More about OpenDX, in case you are interested

OpenDX, `www.opendx.org`, is an open version of IBM's "Visualization Data Explorer". A similar, but not open, system is AVS (Advanced Visual Systems, `www.avs.com`). ParaView, see the assignments, share many of the ideas, as well.
Not quite: "If you have seen one, you have seen them all".

OpenDX is exclusively for visualization (no computational part as in Matlab. Very simple calculations are OK.).

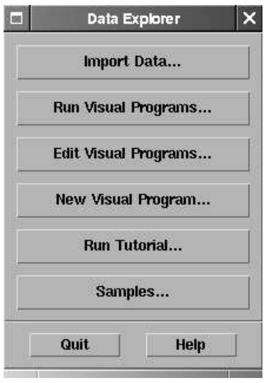
Some, but not all, important points:

- Advanced tools for visualization of 3D-data.
- Takes longer to learn than Matlab, but you can do more. Often faster.
- Modules are connected using a GUI, graphical programming. Visual Program Editor, VPE.
- Input from files (not variables as in Matlab).
- The modules transform the input and sends it to the next module.
- Supports several input formats. Using the "Data Prompter"-program simple inputs can be handled (e.g. uniform, gridded input).
- Lots of documentation. Many demo programs. Few simple examples. Should read a book (or take this course :-)
  David Thompson, Jeff Braun, Ray Ford,
  OpenDX: Paths to Visualization. Consists of a sequence of solved visualization problems.
  `http://www.vizsolutions.com`.

Here comes a brief presentation of OpenDX. For more details see the "QuickStart Guide" (see the course page).

Starting OpenDX (I assume you have the correct path; see the assignment).

```
% dx
```



"Import Data" starts "Data Prompter".
"Run Visual Programs" runs an existing program.
"Edit Visual Programs" starts the VPE with an existing program.
"New Visual Program" starts VPE with an empty work area.
"Run Tutorial" does exactly that.
"Samples" runs demo programs.

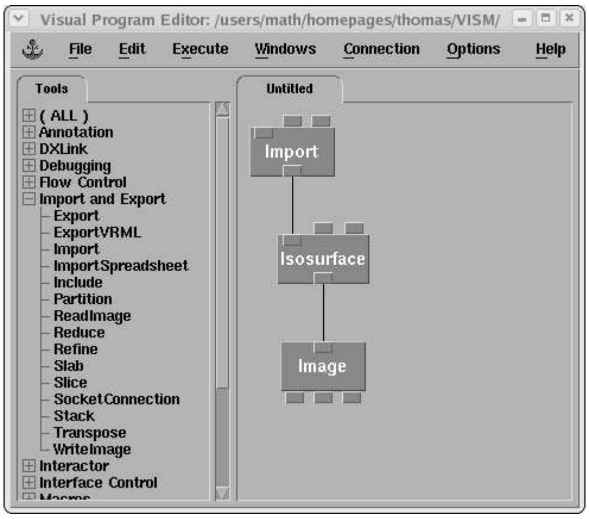
A simple first example. Level curves in 2D

Suppose we would like to copy the following Matlab code:

```
>> [X, Y] = meshgrid(-10:10, -10:10);
>> contour(X, Y, X.^2 + 2 * Y.^2)
```

Choose "New Visual Program".
The first version consists of three modules:

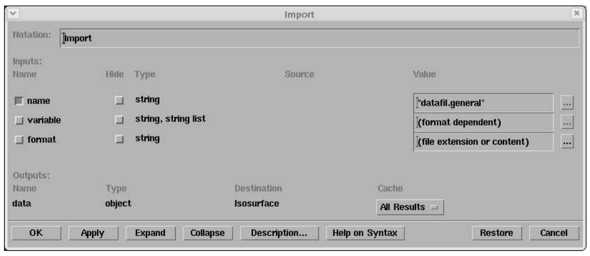
- Import, reads the datafile. The module passes the input on to the next module. The import-module has a input parameter that defines the filename.
- Isosurface takes the input and creates the level curves. There are input parameters to set the number of curves or the levels.
- Image creates a window and draws the image.



We click on Import in the list in the left part of the window and then we click on the canvas (drawing area) to place the icon of the Import module. We can connect tabs on one icon with tabs on another module using the left button mouse (press and drag).

When the Import-icon is placed on the drawing area all the three input tabs are standing up and the leftmost has a different colour; showing that it is a required input (the file name). The other parameters do not have to be specified, they are optional or have default values.

In my program I have defined the file name and the tab is down, indicating that the corresponding input has been defined. If we connected to the wrong tab we can remove the connection by pressing the tab on the receiving end and moving (button still down) the mouse to the canvas releasing the button. If one double-clicks the icon one can inspect and set the values.



Notation: Import

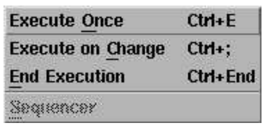
Inputs:

Name	Hide	Type	Source	Value
name		string		"datafil.general"
variable		string, string list		(format dependent)
format		string		(file extension or content)

Outputs:

Name	Type	Destination	Cache
data	object	Isosurface	All Results

To execute the program we choose one of the alternatives from VPE's execute-menu. There are "Execute Once", "Execute on Change" (if we change the levels in Isosurface, for example). There is also "End Execution".



We save the program giving a suffix `.net`. A `.cfg`-file is created as well. Executing once will produce an image window with one level curve. In a real application we would have to fix the input-data before executing, so let us look at how to produce the input file.

In this first example the x- and y-values make up a regular grid and it is not necessary to supply OpenDX with all the coordinates (positions, using OpenDX terminology). It is enough to give starting values, step sizes and the number of values (grid size). So only the z-values (data, using OpenDX terminology) have to be stored. Suppose we do it like this in Matlab:

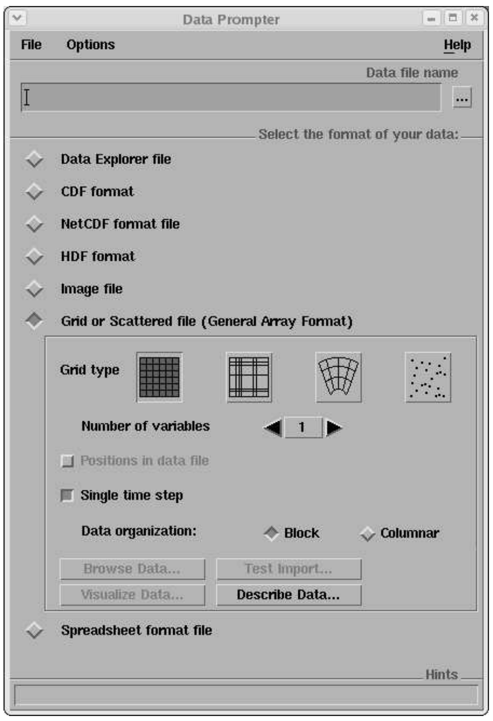
```
>> [X, Y] = meshgrid(-10:10, -10:10);
>> Z = X.^2 + 2 * Y.^2;
>> z = Z(:);
>> save -ascii contour_example.data z
```

