

Tutorial: A Flow-Problem Solver

J. Antoon van Hooft (j.a.v.hooft@gmail.com)

12-Aug-2023

Contents

Code a simple flow solver	3
Pre-requisites	3
Chapter 0: systems check	4
1. The equations for fluid motion	5
The goal of a flow solver	5
The equations for fluid motion	5
The goal of <i>our</i> flow solver	6
2. The method of lines	7
Practice with an example	7
3. Finite differences	12
A test for finite differencing	14
The code in <code>solver.c</code> is getting messy...	15
4. The advection term	18
Tracer advection	18
Computing the advection term	18
5. Antre'acte: Visualization	20
The <code>.ppm</code> file format	20
Movie maker	21
6. Testing the <code>tracer_advection()</code> function	24
A qualitative test	24
(1) A quantitative test	28
(2) A Well-behaved solver?	28
7. The viscous term	30
A stability criterion	32
Testing	32
8. Antre'acte: A Poisson-problem solver	35
Poisson's equation	35
A numerical solving strategy	35
A criterion to stop sweeping	38
Solving for $\nabla\phi$	39

9. The pressure-gradient force	41
A projection method	41
Chorin's projection method	42
10. A Navier-Stokes problem solver	45
A time stepping function	45
Two tests	46
Congratulations!	52
11. Some improved functionalities	53
(1 and 2) A tracer field and a buoyancy field	53
(3) The User interface	53
Finally,	55
12. A gallery of fluid motion	56
A Kelvin-Helmholtz instability	56
Decaying two-dimensional turbulence	57
Colliding dipolar vortices	58
Vortex rebound from a wall	59
Rising warm plume	61
A Rayleigh-Taylor instability	62
13. The good, the bad and the ugly	64
The good	64
The bad	64
The ugly	64
What do other solvers do different?	64

PDF note There exists a web-version of this document: https://www.antoonvanhooft.nl/min_ns/intro. Apart from formatting differences, the movies shown there are replaced here with images of their last frame.

Code a simple flow solver

Computational analysis of flow problems has become an important piece of equipment in the toolbox for science and engineering. The problem of finding approximate solutions to the equations for fluid flow may be reduced to evaluating a numerical recipe, following from a suitable discretization for the problem. These pages aim to provide a tutorial-styled course on numerical solvers for flow problems. It is not only a step-by-step guide on how to write a flow solver in the C-programming language, also, a discussion on the various solver-design choices is included. C is chosen as it is the perfect didactic language for learning how to program due to its explicit and strict syntax.

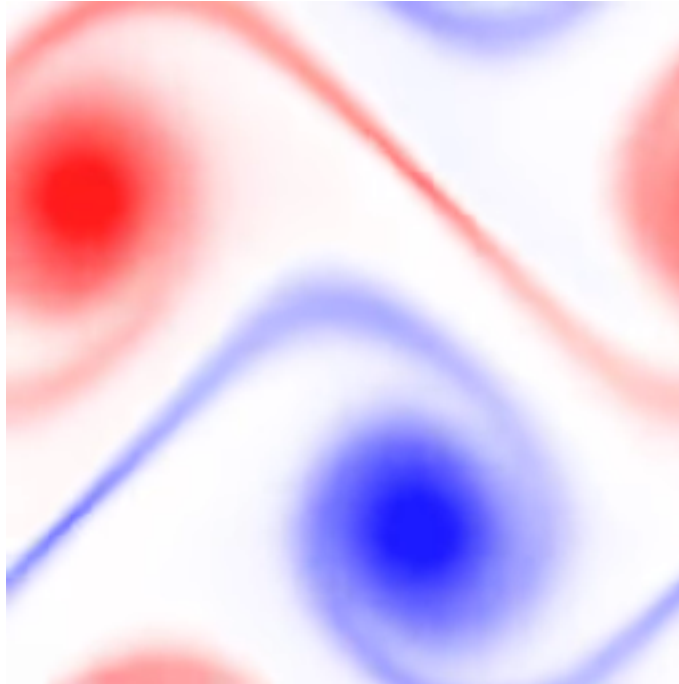


Figure 1: An example result of our efforts; a movie showing the flow evolution by drawing the vorticity field over time

Pre-requisites

Apart from time, interest and a good spirit, a few easier-to-come-by assets are needed to follow along with this course. First, you will need to have a computer that provides you with a C compiler. On a Linux system you may open a terminal-emulator (terminal) and type

```
$ gcc -v
```

and see something like,

```
Using built-in specs.
```

```
...
```

```
gcc version 8.3.0 (Debian 8.3.0-6)
```

Otherwise you can install it on a Debian-based distro,

```
$ sudo apt install gcc
```

If you do not know what the above means, please consult the internet and search for “Install C compiler

on *your OS*", where you replace the italicized text (*your OS*) with the name of your operating system. From hereon, I will assume the C compiler is called `gcc`, although (like always) worthwhile alternatives exists. Furthermore, a so-called text editor is needed. This is a program that lets you write a plain text file. Examples include `atom`, `vim` and `emacs`. The course will mainly go between writing in the editor and typing commands in a terminal. If you prefer a different workflow (e.g. using an IDE), you may need to translate some instructions.

Chapter 0: systems check

This zeroth chapter introduces (and tests) the workflow. You may skip it if you are already familiar with C programming. Open your text editor and make a new file that we will call `hello.c`. Type,

```
#include <stdio.h>

int main() {
    puts ("This is not a flow solver");
}
```

The program code consists of an `#include` directive which tells the C compiler to load the standard input-output header file (`<stdio.h>`), a function declaration (`main()`), a corresponding code block that is marked by curly braces (`{ .. }`) and a function call to the “put string” function (`puts`), that takes a text string as an argument between the round braces (`(...)`). Although the C compiler knows that `puts` only takes a string as an argument, we need to mark our input string with the double quotation marks (`"..."`). Now we can save and exit the editor and come back to the terminal, where we compile our code. Make sure you are in the same folder as where you keep `hello.c`,

```
$ gcc hello.c
```

The absence of error messages marks a successfully run command. If you do get an error, read it carefully and try to fix it. The generated executable can be run from the command line as well,

```
$ ./a.out
```

Any C executable starts with executing the `main()` function, so pretty soon you should see appear,

```
This is not a flow solver
```

on your terminal screen. Great! Lets move on ...

Continue to chapter 1

1. The equations for fluid motion

The goal of a flow solver

What does it mean to “solve” a flow problem? Well, it means that we find a vector flow field that satisfies (i.e solves) the Navier-stokes equations for a specific set of boundary conditions (in both time and space). The pressure field in the fluid may also be regarded as a part of the solution. This aspect is covered in a later chapter. Having access to the full four-dimensional (4D, space + time) flow field allows to compute interesting flow statistics like wind loads, mixing characteristics, flow transport properties, etc.



Figure 2: Example taken from the Basilisk flow solver by Stephane Popinet. Atomizing spray simulations are useful when designing fuel injection systems. Here, the droplet-size distribution is an interesting statistic to help characterize spray pattern efficiency

The equations for fluid motion

In order to achieve an algorithmic solver for flow problems, we must first have a mathematical description of fluid flows. A central result from the classical mechanics era are the Navier-Stokes equations for incompressible flows. In vector form (i.e. it has multiple components) *they* read,

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\frac{1}{\rho} \nabla p^* + \nu \nabla^2 \mathbf{u},$$

with the constraint that,

$$\nabla \cdot \mathbf{u} = 0.$$

Here, $\mathbf{u}(\mathbf{x}, t)$ is the velocity vector as a function of space $\mathbf{x} = \{x, y, z\}$ and time t , ρ is the fluid's density, ∇ is the gradient operator, p^* the (scalar) pressure field in the fluid and ν is the fluid's kinematic viscosity. In this course we limit ourselves to solving for flows of incompressible fluids with a constant density and viscosity parameter. This leads us to introduce a *modified pressure*, $p = \frac{p^*}{\rho}$, so that we can forget about the density of the fluid.

because the computer does not understand vector mathematics, it will prove useful to decompose the equations into their scalar components. For the first velocity component (u_x) we write,

$$\frac{\partial u_x}{\partial t} + u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} + u_z \frac{\partial u_x}{\partial z} = -\frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2} \right),$$

and the other components follow analogous.

The goal of *our* flow solver

In science and engineering practice, efficient and feature-rich flow solvers are required. It appears that for each application, the discretization style, grid structure and algorithms are different. This indicates that there is not a single recipe to solve flow problems and there is no consensus on what is best. Because our goal is more didactical in nature than gaining fame by rivaling solvers such as Openfoam, our solver will aim to be “minimalistic”. Where we will prefer coding simplicity over features and the solver's speed performance. However, we will also write our solver in a modular fashion, separating its components so they may be improved later. The final chapter includes some suggestions.

Continue to Chapter 2

2. The method of lines

The Navier-Stokes equations are an example of a so-called partial differential equation: The solution exists in a multi-dimensional parameter space, and as such the equations contain various corresponding partial derivatives. In the previous chapter we can indeed see derivatives with respect to x, y, z and t . For our solver we choose to delineate between the spatial dimensions and the time dimension for two important reasons:

- The time dimension is the only one associated with causality.
- The equations are isotropic for the group x, y and z (i.e. they maybe rotated)

The actual separation in the treatment of the variables is formalized by the so-called *method of lines*. Consider a dummy variable $\phi(t, x_1, x_2, \dots, x_n)$ that satisfies an equation of the form,

$$\frac{\partial \phi}{\partial t} = \mathcal{L}(\phi),$$

where \mathcal{L} is some differential operator that does *not* include differentiation with respect to t . Because time and physical causality are related concepts, it makes sense to solve for ϕ as an initial value problem. Here $\phi(t = 0, x_1, x_2, \dots, x_n)$ is prescribed as an initial condition, and the equation can be solved by using a time-integration method which advances the solution at the current time (ϕ_n) to the next point in time (ϕ_{n+1}) which differ by a timestep size Δt . It is important to note that the continuously evolving field ϕ is now *discretized* as it only exists on a finite number of points in time. This also means that the proper solution ϕ is fundamentally different from the discrete solution ϕ_n , and this introduces the concepts of convergence, consistency and stability.

Consistency

The timestep parameter is artificially introduced. It is important that for increasingly smaller timesteps, the discretization error (due to truncation) of the numerical schemes vanishes.

Stability

The inevitable errors introduced by discretization should not grow in time.

Convergence

By reducing $\Delta t \rightarrow 0$, the number of timesteps (and effort) required to obtain a solution at some given time goes to infinity. As such, we hope to see the solution *converges* towards an asymptotic solution with increasing effort. This asymptotic solution should match the solution to the original equation at the discrete intervals.

Practice with an example

For many non-linear problems (like flow problems) the above is hard to formalize in numerical algorithms. For example, it is not even known if the Navier-Stokes equations have smooth solutions for the general case. So we will have to think of test cases to gain confidence in our digitally-generated data.

