2 Grid generation

Assume that we want to solve a partial differential equation (PDE) on a complicated domain. It could be a PDE describing the flow of air or water past an object, such as a ship, car, or airplane, or it could be a PDE describing the electric and magnetic fields around an object. We must then *discretize* the domain, i.e., divide it into a number of cells where the solution can be represented. Numerical solutions are often represented as point values of a quantity, (e.g. the local velocity of a fluid) at the grid points, or as the average of the quantity over one cell, see Figure 1.2.

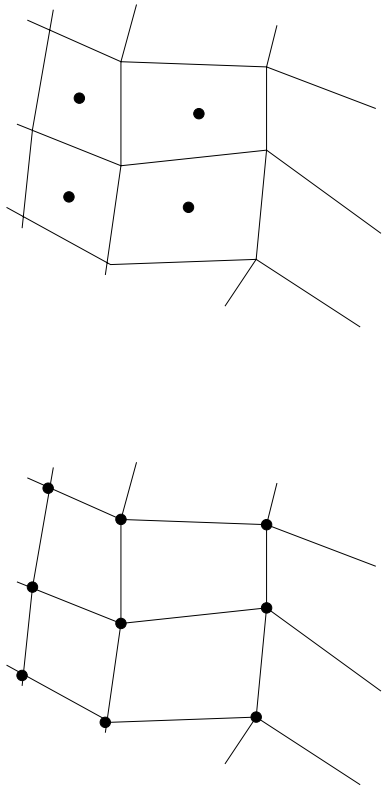


FIG. 1.2. *Grid points and cell centers.*

We next approximate the PDE on the discretized domain, using a finite difference method, finite element method or any other method which is suitable for the problem at hand.

The more densely clustered the grid points are, the better the accuracy will be in the computational time becomes larger. However, with a large number of grid points, the computation becomes slower. There are many problems that even today can not be satisfactory resolved with today's most powerful supercomputers. There are two distinct difficulties when we discretize the domain. How to resolve the geometry, and how to resolve the computed solution.

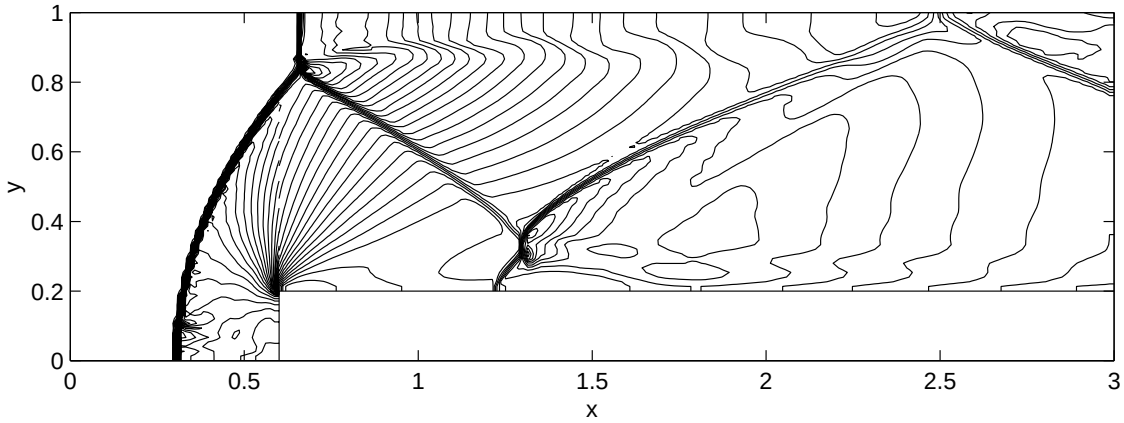


FIG. 1.3. *Compressible fluid flow past a step.*

In Figure 1.3 we show a computation of fluid flow past a forward facing step. Air enters at high speed at the left boundary, flowing in the positive x-direction. The figure shows density contour levels at one instant in time of this unsteady flow. Several shock waves are seen as steep gradient in the density levels. When deciding how many grid points to use in such computations, we have to consider the following points.

- We need to put many grid points where the solution changes quickly, in order to resolve all features. For example, at the shock waves.
- We need put many points near the corner of the step in order to resolve the geometry.
- Which resolution is desired ? If we are only interested in quantities on the boundary, perhaps it is not necessary to resolve all features of the solution far away from the boundary ?

2.1 Handling of Geometry

We distinguish several ways of discretizing a complex geometry

1. Unstructured grids
2. Structured grids
   (a) Rectangular grids
   (b) Multi block grids
   (c) Overlapping grids.

Unstructured grids. The domain is divided into polygons, triangles are often used. See Figure 1.4 for a triangulation of the unit square. Software to generate this type of discretization normally require the user to input an initial, very coarse, triangulation. Perhaps only containing points on the boundary of the domain. Techniques for automatic refinement is then used. In the data structure, each triangle has pointers to its neighbors, but there is no information on coordinate directions. If there are n nodes they are enumerated $i = 1, \ldots, n$, and the coordinates, (x_i, y_i), of each node is stored in an array,

`double x[n], y[n];`.

