Chapter 8

Implicit integration, incompressible flows

The methods we discussed so far work well for problems of hydrodynamics in which the flow speeds of interest are not orders of magnitude smaller than the sound speed. There are, however, many situations in which we are interested in flows that are very subsonic. For instance the flow of air in the Earth's lower atmosphere, the flow of water in oceans, etc. In such cases it is reasonable to assume that there is pressure equilibrium everywhere to the extent that the gas/fluid will not be accelerated to velocities anywhere near the sound speed. In other words, the flow is always very subsonic: $|\vec{u}| \ll C_s$. In such cases we are not interested in the detailed propagation of sound waves, but in the much slower movement of the fluid itself. Can we use the methods of the previous chapter for such problems? Yes, we can, but it will become extremely computationally expensive to model any appreciable flow of the fluid. Here is why: if we determine the time step from the CFL condition (see Chapter 3), then the time step is limited by the fastest characteristics. In this case these are clearly the sound waves. If, for example, our typical flow moves at speeds of less than 0.01 times C_s, then the time step is 100 times smaller than required for the advection of the fluid itself. So the fast characteristics may not be interesting for us, but they do exist, and not obeying the CFL condition for them (i.e. taking the CFL condition for the fluid movement only, which gives a 100 times larger time step in our example) would simply blow up our code. The problem here is that the sound waves exist, if we like them or not. And they limit the time step. Suppose we want to model the flow of fluid from the left boundary of our domain to the right, over 100 grid points. For pure advection (disregarding the sound waves) the CFL condition with CFL number 0.5 requires of the order of 200 time steps for the fluid to move into the domain on the left and out of the domain on the right. But because the sound waves are much faster, say 100$\times$ faster, they limit the time step to a 100$\times$ smaller value, meaning that our model now requires 20000 time steps. It is clear that this is not very practical, and alternative methods must be found to allow us to keep our time step such that it obeys the CFL condition for the fluid motion only, not for the sound waves.

Problems of this kind, where the time scales of interest are much larger than the formal shortest time scale of the problem, are called "stiff". This is a very common complication in numerical integration of ordinary and partial differential equations (PDEs). If the rapid modes of the problem quickly relax to a secular equilibrium (i.e. a quasi-equilibrium), then one may be more interested in the much longer time scale evolution of the system, without wanting to waste CPU time on the short-time-scale modes that are anyway related to their quasi equilibrium, and hence do not change on these short time scales. An example of a typical "stiff" partial differential equation is that of the diffusion equation. The shortest time scales are represented by the smearing-out of the narrowest peaks. In more mathematical terms: the highest $\vec{k}$ modes

damp out the quickest. In typical applications these highest $\vec{k}$ modes damp out so quickly that they have reached their quasi-equilibrium already in the first few time steps. The more interesting longer timescale evolution ("secular evolution") concerns much smaller $\vec{k}$ modes, i.e. much longer wavelength modes. These could in principle be followed sufficiently accurately with time steps much larger than the formal CFL condition. The CFL condition limits the time step to the damping time of the modes with a wavelength as large as two grid points. However, since these modes are expected to be always near their equilibrium, we are not interested in modeling this system at these very small time steps. Numerical stability, however, forces us to do this, thereby forcing us to waste a lot of CPU time.

To solve problems of "stiff" nature, such as the diffusion problem or the problem of very subsonic flow, a fundamentally different way of integration of the partial differential equations has to be introduced. For very subsonic flow we also make an approximation to the flow equations. All these techniques require the solution of large sets of coupled linear equations, and the solution of these equations requires special techniques, which we will briefly outline here (though this is a topic that deserves an entire one-semester lecture!).

We will start out with a simple example of an ordinary differential equation to demonstrate the principle of stiffness. Then we will concern ourselves with the diffusion equation and how to solve this efficiently using "implicit integration". We will then generalize this to 2-D and 3-D, and turn then to general methods for the solution of multi-dimensional parabolic partial differential equations.