For now, we start with an example problem, namely,

$$\begin{aligned}\frac{dx_1}{dt} &= -x_2, \\ \frac{dx_2}{dt} &= x_1.\end{aligned}$$

with initial conditions $x_1(t=0) = 1$ and $x_2(t=0) = 0$. It can be verified that the solution reads,

$$\begin{aligned}x_1 &= \cos(t), \\x_2 &= \sin(t).\end{aligned}$$

Which describes a circular trajectory. Lets start coding a time integrator for this problem in a file called `circle.c`. We will need a few functions defined in some system header files. We begin with,

```
#include <stdio.h>
#include <math.h>

int main() {

}
```

Whoa! those are a lot of lines of code, with ample of room for typos. It is important to check your syntax early on, as spotting a bug becomes harder as we proceed.

```
$ gcc circle.c
$ ./a.out
```

Lets add some variables, including the final time $t_{end} = 2\pi$, when x_1 and x_2 have returned to their original positions.

```
...

int main() {
    double x1 = 1, x2 = 0;
    double Nsteps = 50, tend = 6.2832; // ~2*pi
    double dt = tend/Nsteps;
}
```

The ... dots are added to hint at the omitted code (the `#include` directives in this case). Any errors? It is time for an important design choice, *How do we advance the solution in time?*. For the sake of minimalism, we choose the forward Euler method:

$$\phi_{n+1} = \phi_n + dt\mathcal{L}(\phi_n).$$

If we wrap that in a time loop,

```
...
    double dt = tend/Nsteps;
    for (int i = 0; i < Nsteps; i++) {
        double tmp_x1 = x1;
        x1 = x1 - dt*x2;
        x2 = x2 + dt*tmp_x1;
    }
...
```

Note that we needed to store the value for $x_{1,n}$ because it would have been at stage $n+1$ at the time of updating x_2 , which would not correspond to the Forward Euler method. Compiling and running this code does not give us any feedback. As such, we print the solution:

```
...
    x2 = x2 + dt*tmp_x1;
    printf ("%g %g\n", x1, x2);
```



```
}  
...
```

Running this code produces

```
$ ./a.out  
1 0.125664  
...  
...  
1.4787 -0.0484434
```

We can store the output in a file for analysis

```
$ ./a.out > solution
```

Now we can plot the data using Gnuplot.

```
$ gnuplot  
...  
gnuplot> plot 'solution'
```

Which should show you a bunch of markers. We can add the analytical solution

```
gnuplot> plot sample [t=0:2*pi] '+' using (cos(t)) : (sin(t)) t 'circle' w l lw 2,\  
'solution'
```

It appears that the numerical solution is moving away for the analytical solution. For a more presentable plot, use the following gnuplot code

```
set size square  
set xlabel 'x_1'  
set ylabel 'x_2'  
set grid  
plot sample [t=0:2*pi] '+' using (cos(t)) : (sin(t)) t 'circle' w l lw 2,\  
'solution'  
q
```

where the last command closes Gnuplot.

Finally, we check the convergence of this method for increasing N_{steps} , by wrapping the time loop in a new loop. Also at $t = t_{\text{end}}$, the distance to the analytical solution is computed and printed. The total code note is:

```
#include <stdio.h>  
#include <math.h>  
  
int main() {  
    double tend = 6.2832; // ~2*pi  
    int Nsteps_max = 4000;  
  
    // Convergence loop  
    for (int Nsteps = 50; Nsteps < Nsteps_max; Nsteps *= 2) {  
        double dt = tend/Nsteps;  
        double x1 = 1, x2 = 0;  
        // Time loop  
        for (double t = 0; t <= tend; t += dt) {  
            double tmp_x1 = x1;
```

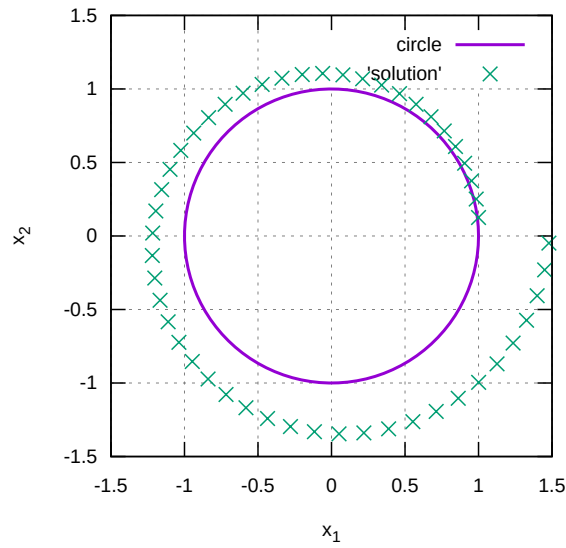


Figure 3: The result stored as svg

```

        x1 = x1 - dt*x2;
        x2 = x2 + dt*tmp_x1;
    }
    // Errors logging
    double error1 = x1 - 1, error2 = x2 - 0;
    double L2 = sqrt(error1*error1 + error2*error2);
    printf ("%d %g\n", Nsteps, L2);
}
}

```

Lets give it a go:

```

$ gcc circle.c
Some error message referencing `sqrt`

```

The compiler (or rather the linker) may throw an error, because it does not know how to compute the square root (`sqrt()`). We must link the math library during compilation, like so;

```

$ gcc circle.c -lm
$ ./a.out > errors

```

Now we can plot the data on a doubly-logarithmic scale, and reveal the dependency of the L_2 error on the number of steps.

```

$ gnuplot
...
gnuplot> set logscale xy
gnuplot> plot 'errors', 20*x**(-1) lw 2 t 'x^{-1}'

```

Which may produce something like:

Because the so-called *scaling* of the error with $L_2 \propto N^{-1}$, fits the data well, we can conclude that the

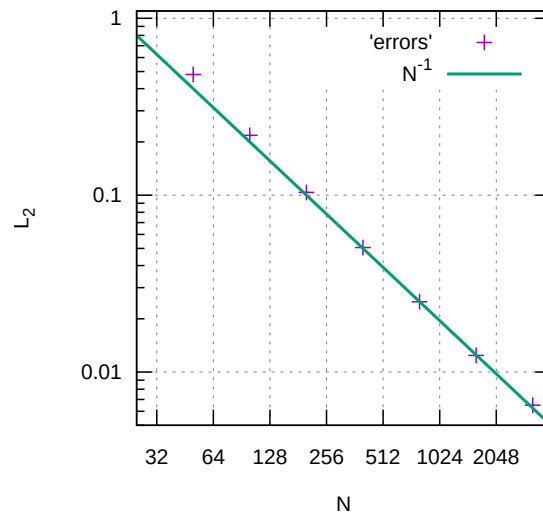


Figure 4: A formatted plot saved as an `.svg` file reveals a First-order convergence rate

solution indeed converges. More specifically, the forward Euler method is *first-order accurate*, which corresponds with the literature.

Continue with Navier-Stokes related programming in chapter 3.

3. Finite differences

Like with time, the spatial variability of the solution also needs to be represented discretely. The purpose of the spatial discretization is two fold: it defines the data structure for the variables that are being time integrated and it enables to compute the right-hand-side for the time-integrator problem,

$$\frac{\partial \phi}{\partial t} = \mathcal{L}(\phi).$$

This entails to evaluate the various spatial derivatives that appear in the Navier-Stokes equations. We choose to adopt the so-called *finite difference* representation of our fields: Any smoothly varying scalar field (e.g. ϕ) is then stored as a collection of its *point values*.

$$\phi_n = \{\phi(t_n, \mathbf{x}_1), \phi(t_n, \mathbf{x}_2), \dots, \phi(t_n, \mathbf{x}_N)\}.$$

We will only consider a two-dimensional domain, ($\mathbf{x} = \{x, y\}$) and the points are distributed on a regular and square Cartesian grid. In principle, each field (i.e. u_x, u_y and p) can be stored for its own set of point locations that differs from the others. However, it is most easy (and minimalist) to have a so-called co-located grid, where the point locations for all variables coincide.

Next, the concept of indexing is introduced, A domain of size $L_0 \times L_0$ can be discretized with $N \times N$ points, giving a grid spacing of $\Delta = L_0 N^{-1}$. Each entry corresponds to a location in physical space, but also in the computer memory. We can use integer Cartesian indexing i, j to refer to some location in space,

$$\phi_{i,j} = \phi \left(x = X_0 + \Delta \left(i + \frac{1}{2} \right), y = Y_0 + \Delta \left(j + \frac{1}{2} \right) \right)$$

where $\{X_0, Y_0\}$ is the origin of the axis.

We will start coding our solver with declaring the solution fields in some file name `solver.c`. As the various solver components are added, this file will become a coding mess, so beware that it will need to be restructured later.

```
#include <stdio.h>

#define N 100

double ux[N][N], uy[N][N], p[N][N]

int main() {

}
```

Here we have declared the fields `ux`, `uy` and `p` on a `N x N` grid. The value of the macro `N` is chosen small enough for quick testing, and large enough to do simple tests. The field values can be set and read in a loop over the grid points.

```
...
int main() {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            ux[i][j] = 0; // example
            printf ("%g\n", p[i][j]);
        }
}
```

```

    }
}

```

The double loop (over *i*, and *j*) will be used many times in our solver. Furthermore, the indexing with *i* and *j* may become tedious. As such we can provide ourselves with some macros that makes coding easier and less prone to errors.

```

#include <stdio.h>
#define foreach() for (int _i = 0; _i < N; _i++) \
    for (int _j = 0; _j < N; _j++)
#define val(s, x_ind, y_ind) s[_i + x_ind][_j + y_ind]

```

The purpose of these underscore indexing (e.g. *_i*) is to reduce the chance that the this variable conflicts with future variables. Using these definitions, the previous sample code becomes more clear,

```

...
int main() {
    foreach() {
        val(ux, 0, 0) = 0;
        printf ("%g\n", val(p, 0, 0));
    }
}

```

Especially when we need to access neighboring field values, which is important to quantify the variability of the solution at some point. However, at the edges of our domain neighbor points do not always exists. The most minimalist way of dealing with this is by choosing periodic boundary conditions for our solver. This can be achieved by wrapping the indices if they are smaller than 0 (e.g. $-1 \rightarrow N - 1$ or larger than $N - 1$ (e.g. $N \rightarrow 0$). Then, the grid point at the edges “see” the data at the opposite boundary.

```

...
#define WRAP(i) (i < 0 ? N + i : (i >= N ? i - N : i))
#define val(s, x_ind, y_ind) s[WRAP(_i + x_ind)][WRAP(_j + y_ind)]
...

```

Where C’s ternary operator is used to define the WRAP() macro. We can test if it works,

```

...
foreach()
    val(p, 0, 0) = 1.;

foreach()
    if (val(p, 1, 1) != 1 || p(-1, -1) != 1) {
        puts ("Error");
        return 1;
    }
puts ("Succes!");
...

```

In the second loop, we check if the top-right or bottom-left neighbor value is indeed the value we had set it to be in the first loop. If it is not the case, it prints an error message and quits the program (using `return` in the `main()` function).

```

$ gcc solver.c
$ ./a.out
Succes!