So the file consists of one long array consisting of the columns in `Z` in a sequence.
To see how the values in the file correspond to the coordinates, we look at a much smaller example:

```
>> [X, Y] = meshgrid(-1:1, -1:1)

X =  -1     0     1
     -1     0     1
     -1     0     1

Y =  -1    -1    -1
      0     0     0
      1     1     1

>> [X(:), Y(:)]

ans =
    -1    -1  % (x_min,           y_min)
    -1     0  % (x_min,           y_min +   dy)
    -1     1  % (x_min,           y_min + 2 dy)
     0    -1  % (x_min +   dx,    y_min)
     0     0  % (x_min +   dx,    y_min +   dy)
     0     1  % (x_min +   dx,    y_min + 2 dy)
     1    -1  % (x_min + 2 dx,    y_min)
     1     0  % (x_min + 2 dx,    y_min +   dy)
     1     1  % (x_min + 2 dx,    y_min + 2 dy)
```

We now press the “Import data”-button to start the “Data Prompter”. Using this program we can describe the layout of the data and store this information on a header-file having the suffix `.general`.

After having clicked in “Grid or Scattered file (General Array Format)”, the window looks like:



Grid type is regular (the leftmost grid-icon).
Press the “Describe Data...”-button.
Another, big, window appears. I have not included a picture of the window.

In the left part of the new window I give the name of the data-file `contour_example.data`. The file has no header (comments in the beginning) so I can skip the next line.

I enter the grid size, 21 × 21. The data format is ASCII (not binary). The data order is “Row”, because when we stored the z-values the row-index (y-direction) changed faster than the column-index.

It is often necessary to hit return after having typed a value (true in the modules as well).

In the bottom part of the window we set the origin for x and y and the increments (delta in the window). So `-10,1` for both grid positions.
So, by now we have said that the z-values (f(x,y)) are stored in the following order:

```
f(-10,  -10)
f(-10,   -9)
 ...
f(-10,   10)
f( -9,  -10)
 ...
```

If we, by mistake, say that we store data by “Column”, OpenDX will interpret our data as:

```
f(-10,  -10)
f( -9,  -10)
 ...
f(10,   -10)
f(-10,   -9)
 ...
```

In the righthand part of the window, we could adjust the data type (float) and structure of data (scalar, vector etc.). In this example no adjustment is necessary.

Finally we save the file as `contour_example.general`. It looks like this:

```
file = contour_example.data
grid = 21 x 21
format = ascii
interleaving = record
majority = row
field = field0
structure = scalar
type = float
dependency = positions
positions = regular, regular, -10, 1, -10, 1

end
```

We recognize some, but not all, keywords.
Once we have stored the file we can do a test visualization of our data (without having written any program) just by pressing "Visualize Data" in the first data prompter window. This will give as an image with coloured level curves. Pressing "Browse Data" a window, with the contents of the data file, is opened.
After having set the filename in the Import-module (**name** is set to **contour_example.general**). It is time to run our program.

It produces a yellow level curve on a black window. Using the Options-menu (for the Image window) one can switch on zoom, rotation (for 3D) etc. The default level is the arithmetic mean of the data values. By double-clicking on the Isosurface module and changing the number-parameter, we get more curves.

How can one find out what the tabs stand for? How do I know that I can change the number or levels?

One way is to take the program and choose "Context-Sensitive Help" from the Help-menu. If one places the questionmark-cursor on the Isosurface icon a help window appears. Another way is to read User's Reference (in PDF). See "Chapter 2. Functional Modules". Here is an edited version of the text.

Category: Realization
Function: Computes isosurfaces and contours.

Syntax (one can use a script language as well)

```
  surface = Isosurface(data, value, number,
                       gradient, flag, direction)
```

Inputs

Name	Type	Default	Description
data	scalar field	none	field from which one or more surfaces are to be derived.
value	scalar or scalar list	data mean	isosurface value or values
number	integer	no default	number of isosurfaces or contours to be computed
gradient	vector field	no default	gradient field
flag	flag	1	0: normals not computed, 1: normals computed
direction	integer	-1	orientation of normals

The first three correspond to the tabs. The last three can be seen if one clicks "Expand" after having double-clicked on Isosurface. One can add tabs to all the inputs (see the Edit-menu in VPE).

Outputs

Name	Type	Description
surface	field or group	isosurface

Functional Details
This module computes any of the following:

- points (for an input field consisting of lines)
- lines (for a surface input field)
- surfaces (for a volumetric input field).

All positions in the output field are isovalues (i.e., they match a specified value or values).

The module also adds a default color to the output (gray-blue for isosurfaces and yellow for contour lines and points) if the input object is uncolored. If the object is colored, its colors are interpolated in the output object.

A "data" component with the same value as the input value is added to the output field.

data is the data object for which an isosurface or contour is to be created.

value is the isovalue or isovalues to be used for computing the isosurface(s) or contour(s).

If this parameter is not specified, the module bases it calculations on the value specified by **number** (see below). If neither parameter is specified, the module uses the arithmetic mean of the data input as a default.

number is ignored if **value** has been specified. If that parameter is not specified, the module uses the value of **number** to compute a set of isosurfaces or contours with the following isovalues:

```
min + delta, min + (2*delta),..., min - delta
```

where delta = (max - min)/(number + 1), and "max" and "min" are the maximum and minimum data values of the input field.

gradient is the gradient field used to compute normals for shading (see Gradient).
If this parameter is not specified, the module adds normals by computing the gradient internally (**flag** can nullify this behavior; see below).
Note: If only one isosurface is to be computed, it is probably more efficient to have module compute the gradient internally. If many are to be generated, it is probably more efficient to compute the gradient of the entire field once, so that the system can use it for every isosurface.

flag specifies whether normals are to be computed for shading. A setting of 0 (zero) prevents the computation of normals. The default is 1 (one)

direction specifies whether the normals should point against (0, the default) or with (1) the gradient.

