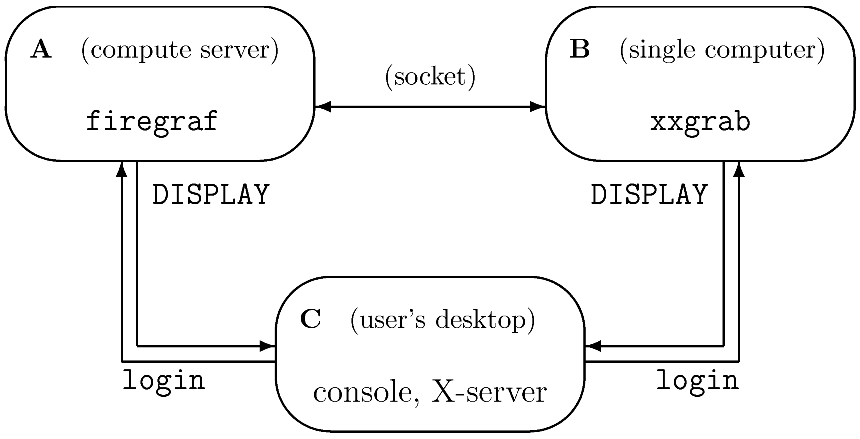


Figure 15: Configuration scheme for saving image sequences

`call grabname`
: This will ask the user to type in the base of a filename, and will send it to `xxgrab`. The program `xxgrab` will extend this filename by a sequence number (4 digits) and the file extension (`.gif`).

`call grabimage`
: Tell `xxgrab` to grab the current contents of the window specified by `GrabInit` and wait for a response by `xxgrab` that allows to use that window for output again. The filename sequence number is increased by 1.
Known bugs:
Overlapping the graphics window by another window will result in bad contents of the grabbed image. Although the program automatically raises the window to the top just before grabbing, it could happen that you were faster with the mouse . . .
If the graphics window is not completely on your screen (i.e. partially outside the screen borders or iconified) then `xxgrab` may crash without any possibility to restart and reconnect to the same running program on computer A.

`call grabimagenr (nr)`
: This is an alternate call similar to `grabimage`, but you can specify your own sequence number `nr` as an integer value, e.g. the number of the time-step or any other indication of the real simulation time.

`call grabdone`
: Close the connection to `xxgrab` if no more `grabimage` is to be done. The program `xxgrab` is terminated by this signal.

What do you have to do to generate a video sequence?

On compute server (A) you must run your program which contains something like this:

```
      include 'include/Grafics.inc'
      ...
C     initialize the graphics window (connect to your X-server)
      call gxinit(0,IER)
      if (IER .NE. 0) goto ...             ! no X-server found ?
      DO ( < for all time steps > )
C       ...
C       ...                                ! compute your solution
C       ...
        call firegraf (...)                ! and display it
        if ( < first loop > ) then
          call grabinit(han,IER)           ! han is defined by gxinit
          if (IER .NE. 0) call grabname    !            in Grafics.inc
        endif
        if (IER .NE. 0) call grabimage
      ENDDO
```

On computer (B) you should change to an empty directory where you want to place all the grabbed images, and from here you start the program `xxgrab`.
On your desktop (C) you may control everything. After you have selected all parameters in your graphics window that is displayed by the call of `firegraf`, you should select `Option`→`Quiet`, and sit and enjoy what happens for the next hours or days.

When you have finished because you lost your patience or your compute server crashed or you are pleased with the result (stop the quiet mode by pressing any key in the focused graphics window), or at any time in between, you can use

```
xanim *.gif &
```

to view the animation of the stored sequence of images. You may also use `animate` from the `ImageMagick` package ([16]).

Finally, you may learn how to use `mpeg_encode` ([13]) or other tools to generate MPEG or Quicktime movies. Suitable tools for producing animated GIF files are e.g. `gifmerge` ([15]), or `gifsicle` .

4.3 More Realistic 3D View

There are at least two effects to improve the "'reality"' of a 3D plot by giving an impression of space depth:

- foreshortening for a realistic perspective view,
- light and shadow to give more feeling for angles and directions in 3D.

Both of them are implemented in our graphics interface.