Information on the connectivity of the nodes is also required. Exactly which information is required depends on the type of discretization used. Unstructured grids are often used with finite element discretizations, but also finite volume methods can be defined on unstructured grids, by considering each triangle as a cell. The solution is represented as volume averages over each triangle. Sometimes the triangles themselves are needed. They are then described by an array,

`int tri[m,3];`,

where `tri[i,0], tri[i,1], tri[i,2]` are the numbers of the three nodes in triangle i. Note that the number of triangles, m, is different from the number of nodes, n. The coordinates of the first node in triangle i, is thus given by the indirect reference

`x[tri[i,0]], y[tri[i,0]]`

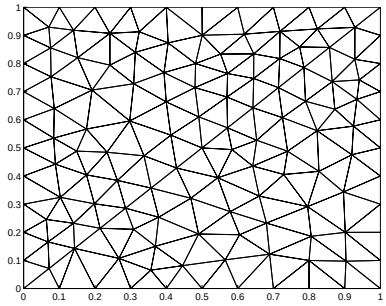


FIG. 1.4. *Unstructured grid on the unit square.*

Advantages with the unstructured grids are:

+ Generality, no need to think about block decomposition.
+ Grid refinement is straightforward.

Disadvantages

- Inefficient on many computer architectures.
- Not straightforward to get an efficient parallelization.
- Discretization formulas often more complicated.

Algorithms for construction of unstructured grids will not be described in this course. We refer to the courses in the finite element method.

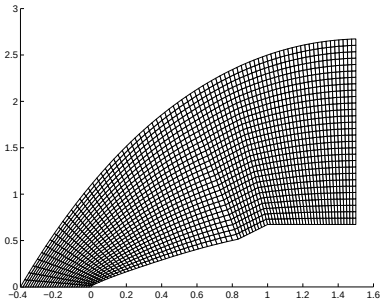


FIG. 1.5. *Structured grid.*

Structured grids. A structured grid is something which is indexed along coordinate directions. We think of a grid as a mapping

$$x(\xi, \eta, \zeta), y(\xi, \eta, \zeta), z(\xi, \eta, \zeta) \tag{2.1}$$

from the unit cube $0 \leq \xi \leq 1, 0 \leq \eta \leq 1, 0 \leq \zeta \leq 1$ to the physical space. See Figure 1.5. For finite difference approximations, we want the grids to be smooth transformations, so that we use (2.1) to transform the PDE to the unit cube, and solve it there. The transformed problem will contain derivatives of the grid as coefficients. This is in analogy with the smooth local coordinate maps of differential geometry. It is possible to define finite volume or finite difference approximations which do not require smoothness of the grid, but these approximations are in general more complicated, and computationally expensive.

Advantages with structured grids are:

+ Easy to implement, and good efficiency

+ With a smooth grid transformation, discretization formulas can be written on the same form as in the case of rectangular grids.

Disadvantages

- Difficult to keep the structured nature of the grid, when doing local grid refinement.

- Difficult to handle complex geometry. (Mapping of an aircraft to the unit cube !).

There are several ways to overcome the difficulty of using structured grids for complex geometry. The most common device is to divide the domain into blocks, where each block can be mapped to the unit cube separately. A division into blocks can sometimes come naturally from a CAD representation of the geometry. The blocks can be adjacent, like in Figure 1.6 a), or overlapping as in Figure 1.6 b). In both cases special interpolating boundary conditions are needed at the interfaces between the grids. In Figure 1.6, the domain is a disk. If a single polar coordinate system had been used, the grid mapping would have been singular at the point $(x, y) = (0, 0)$, and the tranformation formulas would have been undefined there. When using finite volume methods, it is possible to overcome this problem by using a special formula for the tringaular shaped cells closest to the point $(0, 0)$.

Note that block subdivision is like using unstructuredness on a coarser level. The block layout is like an unstructured decomposition of the domain, but where each component is further refined structuredly. The first step in the discretization is to define suitable blocks, and then in the second step, generate one grid for each block.

An alternative to this, is to use purely rectangular grids, and "cut out" the objects as holes in the grid, as shown in Figure 1.7. This is a very general method. However, it is difficult to achieve good accuracy in the boundary conditions for this method. Furthermore, since cells are cut arbitrarily, some cells at the boundary can become very small, and cause stability problems for explicit difference schemes. The discretization is required to work

for all shapes of cells. The method is used when the geometry is extremely complicated, so that generating separate grids around each little detail is not a feasible technique.

We will next concentrate on the problem of generating one single grid. There are some fairly general techniques for doing this. On the other hand, the division into blocks is usually done manually for each particular configuration, and can require many days work by an engineer. Before going to the grid generation techniques, we give an example that shows how grids can be used in a PDE solving computer code.