```

A test for finite differencing

It is important to test our code a bit more quantitatively in the context of finite differences. Consider a dummy scalar field on a domain of size $[-5, 5] \times [-5, 5]$,

$$\phi(x, y) = e^{-x^2 - y^2}.$$

We can analytically differentiate this function,

$$\frac{\partial \phi}{\partial x} = -2x e^{-x^2 - y^2},$$

$$\frac{\partial \phi}{\partial y} = -2y e^{-x^2 - y^2}.$$

the test considers estimating these derivatives numerically using our code. From high school mathematics we may remember that the spatial derivative could be defined like so:

$$\frac{\partial \phi}{\partial x} = \lim_{h \rightarrow 0} \frac{\phi(x+h) - \phi(x)}{h}.$$

However, for this moment, there is no reason to bias in any specific direction (i.e using $x+h$, not $x-h$). We could also write a more accurate centered approximation,

$$\frac{\partial \phi}{\partial x} = \lim_{h \rightarrow 0} \frac{\phi(x+h) - \phi(x-h)}{2h}.$$

Instead of taking the limit of h to 0, our solver will have to do with $h = \Delta$. Further, to initialize the fields, we should also define some more useful macros and variables.

```
...
#include <math.h>

#ifndef N
#define N 100
#endif
...
#define Delta (L0/N)
#define x (X0 + Delta*(_i + 0.5))
#define y (Y0 + Delta*(_j + 0.5))
#define sq(x) ((x)*(x))
double L0 = 10;
double X0 = -5, Y0 = -5;
...
int main() {
    double phi[N][N]; // A dummy field
    foreach()
        val (phi, 0, 0) = exp(-sq(x) - sq(y));

    double L2 = 0;
    foreach() {
        double dphi_dx = (val(phi, 1, 0) - val (phi, -1, 0))/(2*Delta);
```

```

        double dphi_dy = (val(phi, 0, 1) - val(phi, 0, -1))/(2*Delta);
        double errorx = -2*x*exp(-sq(x) - sq(y)) - dphi_dx;
        double errory = -2*y*exp(-sq(x) - sq(y)) - dphi_dy;
        L2 += sq(Delta)*sqrt(sq(errorx) + sq(errory));
    }
    printf ("%d %g\n", N, L2);
}

```

Because we have chosen to set the number of cells in each dimension (N) as a macro (using the `#define` directive), it is not easily possible to make convergence-test loop as we did in the previous chapter. However, the additional `#ifndef` conditional statement allows to define N at compilation time. Say for $N = 32$ we can do

```

$ gcc -DN=32 solver.c -lm
$ ./a.out
32 0.334759

```

If we want to do a convergence study to check our coding, we should write a command-line bash script. Open a new file called `check.sh`,

```

#!/bin/bash
FILE=L2_error
rm $FILE
for ((R = 16 ; R <= 512 ; R *= 2))
do
    gcc -DN=$R solver.c -lm
    ./a.out >> $FILE
done

```

Then in the terminal, make it an executable and run this script,

```

$ chmod +x check.sh
$ ./check.sh
rm: cannot remove 'L2_error': No such file or directory

```

The `rm` error message can be ignored: the `rm` command tried to remove a file it could not find, as it was not produced yet. The bash script uses the `>>` operator, which redirects the output from the `a.out` program that would normally be seen in the terminal to a plain text file. You may check the contents of the newly created file called `L2_error`, which should contain the convergence-study data. Go ahead and plot it on a log-log scale, and try to find a fit. It may look something like:

If all went well, you should see that the so-called 3-point central finite difference is second-order accurate.

The code in `solver.c` is getting messy...

The `solver.c` code is burdened with a lot of moderately useful `#define` statements that effectively defeat the purpose of the somewhat cleaner code that follows. We can offload the code that is not directly related to the main function of the program to elsewhere. I therefore propose to create a file called `common.h` where we keep commonly used utilities. I fill it with:

```

#include <stdio.h>
#include <math.h>

#ifndef N
#define N 100

```

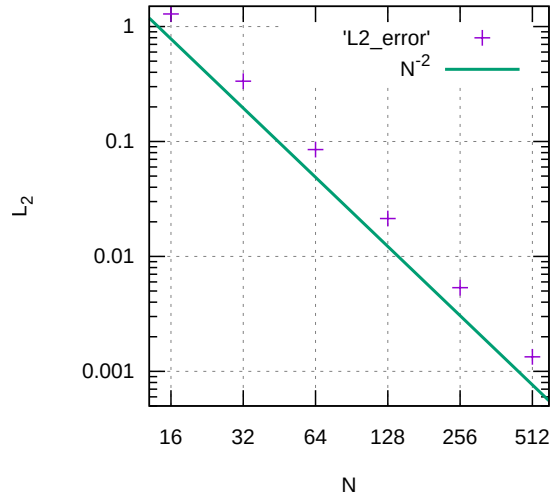


Figure 5: A Second order convergence rate

```
#endif

#define Delta (L0/N)
#define x (X0 + Delta*(_i + 0.5))
#define y (Y0 + Delta*(_j + 0.5))
#define sq(x) ((x)*(x))

#define foreach() for (int _i = 0; _i < N; _i++) \
                  for (int _j = 0; _j < N; _j++)
#define WRAP(i) (i < 0 ? N + i : (i >= N ? i - N : i))
#define val(s, x_ind, y_ind) s[WRAP(_i + x_ind)][WRAP(_j + y_ind)]

//Default domain size
double L0 = 1.;
double X0 = 0, Y0 = 0;
```

The `solver.c` code can become more accessible and focussed now:

```
#include "common.h"

int main() {
    X0 = Y0 = -5.;
    L0 = 10.;
    double phi[N][N];
    foreach()
        val(phi, 0, 0) = exp(-sq(x) - sq(y));
    double L2 = 0;
    foreach() {
        double dphi_dx = (val(phi, 1, 0) - val(phi, -1, 0))/(2*Delta);
        double dphi_dy = (val(phi, 0, 1) - val(phi, 0, -1))/(2*Delta);
```



```

    double errorx = -2*x*exp(-sq(x) - sq(y)) - dphi_dx;
    double errory = -2*y*exp(-sq(x) - sq(y)) - dphi_dy;
    L2 += sq(Delta)*sqrt(sq(errorx) + sq(errory));
}
printf ("%d %g\n", N, L2);
}

```

Mind that the `common.h` file is placed between quotation marks ("`...`") instead of the `<...>` markers that were used for the system header files (`stdio.h` and `math.h`). This hints the compiler where to look for these files. This alteration should not change any of the functionality of the code, and the previous script should still work, as long as `solver.c` is in the same folder as `common.h`. We are effectively creating a rudimentary interface for coding with our flow solver. It is not really user friendly (yet), because it has many quirks. For example, try to declare a variable named `x`. One could say this is an old-fashioned style of “academic” programming, where the programmer is typically the only user of the code.

Continue to chapter 4, where we start approximating the $\mathcal{L}(\mathbf{u})$ vector field.

4. The advection term

Tracer advection

In this chapter we treat the computation of the so-called advection term. It is the under-lined section of the equations,

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

You may also hear this term to be called “the non-linear term” or the “convective term”. In essence, this term describes the transport of the velocity field by *itself*. It will prove useful and instructive to consider a helper equation: The advection equation for a scalar field s ,

$$\frac{\partial s}{\partial t} = -(\mathbf{u} \cdot \nabla) s,$$

describes the evolution of a tracer field (s) advected by a velocity field (\mathbf{u}). Clearly, if we could substitute $s \leftarrow \mathbf{u}$, we retrieve our desired non-linear term. For now, we focus on implementing a function called `tracer_advection`, which computes the tendency for any tracer (s) based on a velocity field.

Computing the advection term

We open a file called `ns.h`, indicating that this is our Navier-Stokes-equations problem-solver file, and start by writing,

```
#include "common.h"

// A function for the advection term of a field `s`
void tracer_advection (double s[N][N], double ux[N][N],
                      double uy[N][N], double ds_dt[N][N]) {
    ...
}
```

Here we have declared a C-language function. In this case, it does not follow a standard input-to-output structure. Rather, the input is implied to be the fields `s`, `ux`, `uy` and `ds_dt`. The values of the latter will be computed and overwritten by this function (output) whereas it will use, but leave the former three fields unchanged (input). The dimension of these field are hard-coded to be 2 with the double `[N][N]`-array syntax, conforming with our earlier design in `common.h` (from the previous chapter).

but what will be the body of this function? It is good to write out the components of the previous equation as the C-compiler does not understand vector notation.

$$\frac{\partial s}{\partial t} = -(u_x \frac{\partial s}{\partial x} + u_y \frac{\partial s}{\partial y})$$

Although the velocity component values are readily available from the input to this function, the derivatives of s need to be *approximated*,

```
...
void tracer_advection (double s[N][N], double ux[N][N],
                      double uy[N][N], double ds_dt[N][N]) {
    foreach() {
        double ds_dx = ...
        double ds_dy = ...
```

```

        val(ds_dt, 0, 0) = -(val(ux, 0, 0)*ds_dx + val(uy,0,0)*ds_dy);
    }
}

```

It turns out that the standard 3-point 2-nd order accurate stencil for the derivative, as tested in the previous chapter, i.e.;

```

...
// Not to use
double dsdx = (val(s, -1, 0) - val(s, 1, 0))/(2*Delta);
...

```

is not suitable for explicit time integration of the advective tendency. This is due to an unfortunate accumulation of the errors introduced by this approximation over time. Instead, we will adopt a so-called upwinding strategy, where the stencil is biased based on the local flow direction. More specifically, we use a 2-point stencil where the gradient is estimated with *upstream* values of the field s . Conceptually, this is the gradient that is being advected towards the current cell.

```

...
void tracer_advection (double s[N][N], double ux[N][N],
                      double uy[N][N], double ds_dt[N][N]) {
    foreach() {
        double ds_dx = val(ux, 0, 0) > 0 ?
            (val(s, 0, 0) - val(s, -1, 0))/Delta :
            (val(s, 1, 0) - val(s, 0, 0))/Delta;
        double ds_dy = val(uy, 0, 0) > 0 ?
            (val(s, 0, 0) - val(s, 0, -1))/Delta :
            (val(s, 0, 1) - val(s, 0, 0))/Delta;
        val(ds_dt, 0, 0) = -(val(ux, 0, 0)*ds_dx + val(uy,0,0)*ds_dy);
    }
}

```

It is not likely that adding so many lines of code goes without typos or other errors. We should at least test the syntax and crate a file, (`advection_test.c`) which `#includes` our new code,

```

#include "ns.h"

int main() {
    ;
}

```

Now we can test if `gcc` finds any issues,

```

$ gcc advection_test.c -lm
$ ./a.out

```

No errors? Then, are we done now with this term? No! We should test our `tracer_advection` function qualitatively and quantitatively.

Before we do that, its time for an antr'acte...