We will then turn to the equations very subsonic (i.e. incompressible) flows. We shall see that the solution requires similar techniques as for parabolic PDEs to solve for the pressure. Once this is done, we can use our classic explicit advection schemes for the propagation of the fluid.

8.1 Simple example: an ordinary differential equation

8.1.1 Stiffness of an ordinary differential equation

Let us start with a simple example of a stiff ordinary differential equation:

$$\frac{d}{dt}\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -100 \end{pmatrix}\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \tag{8.1}$$

Suppose at $t = 0$ we start with $q_1 = 1$ and $q_2 = 1$. Now let us integrate this using forward Euler integration, i.e. simple first order explicit integration:

$$\begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \end{pmatrix} = \begin{pmatrix} q_1^{n} \\ q_2^{n} \end{pmatrix} + \Delta t \begin{pmatrix} -1 & 0 \\ 0 & -100 \end{pmatrix}\begin{pmatrix} q_1^{n} \\ q_2^{n} \end{pmatrix} \tag{8.2}$$

The CLF condition for this equation, with CFL number 0.5, is $\Delta t = 0.005$. If we are interested mainly in the behavior of q_1, then this time step is very small. For q_1 alone a time step of $\Delta t = 0.5$ would be fine. Of course, in this simple example we could indeed integrate q_1 with a different time step than q_2, but in more realistic cases one cannot always separate the variable so easily. In such cases one must integrate the entire set of equations together, and a common time step is required, which must then be the smallest of the two, i.e. $\Delta t = 0.005$. For following our variable of interest, q_1, we now have to perform many hundreds of time steps, much more than we would expect judging solely from the slow variation of q_1. In fact, after the initial fast decay of q_2, even q_2 does not change very much anymore, as it reached its equilibrium state already long before q_1 has changed appreciably. So in this example we are taking very small time steps,

much smaller than strictly required for following the variations of q_1 and q_2. It is only required for numerical stability.

8.1.2 Implicit integration: the backward Euler method

An efficient and very stable way to solve this stiffness problem is to use *implicit integration*, which is also often called *backward Euler integration*. We have seen already an example of this for an ordinary differential equation (ODE) in chapter 3. The trick is to use not the values of q at time $t = t_n$ but their future values at time $t = t_{n+1}$ for the evaluation of the right-hand-side of the equation. Since these future values are not yet known at $t = t_n$, this poses a chicken-or-egg problem. This problem can be solved if one writes the set of equations for q^{n+1} such that all the future variables q^{n+1} that were initially on the right hand side are now on the left hand side. For linear equations this can then be written as a matrix equation which can be solved using standard matrix equation solvers.

Let us put our above example in implicit form:

$$\begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \end{pmatrix} = \begin{pmatrix} q_1^{n} \\ q_2^{n} \end{pmatrix} + \Delta t \begin{pmatrix} -1 & 0 \\ 0 & -100 \end{pmatrix} \begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \end{pmatrix} \tag{8.3}$$

where the difference with Eq.(8.2) is only in the last vector where n was replaced by $n+1$. Now let us reorder the $n+1$ variables to the left and all n variables to the right:

$$\left[1 - \Delta t \begin{pmatrix} -1 & 0 \\ 0 & -100 \end{pmatrix} \right] \begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \end{pmatrix} = \begin{pmatrix} q_1^{n} \\ q_2^{n} \end{pmatrix} \tag{8.4}$$

If we define the matrix $\mathbf{M}$ to be:

$$\mathbf{M} = 1 - \Delta t \begin{pmatrix} -1 & 0 \\ 0 & -100 \end{pmatrix} \tag{8.5}$$

and vector $\mathbf{q}$ to be

$$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \tag{8.6}$$

then the above matrix equation is:

$$\mathbf{M}\mathbf{q}^{n+1} = \mathbf{q}^n \tag{8.7}$$

This is the matrix equation we now have to solve at each time step. For our simple example this becomes:

$$\begin{pmatrix} 1 + \Delta t & 0 \\ 0 & 1 + 100\Delta t \end{pmatrix} \begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \end{pmatrix} = \begin{pmatrix} q_1^{n} \\ q_2^{n} \end{pmatrix} \tag{8.8}$$

which can be easily solved:

$$q_1^{n+1} = q_1^{n+1}/(1 + \Delta t) \tag{8.9}$$
$$q_2^{n+1} = q_2^{n+1}/(1 + 100\Delta t) \tag{8.10}$$