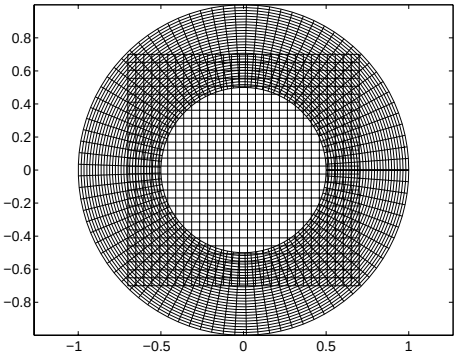


FIG. 1.6a. *Overlapping grids.*

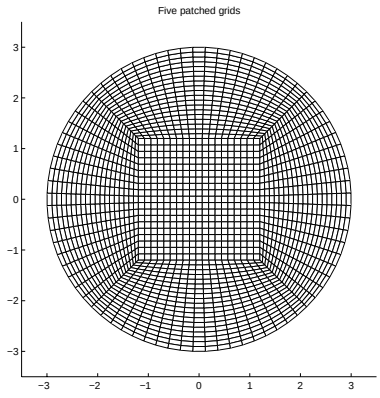


FIG. 1.6b. *Adjacent grids.*

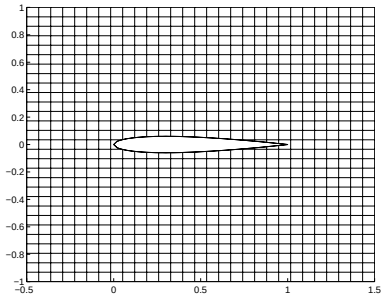


FIG. 1.7 *Geometry cut out from a rectangular grid.*

2.2 Example: Solution of the heat equation on a structured grid

We here show a typical example of how a structured grid is used for solving a simple PDE. Most of the general techniques and potential problems with structured grids can be understood from this example. The PDE is the heat equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

The computational domain is the annular region

$$D = \{(x, y) \mid 1 \leq \sqrt{x^2 + y^2} \leq 3\},$$

and we give the boundary condition $u = 2$ on the inner boundary ($\sqrt{x^2 + y^2} = 1$), and $u = 1$ on the outer boundary. The initial data are given by

$$u(0, x) = 1 + \exp(-(x - 2)^2 + y^2)$$

A grid mapping is easily defined as

$$x(r,s) = (1+2r)\cos 2\pi s \qquad y(r,s) = (1+2r)\sin 2\pi s$$

where $0 < r < 1$ and $0 < s < 1$. Note that two new boundaries, at $s = 0$ and at $s = 1$, that were not present in the original problem definition, now appear. This is a so called periodic boundary, ie, the solution at $s = 0$ should be equal to the solution at $s = 1$. The grid points can be defined as

$$x_{i,j} = x(r_i, s_j), \;\; i = 1, 2, \ldots, m \;\; j = 1, 2, \ldots, n$$
$$y_{i,j} = y(r_i, s_j), \;\; i = 1, 2, \ldots, m \;\; j = 1, 2, \ldots, n$$

where $r_i = (i-1)/(m-1)$ and $s_j = (j-1)/(n-1)$. The partial derivatives with respect to x and y are transformed to the r and s coordinate system and approximated there. This is done by use of the chain rule,

$$\frac{\partial f}{\partial r} = \frac{\partial x}{\partial r}\frac{\partial f}{\partial x} + \frac{\partial y}{\partial r}\frac{\partial f}{\partial y}$$
$$\frac{\partial f}{\partial s} = \frac{\partial x}{\partial s}\frac{\partial f}{\partial x} + \frac{\partial y}{\partial s}\frac{\partial f}{\partial y}$$

which is inverted as a linear system of two unknowns to become

$$\frac{\partial f}{\partial x} = \frac{1}{D}\left(\frac{\partial y}{\partial s}\frac{\partial f}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial f}{\partial s}\right)$$
$$\frac{\partial f}{\partial y} = \frac{1}{D}\left(-\frac{\partial x}{\partial s}\frac{\partial f}{\partial r} + \frac{\partial x}{\partial r}\frac{\partial f}{\partial s}\right.$$

where

$$D = \frac{\partial x}{\partial r}\frac{\partial y}{\partial s} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial s}$$

is the determinant of the coefficient matrix. We let $u_{i,j}$ denote the approximation of the function u at the point $(x_{i,j}, y_{i,j})$. For the numerical approximation of the derivatives, we use the second order accurate centered operator. The r-derivatives are taken in the i-index direction and the s-derivatives are directed along the j-index direction. For example,

$$\left(\frac{\partial x}{\partial r}\right)_{i,j} \approx \frac{x_{i+1,j} - x_{i-1,j}}{2\Delta r}, \qquad i = 2, 3, \ldots, m-1 \quad j = 1, 2, \ldots, n$$

and similarly for $\frac{\partial y}{\partial r}$ and $\frac{\partial u}{\partial r}$. The centered formula can not be used at the boundary points $i = 1$ and $i = m$. We instead use the one-sided differences

$$\left(\frac{\partial x}{\partial r}\right)_{1,j} \approx \frac{x_{2,j} - x_{1,j}}{\Delta r}, \qquad \left(\frac{\partial x}{\partial r}\right)_{m,j} \approx \frac{x_{m,j} - x_{m-1,j}}{\Delta r},$$

there.

