



Dr. Andreas Dedner

Handout Number 1

1 Installation

We need three packages for organizing and visualizing the grid: *Alberta*, *Dune*, and *Grape*. *Dune* (<http://hal.iwr.uni-heidelberg.de/dune>) is a grid interface for accessing and constructing a grid — the actual implementation is in the *Alberta* library (details <http://www.alberta-fem.de>). The library *Grape* (<http://www.mathematik.uni-freiburg.de/IAM/Research/grape/GENERAL>) defines methods for visualizing general grids and discrete solutions defined on the grid. In the following we describe the installation on the CIP-Pool — but basically the steps are the same for an installation under Linux. In the directory `/home/wkurse0/dune` all the required files for the sun installation can be found — for Linux use the files under `/home/wkurse0/Linux/dune`. Login to your account and make a link to the *dune* directory:

```
cd
ln -s /home/wkurse0/dune .
cp -r /home/wkurse0/pde1 .
cd pde1
make
./warmup
```

After these steps you should get some output and the two *Grape* windows (see description in the following section and the files `warmup.cc` and `defines.hh`).

For using the packages with Linux copy the files from `/home/wkurse0/Linux` to your home directory in your Linux installation, i.e., do not use a link in the second step, instead use `cp -r /home/wkurse0/Linux/dune dune`.

2 Grid Structure and Interface

2.1 Grid structure

grid A grid consists of a set of *elements* $T_i \subset \Omega \subset \mathbb{R}^n$ ($i = 0, \dots, N_0 - 1$) where $\bigcup_i T_i = \Omega$ and the measure of $T_i \cap T_j$ is zero for $i \neq j$.

reference element We consider grids consisting of one element type. In this situation there exists one reference element $\widehat{T} \subset \mathbb{R}^n$ which in general is the convex hull of a set of points $\widehat{p}_0, \dots, \widehat{p}_{k-1}$. For each element T_i of the grid we have a smooth isomorphism $F_i : \widehat{T} \rightarrow T_i$ between the reference element and T_i — thus each element in the grid is also a polyeder with k nodes.

Often grids consisting of simplices are used; in this case $k = n + 1$, $\widehat{p}_0 = 0$ and $\widehat{p}_1, \dots, \widehat{p}_n$ are the unit vectors in \mathbb{R}^n . In 2d this leads to a grid consisting of triangles and in 3d of tetrahedral elements. A second possibility is to use the cube $[0, 1]^n$ as reference element.

entities We say that elements are *entities* of co-dimension zero. As a polyeder each is defined by the set of vertices $p_0, \dots, p_{k-1} \in \mathbb{R}^n$ with $p_j = \widehat{T}(\widehat{p}_j)$ ($j = 0, \dots, k - 1$). All vertices p_0, \dots, p_{N_n-1} of the grid are the entities of co-dimension n .

Let us now concentrate on triangular grids in 2d, i.e. $n = 2$ and \widehat{T} is a triangle with the corners $(0, 0)^T, (1, 0)^T, (0, 1)^T$. The vertices of T_i are given by $p_{i_0}, p_{i_1}, p_{i_2}$ ($0 \leq i_1, i_2, i_3 < N_n$). Furthermore we have three edges $e_{i_0}, e_{i_1}, e_{i_2}$ where e_{i_j} is opposite to the vertex p_{i_j} (see Figure 1). Edges of the grid are entities of co-dimension 1 and also have distinct indices between $0, \dots, N_1 - 1$.