The nice property of this solution is that there is no positive Δt for which q_1^{n+1} or q_2^{n+1} becomes negative. When using the explicit Euler method taking Δt larger than the CFL limit will lead to negative values. Implicit integration, as demonstrated here, is therefore *unconditionally stable*, even though for large Δt one should not expect the solution to be accurate. However, suppose we take Δt to be too large for an accurate integration of q_2, this is not too problematic, because we

are anyway interested mainly in q_1 in our example. Could an inaccurate integration of q_2, in more complex examples, compromise the accuracy of q_1? Yes, it could. But typically the faster modes (in this case q_2) quickly reach a semi-steady-state (a quasi-equilibrium), and this state is reached even when taking very large time steps for q_2. The way in which q_2 reaches its quasi-steady state may be inaccurate if one takes too large time steps, but once it reaches it semi-steady state, it stays there and the implicit method gives the right answer for q_2 from that point onward. So as long as we are interested solely in q_1 and we take a time step that is small enough for an accurate integration of q_1, then using an implicit method guarantees stability and accuracy for q_1, even if the problem is more complex than shown above.

So does this mean that the above simple implicit integration method is a simple method that can easily be applied to any problem? The answer is, unfortunately, no. In the above example it was easy to solve the matrix equation because the matrix was diagonal. In more general cases the matrix will not be strictly diagonal, and may be pretty large. For instance, if one has a $\mathbf{q}$ vector of 100 components, and the 100×100 matrix $\mathbf{M}$ has all its elements non-zero, then an analytic solution for $\mathbf{q}^{n+1}$ from $\mathbf{q}^n$ is not practical. One then needs to resort to standard matrix equation solvers.

8.1.3 A standard matrix equation solver: LU decomposition

There are many different methods for solving matrix equations. I can warmly recommend the book by Press et al. *Numerical Recipes in C* or *Numerical Recipes in Fortran*. Here we shall focus on the most popular method, the LU decomposition.

Suppose we wish to solve the following equation:

$$\begin{pmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \tag{8.11}$$

for (x_1, x_2, x_3, x_4). The matrix here is *lower triangular*, meaning that the elements above the diagonal are all zero. For this special case the solution is simple. We start by solving the first equation: $x_1 = y_1/\alpha_{11}$. Once we know this, the second equation becomes $x_2 = (y_2 - \alpha_{21}x_1)/\alpha_{22}$. Since x_1 is known, x_2 is directly obtained by this expression. Next, we find x_3 by $x_3 = (y_3 - \alpha_{31}x_1 - \alpha_{32}x_2)/\alpha_{33}$, etc until all values of x are obtained. Equations of the type 8.11 are therefore trivially solved. Also *upper-triangular* matrix equations are trivially solved in a similar manner:

$$\begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta 23 & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \tag{8.12}$$

Now, suppose we wish to solve the following non-trivial matrix equation:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \tag{8.13}$$

Then unfortunately we do not have such a simple method. We can try to create an upper-triangular or lower-triangular matrix equation out of this by Gaussian elimination. But a more

useful method is to try to write the matrix as a product of a lower-triangular matrix and an upper-triangular matrix:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{pmatrix} \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta 23 & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} \tag{8.14}$$

or in matrix notation:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{8.15}$$

Finding the components of $\mathbf{L}$ and $\mathbf{U}$ for any given matrix $\mathbf{A}$ is not trivial, but numerous off-the-shelf routines are available to make this LU-decomposition of the matrix $\mathbf{A}$, for instance, in the book of Press et al, cited above. But once they are found, the solution to the equation

$$\mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{8.16}$$

is easily found using the above sequential procedure.