The implementation of the solution procedure in the language C is shown in Code 1.1 below.

```
#include <math.h>

void diffr( double * x, double * xr, int m, int n );
void diffs( double * x, double * xr, int m, int n );

int main()
{
  int i, j, m, n, ind, st, maxsteps;
  double *x, *y, *xr, *yr, *xs, *ys, *u, *res, *det, *ur, *us, *ux, *uy;
  double r, s, alpha, pi, dt;

  m = 30;
  n = 100;
  alpha = 1;
  maxsteps = 150;
  pi = 3.1415926535;
  dt = 0.001;

/* Allocate memory */
  x   = (double *)malloc( m*n*sizeof(double));
  y   = (double *)malloc( m*n*sizeof(double));
  xr  = (double *)malloc( m*n*sizeof(double));
  yr  = (double *)malloc( m*n*sizeof(double));
  xs  = (double *)malloc( m*n*sizeof(double));
  ys  = (double *)malloc( m*n*sizeof(double));
  u   = (double *)malloc( m*n*sizeof(double));
  res = (double *)malloc( m*n*sizeof(double));
  det = (double *)malloc( m*n*sizeof(double));
  ur  = (double *)malloc( m*n*sizeof(double));
  us  = (double *)malloc( m*n*sizeof(double));
  ux  = (double *)malloc( m*n*sizeof(double));
  uy  = (double *)malloc( m*n*sizeof(double));

/* Generate grid */
  for( j= 0 ; j<n ; j++ )
     for( i= 0 ; i<m ; i++ )
     {
        ind = i + m*j;
        r  = (i)/(m-1.0);
        s  = (j)/(n-4.0);
        x[ind] = (1+2*r)*cos(2*pi*s);
        y[ind] = (1+2*r)*sin(2*pi*s);
     }

/* Compute metric */
  diffr( x, xr, m, n );
  diffr( y, yr, m, n );
  diffs( x, xs, m, n );
  diffs( y, ys, m, n );
  for( ind = 0 ; ind < m*n ; ind++ )
     det[ind] = xr[ind]*ys[ind]-xs[ind]*yr[ind];

/* Get initial data */
  for( ind = 0 ; ind < m*n ; ind++ )
     u[ind] = 1 + exp(-alpha*( (x[ind]-2)*(x[ind]-2) + y[ind]*y[ind] ) );

/* Main loop for time stepping */
  for( st = 0 ; st < maxsteps ; st++ )
  {

    /* Compute the second derivatives */
     diffr( u, ur, m, n );
     diffs( u, us, m, n );
     for( ind = 0 ; ind < m*n ; ind++ )
     {
        ux[ind] =( ur[ind]*ys[ind]-us[ind]*yr[ind])/det[ind];
        uy[ind] =(-ur[ind]*xs[ind]+us[ind]*xr[ind])/det[ind];
     }
     diffr( ux, ur, m, n );
```

```
     diffs( ux, us, m, n );
     for( ind = 0 ; ind < m*n ; ind++ )
        res[ind] =( ur[ind]*ys[ind]-us[ind]*yr[ind])/det[ind];

     diffr( uy, ur, m, n );
     diffs( uy, us, m, n );
     for( ind = 0 ; ind < m*n ; ind++ )
        u[ind] = u[ind] + dt*( res[ind] +
                  ( -ur[ind]*xs[ind]+us[ind]*xr[ind])/det[ind] );

     /* Inner and outer boundaries */
     for( j=0 ; j < n ; j++ )
     {
       u[m*j] = 2;
       u[m-1+m*j] = 1;
     }
     /* Periodic boundary */
     for( i=0 ; i < m ; i++ )
     {
        u[ i    ] = u[i + m*(n-4) ];
        u[ i +m ] = u[i + m*(n-3) ];
        u[ i +m*(n-2) ] = u[i + m*2];
        u[ i +m*(n-1) ] = u[i + m*3];
     }

  }
 /*  Write out solution and grid */
  for( ind=0 ; ind < n*m ; ind++ )
     printf("%g   %g   %g \n" , x[ind], y[ind], u[ind] );

/* Should give back memory here */

}

/* ---------- Procedure to compute r-derivative --------- */
void diffr( double * x, double * xr, int m, int n )
{
   int i, j, ind;
   for( j= 0 ; j<n ; j++ )
   {
      i = 0;
      ind = i + m*j;
      xr[ind] = x[ind+1]-x[ind];
      for( i= 1 ; i<m-1 ; i++ )
      {
         ind = i + m*j;
         xr[ind] = 0.5*(x[ind+1]-x[ind-1]);
      }
      i = m-1;
      ind = i + m*j;
      xr[ind] = x[ind]-x[ind-1];
   }
}

/* ---------- Procedure to compute s-derivative --------- */
void diffs( double * x, double * xs, int m, int n )
{
   int i, j, ind;
   for( i= 0 ; i<m ; i++ )
   {
      j = 0;
      ind = i + m*j;
      xs[ind] = x[ind+m]-x[ind];
      for( j= 1 ; j<n-1 ; j++ )
      {
         ind = i + m*j;
         xs[ind] = 0.5*(x[ind+m]-x[ind-m]);
      }
      ind = n-1;
      ind = i + m*j;
```

```
      xs[ind] = x[ind]-x[ind-m];
   }
}
```