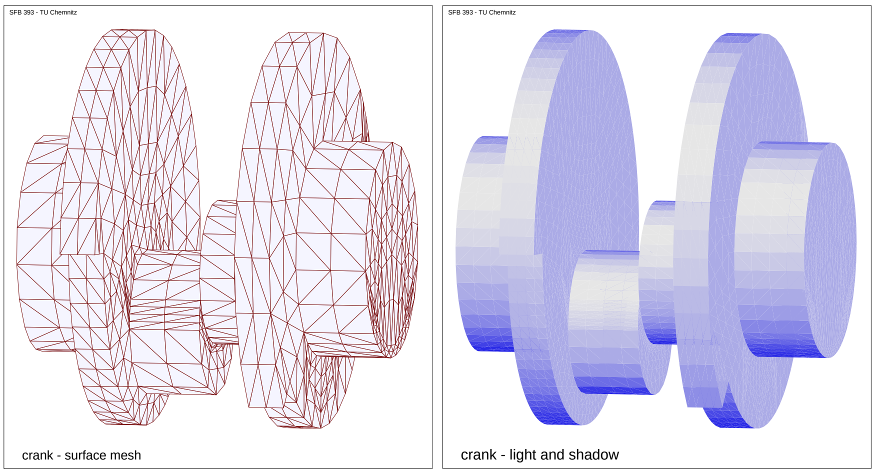


Figure 16: Light and shadow using a grayscaled view

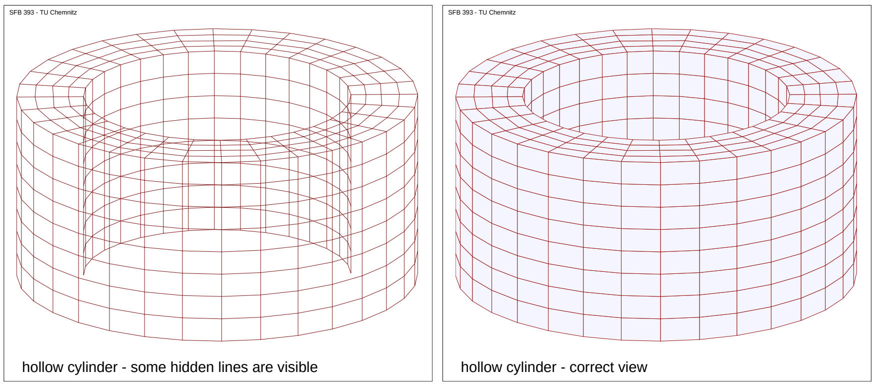

SFB 393 - TU Chemnitz
hollow cylinder - some hidden lines are visible
SFB 393 - TU Chemnitz
hollow cylinder - correct view

Figure 17: Patch mode for a non-convex solid

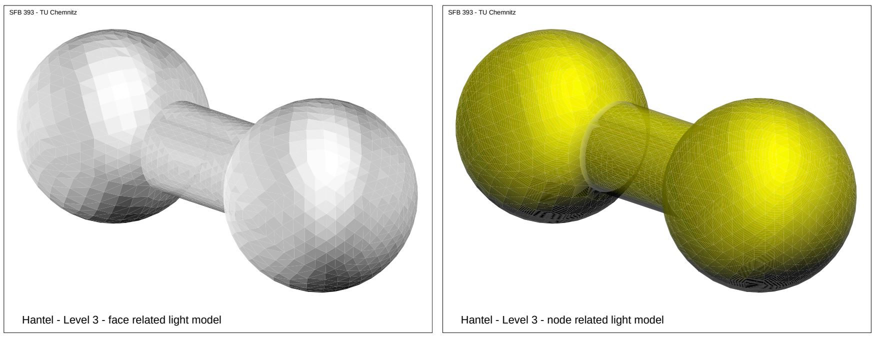

SFB 393 - TU Chemnitz
Hantel - Level 3 - face related light model
SFB 393 - TU Chemnitz
Hantel - Level 3 - node related light model

Figure 18: Face related light (left) or interpolation of node related light (right)