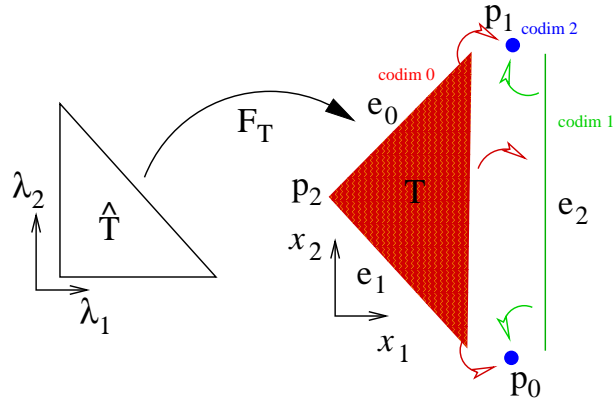


Figure 1: Reference triangle and a general triangle together with the navigation from lower co-dimension to higher co-dimension.

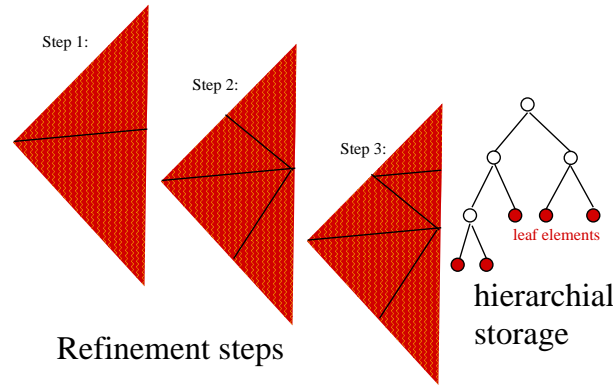


Figure 2: Refinement of the triangle using bisection leads to a binary tree structure for storing the grid. The leaves of the tree correspond to the actual elements of the grid.

intersections We speak of a conform grid if the intersection between two different elements T_i, T_j is either empty, consists of one vertex of the grid or consists of one edge, i.e., $T_i \cap T_j = e_q = e_{i_k} = e_{j_l}$ with $0 \leq q = i_k = j_l < N_1$, and $0 \leq k, l < 3$; in the last case we say that T_i and T_j are neighbors or that they have a common intersection.

hierarchy The grid is constructed starting from a macro-grid (or coarse grid). To reach a higher resolution, elements are marked for refinement. During the adaptation process marked elements are split into a number of *children*; this leads to a tree-like structure in which the grid elements are stored (see Figure 2). The actual grid consists of all *leaf elements* of the grid, i.e., those elements in the tree which have no children.

In the implementation *newest vertex bisection* is used to refine elements. In the macro-grid one vertex in each element is marked as refinement vertex (e.g. vertex p_{i_2} is used). If an element is to be refined the edge opposite to this vertex is split in the middle. With this new vertex two new triangles are formed and the new vertex is used as refinement vertex for the new elements (see Figure 2).

navigation We can navigate over all (leaf) entities of a given co-dimension of the grid and from one entity we can access all entities of a higher co-dimension — from the elements we can access the edges and vertices; from the edges we can access the vertices on that edge. The opposite navigation (for example from vertex p_i to all T_j with $p_i \in T_j$) is not possible! (see Figure 1).

On entities of co-dimension zero, i.e., the triangles, we can also access all entities of co-dimension zero which intersect this element, i.e., the neighboring elements.

indices To define discrete functions on the grid we have to attach *degrees of freedom* (DOFs) to the entities of the grid. This is made possible by assigning distinct indices to each entity of a given co-dimension. These indices are non-negative integers which can be used to access data stored in vectors.

2.2 The Dune grid interface

All necessary types are defined in the file *defines.hh*:

Coordinate types for world coordinates and local coordinates, i.e., $x \in \mathbb{R}^2$ and $\lambda \in \mathbb{R}^2$:

```
typedef Dune::FieldVector<double,dimp> LocalCoordType and
```

```
typedef Dune::FieldVector<double,dimw> GlobalCoordType;
```

these are array-like classes, i.e., with a `[int i]` operator but also with `+=`, `*=` etc.