The nice property of this LU-decomposition method is that once these two triangular matrices are found, the solution $\mathbf{x}$ can be found for a great number of vectors $\mathbf{y}$ without the need for recomputing the matrices $\mathbf{L}$ and $\mathbf{U}$ all the time.

8.2 A 1-D diffusion equation

Suppose we wish to solve the following diffusion equation

$$\frac{\partial q}{\partial t} - \frac{\partial}{\partial x}\left(D\frac{\partial q}{\partial x}\right) = b \tag{8.17}$$

where D is a diffusion constant. Let us assume D =constant and the grid spacing Δx is also constant. The source term b does not necessarily have to be constant. The forward Euler (i.e. explicit) integration of the equation can be written as:

$$q_i^{n+1} = q_i^n + \frac{\Delta t D}{\Delta x^2}(q_{i+1}^n - 2q_i^n + q_{i-1}^n) + \Delta t b \tag{8.18}$$

The CFL condition for this equation is

$$\Delta t \lesssim 0.5\frac{\Delta x^2}{D} \tag{8.19}$$

This condition is very restrictive for fine grid resolution (small Δx). The short time scale comes about because the highest k modes (with a wavelength twice the grid spacing) have the shortest time scale. It turns out, however, that these high-k modes also damp out quickly, so after a short time they have become irrelevant; we do not need to model them any longer. But these modes are still there and if Δt is larger than the CFL condition these modes explode.

So let us write the equation in implicit form:

$$q_i^{n+1} = q_i^n + \frac{\Delta t D}{\Delta x^2}(q_{i+1}^{n+1} - 2q_i^{n+1} + q_{i-1}^{n+1}) + \Delta t b \tag{8.20}$$

Now sort things:

$$q_i^{n+1} - \frac{\Delta t D}{\Delta x^2}(q_{i+1}^{n+1} - 2q_i^{n+1} + q_{i-1}^{n+1}) = q_i^n + \Delta t b \tag{8.21}$$

We can write this as a matrix equation:

$$\mathbf{A}\mathbf{q}^{n+1} = \mathbf{q}^n + \Delta t b \tag{8.22}$$

The solution would then formally be:

$$\mathbf{q}^{n+1} = \mathbf{A}^{-1}(\mathbf{q}^n + \Delta t b) \tag{8.23}$$

but in practice one never calculates the matrix $\mathbf{A}^{-1}$, nor its LU components. The reason is that the matrix $\mathbf{A}$ is a *sparse matrix*, meaning that only a very limited subset of its elements are non-zero. It requires therefore very little storage space to store the matrix $\mathbf{A}$ is the computer. However, the inverse matrix $\mathbf{A}^{-1}$ is non-sparse. Typically all its elements are non-zero. It would therefore require quite a bit of storage space, and moreover it would require an enormous amount of computational effort to create the matrix $\mathbf{A}^{-1}$.

Instead, there are various methods for solving matrix equations where the matrix is sparse. The above matrix has a special structure: it is *tridiagonal*. A tridiagonal matrix of, say 7×7 has the following form:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & 0 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & c_6 \\ 0 & 0 & 0 & 0 & 0 & a_7 & b_7 \end{pmatrix} \tag{8.24}$$

→ **Exercise:** Prove that, for Eq.(8.21), the matrix $\mathbf{A}$ has a tridiagonal form. Give an expression for a_i, b_i and c_i for $2 \leq i \leq N-1$ (where N is the grid size). At the left boundary require that $q = K$ (for some arbitrary number K) and at the right boundary require that $\partial q/\partial x = 0$.