5. Antre'acte: Visualization

The .ppm file format

It would be nice to view our future solutions, so we can inspect its structure which helps with interpreting results. This goes beyond just the apparent beauty of flow visualizations, as so well explained by Ascombe. It may be tempting to offload the visualization burden to a post-processing step in your favorite Matplotlib-enabled language. However, it is actually not that hard to output images in the simple .ppm file format. This was also be used for the movie in the opening of Chapter 0. A code that is compatible with our previous works is implemented below,

```
// Colorbar used values that ramp up and down between 0 and 255
#define RAMPUP (255*(1 - (avg - val(s, 0, 0))/(maxv - avg)))
#define RAMPDOWN (255*(1 + (avg - val(s, 0, 0))/(maxv - avg)))

// Min and Max operators for clipping
#define min(a, b) (a < b ? a : b)
#define max(a, b) (a > b ? a : b)

void output_ppm (double s[N][N], char * fname, double minv, double maxv) {
    // central value (white)
    double avg = (minv + maxv)/ 2.;
    // Open file
    FILE * fp = fopen (fname, "w");
    // Print ascii header for PPM file
    fprintf (fp, "P6 %d %d 255\n", N, N);
    // Upside up left-to-right iterator
    for (int _j = N - 1; _j >= 0; _j--)
        for (int _i = 0; _i < N; _i++) {
            // rgb value for a simple blue-white-red colorbar
            unsigned char rgb[3] = {
                val(s, 0, 0) < avg ? max(0, RAMPUP): 255,           // red
                val(s, 0, 0) < avg ? max(0, RAMPUP) : max(0, RAMPDOWN), // green
                val(s, 0, 0) < avg ? 255 : max(0, RAMPDOWN)};       // blue
            // write binary
            fwrite (rgb, sizeof(char), 3, fp);
        }
    //close file
    fclose (fp);
}
```

I think it maybe instructive if you wrote this code yourself, copying the above by reading and rewriting line by line. But if you do not care about the .ppm format and want to copy-paste code, this bit has perhaps the least to do with the working of our solver. In any case, if we save these approx. 30 lines of codes to a file called `visual.h` we can test it with `test_visual.c`

```
#include "common.h"
#include "visual.h"

double s[N][N];

int main() {
    LO = 10;
    XO = YO = -LO/2;
```

```

foreach()
    val (s, 0, 0) = exp(-sq(x + 2) - sq(y + 2)) - exp(-sq(x - 1) - sq(y - 2));
    output_ppm (s, "s.ppm", -.7, .7);
}

```

In the test, two Gaussian blobs are initialized, one with positive values for s in the bottom left, and the other with negative values for s near the top. One may view the `s.ppm` file with most image viewer software. For display on the website it needs to be converted to the more sensible `.png` format, which can be done using the `convert` command which is part of the `imagemagick` package. In the terminal:

```

$ sudo apt install imagemagick
....
$ gcc -DN=200 test_visual.c -lm
$ ./a.out
$ convert s.ppm s.png

```



Figure 6: The resulting image shows the Gaussian blobs

Movie maker

It is also possible to make a movie, first we need to generate many images, called `s-0XXX.ppm`, where `XXX` indicates the frame number.

```

#include "common.h"
#include "visual.h"

double s[N][N];

int main() {
    L0 = 10;
    X0 = Y0 = -L0/2;
    int frame = 0;

```

```

// Loop over displacement (dx)
for (double dx = -1; dx < 1; dx += 0.02) {
    foreach()
        val (s, 0, 0) = exp(-sq(x - dx + 2) - sq(y + 2)) - exp(-sq(x - dx - 1) - sq(y - 2));
        char fname[99];
        sprintf (fname, "s-%04d.ppm", frame);
        output_ppm (s, fname, -.7, .7);
        frame ++;
    }
}

```

The `%04d` format specifier is used for so-called zero padding. This padding adds leading zeros so that all printed number will have four digits. This is useful to determine that frame 10 comes somewhere after frame 2, which is not what we would get with regular alpha-numeric sorting. We can generate a movie from these frames, again using `convert` in the terminal,

```

$ gcc -DN=200 test_visual.c -lm
$ rm *.ppm
$ ./a.out
$ convert s-*.ppm s.mp4.png

```

the `rm *.ppm` command removes any precious files with the `.ppm` extension in the current folder. Later, we can then use `convert` to append all frames (from files whose names start with “s” and ending with “.ppm”) into a movie called “s.mp4.png”. The result is shown below.

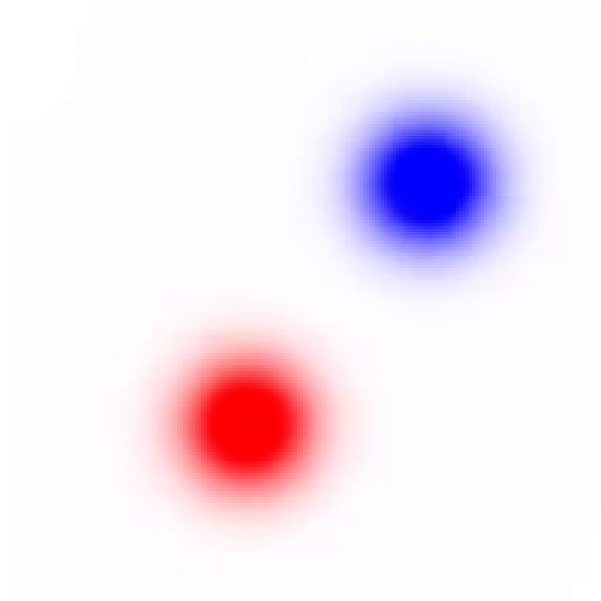


Figure 7: Moving Gaussian blobs

It is not unlikely that you run into issues whilst viewing the generated movie. This is because `convert` is not really designed to handle movies well. If so, an alternative is to use `FFmpeg`, which needs some extra instructions to understand the input image sequence.

```

$ sudo apt install ffmpeg
$ ffmpeg -pattern_type glob -i 'o-*.ppm' o.mp4.png

```

It is less intuitive, but much more powerful. For a web-friendly encoding i use,

```
ffmpeg -pattern_type glob -i 's-*.ppm' -c:v libx264 -vf format=yuv420p s.mp4.png
```

We will use this visualization to test our `tracer_advection` scheme in Chapt. 6

6. Testing the `tracer_advection()` function

A qualitative test

We can now start to put together our previous lessons to setup a tracer advection test case. For any advection problem, we have a velocity field and since we are preparing for a full Navier-Stokes equations solver, it makes sense to declare the fields for its components in the `ns.h` file.

```
# include "common.h"

double ux[N][N], uy[N][N];

double t, dt = 1;

void tracer_advection (...
```

We have also introduced global variables for the time (`t`) and the time step size (`dt`), whose value we can alter in the test file `test_advection.c`. In this file, we will setup the test, which consists of including the relevant header files and declare a scalar field. In `test_advection.c`:

```
#include "ns.h"
#include "visual.h"

// Scalar field
double s[N][N];

int main() {
    // setup domain size
    LO = 10;
    XO = YO = -LO/2;
}
```

All OK? We can continue by initializing the relevant field values. For `s` we could reuse the compact functions from the previous chapter. For the velocity field, we can use a simple translation. The case is fully defined by setting an end time that corresponds to a complete cycle though the periodic domain.

```
...
XO = YO = -LO/2.;
// Initiauze s, ux and uy
foreach() {
    val (s, 0, 0) = exp(-sq(x + 2) - sq(y + 2)) - exp(-sq(x - 1) - sq(y - 2));
    val (ux, 0, 0) = -2.;
    val (uy, 0, 0) = 0.;
}
// end time
double t_end = 5;
}
```

Now we setup the time loop (c.f. Chapt. 2) with a small value for `dt`.

```
...
// end time
double t_end = 5;
double dt = 0.05;
// step counter
int iter = 0;
```



```

// Time loop
for (t = 0; t < t_end; t += dt) {
    // Tendency
    double ds_dt[N][N];
    tracer_advection (s, ux, uy, ds_dt);
    // Advance
    foreach()
        val(s, 0, 0) += dt*val(ds_dt, 0, 0);
    // Visualize
    char fname[99];
    sprintf (fname, "s-%04d.ppm", iter);
    output_ppm (s, fname, -0.7, 0.7);
    // Increment step counter
    iter++;
}
printf ("# Solver finished after %d steps at t = %g\n", iter, t);
}

```

Most of the code above should look familiar from the previous chapters. After some cleaning up, we can compile and run our test.

```

$ rm s-*.ppm
$ gcc test_advection.c -lm
$ ./a.out
# Solver finished after 101 steps at t = 5.05

```

Great! Lets generate a movie,

```

$ convert s-*.ppm s_advect.mp4.png

```

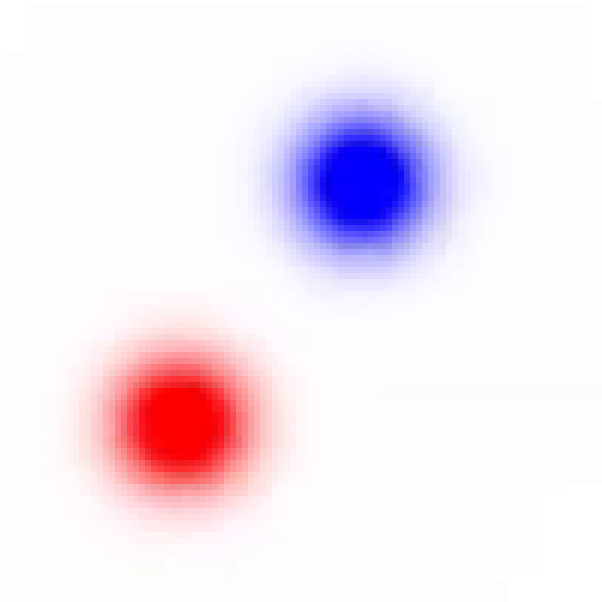


Figure 8: Again, moving blobs

If you have arrived at this point, you should realize that a lot has gone well. Indeed the blobs move(!),

in the expected direction(!) ($u_x < 0$), and the expected distance(!) (single cycle). All is well then? Well... lets push our solver a bit, and increase the resolution.

```
$ rm s-*.ppm
$ gcc -DN=400 test_advection.c -lm
$ ./a.out
# Solver finished after 101 steps at t = 5.05
$ convert s-*.ppm s_400.mp4.png
```

Disaster!

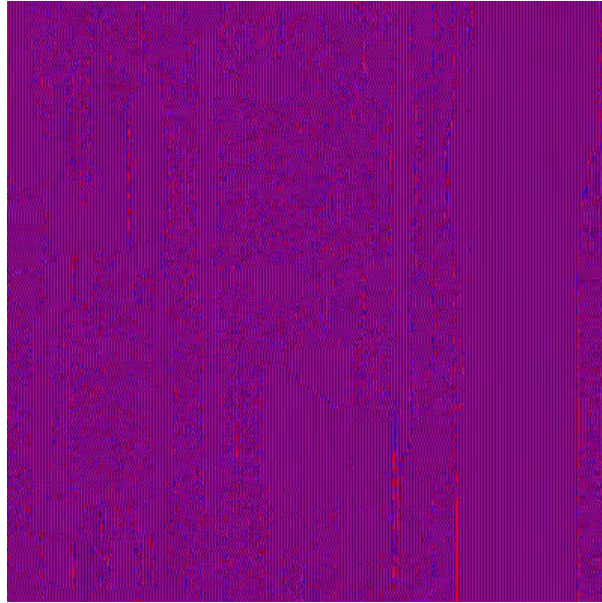


Figure 9: An instability

A stability criterion

We have obtained a nonphysical solution due to the rapidly accelerating growth of errors. Indeed, our upwinding strategy does not yield unconditionally stable time integration. Even before the advent of the digital computer it was known that the time-step size cannot be chosen freely for such advection problems. Rather, for explicit time-integration (such as forward Euler) it must be chosen small enough so that the cells do not “over flow” in a single step. This is formalized by the so-called CFL condition, named after its “inventors”, Richard Courant, Kurt Friedrichs, and Hans Lewy. It can be interpreted as follows: There exists a dimensionless number (CFL) that compares the time-step size (Δt) against the mesh-element size (Δ), based on the (maximum) velocity in the domain.

$$\text{CFL} = \frac{\Delta t \|u\|_{\max}}{\Delta}.$$

For any explicit time integration method, there exists a maximum critical finite value for which the solution will remain stable. Typically, $\text{CFL}_{\text{crit.}} \approx 1$. In practice, this entails selecting a time-step size based on this so-called CFL condition,

$$dt = CFL_{crit.} \frac{\Delta}{\|u\|_{\max}}.$$

Notice that this *stability criterion* is separate from any accuracy criterion. Our test in the second chapter was therefore not representative for the time integration of flow problems. Further, the CFL criterion directly relates the spatial discretization parameter (Δ) to one for time. This is somewhat natural when doing a convergence study for a spatio-temporal problem.