The class for storing and organizing the triangular grid is

```
typedef Dune::AlbertaGrid< 2, 2 > GridType;
```

on the grid we have methods for marking entities and for adapting the grid. The grid is initialized with a filename containing the macro-grid, i.e, we have a constructor taking a `char*`:

```
GridType grid(filename);
```

The macro-grid has a special format, here is an example (see `quadrat1.git`):

```
DIM: 2
DIM_OF_WORLD: 2
number of elements: 2
number of vertices: 4
vertex coordinates:
-1 -1
 1 -1
 1  1
-1  1
element vertices:
2 0 1
0 2 3
element boundaries:
 1  2  0
 3  4  0
```

(see Figure 3). Together with the coordinates of all vertices in the grid (numbered from 0 to $N_2 - 1$) the triangles in the macro-grid are defined by giving the numbers of the three vertices defining the triangle. The last block defines the boundary $\partial\Omega$ of the grid. This is done by defining the neighbors for all elements, neighbors which are elements are identified by a zero, intersections with the boundary are given by positive or negative numbers.

Furthermore we can obtain start and end iterators for traversing the grid. For example a iterator over the leaf entities of co-dimension zero (the triangles of the grid) is obtained and used as followed

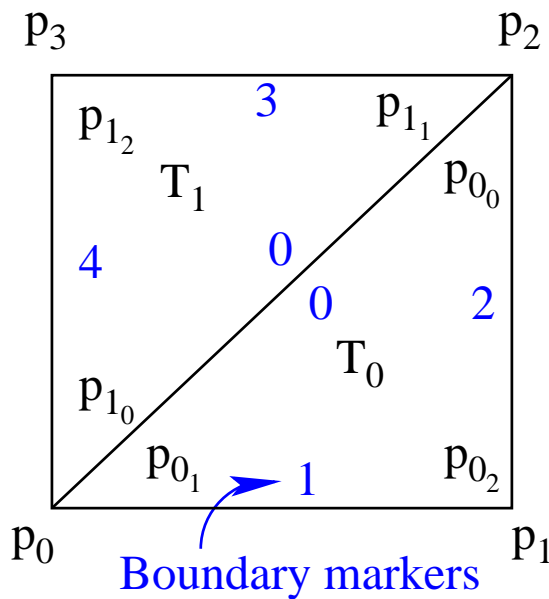


Figure 3: An example of a macro triangulation of the cube $[-1, 1]^2$.

```

LeafIteratorType it = grid.leafbegin<0>();
LeafIteratorType endit = grid.leafend<0>();
for( ; it != endit ; ++it ) {
}

```

*it is of type `EntityType`. The geometry of each entity can be accessed using the method `geometry`. For entities of co-dimension zero:

```
const GeomType& geom=it->geometry();
```

where `it` is a pointer to an instance of `EntityType`. The `GeometryType` class has a `[int i]` operator which returns the coordinates `GlobalCoordType` of the vertices of the given entity.

For solving differential equation on the grid, we need to store data on the grid, for example for a scalar piecewise linear continuous function we need to store one `double` on each vertex of the grid. We want to store these elements in double arrays and use the class `Dune::Array<double>` to do this. These values are called *degrees of freedom* (DOFs). To identify some DOF with a given vertex, i.e., a given entity of co-dimension two, we require an index for each vertex:

```

IndexSetType iset ( grid , grid.maxLevel() );
cerr << "number of vertices: " << iset.size(dimp) << endl;
cerr << "elements in the hierarchy: " << iset.size(0) << endl;
Dune::Array<double> Fh(iset.size(dimp));
LeafIteratorType it = grid.leafbegin<0>();
LeafIteratorType endit = grid.leafend<0>();
for( ; it != endit ; ++it ) {
    cerr << iset.index(*it) << " , ";
    const GeomType& geom=it->geometry();
    for (int c=0;c<geom.corners();c++)
        Fh[iset.subIndex<2>(*it,c)]=...
}

```

3 Visualization with Grape

We have two functions for starting the program Grape:

```
void display(GridType &grid) {...} and
void displayResult(GridType &grid,Dune::Array<double> &uh) {...},
```

the first for the visualization of the grid and the second for the grid together with a discrete function.