CODE 1.1, *C program to solve 2D PDE on curvilinear grid*

Code 1.1 performs the following steps

1. Generate grid points
2. Compute metric coefficients
3. Impose initial data
4. Main time stepping loop. In each iteration, the residual is computed, the solution is updated to the new time level, and boundary conditions are imposed on the new solution.
5. Solution and grid are written to standard output. The results can be read into a visualization program for display.

Note that the grid point generation step could be replaced by reading into the program any curvilinear grid, and the program would work equally well. No specialized features of the known grid mapping are used.

The metric derivatives, $\partial x/\partial r$, $\partial x/\partial s$, etc. are precomputed and stored into the arrays `xr`, `xs`, `yr`, `ys`. The determinant of the metric coefficient matrix is stored into the array `det`. It would have been possible to avoid storing these, and instead compute them whenever they are needed. There is a trade-off between computational speed and use of memory. It has turned out in practise, that it is usually better to precompute them, since memory is less critical than computational speed. However this is dependent on the kind of computer used.

The approximation of the second derivative $\partial^2 u/\partial x^2$ is done by applying the centered approximation twice. On a uniform rectangular grid, this means using the approximation $D_0 D_0 u_j$, where $D_0 u_j = (u_{j+1} - u_{j-1})/(2\Delta x)$, instead of the standard $D_+ D_- u_j = (u_{j+1} - 2u_j + u_{j-1})/\Delta x^2$. The $D_0 D_0$ approximation is somewhat less well-behaved from the point of view of stability, but the implementation becomes very easy.

The difference approximations are done in separate procedures `diffr` and `diffs`, since they are needed to be applied on several different variables. Make sure you understand how these procedure works.

The matrices $u_{i,j}, x_{i,j}$ etc are all stored as vectors. Matrix operations are not well-developed in C, and we prefered in this program to use the 2D to 1D index mapping of a $m \times n$ matrix into a vector of mn elements,

$$ind = i + m(j-1)$$

where $1 \le i \le m$, $1 \le j \le n$, and $1 \le ind \le mn$. In C, an array with n elements is indexed from 0 to $n-1$. Therefore, in Code 1.1, $0 \le i \le m-1$, $0 \le j \le n-1$, and $0 \le ind \le mn-1$, and the mapping becomes

$$ind = i + mj$$

To summarize, the matrix element $x_{i,j}$ is represented as `x[i+m*j]` in the computer code. Furthermore, loops that only involves point-wise operations can be coded as one-dimensional, i.e., can loop over `ind`, instead of using a double loop over `i` and `j`. This usually gives some gain in efficiency.

Note that the steps in the r, s space, Δr and Δs are not used in the implementation. These are cancelled from the differentiation formulas

$$\frac{1}{D}(\frac{\partial y}{\partial s}\frac{\partial f}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial f}{\partial s})$$

because both the numerator and the denominator D contain a factor $\Delta s \Delta r$.

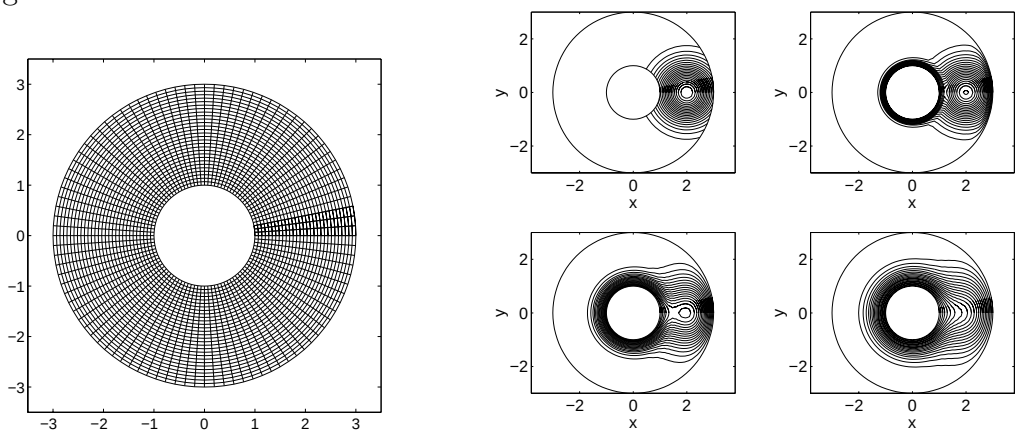
The boundary condition on the inner boundary at `i=0`, $u = 2$, and on the outer boundary at `i=m-1`, $u = 1$, are imposed in a straightforward way. The periodic boundary condition is implemented, by using a few redundant grid points near the boundaries `j=0` and `j=n-1`. The grid is arranged so that

$$\begin{aligned} u_{i,1} &= u_{i,n-3} \quad u_{i,2} = u_{i,n-2} \\ u_{i,n-1} &= u_{i,3} \quad u_{i,n} = u_{i,4} \end{aligned} \tag{1.1}$$

