nag®

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Case Study I

Dwight Barkley

Computational Fluid Dynamics

Case Study: Computational Fluid Dynamics

Navier-Stokes Equations

Computational fluid dynamics (CFD) is concerned with solutions of the Navier-Stokes equations. To avoid too many generalities we shall focus on one of the most important cases, the incompressible Navier-Stokes equations:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} &= -\nabla p + \nu \nabla^2 \mathbf{u} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

where $\mathbf{u}$ is a vector field describing the fluid velocity and p is the pressure. If you are not familiar with vector differential operators, the equations are given in component form in the Appendix. The first equation is called the momentum equation and the second the incompressibility constraint and is due to mass conservation. In addition to the Navier-Stokes equations themselves, some domain and boundary conditions must be specified.

In three space dimensions, $\mathbf{u}$ has three components, which we denote by u, v, w, that are functions of x, y, z and time t. In two dimensions, $\mathbf{u}$ has just components u, v, that are functions of x, y and time t.

Numerically solving the Navier–Stokes equations most often means starting from some initial condition $\mathbf{u}(t = 0)$ and evolving this initial field forward in time. This is generally referred to as simulating the equations.

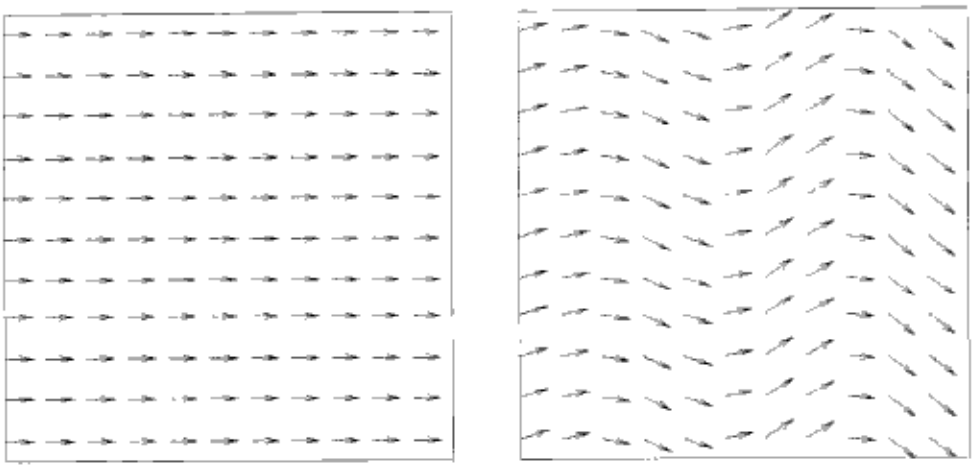

Figure 1: Illustration of a 2D fluid simulation. The left shows the initial vector field (initial condition) and the right the vector field after evolution over some time t.

Examples

Let us right away look at a few real examples of the types of simulations one typically performs in CFD.

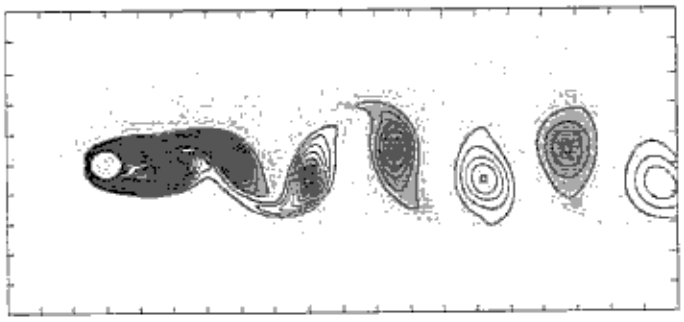

Figure 2: **2D cylinder wake**: Flow is from left to right. Solution is visualized in terms of vorticity $\omega = \nabla \times \mathbf{u}$. Simulation by D. Calhoun.