In both cases Grape is started and the output window (showing the results) and the *manager window* are opened. In the manager window the top row allows the access to number of different controls panels:

mang In this layer (see Figure 4(left)) one can choose the *display* method which is used for visualizing the grid and the discrete function. Also input and output of data is performed here. We will focus on three button: **switches**, **scene display method**, and finally **I/O**.

By clicking the switches button we can choose between **grid**, **patch**, or **texture** mode display (see Figure 4(middle)) Grid allows the visualization of the grid itself or of isolines of the data. Patch and texture mode are very similar — patch mode being slightly faster but not leading to a very satisfying color distribution. These two modes are used for the visualization of discrete functions.

To choose between different methods for displaying the data one has to click on the scene display button. The display methods are organized in directory like structure. The two most important methods for us are in the directory `../Genmesh2d` (see Figure 4(right)). The first is simply the method **display** which is used for showing the grid (remember to switch to grid mode!), the second is the **isoline** display method.

To write the results of the visualization to disk one can use the I/O button which allows us to write a postscript (ps) or an encapsulated postscript (eps) file.

trans In this layer we can rotate or move the displayed scene. At the bottom we see a button **fit2window** (see Figure 5(left)) which is used to reduce or enlarge the size and position of a scene to fit the display window. At the top the button **reset** can be used to return to the original view of the scene.

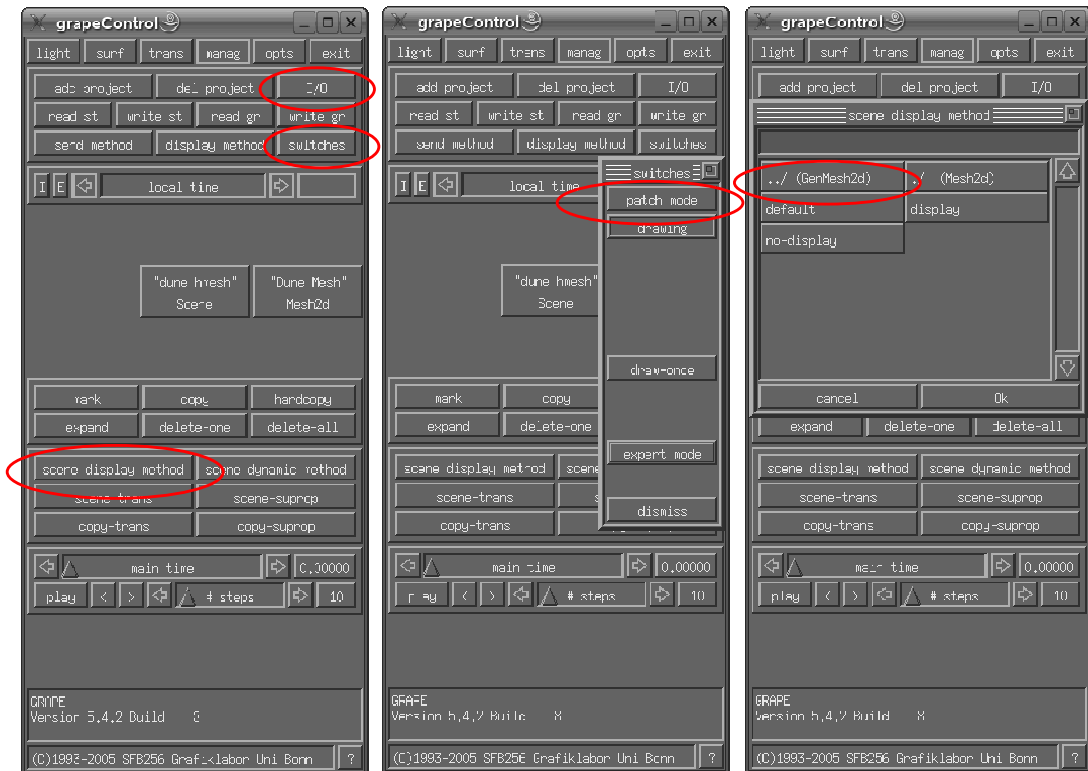


Figure 4: Left: At startup Grape is in the manag-layer. Here the display-style (using the switch-button shown in the middle) and the display-method (right) for the scene can be selected. Also input/output is managed in this layer.