For now, we have to extend our code in `ns.h` with a function that **returns** the CFL-based limit. Notice that the C language does not come with a simple function that finds the absolute maximum of an array. So we need to code it ourselves. In `ns.h`,

```
...
double t, dt = 1;
double CFL = 0.7;

void tracer_advection (...) {
    ...
}

double dt_CFL () {
    double max_v = -1.;
    foreach() {
        if (fabs(val(ux, 0, 0)) > max_v)
            max_v = fabs(val(ux, 0, 0));
        if (fabs(val(uy, 0, 0)) > max_v)
            max_v = fabs(val(uy, 0, 0));
    }
    if (max_v > 0) // Dont divide by zero
        return CFL*Delta/max_v;
    return 1e30;
}
```

After setting a safe value for CFL, the new function without input determines the absolute maximum value of the velocity component fields and computes an appropriate time step. A special check is performed for the case where `max_v = 0`, to prevent dividing by zero. Note that once a **return** statement is executed, the control loop stops further function evaluation. I.e. this function *either* returns with the CFL-based limit or with 10^{30} . Finally, I will admit that it is somewhat debatable and inconsistent to set `ux` and `uy` as input to the function `tracer_advection()`, and rely on the global variables in `dt_CFL()` for the velocity data. You may choose to do otherwise.

While we are in `ns.h`, I would also like our solver to stop at `t = t_end`, and not at `t = t_end + dt`, which was the case in our previous experiment (see the terminal output). Once the time parameter `t` is close to `t_end`, `t + dt` should be smaller or equal to `t_end`, not to over step. For this purpose, we add a new function in `ns.h`:

```
...
    return 1e30;
}

double dt_next (double t_end) {
    double DT = dt_CFL();
    if (t + DT > t_end)
        return t_end - t;
}
```

```

return DT;
}

```

We use this to update our time stepping in `test_advection.c`

```

...
// end time
double t_end = 5; //L0/ux
// step counter
int iter = 0;
// Time loop
for (t = 0; t < t_end; t += dt) {
    // compute timestep size
    dt = dt_next(t_end);
    // Tendency
    double ds_dt[N][N];
    ...
}

```

Now test

```

$ rm s-*.ppm
$ gcc -DN=400 test_advection.c -lm
$ ./a.out
# Solver finished after 572 steps at t = 5
$ convert s-*.ppm s_400.mp4.png

```

It appears to solver took more steps and ends at to correct time. You can verify the contents of `s.mp4.png` yourself. To test our upwinding implementation, it would also be good to redo the experiment with positive and negative values for both `ux` and `uy`.

This chapter continues with two exercises.

(1) A quantitative test

It would be good to not only visually inspect the solution, but also diagnose the convergence rate of our formulations. For this purpose, you could setup a workflow as in Chapt. 3 using the initialized solution as the reference analytical solution after one full cycle. Indeed, pure translation should not alter the shape of the solution. For this case, the domain size needs to be enlarged as the initialized solution is not consistent with the periodic boundaries. This effect is exponentially reduced by increasing `L0`. Note that when analyzing your results, both spatial and temporal discretization are only first-order accurate.

(2) A Well-behaved solver?

Although the speed performance of our solver is not a critical concern, we should verify if the time efficiency at least behaves well. This entails verifying the scaling of the time effort against the expected scaling of the iteration effort. For this purpose we turn of the output routine,

```

...
// output_ppm (...)
...

```

so that the disk-writing performance does not affect our wall-clock time to solution. Next, we generate 3 executables with increasingly finer grids. In order to not overwrite a previously generated program, we name them differently using the `-o` option, to name the output programs other than the default `a.out`.

```
$ gcc -DN=100 test_advection.c -lm -o ta-100
$ gcc -DN=200 test_advection.c -lm -o ta-200
$ gcc -DN=400 test_advection.c -lm -o ta-400
```

We can use a stopwatch that is build into most shells to measure the time spend on the execution of the programs. I get (using bash for illustrative purposes):

```
$ time ./ta-100
# Solver finished after 143 steps at t = 5

real 0m0.136s
...
$ time ./ta-200
# Solver finished after 286 steps at t = 5

real    0m0.885s
...
$ time ./ta-400
# Solver finished after 572 steps at t = 5

real    0m6.819s
...
```

As we increase the resolution by a factor of two, the effort required to run the simulation increased by a factor of eight! This somewhat worrying scaling behavior is in fact the expected result, and so far, our solver is well behaved. Can you see why?

Lets continue with the next term in the Navier-Stokes equations in chapter 7

7. The viscous term

In this chapter, we concern ourselves with the viscous term. I.e. the under-lined term in the equations below,

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

This term aims to describes the diffusion of momentum originating from the exchange of momentum by colliding molecules. For sufficiently continuous fluids, the effects of the incredibly-complicated statistical dynamics of these collisions are remarkably well modeled with the relatively simple diffusion term with a constant diffusivity of momentum (kinematic viscosity, ν).

Again, it will prove useful to consider a helper equation: The diffusion of a scalar field s in a medium with diffusivity κ ,

$$\frac{\partial s}{\partial t} = \kappa \nabla^2 s,$$

where $\nabla^2 s$ symbolizes the divergence of the gradient of s ,

$$\frac{\partial s}{\partial t} = \kappa (\nabla \cdot \nabla s),$$

but if we write it without vector-operator notation it reads,

$$\frac{\partial s}{\partial t} = \kappa \left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} \right).$$

It appears we need to estimate the second derivative of the field s with respect to the spatial directions. The second derivative in any direction can be evaluated by differentiating the first derivative. Conceptually, it would be an OK idea to implement a `gradient_x(s[N][N], ds_dx[N][N])` function (and an `gradient_y()` counterpart), which compute the spatial derivatives of a field. We could then use it to compute the higher-order derivatives by calling it successively. For our second derivative,

```
// We will not use this!
double s[N][N];
...
// Storage for first and second derivative in x direction
double ds_dx[N][N], d2s_dx2[N][N];
// Differentiate twice
gradient_x(s, ds_dx);
gradient_x(ds_dx, d2s_dx2);
```

Although performance was not a concern for the design of our solver, the code above is wasteful to such a degree that it would be an insult to the didactic nature of this text. This is because we can achieve the same result without introducing new fields by analyzing the stencil operations. Say we use the 3-point second-order accurate gradient,

$$\frac{\partial s}{\partial x}[0,0] \approx \frac{s[1,0] - s[-1,0]}{2\Delta},$$

where $\frac{\partial s}{\partial x}[0, 0]$ denotes the derivative in the current cell and $s[1, 0]$ denotes the right-hand-side neighbor value (`val(s, 1, 0)`). The second derivative is then approximated by,

$$\frac{\partial^2 s}{\partial x^2}[0, 0] \approx \frac{\frac{\partial s}{\partial x}[1, 0] - \frac{\partial s}{\partial x}[-1, 0]}{2\Delta},$$

Combining the equations,

$$\frac{\partial^2 s}{\partial x^2}[0, 0] \approx \frac{s[-2, 0] - 2s[0, 0] + s[2, 0]}{4\Delta^2}.$$

Now we can directly evaluate this expression when needed instead of declaring fields for storage, which is so much better, that the “successive gradients” method becomes a bit silly. It is however odd that our expression is 5 points wide (from $s[-2, 0]$ to $s[2, 0]$), but only uses three values. We have inherited this from the original conception where the (intermediate) first derivative was evaluated at the grid points. The field s and its two derivatives needed to be co-located for the successive gradient approach. But for the theoretical analysis we could have computed these values at the mid points in between cells,

$$\frac{\partial s}{\partial x}[\frac{1}{2}, 0] \approx \frac{s[1, 0] - s[0, 0]}{\Delta}.$$

Redoing the exercise in the so-called *staggered* fashion, where the gradients are (on paper) evaluated in between the grid points, results in,

$$\frac{\partial^2 s}{\partial x^2}[0, 0] \approx \frac{s[-1, 0] - 2s[0, 0] + s[1, 0]}{\Delta^2}.$$

We are now ready to implement the diffusion term for a scalar field s . In `ns.h`, insert

```
...

void advection (...) {
...
}

void diffusion (double s[N] [N], double kappa, double ds_dt[N] [N]) {
    foreach() {
        val(ds_dt, 0, 0) += kappa/sq(Delta)*
            (val(s, -1, 0) - 2*val(s, 0, 0) + val(s, 1, 0));
        val(ds_dt, 0, 0) += kappa/sq(Delta)*
            (val(s, 0, -1) - 2*val(s, 0, 0) + val(s, 0, 1));
    }
}

double dt_CFL () {
...
}
```

Notice that we have chosen to *add* (using `+=`) the diffusive tendency to the input filed `ds_dt`. This way, we can simply add the viscous tendency to an earlier initialized (advection) tendency.

A stability criterion

Just as for the advection equation, explicit time integration of the diffusion equation is prone to instabilities for too large time steps (dt) on relatively fine grids. We can again compare the time-step size and the grid size (Δ) using the equation parameter κ with a dimensionless number (Pe),

$$Pe = \frac{\kappa dt}{\Delta^2}.$$

A (sometimes) so-called critical cell-Peclet number (Pe_{crit}) can be used to compute a stable time-step size,

$$dt = Pe_{\text{crit}} \frac{\Delta^2}{\kappa}.$$

Clearly, this diffusion-based limit gets smaller faster for small Δ compared to the CFL-based limit. For our solver, we will simply take the minimum of these constraints. This warrants to bump our `min/max` definitions in the `visual.h` file to the common utilities file `common.h`. In `ns.h`, we will consider the diffusivity for a scalar field (κ), and the fluid' viscosity (ν) separately. Since the latter is just a fancy word for the diffusivity of momentum, we take their maximum to compute the Peclet-based limit.

```
...
double CFL = 0.8, Pe = 0.1;
double kappa = 0, nu = 0;
...
double dt_CFL() {
    ...
}

double dt_diffusion() {
    return Pe*sq(Delta)/max(kappa, nu);
}

double dt_next (double t_end) {
    double DT = min(dt_CFL(), dt_diffusion());
    if (t + DT > t_end)
        return t_end - t;
    return DT;
}
```

Testing

The diffusion of a Gaussian bump has an analytical solution. In 2D,

$$s(t, x, y) = \frac{e^{\frac{-x^2-y^2}{4\kappa(t+t_0)}}}{4\pi\kappa(t+t_0)},$$

With arbitrary time-shift parameter t_0 . We can set up a test using this. First qualitatively in `diffusion_vis.c`,

```
#include "ns.h"
#include "visual.h"
#define pi (3.14159265)
```



```

#define SOL(t) ((exp((-sq(x) - sq(y))/(4*kappa*(t + t0)))/(4*pi*(t + t0)))

double t0 = 0.5, t_end = 1;
double kappa_val = 2.;

double s[N][N];

int main() {
    LO = 20;
    XO = YO = -LO/2.;
    kappa = kappa_val;

    foreach()
        val (s, 0, 0) = SOL(t);

    int iter = 0;
    for (t = 0; t < t_end; t += dt) {
        dt = dt_next(t_end);
        double ds_dt[N][N];
        foreach()
            val (ds_dt, 0, 0) = 0;
        diffusion (s, kappa, ds_dt);
        foreach()
            val (s, 0, 0) += dt*val(ds_dt, 0, 0);
        char fname[99];
        sprintf (fname, "s-%04d.ppm", iter);
        output_ppm (s, fname, -0.1, 0.1);
        iter++;
    }
    printf ("# Solver finished at t = %g after %d iterations\n",
           t, iter);
}

```

Compile and run,

```

$ rm s-*.ppm
$ gcc -DN=200 diffusion_vis.c -lm
# Solver finished at t = 1 after 2001 iterations

```

Oof! 2001 iterations, that movie will take about 80 seconds at 25 frames per second. Lets reduce it a bit by only writing a frame every `iter_interval` iterations.

```

...
int iter_interval = 20;

int main() {
    ...
    if ((iter % iter_interval) == 0) {
        char fname[99];
        sprintf (fname, "s-%04d.ppm", iter);
        output_ppm (s, fname, -0.1, 0.1);
    }
    iter++;
}

```

...