Once Eq.(8.21) is written in matrix form Eq.(8.22) with $\mathbf{A}$ a tridiagonal matrix, then the solution is relatively simple. Let's assume we have $N = 7$, i.e. 7 grid points, so that we can write this equation in the following way:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & 0 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & c_6 \\ 0 & 0 & 0 & 0 & 0 & a_7 & b_7 \end{pmatrix} \begin{pmatrix} q_1^{n+1} \\ q_2^{n+1} \\ q_3^{n+1} \\ q_4^{n+1} \\ q_5^{n+1} \\ q_6^{n+1} \\ q_7^{n+1} \end{pmatrix} = \begin{pmatrix} r_1^{n+1} \\ r_2^{n+1} \\ r_3^{n+1} \\ r_4^{n+1} \\ r_5^{n+1} \\ r_6^{n+1} \\ r_7^{n+1} \end{pmatrix} \tag{8.25}$$

where $\mathbf{r}$ contains the entire right-hand-side of the equation. One can simply start an elimination procedure:

$$q_1^{n+1} = (r_1 - c_1 q_2^{n+1})/b_1 \tag{8.26}$$

$$q_2^{n+1} = (r_2 - a_2 r_1/b_1 - c_2 q_3^{n+1})/(b_2 - a_2 c_1/b_1) \tag{8.27}$$

$$q_3^{n+1} = ... \tag{8.28}$$

One sees that every new equation solves on variable but introduces a new one. However, once one arrives at $i = N$ then the equation does not introduce any new variables and the equation gives the first actual number. One can then back-substitute everything from $i = N$ to $i = 1$ and the solution is done. This procedure is called the *forward-elimination backward-substitution method*. It works very well and robustly for problems involving tridiagonal matrices.

Once this equation is solved, the q_i^{n+1} values are found and the next time step can be made.

8.3 Diffusion equation in 2-D and 3-D: a prelude

Now let us do the same thing for a 2-D diffusion equation:

$$\frac{\partial q}{\partial t} - \frac{\partial}{\partial x}\left(D\frac{\partial q}{\partial x}\right) - \frac{\partial}{\partial y}\left(D\frac{\partial q}{\partial y}\right) = b \tag{8.29}$$

In discrete implicit form:

$$q_{i,j}^{n+1} = q_{i,j}^{n} + \frac{\Delta t D}{\Delta x^2}(q_{i+1,j}^{n+1} - 2q_{i,j}^{n+1} + q_{i-1,j}^{n+1}) + \frac{\Delta t D}{\Delta y^2}(q_{i,j+1}^{n+1} - 2q_{i,j}^{n+1} + q_{i,j-1}^{n+1}) + \Delta t b \tag{8.30}$$

and sort this:

$$q_{i,j}^{n+1} - \frac{\Delta t D}{\Delta x^2}(q_{i+1,j}^{n+1} - 2q_{i,j}^{n+1} + q_{i-1,j}^{n+1}) - \frac{\Delta t D}{\Delta y^2}(q_{i,j+1}^{n+1} - 2q_{i,j}^{n+1} + q_{i,j-1}^{n+1}) = q_{i,j}^{n} + \Delta t b \tag{8.31}$$

How to write this in a matrix form? First we have to make a 1-D vector $\mathbf{q}$ out of the 2-D arrangement of variables $q_{i,j}$. This can be done in the following way (here: a 4×4 grid):

$$\mathbf{q} = (q_{1,1}, q_{2,1}, q_{3,1}, q_{4,1}, q_{1,2}, q_{2,2}, q_{3,2}, q_{4,2}, q_{1,3}, q_{2,3}, q_{3,3}, q_{4,3}, q_{1,4}, q_{2,4}, q_{3,4}, q_{4,4})^T \tag{8.32}$$

So for an $N_x \times N_y$ grid we get a vector $\mathbf{q}$ of length $N = N_x N_y$. The corresponding matrix $\mathbf{A}$ will then have $N \times N$ components, i.e. $(N_x N_y)^2$ components. Again, most of these components will be 0, so we will want to store these matrix elements in a clever way, like the a, b, c case for the tridiagonal case. In this way we can keep the matrix storage manageable. For a 3-D grid all of this goes similarly, and for an $N_x \times N_y \times N_z$ grid we get a vector $\mathbf{q}$ of length $N = N_x N_y N_z$ and a matrix of $(N_x N_y N_z)^2$ elements, again most of which are 0. But let us stick to the 2-D case for now.

So what is the structure of this matrix? The matrix will be partly tridiagonal, but it will have side-bands too. See Fig.8.1.