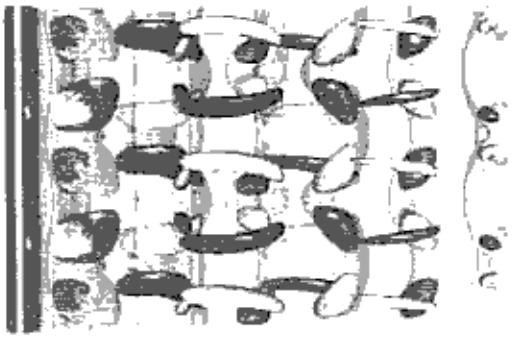

Figure 3: **3D cylinder wake**: Flow is from left to right; the axis of the cylinder is vertical. Colors show isosurfaces of streamwise vorticity. Transparent tubes are isosurfaces of spanwise vorticity. Simulation by H. Blackburn.

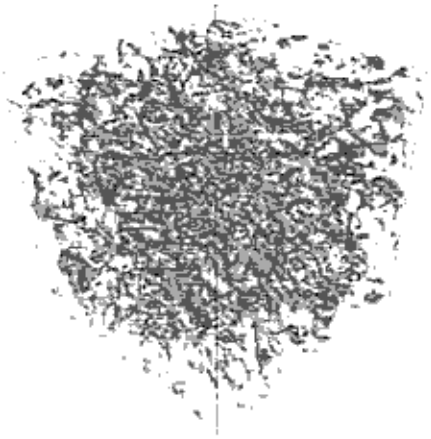

Figure 4: **Turbulence**: Isosurfaces of one vorticity component in a 3D simulation of decaying turbulence. From *www.innovative-cfd.com*

Overview

- The main purpose of this case study is to highlight the role of core numerical algorithms and libraries in computational fluid dynamics. You should gain some sense of where the computational complexity lies and what types of core subroutines are required and why.
- The case study must necessarily have a limited focus. It can only cover the most basic features of Navier-Stokes simulations - it is not meant to be a replace a proper course on CFD.

The following is a fairly typical picture representing what a CFD code might look like and will be a useful reference in what follows. Many of the points are rather general and apply equally to other large-scale scientific applications.

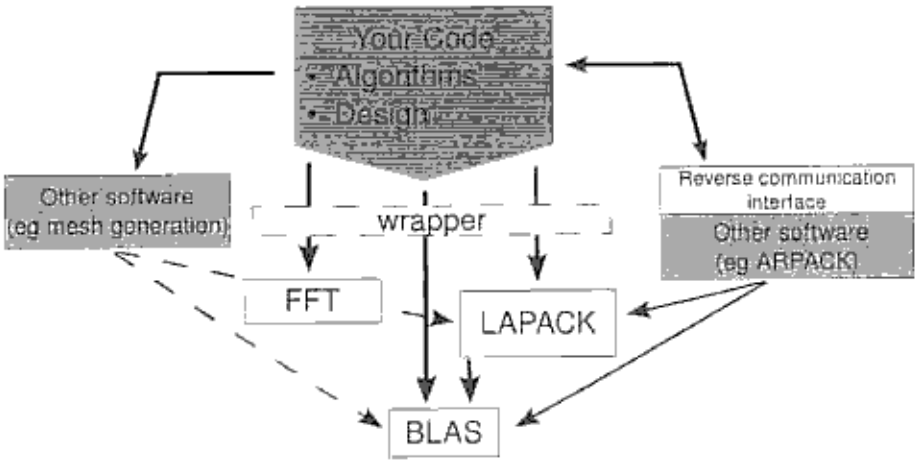

Figure 5: Illustration of what a typical CFD code might look like.

- The high-level code is based on efficient algorithms for the set of problems to be solved. Design is dictated by ease of maintenance, reliability, readability etc.
- The bulk of the actual floating-point computations are passed down to low-level libraries, possibly in several stages. At the bottom-most level the efficiency of actual operations is paramount.
- This approach leads to both computational efficiency and also portability across platforms.