Notes:

1. This module adds an attribute called “Isosurface value,” which has as its value the isovalue(s) used. To extract this attribute (e.g., for use in a caption for an image), use the Attribute module.
2. For contour lines, this module adds a “fuzz” attribute so that the line will be rendered slightly in front of a coincident surface (see Display).
3. A surface or contour is considered to be undefined if every point in the input volume or surface, respectively, is equal to value. In such cases, the module output is an empty field.
4. Isosurface does not accept connection-dependent data.
5. With disjoint data fields, there may be no data crossings (i.e., points along a connection element where the interpolated data value equals the isovalue), even though the isovalue itself falls in the range of the actual data.

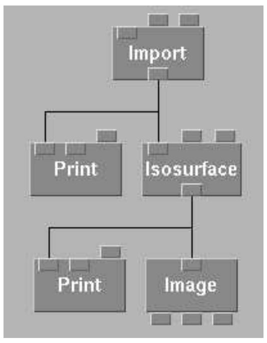
Components

Creates new “positions” and “connections” components. For surfaces output, the default is to create a “normals” component. Any component dependent on “positions” is interpolated and placed in the output object.

Example Visual Programs
Many example visual programs use the Isosurface module, including:
AlternateVisualizations.net ContoursAndCaption.net InvalidData.net MappedIso.net Sealevel.net UsingIsosurface.net SIMPLE/Isosu

See Also
Band, Color, Gradient, Map, SimplifySurface

We can try to understand this manual page if we add Print-modules to our program.



The printout comes in a separate Message Window. By setting the options-parameter in Print to d (in this case) everything will be printed for our small example (3 × 3-matrices).

First some definitions: Data is stored in a hierarchy of objects. The top level is the group object which can contain other groups and so-called fields. A field object is the basic object in OpenDX. A field stores component objects. A component is typically an array object with an associated name. An array object is an array having elements of a given datatype. We refer to the elements using an index.

An attribute gives a connection between an object and a value. Attributes are used to store metadata (data about the data).

In our simple example data will consist of some arrays stored in a field. The first Print produces (after some editing).
`%` are my comments.

```
Begin Execution
Field.  5 components.

Component number 0, name 'data':
 Generic Array.  9 items, float, real, scalar
 data values:              % The z-values
         3         1         3         2         0
         2         3         1         3
 Attribute.  Name 'dep':  % data depends
  String.  "positions"    % on the positions (x, y)

Component number 1, name 'positions':  % (x, y)-values
 Product Array.  2 terms.
 Product term 0: Regular Array.
                 3 items, float, real, 2-vector
  start value [-1, 0], delta [1, 0], for 3 repetitions
 Product term 1: Regular Array.
                 3 items, float, real, 2-vector
  start value [0, -1], delta [0, 1], for 3 repetitions
 Attribute.  Name 'dep':
  String.  "positions"

% connections is new to us. It describes how data
% is connected. Used for interpolation.
Component number 2, name 'connections':
 Mesh Array.  2 terms.
 Mesh offset: 0, 0
 Mesh term 0:   Path Array.  connects 3 items
 Mesh term 1:   Path Array.  connects 3 items
 Attribute.  Name 'element type': % what kind of
  String.  "quads"  % interpolation that has been used
 Attribute.  Name 'dep':
  String.  "connections"
 Attribute.  Name 'ref': % refers to
  String.  "positions"
```

```
% Min and max
Component number 3, name 'box':
 Generic Array.  4 items, float, real, 2-vector
 data values:
            -1             -1
            -1              1
             1             -1
             1              1

 Attribute.  Name 'der':  % derived from
  String.  "positions"    % positions

Component number 4, name 'data statistics':
 Generic Array.  8 items, double, real, scalar
 data values:
      0               3                  2
      0              18                 46
      9               1
 Attribute.  Name 'der':
  String.  "data"
Attribute.  Name 'name':
 String.  "field0"
```

The second Print produces quite different output:

```
Begin Execution
Field.  5 components.

% Coordinates along the level curves
Component number 0, name 'positions':
 Generic Array.  6 items, float, real, 2-vector
 data values:
            -1          -0.5
             0            -1
            -1           0.5
             0             1
             1          -0.5
```

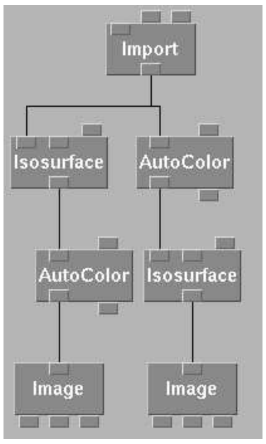
```
              1            0.5

 Attribute.  Name 'dep':
  String.  "positions"

Component number 1, name 'colors':
 Constant Array.  6 items, float, real, 3-vector
 constant value [ 0.7, 0.7, 0 ]   % Yellow
 Attribute.  Name 'dep':
  String.  "positions"

Component number 2, name 'data':
 Constant Array.  6 items, float, real, scalar
 constant value 2            % mean of z-values
 Attribute.  Name 'dep':
  String.  "positions"

Component number 3, name 'connections':
 Generic Array.  4 items, integer, real, 2-vector
 data values:
         0         1
         2         3
         1         4
         3         5

 Attribute.  Name 'ref':
  String.  "positions"
 Attribute.  Name 'element type':
  String.  "lines"
 Attribute.  Name 'dep':
  String.  "connections"

Component number 4, name 'box':
 Generic Array.  4 items, float, real, 2-vector
 data values:
             -1              -1
             -1               1
```

```
              1             -1
              1              1

 Attribute.  Name 'der':
  String.  "positions"
Attribute.  Name 'fuzz':
 Generic Array.  1 item, float, real, scalar
 data values:
              6
Attribute.  Name 'shade':
 Generic Array.  1 item, integer, real, scalar
 data values:
          0
Attribute.  Name 'name':
 String.  "field0"
Attribute.  Name 'Isosurface value':
 Generic Array.  1 item, float, real, scalar
 data values:
              2
Attribute.  Name 'series position':
 Generic Array.  1 item, float, real, scalar
 data values:
              2
```

So positions has changed from the grid points to the points on the level curves. Data has changed from z-values to one z-level value. Colour information has been added.
If we require three level curves we get a group containing three fields, one curve in each.

Let us write some more programs.