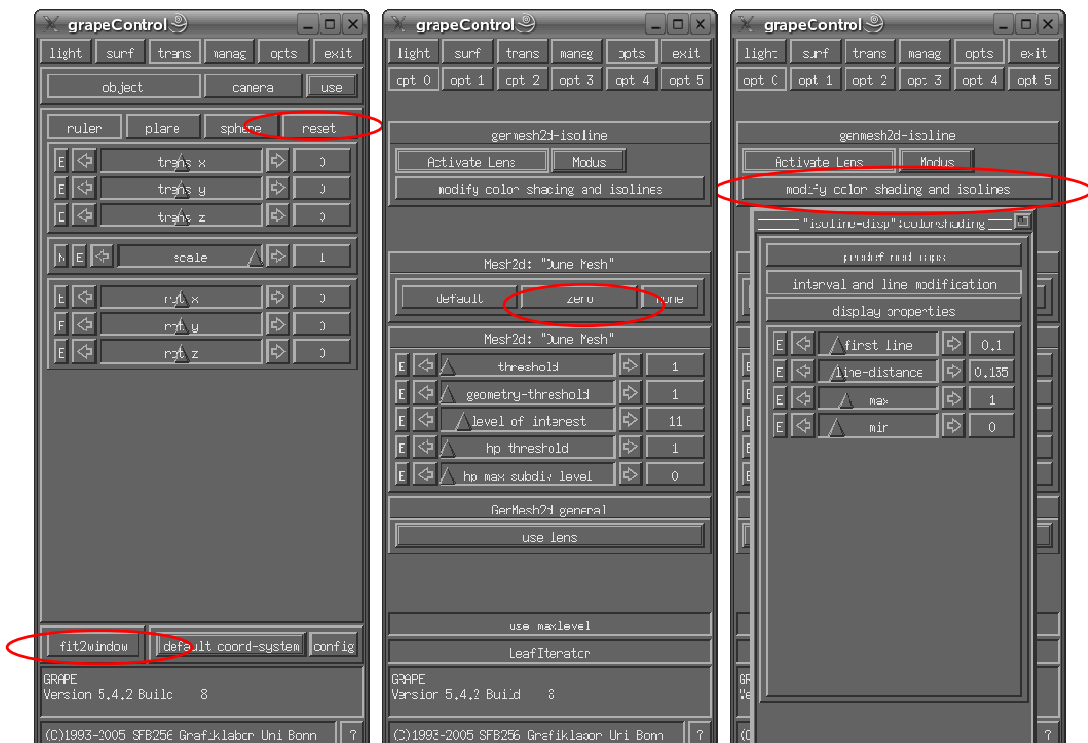


Figure 5: The left picture shows the trans-layer where the scene can be moved and rotated. The middle and the right show the opts-layer where parameters for the display method can be set — for example the discrete function to be displayed can be selected (middle) and the properties of the colorbar (right).

opts The buttons in the opts layer depend on the display method used. For isoline display method we can define the colormap (the way values of the discrete function are mapped to specific colors); and we can choose the discrete function which is to be displayed. This is done with the button in the middle of the layer (see Figure 5(middle)). At the start of Grape this button shows a constant **zero** function. By clicking on the button and holding down the mouse button we get a list of all available discrete function. The function used when calling `displayResult(...)` is at the top denoted **myFunc[0]**.

The properties of the colormap are modified by clicking on the button marked **modify color shading and isolines** — which is only available if the isoline display method was activated in the mang-layer (see Figure 5(right)).