To see in more detail the connection between core algorithms and simulations of the Navier-Stokes equations, we present a brief summary capturing essential ingredients of typical approaches.

Spatial Discretization

In all cases it is necessary to discretize in space. We must represent continuous functions of space, e.g. u and p, by a finite number of quantities and also obtain the form of the differential operators acting on these discrete quantities. There is a tremendous variety of approaches to spatial discretization going by names

such as: **finite differences, finite elements, finite volumes, spectral, pseudospectral, spectral-elements**, and others. Moreover, the discretization may be applied to variables other than $\mathbf{u}$ and p, e.g. streamfunction and vorticity.

For our purposes it is sufficient to consider two basic cases and to restrict to 2D for ease of presentation.

1. **Finite-differences.** Discretizing $\mathbf{u}$ on a two-dimensional $N \times N$ grid (x_j, y_k) leads for example to

$$U_i = u(x_j, y_k) \quad V_i = v(x_j, y_k) \quad \text{where } i = j + Nk$$

Thus the vector field $\mathbf{u}$ is represented by $2 \times N^2$ values. For simplicity we write the discrete field as a single vector $\mathbf{U}$, whether or not the discrete field will be stored as one long vector in the computer.

As we have seen in examples in the Maths Lectures, derivatives can be approximated by combinations of grid values. These are called finite difference approximation.

The essential point is

$\mathbf{u} \to \mathbf{U}$	Components of $\mathbf{U}$ are field values at grid points
$\nabla \mathbf{u} \to \mathbf{D}_1 \mathbf{U}$	Derivatives are approximated by linear combinations
$\nabla^2 \mathbf{u} \to \mathbf{D}_2 \mathbf{U}$	of values of $\mathbf{U}$ and are equivalent to matrix multiplication

2. **Pseudopectral.** $\mathbf{u}$ is discretized not by a set of values on a grid, but by a set of coefficients in an expansion

$$u(x, y) = \sum_m^M \sum_n^N a_{mn} \phi_{mn}(x, y)$$

The $\phi_{mn}(x, y)$ are either polynomials or trigonometric functions. There are a finite number of amplitudes a_{mn} representing each field. Collectively we write the full set of amplitudes for all fields as $\mathbf{a}$.

Derivatives become linear combinations of amplitudes and we write $\nabla u \to \hat{\mathbf{D}}_1 \mathbf{a}$, $\nabla^2 u \to \hat{\mathbf{D}}_2 \mathbf{a}$. For the same computational cost, these derivative operations are much more accurate than the equivalent finite-difference versions.

In the pseudospectral approach one uses both the amplitudes and a corresponding set of grid values related by some fast transform, almost always based on the FFT.

The essential point is

$$\mathbf{u} \to \begin{cases} \mathbf{a} & \text{Amplitudes} \\ \mathbf{U} & \text{Grid values} \end{cases}$$
$$\mathbf{a} \longleftrightarrow \mathbf{U} \quad \text{by fast transform}$$

One uses amplitudes where they work best and grid values where they work best. The fast transform is essential.

In either case the fields become (long) vectors. One might construct other data structures, but the more closely these can be tied to basic BLAS operations the better.

Time Stepping

It is convenient to denote the value of $\mathbf{U}$ at time step n by $\mathbf{U}^n$. $\mathbf{U}^0$ is the initial condition. A simulations is viewed as follows:

$$\begin{array}{ll} \mathbf{U}^0 & \leftarrow \text{set initial condition} \\ \downarrow & \text{advance forward } \triangle t \\ \mathbf{U}^1 & \\ \downarrow & \text{continue} \\ \vdots & \\ \mathbf{U}^n & \end{array}$$

We do not in practice allocate memory for and store all of $\mathbf{U}^1, \mathbf{U}^2 \cdots, \mathbf{U}^n \cdots$. We will almost surely need to store a few time slices, but otherwise this this is only notational.