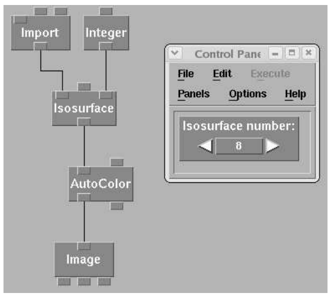
To change the colour of the curves we add an AutoColor-module. This can be done in two ways. The following program produces two windows (two Image-modules) both having coloured level curves (of varying colour). Note that Import can feed several modules.



In the left part of the program the Isosurface data is fed to AutoColor which (like Matlab) maps the different values onto the colormap. In the right part the original data is coloured and a subset of data is extracted by the Isosurface module.
So if we have levels that are close to each other the two images will have rather different colours. The left will contain the full spectrum (with a suitable colormap) while the right may have lines of almost the same colour.

Sometimes modules commute, but that is not always the case.
It is somewhat inconvenient to change the parameters in the Isosurface module. A novice to OpenDX may use your program so you should include an easier way to change the parameters.

We add a so-called interactor. The interactor is manipulated in a separate window, a control panel. We place an Integer interactor in the programs. The module is connected to the number-parameter of the Isosurface.
Here is the program, together with the control panel window:



Several types of interactors exist, and they can be put in the same window. It is also possible to make this window look less crude by gluing buttons together, adding text, separators etc.

If we set the number of curves to zero (or less) we get an error message. To avoid this we can use the “Set Attributes...” alternative from the menu in the panel window. We set min to be 1 and max to be 20 (do not forget to hit return). One can also change the increment (how much a click on one of the arrows should change the value).
By selecting “Execute on Change”, the image will be updated for each click.

Let us now draw a surface $z = f(x, y)$. In Matlab we would use **`mesh`**.
There are several ways to do this. One easy way is to use the RubberSheet-module. It uses the data part as z-values and the positions as x- and y-coordinates.

The second input to the module is a magnification factor (to in- or decrease the z-values). Negative values are allowed. I have added a Scalar (i.e. floating point) interactor to this tab.

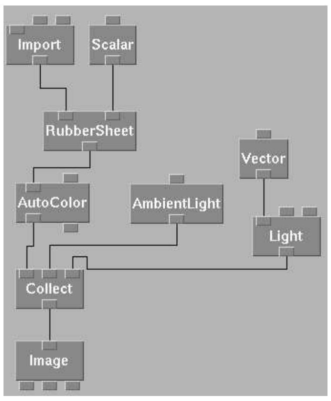
We would like some light, so I have added a Light-module and a Vector-interactor to control the direction to the light source.

We are feeding two different inputs to Image. Since Image only has one input tab we collect the inputs using a Collect-module. The Collect-module comes with two inputs, but one can add tabs using Edit and then “Input/Output Tabs”.

To get one panel window, we move the Vector interactor by marking it with the middle mouse button, moving it to the other panel window and releasing the button. Remove the interactor from the first panel window (Edit/Delete).

To get axes we choose “AutoAxes...” from the Options-menu in the image window. Click in “AutoAxes enabled” and add whatever attributes you would like to have.

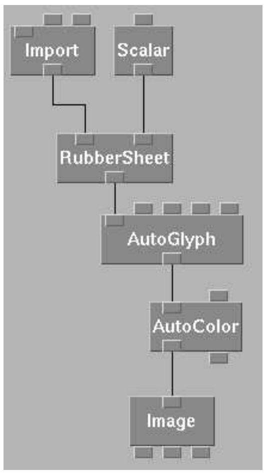
To move the surface use “Options/View Control...” or “Options/Mode”. Try the different mouse-buttons.
“Options/Reset” (or CTRL+F) resets the image to its original position.
Here comes the program where also AmbientLight
has been added.

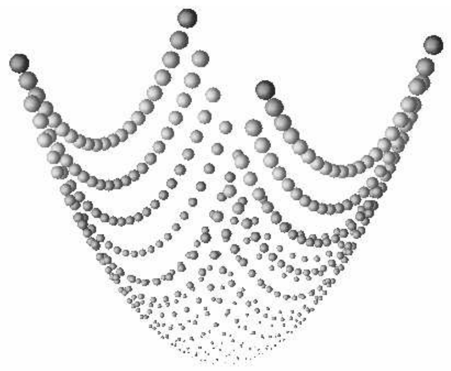


We can get an amusing picture by drawing glyphs in each point on the surface. (Glyph from from Greek Glyphe, carved work, from
glyphein to carve.
1: an ornamental vertical groove especially in a Doric frieze
2: a symbolic figure or a character (as in the Mayan system of writing) usually incised or carved in relief
3: a symbol (as a curved arrow on a road sign) that conveys information nonverbally).

Glyphs in computer graphics are usually small symbols, such as circles, squares in 2D, and spheres, cubes in 3D. It is easily to draw such a surface, where the glyphs vary in colour and size depending on the value of z.

We can add glyphs by using the AutoGlyph-module. This module has several parameters (type, size, quality of the glyph etc.).
The image should be seen in colour on the monitor.





Let us now try to visualize data produced by $w = f(x, y, z)$. We start with som simple data produced in Matlab. The reason I am using **`ndgrid`** and not **`meshgrid`** will be explained below.

```
>> [X, Y, Z] = ndgrid(0:0.1:1, 10:2:40, -1:0.1:1);
>> W = X.^2 + (0.05 * (Y - 10)).^2 + Z.^2;
>> w = W(:);
>> save -ascii wdata w
```

To understand how the values are stored in the file we look at a much smaller example.

```
>> [X, Y, Z] = ndgrid(0.1:0.1:0.3, -1:1, 20:10:40)

X(:,:,1) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000
X(:,:,2) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000
X(:,:,3) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000

Y(:,:,1) =
    -1     0     1
    -1     0     1
    -1     0     1
Y(:,:,2) =
    -1     0     1
    -1     0     1
    -1     0     1
Y(:,:,3) =
    -1     0     1
    -1     0     1
    -1     0     1
```

```
Z(:,:,1) =
    20    20    20
    20    20    20
    20    20    20
Z(:,:,2) =
    30    30    30
    30    30    30
    30    30    30
Z(:,:,3) =
    40    40    40
    40    40    40
    40    40    40

>> [X(:), Y(:), Z(:)] % I have added blank lines
ans =
    0.1000   -1.0000   20.0000  % (x1, y1, z1)
    0.2000   -1.0000   20.0000  % (x2, y1, z1)
    0.3000   -1.0000   20.0000  % (x3, y1, z1)

    0.1000         0   20.0000  % (x1, y2, z1)
    0.2000         0   20.0000  % (x2, y2, z1)
    0.3000         0   20.0000  % (x3, y2, z1)

    0.1000    1.0000   20.0000  % (x1, y3, z1)
    0.2000    1.0000   20.0000  % (x2, y3, z1)
    0.3000    1.0000   20.0000  % (x3, y3, z1)

    0.1000   -1.0000   30.0000  % (x1, y1, z2)
    0.2000   -1.0000   30.0000  % etc.
    0.3000   -1.0000   30.0000

    0.1000         0   30.0000
    0.2000         0   30.0000
    0.3000         0   30.0000

    0.1000    1.0000   30.0000
```

```
    0.2000    1.0000   30.0000
    0.3000    1.0000   30.0000

    0.1000   -1.0000   40.0000
    0.2000   -1.0000   40.0000
    0.3000   -1.0000   40.0000

    0.1000         0   40.0000
    0.2000         0   40.0000
    0.3000         0   40.0000

    0.1000    1.0000   40.0000
    0.2000    1.0000   40.0000
    0.3000    1.0000   40.0000
```