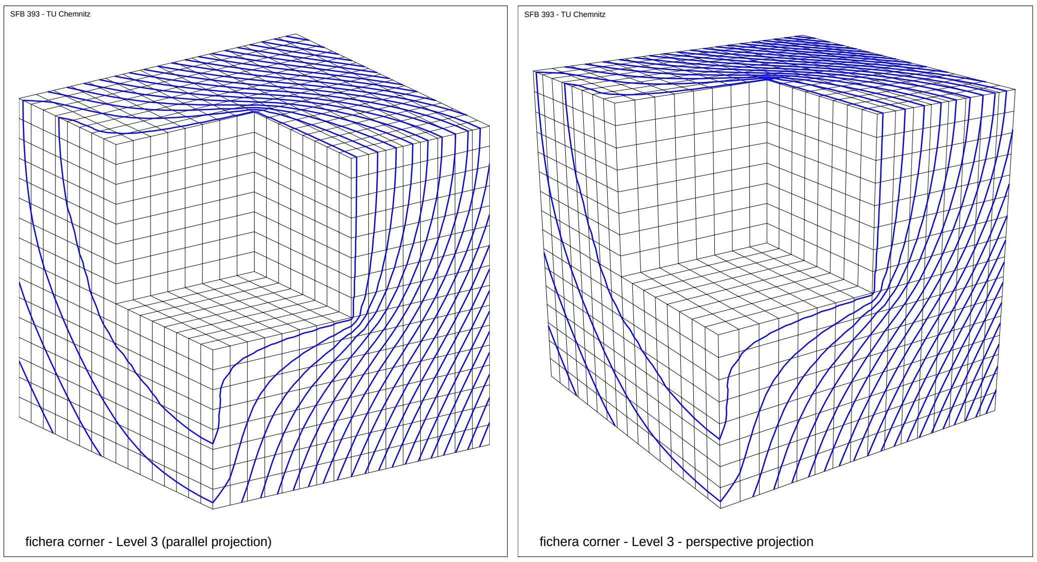

SFB 393 - TU Chemnitz
fichera corner - Level 3 (parallel projection)
SFB 393 - TU Chemnitz
fichera corner - Level 3 - perspective projection

Figure 19: Parallel and perspective projection of a 3D object

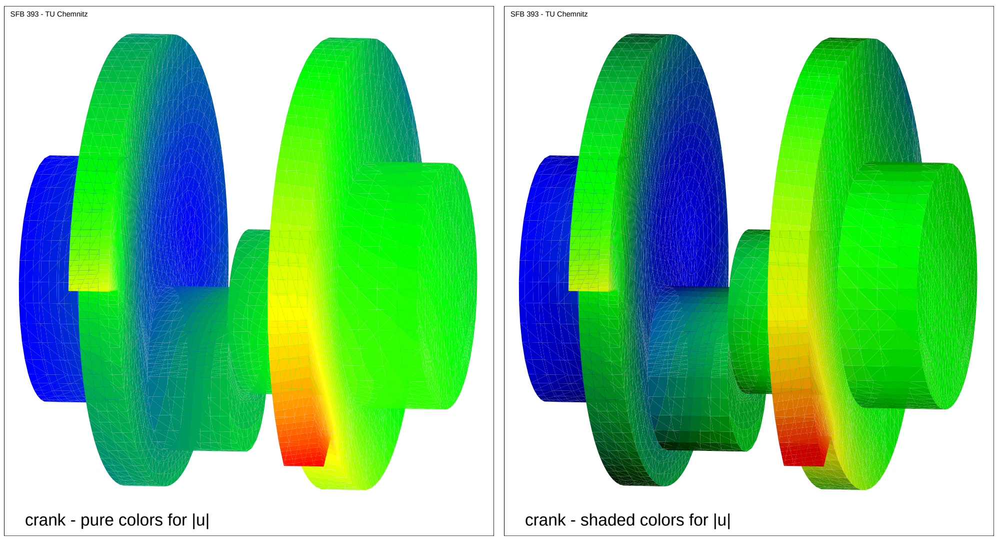


Figure 20: Pure colors for a solution (left) and overlaying shadows (right)

Perspective View

The user interfaces described in Sections 3.3.1 and 3.3.2 have an option to select both the direction $(x, y, z)^\top$ and the distance d of the viewer's position. The default distance $d = 0$ is used for parallel projection of the 3D object to the screen plane. A positive value of d indicates a perspective projection (in Figure 19 $d = 8$ for an object of size 1). The distance should be selected large enough to place the viewer outside the object to be viewed. The prompt for the selection of d includes a suggestion for a minimum value. A very large value for d will approach the parallel projection again.

Light, Shadow and Visibility

In the Sections about the user's interface for 3D we shortly mentioned the option `'L'` for lightening the object. This effect may be switched on and off by entering this option successively. If switched on the behavior of this option is the following:

- For each 2D polygon (projected face of the 3D object) there is stored a "material" value for brightness which can be displayed as described in Section 2.2.2 (`DoF`$\rightarrow$`Mater`). The best impression will be obtained chosing `Gray` from the color menu (2.2.3).
- The 2D elements are sorted by space depth. Hence, the option `Patch` (2.2.5) may be selected to draw the elements in sequence (instead of ordering the polygons by colors). This depth sort method should be used for non-convex solids to obtain a better representation of hidden surfaces (Figures 17 and 16). For convex solids this method is unnecessary.