Lets check the movie (remember to remove the old images).

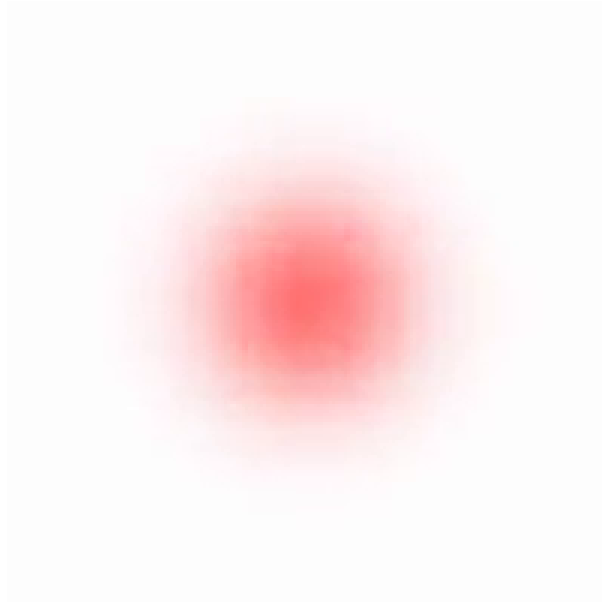


Figure 10: Diffusion seems to smoothing out the gradients

This looks good, and we can continue to do the convergence test, which is left as an exercise for the reader. Notice that for a well-behaved solver, the effort now scales with $\propto \Delta^4$, which is truly horrific. At this point it can become advantageous to compile with *optimization*: At the expense of a more time-consuming compilation step (i.e. the `gcc ...` command), a more optimized executable is generated. The steps taken in this optimization process are truly remarkable (see), but are far outside the scope of this course. Instead, we simply “turn on optimization” as a command-line option. The highest level of optimization supported by the gcc compiler is achieved with the option `-O3`.

```
$ gcc -O3 -DN=200 diffusion_vis.c -lm
```

It is now time for another Antr’acte before we add the last term (or is it?).

8. Antre'acte: A Poisson-problem solver

Poisson's equation

Poisson problems arise in many fields in science and engineering. For example, it relates the electric or gravity potential (ϕ) to the spatial charge-density or mass-density distribution (ρ), respectively. It reads,

$$\nabla^2 \phi = \rho.$$

From the previous chapter, we know how to approximate the left-hand side ($\nabla^2 \phi$) to obtain an approximation of the scalar field ρ , so what is Poisson's problem? It is about the reversed case, finding ϕ , for a given so-called source term (i.e. the right-hand side, ρ) that satisfies the above equation. This is an implicit problem, as we can only verify a solution after it has already been computed. On the other hand, it is a relatively simple linear expression, and an analytical solution has been found. In 2D,

$$\phi(\mathbf{x}) = - \iint \frac{\rho(\mathbf{x}')}{2\pi \|\mathbf{x} - \mathbf{x}'\|} d^2 x'.$$

Our goal is to implement a function `poisson (double phi[N][N], double rho[N][N])`, that approximates this solution. It is tempting to use this analytical form, and integrate the solution over our domain. Although this is easier said than formalized, it would indeed be a valid strategy. However, many alternative methods exist, and we will take another route where we have more easy control over some *numerical properties* of our solution.

Notice that the Poisson equation alone does not fully specify the problem as any solution (ϕ_1) that solves the Poisson problem, can be altered by a solution (ϕ_2) to an Laplace equation

$$\nabla^2 \phi_2 = 0.$$

Given the linearity of the problem, $\phi_3 = \phi_1 + \phi_2$ would also solve the Poisson problem. Examples for ϕ_2 include a constant field, and fields with constant gradients. This ambiguity can be taken away by setting suitable boundary conditions for ϕ on the domain. However, since we limit ourselves to periodic boundaries on square domains, the solution is ambiguous and can be changed by adding and subtracting an arbitrary constant.

No worries, there could exist many solutions and that could make it easier to find at least one?. Well, that is assuming that there exists a regular solution. Consider for example the case of an infinitely periodically repeating positive charge distribution. The corresponding electric potential well would be infinitely deep $\phi \rightarrow -\infty$, making it obvious to state that for such a case, no regular solution exists. As such, the boundary conditions to a Poisson problem often impose a compatibility constraint on ρ . For our periodic solver, it means that the domain (\mathcal{D}) integral of ρ must net zero.

$$\iint_{\mathcal{D}} \rho dx dy = 0.$$

A numerical solving strategy

From the previous chapter, we know a discrete second-order accurate version of the problem is that for all cell indices i and j ,

$$\frac{\phi[i-1, j] + \phi[i, j-1] - 4\phi[i, j] + \phi[i+1, j] + \phi[i, j+1]}{\Delta^2} = \rho[i, j].$$

A rather crude, yet incredibly simple, strategy for finding such ϕ is to iteratively solve for it. For a given cell we could satisfy the equation *locally* by assigning the corresponding value to the *current* cell,

$$\phi[0,0] \leftarrow \frac{-\Delta^2 \rho[0,0] + \phi[-1,0] + \phi[0,-1]\phi[1,0] + \phi[0,1]}{4}.$$

If we do this for every cell in the grid, we are only guaranteed to satisfy the Poisson equation in the last cell of our iterator. However, if the initial guess of the ϕ -field is not particularly bad, the changes to the local cell ($\phi[0,0]$) can be small. Of the then repeatedly sweep over all cells, the difference between the replacement values and the current solution may shrink to a very small value. This *could* indicate that the solution converges by successive direct replacement. Lets implement this idea in a file called `poisson.h`.

```
int max_sweeps = 100;

void poisson (double phi[N][N], double rho[N][N]) {
    for (int sweep_nr = 0; sweep_nr < max_sweeps; sweep_nr++) {
        foreach() {
            double c = 0;
            for (int i = -1; i <= 1; i += 2)
                c += val(phi, i, 0) + val(phi, 0, i);
            val(phi, 0, 0) = (c - val(rho, 0, 0)*sq(Delta))/4.;
        }
    }
}
```

It would also be nice to trace the so-called residual (`res`) as the ϕ field “relaxes” towards the solution.

$$\text{res} = \nabla^2 \phi - \rho$$

We are interested in its absolute maximum,

```
...
    foreach() {
        ...
    }
    double max_res = -1;
    foreach() {
        double c = 0;
        for (int i = -1; i <= 1; i += 2)
            c += val(phi, i, 0) + val(phi, 0, i);
        double res = (c - 4*val(phi, 0, 0))/sq(Delta) - val(rho, 0, 0);
        if (fabs(res) > max_res)
            max_res = fabs(res);
    }
    printf ("%d %g\n", sweep_nr, max_res);
}
```

We can now see if our approach works using `test-poisson.c`,

```
#include "common.h"
#include "poisson.h"
```

```

int main() {
    LO = 2*3.1415;
    // phi and rho (a and b)
    double a[N][N], b[N][N];
    foreach() {
        val (a, 0, 0) = 0;
        val (b, 0, 0) = sin(x) + cos(y);
    }
    poisson (a, b);
}

```

Note that we have initialized the **a**-field values to some initial guess as it could contain garbage upon declaration. Further, the source field **b** is carefully chosen such that it satisfies the compatibility requirement. Compile and run,

```

$ gcc test_poisson.c -lm
$ ./a.out
0 2.77167
1 2.91396
2 2.94642
3 2.95452
4 2.95533
...
95 2.48183
96 2.47694
97 2.47206
98 2.46719
99 2.46233

```

The output reveals that the residual is somewhat decreasing, but after 100 iterations the maximum residual (which should go to zero) is still relatively close to its initial value. We can change `test_poisson.c` to take more sweeps in to hope for a better solution,

```

...

    max_sweeps = 10000;
    poisson (a, b);
}

```

Resulting in,

```

$ gcc -O3 test_poisson.c -lm
$ ./a.out
...
9996 5.89834e-05
9997 5.89834e-05
9998 5.89834e-05
9999 5.89834e-05

```

Much better. lets investigate the convergence of the solution a bit,

```

$ ./a.out > out
$ gnuplot
gnuplot> set logscale y
gnuplot> plot 'out'

```

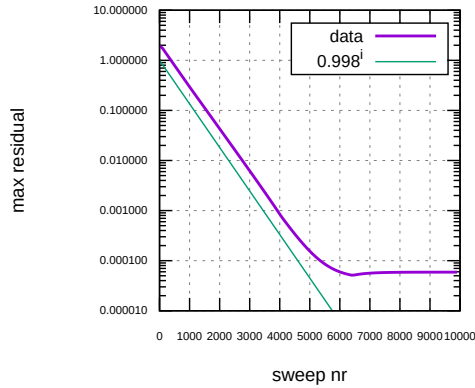


Figure 11: A formatted plot stored as .svg

It seems that for the first few thousand iterations, there is a constant reduction factor for the residual with each sweep (≈ 0.998), but after some point the solution stops to improve. This is a reminder of the compatibility requirement. Because we have only defined π with a few digits, the discrete integral of the source term is not exactly zero, and there does not exist a proper solution to the discrete problem. You can verify by setting $L0 = 2*3.14159265;$, that the residual reduces to $6.62\dots e-9$. It is nice to know that our iterative solver was quite robust in the previous case, as it did manage to find a field ϕ that *nearly* solves the impossible problem.

A criterion to stop sweeping

The accurate solution to the discrete problem (i.e. a small residual), is rather expensive to obtain due to the high number of relaxation sweeps. It would be nice to stop iterating once the field ϕ is *sufficiently* close to solving the discrete problem. This can be especially rewarding for cases when the initial guess is quite close to the solution and only a few iterations are needed for a small residual. As such we will tolerate a small maximum absolute residual on our solution, and stop sweeping once it is achieved. In `poisson.h`, we replace our `for` loop with a `do-while` construction.

```
int max_sweeps = 1e4;
double tolerance = 1e-3;

void poisson (double phi[N][N], double rho[N][N]) {
    int sweep_nr = 0;
    double max_res = -1;
    do {
        foreach() {
            ...
        }
        sweep_nr++;
        max_res = -1;
        foreach() {
            ...
        }
    } while (max_res > tolerance && sweep_nr < max_sweeps);
}
```

```

// Warn when the solution did not converge within `max_sweeps`
if (max_res > tolerance)
    printf ("# Warning: max res = %g is greater than its tolerance (%g)\n",
            max_res, tolerance);
printf ("%d %g\n", sweep_nr, max_res);
}

```

We can test it,

```

$ gcc test_poisson.c -lm
$ ./a.out
4054 0.000999368

```

The relevance becomes apparent by perturbing the solution with a high-frequency component. In `test_poisson.c`,

```

...
    poisson (a, b);
    foreach()
        val (a, 0, 0) += 0.1*(sin(10*x)*cos(20*y));
    poisson (a, b);
}

```

Gives as output,

```

...
4054 0.000999368
38 0.000992947

```

The second line shows that when the initial guess of the solution was already close to the final solution, the Poisson solver automatically stopped after only a few sweeps (38). Note that the residual reduction per sweep is likely to be smaller (i.e. favorable) than the 0.998 from the previous case.