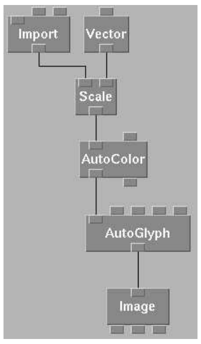
So we get 3D-matrices and we see that when W is computed, x varies faster than y which changes faster than z. Had I used **meshgrid** the order would have been y, x, z, which is less regular.

We use the Data Prompter as before, to create a header file, **wdata.general**. The grid is 11 × 16 × 21. The data order is column, which means that the last grid position, the column i.e. the x-values, in the 2D-case changes fastest. Row means that the first grid position, the row i.e. the y-values in the 2D-case, changes fastest. So row and column have to be generalized to the 3D-case. This might be confusing, but think of the 2D-case. Suppose that we have 9 w-values presented in two ways (the numbers give the order in the input file):

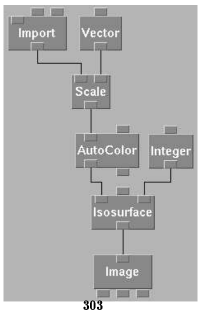
```
  7 8 9     3 6 9
  4 5 6     2 5 8
  1 2 3     1 4 7
 column     row
```

In the left case the column (x) changes faster and in the right case, the row (y) changes faster. We see that **meshgrid** does not fit any of these in the 3D-case.

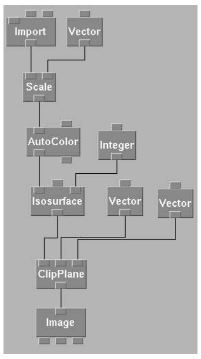
Origins and deltas are, (0,0.1), (10,2) and (-1,0.1) respectively. As before we can make a test visualization of data, but it is not easy to interpret. One reason is that the y-values have a different scale, so a coordinate system with equal scaling will be very drawn out. So, let us write a program.



Scale scales the positions with a scaling matrix. I give the diagonal by using a Vector-module. This way I can scale the y-component to a reasonable size. In each (x,y,z)-point there will be a coloured sphere whos size and colour reflects the corresponding w-value.
Isosurfaces work well with this data (we get parts of ellipsoids).

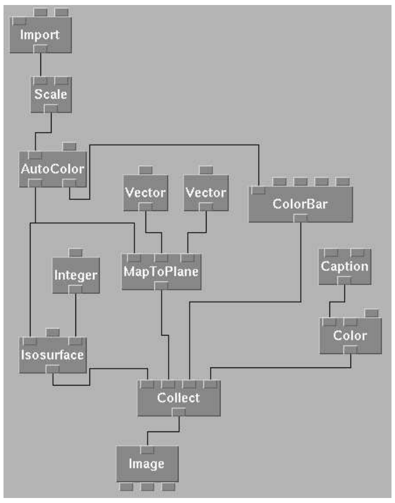


It is possible to remove part of the data (everything on one side of a plane). We use the module ClipPlane. It takes the data, a point in the plane and a normal defining the clip plane. I have used Vector interactors to set these parameters. Everything on the side of the plane (in the direction of the normal) is removed. As I have made my program, the point is given in the scaled coordinates.



Here is a related construct. The MapToPlane-module creates an arbitrary cutting plane through 3D-space and interpolates data values onto it. The plane is defined by a point a normal, just as the ClipPlane. Using the Vector interactors we can move to plane.
I have combined MapToPlane with Isosurface. I have also added Colorbar which draws a scale (as in Matlab). Finally there is Caption which corresponds to Matlab's title. In the image-window (under Options) I set the background colour to white (suitable when including the image in this handout). This made it necessary to include a Color-module to change the colour of the title from white to black.

I had the same problem in ColorBar, but I fixed it in a different way. I double-clicked on ColorBar, and then I had to click on Expand, to see all parameters. I set colors to black and annotation to labels (this caused the labels to be black). In a similar way I changed the positions in Caption and ColorBar.
Here is the program



and here is a (bad) version of the resulting image.

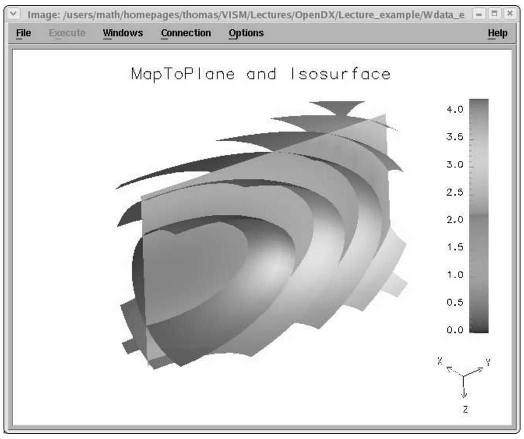


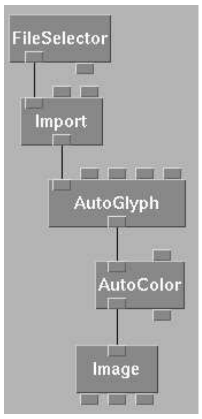
Image: /users/math/homepages/thomas/VISM/Lectures/OpenDX/Lecture_example/Wdata_e
File
Execute
Windows
Connection
Options
Help
MapToPlane and Isosurface
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
X
Y
Z

Vector fields

Suppose we have a grid in 2D (or a 3D) and we have a vector in every grid point. OpenDX can be used to visualize such data in several ways.
Let us start with a 2D-example. We make a datafile containing 2D-vectors. Each grid point has one vector.

```
n = 14;
[X, Y] = ndgrid(linspace(0, n, n+1));
V = [-Y(:), X(:)];
save -ascii Vdata V
```