  However, for parallel computations there was not implemented any global comparison between subdomains to decide which faces are visible. Each processor operates on its local subdomain only. Thus, an absolutely correct handling of visibility may be expected either for a convex solid or if the program runs on a single processor.

Smoothening of the Surface by Light Effects

The previously described method to show light effects is related to faces, i. e. each face has a constant brightness. So one may identify each patch of a curved surface by its own color (Figure 18). But the user may define a special "degree of freedom". Then this value is related to nodes and may be interpolated across the faces yielding a very smooth view of the surface.

Shadows On a Colored Surface

As shown above, the "light effect" of our graphics programs give a nice view on the screen for a grayscale. Moreover, those shadows may overlay the colors that represent any numerical solution on the surface (Figure 20). But this is implemented only for postscript output – the screen colors are always fixed and have no overlaying shadows.

References

[1] T. Apel, G. Haase, A. Meyer, and M. Pester. Parallel solution of finite element equation systems: efficient inter-processor communication. Preprint SPC 95_5, TU Chemnitz-Zwickau, Februar 1995.
file://ftp.tu-chemnitz.de/pub/Local/mathematik/SPC/spc95_5.ps.Z 17, 19

[2] T. Apel and U. Reichel. SPC-PM Po 3D V 3.3, User's Manual. Preprint SFB393/99-06, TU Chemnitz, February 1999.
http://www.tu-chemnitz.de/sfb393/Files/PS/ssfb99-06.ps.gz

[3] Thomas Apel, Frank Milde, and Uwe Reichel. SPC-PM Po 3D v 4.0 - Programmers manual II. Preprint SFB393/99-37, TU Chemnitz, December 1999.
http://www.tu-chemnitz.de/sfb393/Files/PS/ssfb99-37.ps.gz 17, 19

[4] U. Groh. Lokale Realisierung von Vektoroperationen auf Parallelrechnern. Preprint SPC 94_2, TU Chemnitz-Zwickau, März 1994.
file://ftp.tu-chemnitz.de/pub/Local/mathematik/SPC/spc94_2.ps.gz 17

[5] M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen – Basisroutinen für Kommunikation und Grafik. Preprint SFB 02-01, TU Chemnitz, Januar 2002. *(updated from SPC 95_20)*.
http://www.tu-chemnitz.de/sfb393/Files/PDF/sfb02-01.pdf 38

[6] M. Meisel and A. Meyer. Kommunikationstechnologien beim parallelen vorkonditionierten Schur-Komplement CG-Verfahren. Preprint SPC 95_19, TU Chemnitz-Zwickau, Juni 1995.
file://ftp.tu-chemnitz.de/pub/Local/mathematik/SPC/spc95_19.ps.gz 17

[7] M. Meyer. Grafik-Ausgabe vom Parallelrechner für 3D-Gebiete. Preprint SPC 95_4, TU Chemnitz-Zwickau, Januar 1995.
file://ftp.tu-chemnitz.de/pub/Local/mathematik/SPC/spc95_4.ps.gz 1, 32

[8] A. Nye. Xlib programming manual for version X11. Technical report, O'Reilly & Associates, Inc., 1990. 2

[9] M. Pester. Grafik-Ausgabe vom Parallelrechner für 2D-Gebiete. Preprint SPC 94_24, TU Chemnitz-Zwickau, November 1994.
file://ftp.tu-chemnitz.de/pub/Local/mathematik/SPC/spc94_24.ps.gz 1

[10] M. Pester. On-line visualization in parallel computations. In W. Borchers, G. Domick, D. Kröner, R. Rautmann, and D. Saupe, editors, *Visualization Methods in High Performance Computing and Flow Simulation*, Proceedings of the International Workshop on Visualization, Paderborn, 18-21 January 1994, pages 91–98. VSP Utrecht and TEV Vilnius, 1996. 1, 32

[11] R. Ruhmer. 3D-Finite-Elemente-Rechnungen und ihre Visualisierung mit Hilfe von OpenGL. Diplomarbeit, TU Chemnitz, 2000. 32

[12] Andreas Wierse and Martin Rumpf. GRAPE Eine objektorientierte Visualisierungs- und Numerikplattform. *Informatik Forsch. Entw.*, 7:145–151, 1992. 1, 32