The operator

$$\left(\frac{\partial u}{\partial s}\right)_{i,j} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta s}$$

is applied twice for the second derivative in the heat equation. Therefore, the difference approximation depends on the points $j-2, j-1, j, j+1, j+2$, i.e., a five point stencil in the j-direction (the same holds in the i-direction). The periodicity is easily enforced by computing from $j = 3$ to $j = n-2$ and then impose periodicity conditions (1.1) to find the values of u at $j = 1, 2, n-1, n$. Note also that use of $s_j = j/(n-4)$ instead of $s_j = j/(n-1)$ in the grid point generation statement, is due to this periodic 'wrap-around' of the grid.



Grid and solution at different times for example problem.

In the figure above we show the grid and the solution after 0, 10, 70, and 140 time steps.

2.3 Single structured grids

For single grid components, we have several methods of constructing grids.

1. Analytical transformations
2. Algebraic grid generation
3. Elliptic grid generation
4. Variational grid generation
5. Hyperbolic grid generation

In general, the procedure to generate a grid goes as follows. First generate grids on the edges of the domain, second from the given edges, generate the sides, and finally generate the volume grid from the six given side grids. In the description below, we will assume two space dimensions, but the methods extends straightforwardly to three dimensions.

Analytical transformations. This is the most obvious type of grid generation. If the geometry is simple enough that we know an analytical transformation to the unit square, the grid generation becomes very easy. E.g., the domain between two circles,

$$D = \{(x, y) \mid r_i^2 \leq x^2 + y^2 \leq r_o^2\},$$

where r_i is the radius of the inner circle and r_o is the radius of the outer circle, can be mapped to the unit square by the mapping

$$\begin{aligned} x &= (r_i + (r_o - r_i)r)\cos 2\pi\theta \\ y &= (r_i + (r_o - r_i)r)\sin 2\pi\theta \end{aligned}$$

where now $0 \leq r \leq 1$ and $0 \leq \theta \leq 1$. The grid is then obtained by a uniform subdivision of the (r, θ) coordinates. This method of generating grids is simple and efficient. Another advantage is that we can generate orthogonal grids if a conformal mapping can be found. Orthogonal grids are grids where grid lines always intersect at straight angles. Often PDE approximations becomes more accurate, and have stencils with fewer points on orthogonal grids. Of course, this method has very limited generality, and can only be used for very special geometries.

Algebraic grid generation. This type of grid generation is also called transfinite interpolation. In its simplest form, it is described by the formula

$$\begin{aligned} x(\xi,\eta) = \;& (1-\xi)x(0,\eta) + \xi x(1,\eta) + (1-\eta)x(\xi,0) + \eta x(\xi,1) - \\ & (1-\xi)(1-\eta)x(0,0) - \xi(1-\eta)x(1,0) - (1-\xi)\eta x(0,1) - \xi\eta x(1,1) \\ y(\xi,\eta) = \;& (1-\xi)y(0,\eta) + \xi y(1,\eta) + (1-\eta)y(\xi,0) + \eta y(\xi,1) - \\ & (1-\xi)(1-\eta)y(0,0) - \xi(1-\eta)y(1,0) - (1-\xi)\eta y(0,1) - \xi\eta y(1,1) \end{aligned} \tag{2.2}$$

Here it is assumed that $(x(\xi,\eta), y(\xi,\eta))$ are known on the sides of the domain. Formula (2.2) gives then the grid in the entire domain. The formula is an interpolation from the sides to the interior. The two first terms is the interpolation between the sides $\xi = 0$ and $\xi = 1$, the next two terms are the same for the η direction. Finally four corner terms are subtracted. One can easily verify that (2.2) is exact on the boundary by putting $\xi = 0$, $\xi = 1$, $\eta = 0$, or $\eta = 1$.

This formula can be generalized to the case where

- x and y are given on several coordinate lines in the interior of the domain.
- both x and y and its derivatives are given on the sides.