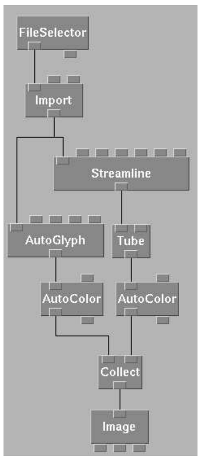
We can import the data almost as usual. The Vector interleaving should be $x_0y_0, x_1y_1, \ldots, x_ny_n$ and not $x_1x_2, \ldots, x_n, y_0, y_1, \ldots, y_n$.
We have to change the Structure from scalar to 2-vector after which we have to press the Modify button.
The following simple program imports the data. Using the FileSelector module we can browse our files and pick the one we want.
AutoGlyph changes the glyph to arrows, to fit the new datatype.



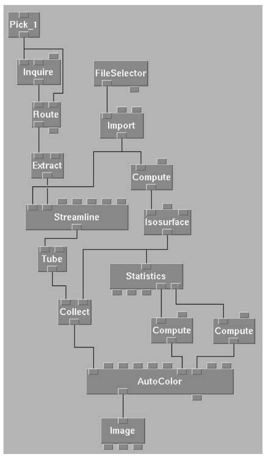
The image contains arrows which are tangents to circles centered at the origin. The lengths of the arrows increase with the radius and the colour increases from blue to red. Length and colour depend on the norm of a vector.
Here is a simple 3D-example. What does the vector field look like?

```
n = 9;
[X, Y, Z] = ndgrid(linspace(0, n, n+1));
x = X(:);   y = Y(:);   z = Z(:);
V = [-y .* z, x .* z, 0 * z];
save -ascii Vdata3D V
```

It is easy to import the data; the Structure is 3-vector. We can actually use the same program to visualize the 3D-field.
AutoGlyph is a smart module which adapts to the input.
Let us draw some streamlines.



This program draws arrows and streamlines. To see the streamlines better I have used the Tube-module which makes a tube of the streamlines (but Tube is not necessary). It is possible to use the Ribbon-module instead of Tube. The ribbons can twist, showing the curl of the field. One specifies the starting point of the streamline(s) by supplying a vector list.
In the following program we use the mouse and the Pick-module to specify the starting position for a tube. The program also illustrated one way to use an “if-statement”.



The program draws a set of isosurfaces; the first Compute-module computes the norm of the vectors (Isosurface requires scalars, not vectors). One could use the expression
`sqrt(a.x*a.x+a.y*a.y+a.z*a.z)`, but **`mag(a)`** is a shorter alternative. There is also a function, norm, but it normalizes a vector.

If I click somewhere on one isosurface I get the coordinates, in 3D, for the click. If I miss the surface I do not get any coordinates. This would break the program, which is the reason an if-construct has to be used.

In the Pick_1-module I have set the first-parameter to 1, indicating that I only want the first pick (click) (one can get a vector of coordinates sometimes). I have set the persistent-parameter to 0; I do not want to save the clicks.

In the Inquire-module I have set the inquiry-parameter to “is not empty”, which sends 1 to Route if we got coordinates. If we missed the surface Inquire sends a zero.

If the Route-module gets 0 none of the modules, that use the output from Route, are executed. If Route gets 1 it will pass on the input to the first tab (if the leftmost one is number zero).

So it will send the coordinates to the Extract-module which extracts the positions-part of the input. This is sent to the start-parameter of Streamline. I have changed the diameter of the Tube (that is the reason the second tab is down).

The Statistics-module is used to get a consistent colouring of surfaces and tubes. I used the min- and max-outputs from Statistics. The Compute-modules decrease min and increase max slightly to avoid some problems with the colour of the tubes. The new min/max-values are then fed to AutoColor.

In order for Pick to work I have to set the Mode to Pick (Options in the Image window).

Data formats

OpenDX supports several data formats:

1. The format supported by the General Array Importer. It usually consists of two files, one with suffix **`general`**. The **`general`**-files contains keywords and data defining the layout.
2. Data Explorer native file format. Suffix **`dx`**.
3. netCDF, Network Common Data Form.
   `http://my.unidata.ucar.edu/content/software/netcdf/index.html`
4. CDF, Common Data Format.
   `http://cdf.gsfc.nasa.gov/`
5. HDF, Hierarchical Data Format.
   `http://hdf.ncsa.uiuc.edu/`

We have used the first format so far. The second is more general. The last three, which is not part of OpenDX, consist of standardized, self-describing formats and library routines. These formats are used for data in general, and not only for computer graphics.

Let us now visualize data where the positions do not form a regular grid. The data format in OpenDX supports several types of grids. Not all forms are supported by the General Array Importer.

- Regular grid. What we have seen so far. In 2D, the coordinates are given by (x_j, y_k), $x_j = x_0 + j\delta_x, j = 0, \dots, m-1$ and $y_k = y_0 + k\delta_y, k = 0, \dots, n-1$. So the grid is determined by $x_0, y_0, \delta_x, \delta_y, m, n$.
- Deformed grids. The grid points are connected in a regular fashion (the relation between neighbours remain). Think of taking a regular grid and perturbing the x- and y-values somewhat. Another example is using polar coordinates, so $(r_j \cos\varphi_k, r_j \sin\varphi_k)$

- Irregular grid. Arbitrary (x_j, y_k)-values but with defined connections between the points.
  Think of a planar graph (in 2D).

OpenDX also makes a distinction between how a data-value is related to the coordinates. So far we have seen so-called position-dependent data, to each (x_j, y_k) ((x_i, y_j, z_k) in 3D) corresponds one datavalue.

In some situations it may be more natural to associate a value with an area (volume in 3D). A biologist may be interested in the distribution of a special plant in a forest. The number of occurrences per km^2 may be a good measure.
Using a regular grid we define a value for each rectangle, $(x_j, y_k), (x_{j+1}, y_k), (x_{j+1}, y_{k+1}), (x_j, y_{k+1})$.

This is called connection-dependent data. We may think of the data as defined for a point in the centre of each rectangle and/or having a constant value for every point in the rectangle.

Here comes sequence of examples. In the first we will produce polar coordinates using stuff we already know.

Let us produce some data in Matlab, as usual.