There are 3 sub-problems to time marching which correspond to the various terms in the Navier–Stokes equations. We discuss these one by one.

viscous term

Consider time-stepping just the viscous part of the problem:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u}$$

This equation is known as the diffusion, or equivalently, the heat equation. It is irrelevant for the present discussion whether or not we consider the vector quantity $\mathbf{u}$ or just a component of it. Using our spatial discretization and taking the simplest possible time discretization of this equation we have

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\triangle t} = \nu \mathbf{D_2} \mathbf{U}^n$$

This is forward Euler time stepping and gives the following rule for going from time step n to time step $n+1$

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \nu \triangle t \mathbf{D_2} \mathbf{U}^n$$

While this is a perfectly valid method, it is neither very accurate (it is only first order in $\triangle t$) nor is it stable for very large $\triangle t$. It is almost never used in practice, although it is useful for testing purposes and should not be overlooked in this regard.

There are several possible improvements. Here we consider the Crank-Nicolson method based on the following discretization of the diffusion equation

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\triangle t} = \nu \mathbf{D_2} \left(\frac{\mathbf{U}^n + \mathbf{U}^{n+1}}{2} \right)$$

Putting all terms with $\mathbf{U}^{n+1}$ on the left-hand-side gives:

$$(\mathbf{I} - \frac{\nu \triangle t}{2} \mathbf{D_2}) \mathbf{U}^{n+1} = (\mathbf{I} + \frac{\nu \triangle t}{2} \mathbf{D_2}) \mathbf{U}^n$$

which after solving for $\mathbf{U}^{n+1}$ gives:

$$\mathbf{U}^{n+1} = (\mathbf{I} - \frac{\nu \triangle t}{2} \mathbf{D_2})^{-1} (\mathbf{I} + \frac{\nu \triangle t}{2} \mathbf{D_2}) \mathbf{U}^n$$

The last two expressions are equivalent. The second is more common, while the first emphasizes that $\mathbf{U}^{n+1}$ must be found as the solution of a (large) linear system. The equation is equivalent to the Helmholtz equation discussed in Maths Lecture II. Note that the matrix $A = (\mathrm{I} - \frac{\nu \triangle t}{2}\mathrm{D}_2)$ does not change during the simulations, hence it need be factored only once in a preprocessing step before any timesteps are taken.

Any reasonable CFD code for the incompressible Navier–Stokes equations will use implicit time stepping for the viscous term (whether or not Crank-Nicolson) and hence require something equivalent to the solution of a large linear system of equations every time step. This implicit timestepping, and the pressure term discussed next, are very much at the core of the difficulties and computational expense of CFD.

pressure term

There are nearly as many ways to treat pressure as there are ways to simulate the Navier–Stokes equations. The following, nevertheless, gives a flavor of what is involved. Take the divergence $\nabla \cdot$ of the momentum equation

$$\nabla \cdot \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} \right)$$

Using the divergence condition $\nabla \cdot \mathbf{u} = 0$, this gives

$$\nabla \cdot (\mathbf{u} \cdot \nabla)\mathbf{u}) = -\nabla^2 p$$

or

$$\nabla^2 p = -\nabla \cdot (\mathbf{u} \cdot \nabla)\mathbf{u})$$

The pressure thus satisfies a Poisson equation (known as the pressure Poisson equation) where the source term or the right-hand side comes from the nonlinear term. When discretized, this will again result in a linear system of equations. On the surface this looks about the same as the Helmholtz equation for the viscous term, but in many implementations the Poisson problem is much tougher numerically. The Helmholtz equation in the viscous time stepping is a perturbation of the identity and is very well conditioned; this is not the case for the Poisson problem:

$(\mathrm{I} - \frac{\nu \triangle t}{2}\mathrm{D}_2)$	Helmholtz Matrix
D_2	Poisson Matrix