[13] Berkeley Multimedia Research Center. MPEG Tools. `http://www-vis.lbl.gov/multimedia/mpeg/`. 40

[14] E. Kohler. Gifsicle: Animated GIFs for UNIX. `http://www.lcdf.org/~ediietwo/gifsicle/`

[15] The Labs: GIFMerge - Merge GIFs to a GIF animation. `http://www.the-labs.com/GIFMerge/` 40

[16] ImageMagick - Convert, Edit and Compose Images. `http://www.imagemagick.org`. 40

[17] NAG. IRIS Explorer Documentation. `http://www.nag.co.uk/visual/IE/iecbb/DOC/Index.html`. 1, 27, 31

Appendix: Socket Interface for IRIS Explorer

This is a short description of the internal protocol used for socket communication between `out3dexpl` and the interface modules `PFEMread` and `PFEMdata` (Section 3.3.3). Messages are sent as packages of varying size either as ASCII strings or in binary mode. One package is organized in this way:

$$\boxed{\boxed{\textit{length}} : \boxed{\qquad\qquad\qquad \textit{data} \qquad\qquad\qquad}}$$

The data part may consist of multiple logical partitions separated by a delimiter. After establishing the connection by a client the server accepts any request message. Requests may have the following data format:

ID	request arguments	description of the request
`0[B]`		send surface polygons
`1[B]`		send all face polygons
`2[B]`		send volume elements
`3`		go on, finish server mode
`4`	*`s e`*	set shrink and explode parameters ($0 < s \leq 1,\ 1 \leq e < 2$) for subdomains
`6`		send names for each component of the solution (to be placed in the pull down menu of the module `PFEMdata`)
`9[B]`	*`n`*	send data vector for the n-th component of the solution (where $n = 0$ means: all components)

The optional extension `'B'` in the request code indicates to open an extra binary socket for data response which is much faster than ASCII conversion and transfer.

The server responds to those requests sending the following data (the character '`#`' is used for ASCII streams as a delimiter between certain blocks of information):

Req.ID	**response data**	**description of response data**
0, 1	`PFEMread 1.2` *`filename levels`* `#`	identification string and description
	`nproc` `#`	number of processors of the parallel machine where PFEM runs
	`2 #`	type information (as in 3.3.3)
	$n_{pol}\ n_{nod}\ n_{dof}$ `#`	number of polygons, nodes, and degrees of freedom per node
	$i\ n\ k_1 \ldots k_n$ `#`	polygons defined by node numbers,
	...	for $i = 1, \ldots, n_{pol}$; (restricted to $n = 3$ or $n = 4$)
	$j\ x_j\ y_j\ z_j$ `#`	coordinates of node j,
	...	for $j = 1, \ldots, n_{nod}$

Req.ID	**response data**	**description of response data**
2	`PFEMread 1.2` *`filename levels`* `#`	identification string and description
	`nproc` `#`	number of processors of the parallel machine where PFEM runs
	`type` `#`	type information (3 for tetrahedrons, 4 for hexahedrons)
	n_{vol} n_{nod} n_{dof} `#`	number of volumes, nodes, and degrees of freedom per node
	i $k_1 \ldots k_n$ `#`	volumes defined by node numbers,
	`...`	for $i = 1, \ldots, n_{vol}$, where *`type`*$= 3$ implies $n = 4$ and *`type`*$= 4$ implies $n = 8$
	j x_j y_j z_j `#`	coordinates of node j,
	`...`	for $j = 1, \ldots, n_{nod}$
3	`%% EOF %%`	acknoledgement for closing connection
4		no response, modification will affect next data transmission for request 0, 1 or 2
6	n_{dof} `#`	number of components per node in the solution
	*`string`*_1 `#`	menu name for the first component
	`...`	(max. 6 characters each)
	*`string`*_n_{dof} `#`	
9	(for 1 ≤n≤ n_{dof})	(n = argument of the request string)
	`PFEMread 1.3 #`	identification string (for `PFEMdata`)
	k u_{kn} `#`	n-th component,
	`...`	for $k = 1, \ldots, n_{nod}$
9	(for $n = 0$)	
	`PFEMread 1.3 #`	identification string (for `PFEMdata`)
	k u_{k1} $\ldots$ $u_{kn_{dof}}$ `#`	all components node by node
	`...`	for $k = 1, \ldots, n_{nod}$
9	(for $n < 0$ or $n > n_{dof}$)	
	`PFEMread 1.3 #`	only acknoledgement, no data

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.