→ **Exercise:** What is the location of the side bands in the matrix (i.e. how many rows or colums offset from the true diagonal)?

For this kind of matrix the forward elimination backward substitution method does not work. For this we need iterative solvers for sparse matrices.

8.4 Iterative solvers for sparse matrix equations

There is a large variety of iterative methods for solving sparse matrix equations. Some basic methods are *Jacobi iteration*, *Successive Overrelaxation (SOR)*, *Incomplete LU factorization*, *Gauss-Seidel iteration* etc. These methods are not very efficient, but it turns out that in combination with other methods they still have their use.

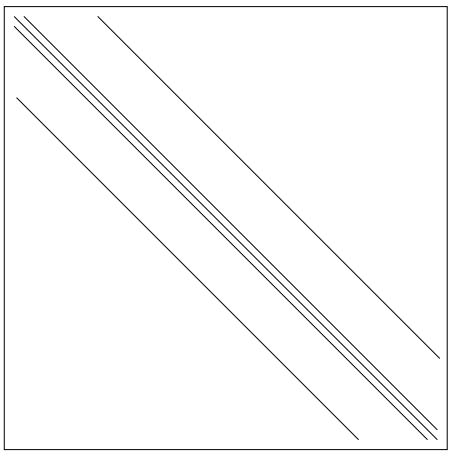

Figure 8.1. Graphical representation of the $\mathbf{A}$ matrix resulting from the 2-D diffusion equation.

8.4.1 Conjugate gradient methods

Here we will focus on a particular class of methods called *conjugate gradient methods* (again we refer here to the book by Press et al. "Numerical Recipes", but also to the book by Ferziger & Peric "Computational Methods for Fluid Dynamics"). Consider the general matrix equation

$$\mathbf{A}\mathbf{q} = \mathbf{b} \tag{8.33}$$

If *and only if* the matrix $\mathbf{A}$ is symmetric and positive definite one can devise the following method. Define a function

$$f(\mathbf{q}) = \frac{1}{2}\mathbf{q} \cdot \mathbf{A} \cdot \mathbf{q} - \mathbf{b} \cdot \mathbf{q} \tag{8.34}$$

At the point where $f(\mathbf{q})$ has a minimum, $\mathbf{q}$ is a solution of the matrix equation. This minimum is located where

$$\nabla f = \mathbf{A}\mathbf{q} - \mathbf{b} = 0 \tag{8.35}$$

proving that indeed this minimum corresponds to the solution of the matrix equation. The trick is now to start with an initial guess and walk toward the point where the minimum is. At each iteration k we make a search direction $\mathbf{p}_k$ and we find the value of a quantity α_k such that $f(\mathbf{q}_{k+1})$ is minimized, where $\mathbf{q}_{k+1}$ is given by:

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \alpha_k \mathbf{p}_k \tag{8.36}$$

So how do we determine $\mathbf{p}_k$? This could be done using the "steepest descent" method. This method is guaranteed to converge, but it can converge very slowly. In particular if the minimum lies in a very long and narrow valley. The *conjugate gradient* method is a version of this method in which the new search direction $\mathbf{p}_{k+1}$ is chosen to be as different from all the previous ones as possible. Let us define $\mathbf{p}_k$ and $\mathbf{p}_l$ too be *conjugate vectors* if they obey:

$$\mathbf{p}_k \cdot \mathbf{A} \cdot \mathbf{p}_l = 0 \tag{8.37}$$

i.e. they are, so to speak, orthogonal with respect to the "metric" $\mathbf{A}$. In the conjugate gradient method each new search direction is required to be conjugate to all previous ones. This choice of $\mathbf{p}_{k+1}$ ensures that if one finds the α_{k+1} such that $f(\mathbf{q}_{k+1} + \alpha_{k+1}\mathbf{p}_{k+1})$ is minimized with respect to α_{k+1} it is also minimized with respect to all previous $\alpha_{l<k+1}$ and $\mathbf{p}_{l<k+1}$. THis means that if

one has $N = N_x N_y N_z$ grid points, then this method is guaranteed to find the exact solution in at most N iterations. In practice, however, one does not want to store all previous search directions, nor do the large amount of work to find a conjugate to a large number of vectors. So in practice the number of successively conjugate search directions is taken to be a limited value, and after that the algorithm is restarted. This does not guarantee convergence in a finite number of steps, but still it converges relatively rapidly.