A general form of (2.2) is obtained by introducing the blending functions φ_0 and φ_1, with the properties

$$\varphi_0(0) = 1 \quad \varphi_0(1) = 0 \quad \varphi_1(0) = 0 \quad \varphi_1(1) = 1$$

The general transfinite interpolation is then defined by

$$\begin{aligned} x(\xi,\eta) = \quad & \varphi_0(\xi)x(0,\eta) + \varphi_1(\xi)x(1,\eta) + \varphi_0(\eta)x(\xi,0) + \varphi_1(\eta)x(\xi,1) - \\ & \varphi_0(\xi)\varphi_0(\eta)x(0,0) - \varphi_1(\xi)\varphi_0(\eta)x(1,0) - \varphi_0(\xi)\varphi_1(\eta)x(0,1) - \varphi_1(\xi)\varphi_1(\eta)x(1,1) \end{aligned}$$

By introducing the projector

$$P_\xi(x) = \varphi_0(\xi)x(0,\eta) + \varphi_1(\xi)x(1,\eta)$$

we can write the transfinite interpolation above as the boolean sum

$$x(\xi,\eta) = P_\xi(x) + P_\eta(x) - P_\xi(P_\eta(x)) \tag{2.3}$$

This form is convenient for describing generalizations of the method, e.g., if the derivatives of the grid are prescribed on the boundary, we can use the projector

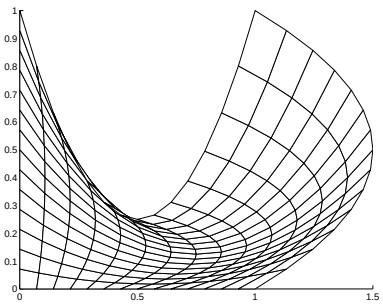
$$P_\xi(x) = \varphi_0(\xi)x(0,\eta) + \varphi_1(\xi)x(1,\eta) + \psi_0(\xi)\frac{\partial x(0,\eta)}{\partial \xi} + \psi_1(\xi)\frac{\partial x(1,\eta)}{\partial \xi}$$

in (2.3) instead. Here the new blending function ψ_0 satisfies

$$\begin{aligned} \psi_0(0) = 0 \quad \psi_0(1) = 0 \quad \psi_0'(0) = 1 \quad \psi_0'(1) = 0 \\ \psi_1(0) = 0 \quad \psi_1(1) = 0 \quad \psi_1'(0) = 0 \quad \psi_1'(1) = 1 \end{aligned}$$

An example of when we would like to prescribe the derivatives on the boundary, is when we want the grid to be orthogonal to the boundary. This can sometimes be needed in order to simplify boundary conditions. Of course, it is possible to use different projectors and blending functions in the different coordinate directions.

The advantage of algebraic grid generation is its efficiency, and ease of implementation. A disadvantage is that there is no guarantee that the method will be successful. For very curved boundaries, it can often happen that grid lines intersect, like the example in Figure 1.8 a). Another problem is the propagation of singularities from the boundary. If the boundary has an interior corner, as in Figure 1.8 b), the corner will be seen in the interior domain, making it difficult to generate smooth grids.



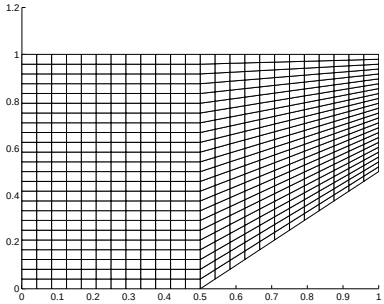


FIG. 1.8a. *Folding of grid lines.* FIG. 1.8b. *Propagating corner.*

Elliptic grid generation. This type of grid generation is motivated by the maximum principle for elliptic PDEs. We define the inverse grid transformation, $\xi(x,y), \eta(x,y)$ as the solution of

$$\begin{aligned} \xi_{xx} + \xi_{yy} &= 0 \\ \eta_{xx} + \eta_{yy} &= 0 \end{aligned} \tag{2.4}$$

We know that $0 \le \xi \le 1$ and $0 \le \eta \le 1$ are monotone on the boundaries. It then follows from the maximum principle that ξ and η will stay between these values. Furthermore, there will be no local extrema in the interior, and thus grid lines can not fold. The equations (2.4) are formulated in the x-y domain, and has to be transformed to the unit square, so that we can solve them there. We use the unknown transformation itself to transform the equations (2.4). The transformed system then becomes

$$\begin{aligned} (x_\eta^2 + y_\eta^2)x_{\xi\xi} - 2(x_\xi x_\eta + y_\xi y_\eta)x_{\xi\eta} + (x_\xi^2 + y_\xi^2)x_{\eta\eta} = 0 \\ (x_\eta^2 + y_\eta^2)y_{\xi\xi} - 2(x_\xi x_\eta + y_\xi y_\eta)y_{\xi\eta} + (x_\xi^2 + y_\xi^2)y_{\eta\eta} = 0 \end{aligned} .$$