The last coding in this chapter is to remove the not strictly needed messaging. It would clutter output as we call it many times during a flow simulation.

```

...
// printf ("%d %g\n", sweep_nr, max_res);
...

```

Solving for $\nabla\phi$

Often, and not in the last place for computational fluid dynamics, we are not really interested in ϕ , but use it to compute its gradients. This is why we do not really care about the constant offset any solution may have on a periodic domain. It should be noted however, that the computed discrete gradients of our solution field ϕ may not inherit the expected properties of such gradient fields from the small tolerance on ϕ to solve the discrete Poisson problem. This is a bit cryptic way of repeating that the 3-point stencil for the Laplacian operator was derived from gradients evaluated at staggered locations. So when we are computing gradients of our solution at grid points, the divergence of these gradients will not match the source term exactly (irrespective of the value for `tolerance`). A way around this problem is to change the discrete problem to the earlier derived 5-point stencil,

$$\frac{\phi[i-2, j] + \phi[i, j-2] - 4\phi[i, j] + \phi[i+2, j] + \phi[i, j+2]}{4\Delta^2} = \rho[i, j],$$

and require that the gradients must be evaluated using the central 3-point stencil. Another way around this issue it is to define the variables that interact with the gradients to also be staggered compared to

ϕ . We will discuss what this narrative actually entails for our solver in a future chapter. But it turns out, we can mostly ignore it for now.

We can now apply our linear solver to compute the pressure-gradient term in Chapt. 9.

9. The pressure-gradient force

In this chapter we will consider the pressure term. That is the under-lined term in the Navier-Stokes equations below,

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

This term is perhaps the most involved and least intuitive of these equations. This is perhaps because the pressure in a fluid is a somewhat illusive concept. As already mentioned, we will find an implicit problem that needs to be solved in an intermediate step before actually computing the pressure gradient. To add insult to injury, we cannot really conceptualize this term for a helper scalar field, so we will do the full vector calculus.

Before we start, we should get some common misconceptions out of the way. If you have learned about forced flows (e.g. pipe flow) in your fluid dynamics classes, your intuition may tell you that the pressure gradient is externally applied and prescribed by some sort of pumping mechanism. Although a pressure drop can be treated as a boundary condition, a more relevant view (from the perspective of our solver) is demonstrated by the Bernoulli principle. If a steady pipe flow contains some constriction (e.g. a Venturi), the approaching flow accelerates towards the narrowing. This accelerating force comes from the pressure gradient as Bernoulli realized this region of high velocities is associated with a lower pressure. It thus appears that the pressure field is directly related to the flow itself. But how?

Another confusing conception is the thermodynamic meaning of pressure. Especially for those who are versed in compressible flows it can be hard to go without some equation of state. Indeed, for incompressible flows in simple (i.e. at least rigid) geometries, pressure loses its thermodynamic meaning. The pressure can be seen to adjust such that the flow remains incompressible. This entails that we do not need to time integrate some evolution equation for the pressure as we can derive it directly as a function of the flow.

A projection method

The fundamental theorem of vector calculus states that any vector field \mathbf{v} can be decomposed as,

$$\mathbf{v} = \mathbf{R} + \nabla \phi,$$

where,

$$\nabla \cdot \mathbf{R} = 0.$$

such that the divergence in \mathbf{v} is captured by the scalar-gradient term. We already have the tools to make this so-called Helmholtz decomposition numerically. If we take the divergence of the equation,

$$\begin{aligned}\nabla \cdot \mathbf{v} &= \nabla \cdot \mathbf{R} + \nabla \cdot \nabla \phi, \\ \nabla \cdot \mathbf{v} &= 0 + \nabla \cdot \nabla \phi.\end{aligned}$$

We see the structure of a Poisson equation where the source term is the divergence of the vector field \mathbf{v} . If we have solved for ϕ , we can also find \mathbf{R} by,

$$\mathbf{R} = \mathbf{v} - \nabla \phi,$$

completing the decomposition. At the risk of sounding cryptic, we could say that \mathbf{R} is the projection of \mathbf{v} onto the space of divergence-free vector fields¹. Lets code it up! In `ns.h`,

```
...

void diffusion (...) {
    ...
}

void projection (double vx[N][N], double vy[N][N], double phi[N][N]) {

}

...
```

Note that it will prove important to have the `phi` field as input, although we could in principle declare and initialize it in the function's scope. The function takes the steps described above, overwriting the input vector field.

```
#include "common.h"
#include "poisson.h"
...
void projection (double vx[N][N], double vy[N][N], double phi[N][N]) {
    // Compute the divergence
    double div[N][N];
    foreach() {
        val(div, 0, 0) = (val(vx, 1, 0) - val(vx, -1, 0))/(2*Delta);
        val(div, 0, 0) += (val(vy, 0, 1) - val(vy, 0, -1))/(2*Delta);
    }
    // Compute the helper scalar field
    poisson(phi, div);
    // Reject the divergent part
    foreach() {
        val(vx, 0, 0) -= (val(phi, 1, 0) - val(phi, -1, 0))/(2*Delta);
        val(vy, 0, 0) -= (val(phi, 0, 1) - val(phi, 0, -1))/(2*Delta);
    }
}
...
```

For now, I propose to only test the syntax of this function (using gcc), and not do a proper test, as we will already change it in the next section and we will test it soon enough in a proper flow setting.

Chorin's projection method

In Chorin's seminal paper² on numerical flow-problem solving, the above projection method is used to compute the pressure term. His special insight was on how to compute it in conjunction with the advection and diffusion terms. In the first stage of our time step, a provisional velocity field (\mathbf{u}^*) is computed at $t_{n+1} = t_n + dt$, which considers every term except the pressure term.

$$\mathbf{u}^* = \mathbf{u}_n + dt \left(-(\mathbf{u}_n \cdot \nabla) \mathbf{u}_n + \nu \nabla^2 \mathbf{u}_n \right).$$

We can find \mathbf{u}_{n+1} by projecting \mathbf{u}^* on to the space of divergence-free vector fields. But what about the

¹It seems more like a vector rejection to me.

²Chorin, Alexandre Joel. "Numerical solution of the Navier-Stokes equations." *Mathematics of computation* 22.104 (1968): 745-762.

pressure? `phi` can not be it as it does not have the right units. Well, it is the pressure, but its off by a factor `dt`. See,

$$\mathbf{u}_{n+1} = \mathbf{u}^* - dt \nabla p,$$

taking the divergence,

$$\begin{aligned}\nabla \cdot \mathbf{u}_{n+1} &= \nabla \cdot \mathbf{u}^* - \nabla \cdot dt \nabla p, \\ 0 &= \nabla \cdot \mathbf{u}^* - \nabla^2 dt p,\end{aligned}$$

This is the previous Poisson equation with $\phi \leftarrow dt p$.

$$\nabla \cdot \mathbf{u}^* = \nabla^2 dt p,$$

Or even better, we can divide by the time-step size (`dt`)

$$\frac{\nabla \cdot \mathbf{u}^*}{dt} = \nabla^2 p.$$

In view of this, I propose to alter the `projection()` function a little,

```
void projection (double vx[N][N], double vy[N][N],
                double phi[N][N], double dt) {
    // Compute the divergence divided by `dt`
    double div[N][N];
    foreach() {
        val(div, 0, 0) = (val(vx, 1, 0) - val(vx, -1, 0))/(2*Delta);
        val(div, 0, 0) += (val(vy, 0, 1) - val(vy, 0, -1))/(2*Delta);
        val(div, 0, 0) /= dt;
    }
    // Compute the pressure field
    poisson (phi, div);
    // Reject the divergent part
    foreach() {
        val (vx, 0, 0) -= dt*(val(phi, 1, 0) - val(phi, -1, 0))/(2*Delta);
        val (vy, 0, 0) -= dt*(val(phi, 0, 1) - val(phi, 0, -1))/(2*Delta);
    }
}
```

Using p as the helper scalar field for projection instead of any other is important as it means that for slowly evolving flows, the pressure field from the previous time step will serve as a good initial guess. This would not be the case for ϕ , which could change a lot with varying time-step sizes. Further, scaling the divergence with `dt` introduces a good moment to consider the meaning of our tolerance on the maximum residual (`tolerance`). If we wish the tolerance to represent the maximum divergence in our solution we should scale it with the time step size.

```
void projection (double vx[N][N], double vy[N][N],
                double phi[N][N], double dt) {
    // store desired tolerance and set tolerance on divergence/dt
    double tol = tolerance;
    tolerance /= dt;
```

```
...  
// Reset tolerance  
tolerance = tol;  
}
```

We cant really diagnose this low-tolerance divergence as the gradients in our Poisson solver were defined in a staggered fashion. Any divergence we can approximate will be polluted with an additonal discretization error. Further, our projection method is only approximately as we did not opt for the 5-point stencil version.

At this moment, we can combine our functions in to a Navier-Stokes equation time-stepping function. We will do this in the next chapter.

10. A Navier-Stokes problem solver

A time stepping function

We have arrived at the point where we can code our time-stepping function. This function of the form,

```
void advance_ns (double dt)
```

will advance the solution field (u_x and u_y) in time by a timestep dt . We already know that this will require the computation of the associated pressure field which we like to keep in between steps. So in `ns.h` we declare a global-scope field `p[N][N]` along with the velocity-component fields.

```
#include "common.h"
#include "poisson.h"

double ux[N][N], uy[N][N], p[N][N];

...

void projection (...) {
    ...
}

void advance_ns (double dt) {
}

...
```

As aforementioned, we will first compute the provisional velocity field. This is a separate update to the u_x and u_y scalar field according to the advection + diffusion equation. As such, we will first implement an `advance_scalar ()` function that makes a time step for scalar fields.

```
...
void projection (...) {
    ...
}

void advance_scalar (double s[N][N], double ux[N][N],
                    double uy[N][N], double kappa, double dt) {
    double ds_dt[N][N];
    tracer_advection (s, ux, uy, ds_dt);
    if (kappa > 0)
        diffusion (s, kappa, ds_dt);
    foreach()
        val(s, 0, 0) += dt*val(ds_dt, 0, 0);
}

void advance_ns (...) {
    ...
}
```

We can use this to compute the provisional velocity field \mathbf{u}^* in `advance_ns()` which we can not store in the original arrays, as they would be updated with different velocity fields.

```
...
void advance_ns (double dt) {
    // Declare and initialize and compute u_star
    double ux_star[N][N], uy_star[N][N];
```

```

foreach() {
    val (ux_star, 0, 0) = val(ux, 0, 0);
    val (uy_star, 0, 0) = val(uy, 0, 0);
}
advance_scalar (ux_star, ux, uy, nu, dt);
advance_scalar (uy_star, ux, uy, nu, dt);
}
...

```

To complete Chorin’s method, project \mathbf{u}^* and update the solution.

```

...
advance_scalar (uy_star, ux, uy, nu, dt);
// Project and update
projection (ux_star, uy_star, p, dt);
foreach() {
    val (ux, 0, 0) = val(ux_star, 0, 0);
    val (uy, 0, 0) = val(uy_star, 0, 0);
}
}
...