Another version of this algorithm uses another scalar function to minimize: the length of the residual vector $\mathbf{r}$. This residual is defined as:

$$\mathbf{r}_k = \mathbf{A}\mathbf{q}_k - \mathbf{b} \tag{8.38}$$

The function to minimize is then:

$$f(\mathbf{q}) = \frac{1}{2}\mathbf{r} \cdot \mathbf{r} = \frac{1}{2}|\mathbf{A}\mathbf{q} - \mathbf{b}|^2 \tag{8.39}$$

This is the *minimum residual method*. It works for symmetric matrices, also those which are non-positive-definite.

The above methods work fine, but they are restricted to symmetric matrices. There are several generalizations of this to generalize it do more general matrices. For the above mentioned basic conjugate gradient method there is a *biconjugate gradient* method (BCG method) which does not have this restriction. But it is also not clearly linked to minimizing a function.

For the minimum residual method there is a generalized version called the *generalized minimum residual* method (GMRES). This is a very stable method and does not have any restriction on the matrix, except that it is not degenerate.

8.4.2 Implementation and preconditioning

The nice property of this class of methods, also called *Krylov subspace methods*, is that it involves only multiplications of the matrix $\mathbf{A}$ or $\mathbf{A}^T$ with vectors supplied by the algorithm and the computation of inner products between two vectors supplied by the algorithm. In fact, you can, as a user, supply your own subroutine for multiplying either $\mathbf{A}$ or $\mathbf{A}^T$ with a vector that the conjugate gradient supplies, as well as a subroutine for computing the inner product between two vectors. This leaves all the implementation and all the time-consuming computations in your own hands and you can figure out how to most efficiently compute these matrix-vector and vector-vector products.

This nice property also reveals a weakness of these methods. By definition a matrix product with a vector only propagates information from one grid point to its neighbors. So if one has an $N_x \times N_y$ grid then one cannot hope to get a solution with fewer than $\max(N_x, N_y)$ iterations, and generally it may take up to $\max(N_x^2, N_y^2)$ iterations. The reason is simply that one cell on the left of the grid does not "feel" anything from another cell on the right side of the grid until this information is propagated. Each iteration of the Krylov method propagates information only a single cell, and since information may have to travel multiple times back-and-forth before a solution is found, it can take many iterations.

More mathematically one can say that this matrix is *ill conditioned*. If $\lambda_1 \cdots \lambda_N$ are the eigenvalues of the matrix, then the condition number κ is defined as

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{8.40}$$

If this number is large then the matrix is ill conditioned

One way of improving the situation is to use *preconditioning* of the matrix equation by introducing a preconditioning matrix $\mathbf{\Lambda}$ such that one solves the following equation:

$$\mathbf{\Lambda A \Lambda}^{-1}\mathbf{\Lambda q} = \mathbf{\Lambda b} \tag{8.41}$$

This equation is mathematically identical to the original one, but it is numerically different. If one can find a suitable preconditioning matrix $\mathbf{\Lambda}$, then the convergence can be sped up much. Ideally $\mathbf{\Lambda}$ is as closely similar as $\mathbf{A}$ but still easy to invert. In the Krylov methods what happens in practice is that the algorithm will ask you to solve the equation $\mathbf{\Lambda y} = \mathbf{w}$ for $\mathbf{y}$ for some given vector $\mathbf{w}$. If you take $\mathbf{\Lambda} = \mathbf{1}$, then this is trivial: $\mathbf{y} = \mathbf{w}$ and the method reduces to the non-preconditioned method. If, however, $\mathbf{\Lambda}$ is cleverly chosen, then you can solve the equation each time the algorithm asks you to do so, and the method can speed up a lot. Basically what you need to do is to find a $\mathbf{\Lambda}$ that propagates information quickly across the grid without making it difficult to invert.