This problem can be solved as a Dirichlet problem if (x,y) are given on the boundaries, or as a Neumann problem if the normal derivatives of (x,y) are specified on the boundaries. Specifying normal derivatives is here equivalent to specifying the distance between the first and second grid lines. These equations are then approximated by, e.g.,

$$\begin{aligned} x_\xi &\approx (x_{i+1,j} - x_{i-1,j})/2 \quad x_\eta \approx (x_{i,j+1} - x_{i,j-1})/2 \\ x_{\xi\xi} &\approx x_{i+1,j} - 2x_{i,j} + x_{i-1,j} \quad \text{etc.} \end{aligned}$$

where now the index space $1 \le i \le n_i$ and $1 \le j \le n_j$ is a uniform subdivision of the (ξ,η) coordinates, $\xi = (i-1)/(n_i-1)$, $\eta = (j-1)/(n_j-1)$. The number of grid points is specified as $n_i \times n_j$.

```
      subroutine GAUSEI2D( ni, nj, x, y, err )

      integer ni, nj, i, j
      real*8 x(ni,nj), y(ni,nj), xtemp, ytemp, err
      real*8 g11, g12, g22

      err = 0

      do j=2,nj-1
         do i=2,ni-1

            g11 =((x(i+1,j)-x(i-1,j) )**2 + (y(i+1,j)-y(i-1,j))**2 )/4
            g22 =((x(i,j+1)-x(i,j-1) )**2 + (y(i,j+1)-y(i,j-1))**2 )/4
            g12 =(x(i+1,j)-x(i-1,j))*(x(i,j+1)-x(i,j-1))/4+
     *          (y(i+1,j)-y(i-1,j))*(y(i,j+1)-y(i,j-1))/4

            xtemp = 1/(2*(g11+g22))*(
     *          g22*x(i+1,j) - 0.5*g12*x(i+1,j+1) + 0.5*g12*x(i+1,j-1)+
     *          g11*x(i,j+1) + g11*x(i,j-1) +
     *          g22*x(i-1,j) - 0.5*g12*x(i-1,j-1) + 0.5*g12*x(i-1,j+1) )

            ytemp = 1/(2*(g11+g22))*(
     *          g22*y(i+1,j) - 0.5*g12*y(i+1,j+1) +0.5*g12*y(i+1,j-1)+
     *          g11*y(i,j+1) + g11*y(i,j-1) +
     *          g22*y(i-1,j) - 0.5*g12*y(i-1,j-1) +0.5*g12*y(i-1,j+1) )
```

```
      err = err + (x(i,j)-xtemp)**2+(y(i,j)-ytemp)**2

      x(i,j) = xtemp
      y(i,j) = ytemp

    enddo
  enddo

  err = SQRT( err/((nj-2)*(ni-2)) )

  return
  end
```

CODE 1.2. *Gauss-Seidel iteration for elliptic grid generator.*

The equations can then be solved by a standard elliptic solver such as, e.g., conjugate gradients, Gauss-Seidel or the multi grid method. In Code 1.2, we show a fortran subroutine for doing one Gauss-Seidel iteration on the system.

If you want to specify both the grid and its normal derivatives on the boundary, the second order elliptic PDE above can not be used, but it is possible to define an elliptic problem with fourth order derivatives instead.

Elliptic grid generation is very reliable, and will always produce a grid. However it might not always be the grid you want. For example, grid lines tend to cluster near convex boundaries, but will be very sparsely distributed near concave boundaries. In Figures 1.9a and 1.9b we show the same example as in Figures 1.8a and 1.8b, but now the grid is generated by the elliptic equations (2.4). Clearly, the problems which occured with transfinite interpolation does not happen here.

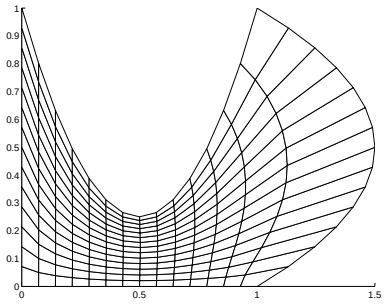


FIG. 1.9a. *No folding.*

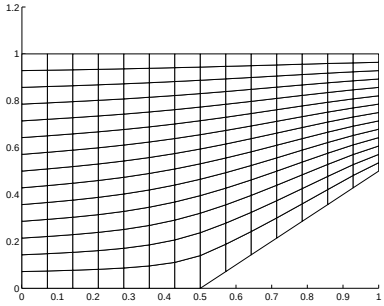


FIG. 1.9b. *Corner smoothed.*

To introduce more control over the grid, so called control functions are introduced into (2.4). The system then becomes

$$\xi_{xx} + \xi_{yy} = P$$
$$\eta_{xx} + \eta_{yy} = Q$$

Alternatively the functions (P, Q) can be scaled so that the right hand side becomes $(\xi_x^2 + \xi_y^2)P$ and $(\eta_x^2 + \eta_y^2)Q$. The maximum principle is now lost, so that we have no guarantee that a grid will be successfully generated. There is a risk that grids with folded coordinate lines occur.

The functions P and Q can be chosen to attract grid lines to certain lines or points in the space. To do this by hand is very difficult. Often P and Q are specified from a weight function, which specifies the grid density at each point in the domain. This is often