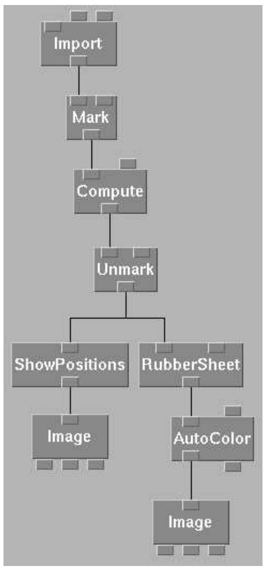
```
>> [R, PHI] = ndgrid(linspace(1, 2, 11),  ...
                     linspace(0, 1, 21));
>> X = R .* cos(PHI);
>> Y = R .* sin(PHI);
>> Z = Y .* (2 * X - 3 * Y);
>> z = Z(:);
>> save -ascii polar z
```

I have specified the positions as $(x_0, \delta_x) = (1, 0.1)$
and $(y_0, \delta_y) = (0, 0.05)$ and using a 10×20-grid.
This corresponds to **`R`** and **`PHI`** in the Matlab program.

The following program uses the Compute-module to compute the corresponding x and y. The Compute-module usually works on the data-part, so in order to form $x = r\cos\varphi$ and $y = r\sin\varphi$ I have to make the positions appear as data.

I have set the name-parameter in the module to positions. The Mark-module then moves the data component to the "saved data" component and copies the positions component to the data component.

Compute then performs `[a.x*cos(a.y), a.x*sin(a.y)]` after which Unmark moves the data back to positions (I have set the name parameter to positions). It also moves the saved data to data. **dx** names the input(s) **a** (and **b**).



ShowPositions plots the computed positions as dots in an x-y-diagram as a control).
Suppose now that the positions cannot be generated by (x_0, δ_x) and (y_0, δ_y) but that they are regular as in the following example:

```
x = [1 2 5 15],  y = [0 3 10 20]

All (x, y)-pairs

   ( 1, 0), ( 1, 3), ( 1, 10), ( 1, 20)
   ( 2, 0), ( 2, 3), ( 2, 10), ( 2, 20)
   ( 5, 0), ( 5, 3), ( 5, 10), ( 5, 20)
   (15, 0), (15, 3), (15, 10), (15, 20)
```

Using the dataprompter we click on the second Grid type-button. Note that the Data Prompter window, and the window that is opened, have different designs depending on which Grid type-button we choose.

Grid positions, choose Partially regular. Choose irregular for the positions list and type the values. We can write the values in the space to the right of the buttons. An alternative is to create a general-file like this:

```
file = testdata
grid = 4 x 4
format = ascii
interleaving = record
majority = column
field = field0
structure = scalar
type = float
dependency = positions
positions = irregular, irregular, 1, 2, 5, 15, 0, 3,
10, 20

end
```

Now to a warped regular grid: each position must be explicitly specified, but there is still a grid structure to the connections between data points. First some Matlab to make a data file.

```
>> x
x =
     1     2     5    15
>> y
y =
     0     3    10    20
>> [X, Y] = ndgrid(x, y);
>> V=[X(:)+0.1*rand(16,1), Y(:)+0.1*rand(16,1), (1:16)']
V =
   1.0050e+00   2.6768e-03   1.0000e+00
   2.0813e+00   1.1882e-02   2.0000e+00
   5.0464e+00   4.8813e-02   3.0000e+00
   1.5012e+01   1.8067e-02   4.0000e+00
   1.0354e+00   3.0942e+00   5.0000e+00
   2.0178e+00   3.0320e+00   6.0000e+00
   5.0062e+00   3.0741e+00   7.0000e+00
   1.5067e+01   3.0388e+00   8.0000e+00
   1.0368e+00   1.0080e+01   9.0000e+00
   2.0103e+00   1.0030e+01   1.0000e+01
   5.0615e+00   1.0046e+01   1.1000e+01
   1.5019e+01   1.0045e+01   1.2000e+01
   1.0192e+00   2.0036e+01   1.3000e+01
   2.0012e+00   2.0058e+01   1.4000e+01
   5.0289e+00   2.0048e+01   1.5000e+01
   1.5032e+01   2.0073e+01   1.6000e+01
>> save -ascii warptest V
```

I added the random numbers just to make this example different from the previous. Note that the rows can be sorted in any way, in this case.

Using the dataprompter we click on the third Grid type-button. Number of variables is one and Dimension (for positions in data file) is two (x and y).

Switch to the Describe Data-window. The Grid size is 4 × 4, the data order is not relevant in this case. To get more alternatives, choose "Full prompter" from the Options-menu. Field interleaving should be Columnar (we have a table with x, y, data. Block would mean, first all the coordinates and then the data, or vice-versa). In the right-hand part of the window we say that locations (the coordinates is a 2-vector of floats).
field0, the data, is a scalar float.

This is how the general-file looks.

```
file = warptest
grid = 4 x 4
format = ascii
interleaving = field
majority = column
field = locations, field0
structure = 2-vector, scalar
type = float, float
dependency = positions, positions

end
```

And finally Scattered data: there are no connections between data points. We continue with 16 points, where x and y are arbitrary. Click on the right-most Grid type and click in "Positions in data file". Two dimensions. Switch to the Describe Data-window and set # of Points to 16. Field interleaving should be Columnar.

This is how the general-file looks.

```
file = scattertest
points = 16
format = ascii
interleaving = field
field = locations, field0
structure = 2-vector, scalar
type = float, float

end
```

If we are trying to use the Isosurface-module we get the following error message:
ERROR: Isosurface: Invalid data: 'data' parameter is missing "connections"component.

To compute the level curves, the module must know how the points are related. We can fix that by first feeding the input through the AutoGrid-module which generates point on a regular grid.

Thinking about the biologist we may create connection-dependent data as well. Set Dependency to connections. Notice also, that if we have 16, say, data values it may correspond to a 5 × 5-grid (framing 16 rectangles).

To understand the details of the general-files, see:
The "Quickstart Guide", "5.3 Header File Syntax: Keyword Statements".
Chapter 5.4 deals with the Data Prompter.

Animation in OpenDX

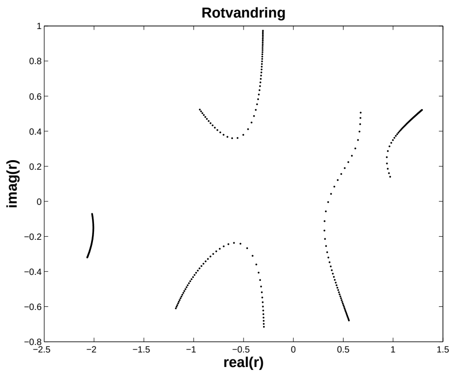
Here comes a very simple animation in 2D. The example is somewhat special in that we only have positions but no corresponding data.