8.4.3 Libraries

It is in general not a good idea to try to program one's own matrix equation solver. Libraries of subroutines for doing this have been developed over decades, and it is generally a good idea to look for an appropriate library and use that library as a "black box". Here is a very incomplete list of some known libraries, some of which are freeware, some of which are proprietary software:

- The subroutines associated with the book by Press et al. "Numerical Recipes in C" and/or "Numerical Recipes in Fortran-77".
  `http://www.nr.com/com/storefront.html`
- SixPack: Linear solvers for Finite Volume / Finite Difference equations.
  `http://www.engineers.auckland.ac.nz/~snor007/software.html`
- Hypre: Scalable Linear Solvers.
  `https://computation.llnl.gov/casc/hypre/software.html`
- PETSc: Portable, Extensible Toolkit for Scientific Computation.
  `http://www-unix.mcs.anl.gov/petsc/petsc-as/`

But there are many more libraries available.

8.5 Incompressible fluid equations

Now let us return to the problem of very subsonic fluids. Very subsonic means in fact that one can approximate the system as being *incompressible*. This means that

$$\nabla \cdot \vec{u} = 0 \tag{8.42}$$

and the density ρ=constant. Since the density is assumed to be constant, we do not need to solve the continuity equation anymore. But instead we get a condition on the pressure gradients. Let us look at the momentum equation:

$$\frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla P \tag{8.43}$$

If we take the divergence of this equation, then we obtain:

$$\frac{\partial \nabla \cdot \vec{u}}{\partial t} = -\nabla \cdot (\vec{u} \cdot \nabla \vec{u}) - \frac{1}{\rho} \nabla^2 P \tag{8.44}$$

where we have used $\nabla \rho = 0$. If we say $\nabla \cdot \vec{u} = 0$ then we obtain

$$\nabla^2 P = -\rho \nabla \cdot (\vec{u} \cdot \nabla \vec{u}) \tag{8.45}$$

In index notation, and with use again of $\nabla \cdot \vec{u} = 0$ we obtain

$$\nabla_k \nabla_k P = -\rho \nabla_k u_l \nabla_l u_k \tag{8.46}$$

This means that the condition of incompressibility creates a *Poisson equation for the pressure*. In numerical form this becomes:

$$\frac{\partial}{\partial x_i} \left(\frac{\partial P}{\partial x_i} \right) = -\frac{\partial}{\partial x_i} \left[\frac{\partial (\rho u_i u_j)}{\partial x_j} \right] \tag{8.47}$$

How to solve this? One way to do so is to see the similarity between the Poisson equation in 2-D and stationary solutions of the 2-D time-dependent diffusion equation. So one can use the same methods as above, but then taking $\Delta t \to \infty$. Or one can keep Δt finite or even small enough to do explicit time integration until a steady state is reached. This is, however, very CPU time consuming.

8.5.1 Boundary conditions

Suppose we have flow in a restricted volume (a pipe or a river), then the boundary conditions are such that $\vec{u} \cdot \vec{n} = 0$, where $\vec{n}$ is the normal to the surface. The pressure gradient must be such that this remains true after one time step. Multiplying the momentum equation with $\vec{n}$ gives:

$$\frac{\partial \vec{n} \cdot \vec{u}}{\partial t} = -\vec{n} \cdot (\vec{u} \cdot \nabla \vec{u}) - \frac{1}{\rho} \vec{n} \cdot \nabla P \tag{8.48}$$

which reduces to

$$\frac{1}{\rho} n_k \nabla_k P = -\vec{n}_k u_l \nabla_l u_k \tag{8.49}$$

which can be implemented as boundary condition.

8.5.2 Numerical issues

In solving the Poisson equation for the pressure one must make sure that one uses the same numerical methods for computing the source term of this equation as one uses for the update of the velocities. If one does not do this, the zero divergence cannot be guaranteed. Also one must start from a velocity field that is already divergence free, otherwise the solution of the Poisson equation will not help.