Very briefly, the essence of how this folds into the overall time stepping is as follows: after the nonlinear term is stepped (as discussed next) the pressure is computed by solving a pressure Poisson equation and then ∇p is subtracted from the right-hand-side of the Navier–Stokes equations. The process: compute p and subtract ∇p, is sometimes call pressure projection and is written as a projection Π.

nonlinear term

This is the most interesting term in many respects, but for grid-based methods it is often the easiest to deal with. Consider the equation for just the nonlinear term.

$$\frac{\partial \mathbf{u}}{\partial t} = \mathrm{N}(\mathbf{u})$$

where

$$\mathrm{N}(\mathbf{u}) = \quad (\mathbf{u} \quad \nabla)\mathbf{u}$$

An implicit method here would require somehow inverting the nonlinear term. Moreover, because the term has only first order spatial derivatives, the stability issue is not as great as for the viscous term. Hence explicit methods are generally used and typically involve past time information (multi-step method). The simplest such scheme which is second order in $\triangle t$ is the Adams-Bashforth scheme which gives $\mathbf{U}^{n+1}$ explicitly from the current $\mathbf{U}^n$ and past $\mathbf{U}^{n-1}$ values

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\triangle t}{2}\left\{3\mathbf{N}(\mathbf{U}^n) - \mathbf{N}(\mathbf{U}^{n-1})\right\}$$

Summarizing for a grid based method

$$\boxed{\mathbf{U}^{n+1} = (\mathbf{I} - \frac{\nu\triangle t}{2}\mathbf{D}_2)^{-1}\left\{(\mathbf{I} + \frac{\nu\triangle t}{2}\mathbf{D}_2)\,\mathbf{U}^n + \mathbf{\Pi}\cdot\left[\frac{\triangle t}{2}\left(3\mathbf{N}(\mathbf{U}^n) - \mathbf{N}(\mathbf{U}^{n-1})\right)\right]\right\}}$$

where $\mathbf{\Pi}$ represents the contribution from the pressure term.

The main computational issues are the implicit time stepping of the viscous term and the determination of the pressure. We have completely ignored the complicated issues associated with boundary condition.

Considerations for Spectral Methods

In just a few words, a pseudospectral time step can be written pretty much the same as a grid based time step

$$\mathbf{a}^{n+1} = (\mathbf{I} - \frac{\nu\triangle t}{2}\hat{\mathbf{D}}_2)^{-1}\left\{(\mathbf{I} + \frac{\nu\triangle t}{2}\hat{\mathbf{D}}_2)\,\mathbf{a}^n + \mathbf{\Pi}\left[\frac{\triangle t}{2}\left(3\mathbf{N}(\mathbf{U}^n) - \mathbf{N}(\mathbf{U}^{n-1})\right)\right]\right\}$$

For simplicity, consider a Fourier pseudospectral method. The matrices $(\mathbf{I} - \frac{\nu\triangle t}{2}\hat{\mathbf{D}}_2)$ are then diagonal and easily dealt with. The computational issue is, as we have written the scheme, that the nonlinear term contains grid values $\mathbf{U}$ and not amplitudes. The evaluation of the nonlinear term from the amplitudes requires a Fourier transform and its inverse.

$$\mathbf{N}(\mathbf{a}^n) = FT\left[\mathbf{N}(FT^{-1}\,\mathbf{a}^n)\right]$$

Hence one sees that efficient Fourier transforms are essential for such a method.

Afternoon Workshop

This workshop will focus on simulating Burger's equation

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

which is like a 1D version of the Navier-Stokes equations, but does not contain pressure. The numerical methods for solving Burger's equation are like those for the full Navier-Stokes equations. Your goal will be to write basic matrix operations, using calls to BLAS and LAPACK, needed for implicitly timestepping this equation.