We are going to study the roots, $r_1, \dots, r_p$, of the polynomial equation:

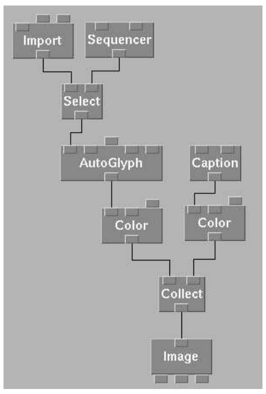
$$c_p x^p + c_{p-1} x^{p-1} + \cdots + c_1 x + c_0 = 0$$

The coefficients are complex numbers.

We perturb c_0 by a multiple of δ, so $c_0^{(k)} = c_0 + k\delta, k = 0, \dots, n-1$ and get the roots $r_1^{(k)}, \dots, r_p^{(k)}, k = 0, \dots, n-1$. We would like to animate this sequence, i.e. we would like to plot $r_1^{(k)}, \dots, r_p^{(k)}$ stepping in k (think of k as a time). This is what a static image may look like:



Here is a program:



The Sequencer-module works like a loop generating a sequence of integers (think k). The integer is passed on to the Select-module which extracts the plot-data for the k:th step. This plot-data is then fed through the rest of the program, generating an image.

The first run of the program takes more time, since the sequence of images is created and stored in a cache. After the first run, the speed is much better. When double-clicking on the Sequencer a window pops up containing VCR-like controls. Read the manual for details.

We must adjust the min and max of k. Do this by clicking once on the Sequencer-module and then choosing Configuration... from the Edit-menu in VPE. We set min to 0 and max to 49 (in this example I have 50 animation steps).

Since I have not supplied any data-part, I cannot use AutoColor, so the roots are plotted like red dots. If I number the roots and add that number as a data-part I can easily have different colours.

Now to the input. The degree of the polynomial is five and we have 50 time steps. I used Matlab to create a file, `data_roots`, containing

$\mathrm{re}(r_1^{(0)})$	$\mathrm{im}(r_1^{(0)})$
$\mathrm{re}(r_2^{(0)})$	$\mathrm{im}(r_2^{(0)})$
$\mathrm{re}(r_3^{(0)})$	$\mathrm{im}(r_3^{(0)})$
$\mathrm{re}(r_4^{(0)})$	$\mathrm{im}(r_4^{(0)})$
$\mathrm{re}(r_5^{(0)})$	$\mathrm{im}(r_5^{(0)})$
$\mathrm{re}(r_1^{(1)})$	$\mathrm{im}(r_1^{(1)})$
$\mathrm{re}(r_2^{(1)})$	$\mathrm{im}(r_2^{(1)})$
$\mathrm{re}(r_3^{(1)})$	$\mathrm{im}(r_3^{(1)})$
$\mathrm{re}(r_4^{(1)})$	$\mathrm{im}(r_4^{(1)})$
$\mathrm{re}(r_5^{(1)})$	$\mathrm{im}(r_5^{(1)})$
$\mathrm{re}(r_1^{(2)})$	$\mathrm{im}(r_1^{(2)})$

etc.

I created the general-file using an editor in this case (it was easier than using the Data Prompter).

```
file = data_roots
points = 5
series = 50
field = locations
structure = 2-vector
interleaving = field
```

If I had wanted to number the roots, `field` and `structure` can be changed to:

```
field = n_root, locations
structure = scalar, 2-vector
```

- `data_roots` contains the plot-data.
- `points` is the number of positions in each time step.
- The `series`-keyword is used to specify the number of time steps.
- `field` specifies the name and number of individual fields in a data file. We have only a positions-part, and the keyword locations informs OpenDx about this fact. In the second case we have data as well, `n_root`.
- `structure` gives the type of data we have. In the first case a 2D-vector (real - and imaginary part) and in the second case we have a scalar as well.
- `interleaving` specifies how data is interleaved.
  In our case each time step corresponds to a row in the file (field-interleaving). There are other forms available.

The Sequencer can be used for other types of animation as well. It is possible to let a Plane (MapToPlane-module) move through a volume, for example.

Sometimes the animation is too fast (simple model, few frames) and we would like to lower the frame rate by adding a time delay to each frame. The image window has a Throttle alternative (under Options) and the Image module has a throttle attribute where the delay can be specified. It does not work on our system however (unless one has a delay of more than one second). The reason was simple enough to find (even though OpenDX consists of some 470000 lines of C-code and some 100000 lines of header files). After some reading of the code I found the function `DXWaitTime`, where there is a line `sleep((int)seconds);`. I changed the code and used `usleep` that can sleep in micro seconds, and now everything works.

Some programming tips for use in the VPE

One can undo editing commands (sometimes), see Edit/Undo.

If you change a data file or a header file (general) during a run, you must "Reset Server" under the Connection-menu to see the changes (since OpenDX caches data).

You can duplicate modules on the canvas by pressing the middle button, dragging and releasing. By shift-clicking you can mark several modules (e.g. for copying).

Edit/Layout Graph tries to make a nice layout of the program. Can undo if it becomes ugly.

One can add comments. "Choose Edit/Add Annotation".

When the size of programs grow it may be hard to read them. There are two tools for structuring OpenDX-programs. Suppose we have part of a program like:

```
  ...
Module_a
   |
Module_b
  ...
```

By inserting a Transmitter/Receiver-pair we can break the program into two pieces.

```
  ...
Module_a               Receiver
   |                      |
Transmitter            Module_b
                         ...
```

It is possible to have several pairs. The Transmitter and the Receiver, in a pair, should be given the same name.
In the example above we can use the width of the window to make the program more readable.

We can also move a part of a program to a separate page on the canvas (the canvas will have tabs, each representing a separate page).
Do for example like this: Click on a module which belongs to the part of the program that should be moved.
Choose "Edit/Select/Deselect Tools/Select Connected". This will mark all the modules which are connected with the first we marked. Then choose "Edit/Page/Create with Selected Tools". This will create a new page with part of the program.
One can name the pages using "Edit/Page/Configure Page..".

You can make the interactors look nicer by collecting them in one window. In the Edit-menu there are several options to change the appearance. You can "glue" the interactors together, creating a uniform background using Options/Dialog Style. To get back to the edit mode, click on Close in the Control Panel and answer Yes to the question.