You will start with a PDE solver based around the following very typical design:

driver.cpp: handles I/O, problem set up, allocation of memory (at least for the solution variables, perhaps not for the hidden variables needed by the stepper, etc), and in contains the main time-stepping loop.

stepper.cpp: performs the time stepping. It useful to keep this as modular as possible so that the operators can be re-combined as desired.

analyzer.cpp: computes and outputs diagnostics. Much of the testing code can go here to keep the stepper and driver relatively clean.

ezplot: Library for plotting solutions in real time and allowing interaction with the user.

I have separated out all the BLAS and LAPACK calls to separate files matrix.c and matrix.f90. (This was a bit of a hack so please do not pay close attention to the design of the current stepper.cpp.) You will work almost exclusively with one of the two files matrix.c or matrix.f90 depending on your preferred language.

Appendix: Navier-Stokes Equations in Component Form

The Navier-Stokes equations in Cartesian coordinates (x, y, z) are:

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} &= -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \\
\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} &= -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) \\
\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} &= -\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} &= 0
\end{aligned}$$

where u, v, w are the three components of the velocity vector $\mathbf{u}$.

Appendix: General Suggestions for Testing

- **Test parts of your program that cannot possibly be wrong.**
- Test scaling.

  Even without exact solutions one can test convergence rates. Use a very high resolution solution as "exact" even if it may not be so.
- If possible test linear and nonlinear part of the code separately.
- Linear Operators.

  Generally straightforward but here are some hints.
  - Build up your operators in pieces that can be tested independently if necessary.
  - It is usually easy to choose some test functions and verify scaling of operators:

    For example, $(\mathbf{1} + \triangle t\mathbf{D}_2)\mathbf{V}$ converges to $(1 + \triangle t\frac{\partial^2}{\partial x^2})v(x)$ with the correct scaling for the finite-difference approximation $\mathbf{D}_2$.

  - Verify that all linear operators are linear.
    That is $(\mathbf{I} - \triangle t\mathbf{D_2})^{-1}\ (\mathbf{I} + \triangle t\mathbf{D_2})$ ought to act linearly on $\mathbf{U}$.
  - Verify matrix inverses. Note that $(\mathbf{I} + \triangle t\mathbf{D_2})$ and $(\mathbf{I} - \triangle t\mathbf{D_2})^{-1}$ are inverses of one another apart from the sign of $\triangle t$. Changing $(\triangle t)$ to $(-\triangle t)$ in one of the two places should result in the identity:
    $$\mathbf{U} = (\mathbf{I} - (-\triangle t)\mathbf{D_2})^{-1}\ (\mathbf{I} + \triangle t\mathbf{D_2})\mathbf{U}$$
    for any $\mathbf{U}$. This will verify that the explicit and implicit parts of a code are at least consistent. **I strongly recommend this test.**

- **Exact solutions.**

  Exact solutions are great, but even without one you can (usually) test in at least two ways.

  - Change the problem.
    Given a nonlinear equation
    $$\frac{\partial u}{\partial t} = \nabla^2 u + \mathcal{N}(u),$$
    choose a suitable test function u_T for which one can analytically compute
    $$g = -(\nabla^2 u_T + \mathcal{N}(u_T)).$$
    Add this *constant forcing function* to the right-hand-side and solve the PDE:
    $$\frac{\partial u}{\partial t} = \nabla^2 u + \mathcal{N}(u) + g.$$
    u_T will necessarily be an exact steady solution to this equation.
  - Use your pencil or your favorite algebraic manipulation package.
    Choose a suitable function v and work out the effect of one time step on this function. Verify that the code does as expected over one time step.
    Often very trivial functions such as linear or constant functions can provide limited testing. These are surprisingly good at finding mistakes in boundary conditions.

- **Benchmarks.**

  Test against benchmarks and other code if available.

- **Leave testing code in place.**

  You will need if more often than you think. Keep in a separate file if you wish.

  For testing strive for complete clarity, not cleverness. Computational cost is not an issue for testing.

  Produce a documented test suit which can be easily re-run frequently.

- And finally: **Test parts of your program that cannot possibly be wrong.**