```

Two tests

We should now test the combined functions of our code. A difficulty with the non-linear Navier-Stokes equations is that finding non-trivial solutions is hard, which limits the ability to test a solver. Here we will consider two flow solutions, one will be investigated qualitatively, the other quantitatively.

The Lamb-Chaplygin dipole

I am proud to say that I lead authorship for the Wikipedia page on the Lamb-Chaplygin dipole. Quoting myself,

The Lamb–Chaplygin dipole model is a mathematical description for a particular inviscid and steady dipolar vortex flow. It is a non-trivial solution to the two-dimensional ~~Euler~~ [Navier-Stokes] equations. The model is named after Horace Lamb and Sergey Alexeyevich Chaplygin, who independently discovered this flow structure.³

Here we test if our solver can represent this steady solution. We will use the stream function (ψ) to initialize the flow field. Which are related by,

$$u_x = \frac{\partial \psi}{\partial y}, u_y = -\frac{\partial \psi}{\partial x}.$$

The stream function for the Lamb-Chaplygin dipole in cylindrical coordinates (r, θ) reads,

$$\psi = \begin{cases} U \left(\frac{-2J_1(kr)}{kJ_0(kR)} + r \right) \sin(\theta), & \text{for } r < R, \\ U \frac{R^2}{r} \sin(\theta), & \text{for } r \geq R, \end{cases}$$

However, this formulation is based on an infinitely large domain size. As such, we will focus on the (2D scalar) vorticity ($\omega = -\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}$) distribution,

³Meleshko, V. V., and G. J. F. Van Heijst. “On Chaplygin’s investigations of two-dimensional vortex structures in an inviscid fluid.” *Journal of Fluid Mechanics* 272 (1994): 157-182.

$$\omega = \begin{cases} kU \left(\frac{-2J_1(kr)}{J_0(kR)} \right) \sin(\theta), & \text{for } r < R, \\ 0, & \text{for } r \geq R, \end{cases}$$

where R is the size of the circular dipole, U is its translation velocity, J_0 and J_1 are the zeroth and first Bessel functions of the first kind, and the value for k is such that $kR = 3.8317$, the first non-trivial zero of the first Bessel function of the first kind. We can compute a suitable stream function via the (Poisson) relation,

$$\nabla^2 \psi = -\omega,$$

which can then be used to compute $\mathbf{u}(t=0)$.

Lets code it up in `lamb.c`. We start by carefully implementing the vorticity field (ω, omega) , solving for ψ (`psi`) and computing the initial velocity field.

```
#include "ns.h"
#include "visual.h"
#define RAD (sqrt(sq(x) + sq(y)))
#define SIN_THETA (RAD > 0 ? y/RAD : 0)
double U = 1, R = 1;

int main () {
    LO = 15;
    XO = YO = -LO/2.;
    double k = 3.8317/R;
    double omega[N][N], psi[N][N];
    foreach() {
        if (RAD < R)
            val (omega, 0, 0) = -k*U*(-2*j1(k*RAD))/(j0(k*R))*SIN_THETA;
        else
            val (omega, 0, 0) = 0;
            val (psi, 0, 0) = 0;
    }
    max_sweeps = 10000;
    poisson (psi, omega);
    // Compute initial velocity field
    foreach() {
        val (ux, 0, 0) = (val(psi, 0, 1) - val(psi, 0, -1))/(2*Delta) - U;
        val (uy, 0, 0) = -(val(psi, 1, 0) - val(psi, -1, 0))/(2*Delta);
    }
}
```

We should visually inspect the initialization by looking at the velocity components, and check if the vorticity is still entirely concentrated in a circular region. Hence we *diagnose* the vorticity field, reusing the `omega` field.

```
...
    val (uy, 0, 0) = -(val(psi, 1, 0) - val(psi, -1, 0))/(2*Delta);
}
output_ppm (ux, "ux.ppm", -U, U);
output_ppm (uy, "uy.ppm", -U, U);
foreach() {
```

```

    val(omega, 0, 0) = -(val(ux, 0, 1) - val(ux, 0, -1))/(2*Delta);
    val(omega, 0, 0) += (val(uy, 1, 0) - val(uy, -1, 0))/(2*Delta);
  }
  output_ppm (omega, "vort.ppm", -U*k, U*k);
}

```

Compile, run and inspect,

```

$ gcc -DN=300 lamb.c -lm
$ ./a.out
# Warning: max res = 0.00315803 is greather than its tolerance (0.0001)
$ convert ux.ppm uy.ppm vort.ppm +append xyo.png

```

After the inevitable syntax-error fixes, we can inspect `xyo.png`.

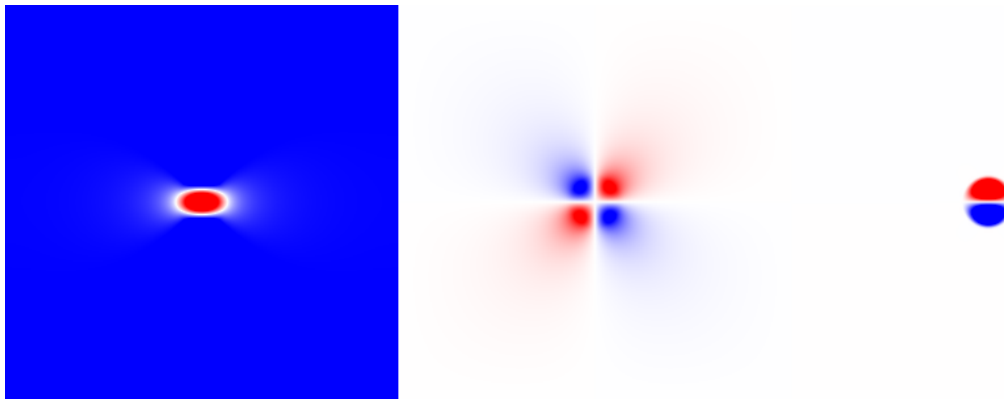


Figure 12: For inspection: u_x (left), u_y (middle) and ω (right)

This looks good and we can setup a time loop,

```

...
  output_ppm (omega, "vort.ppm", -U*k, U*k);

  // Time loop
  int iter = 0;
  double t_end = 2;
  for (t = 0; t < t_end; t += dt) {
    printf ("# i = %d, t = %g\n", iter, t);
    dt = dt_next(t_end);
    advance_ns (dt);
    iter++;
  }
}

```

To verify if our solution remained steady, we can output the vorticity field at the end of the simulation,

```

...
  iter++;
}
// Output vorticity field
foreach() {
  val(omega, 0, 0) = -(val(ux, 0, 1) - val(ux, 0, -1))/(2*Delta);
}

```



```

    val(omega, 0, 0) += (val(uy, 1, 0) - val(uy, -1, 0))/(2*Delta);
}
char fname[99];
sprintf (fname, "o-%d.ppm", N);
output_ppm (omega, fname, -U*k, U*k);
}

```

Compile run and check,

```

$ gcc -O3 lamb.c -lm
$ ./a.out
...
# i = 24, t = 1.7463
# i = 25, t = 1.83109
# i = 26, t = 1.91627

```

After a second or two of simulation I increase the resolution a bit and add an annotation whilst converting it to a .png to show here.

```

$ convert o-100.ppm -resize 300x300 -pointsize 30\
    -annotate +100+70 'N = 100' o-100.png

```

N = 100



Figure 13: The result for $N = 100$

The vortex structure seems to have survived(!) but in a deformed state. We should also check if this deformation at least decreases if we increase the accuracy of our computations.

```

$ gcc -DN=200 -O3 lamb.c -lm
$ ./a.out
...
$ gcc -DN=400 -O3 lamb.c -lm
$ ./a.out
...

```

Then we prepare our comparison (I did not put the ampersands (&)) so the block below maybe copied and pasted.

```
convert o-200.ppm -resize 300x300 -pointsize 30\  
    -annotate +100+70 'N = 200' o-200.png  
convert o-400.ppm -resize 300x300 -pointsize 30\  
    -annotate +100+70 'N = 400' o-400.png  
convert vort.ppm -resize 300x300 -pointsize 30\  
    -annotate +80+70 'Reference' vort.png  
convert o-100.png o-200.png +append o-top.png  
convert o-400.png vort.png +append o-bottom.png  
convert o-top.png o-bottom.png -append o-2x2.png
```

The resulting image (o-2x2.png) is shown below,

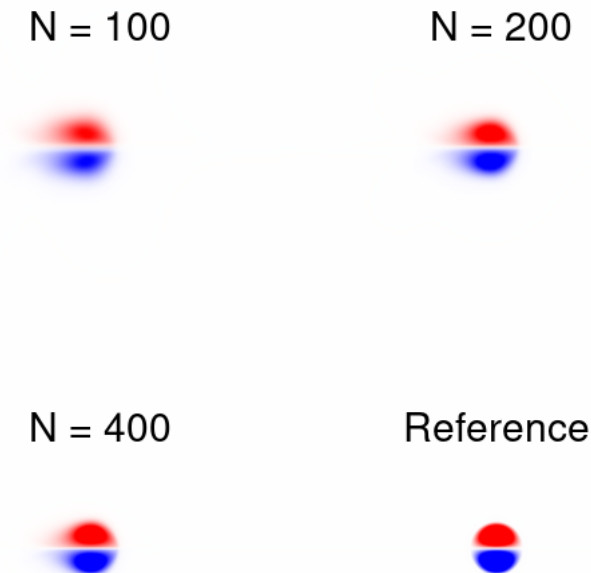


Figure 14: The obtained vorticity field does seem to match the reference better for larger values of N

I consider this a pass. Notice that this test is far from perfect, it is unclear what the effects of our finite $L0$ and the periodic boundaries are. But I do know (from experience) that this test is more critical than the qualitative test below.

Taylor-Green vortices

We will adhere to the tradition of testing new flow solvers against the Solution of Taylor and Green. On a periodic domain of size $2\pi \times 2\pi$,

$$u_x = \cos(x) \sin(y) e^{-2\nu^* t},$$

$$u_y = -\cos(y)\sin(x)e^{-2\nu*t}.$$

We will make it more interesting by evaluating it in a moving frame of reference. We will do a convergence test for the L_2 error. Using `tg.c`,

```
#include "ns.h"

#define UX (cos(x)*sin(y)*exp(-2*nu*t) + 1)
#define UY (-cos(y)*sin(x)*exp(-2*nu*t) + 1)

int main() {
    L0 = 2*3.14159265;
    nu = 0.1;
    // Initialize
    foreach() {
        val(ux, 0, 0) = UX;
        val(uy, 0, 0) = UY;
    }
    // Solve until t = 2*pi
    double t_end = L0;
    int iter = 0;
    for (t = 0; t < t_end; t += dt) {
        dt = dt_next(t_end);
        advance_ns (dt);
        iter++;
    }
    // Compute L2 for ux and uy combined
    double L2 = 0;
    foreach() {
        L2 += sq(Delta)*sq(val(ux, 0, 0) - UX);
        L2 += sq(Delta)*sq(val(uy, 0, 0) - UY);
    }
    printf ("%d %d %g %g\n", N, iter, t, L2);
}
```

Compile and compare the results

```
$ gcc -O3 tg.c -lm
$ ./a.out
100 1592 6.28319 0.18054
$ gcc -O3 -DN=200 tg.c -lm
$ ./a.out
200 6367 6.28319 0.0581281
$ gcc -O3 -DN=400 tg.c -lm
$ ./a.out
$ 400 25465 6.28319 0.0166069
```

Convergence looks OK since $\frac{0.18}{0.058} \approx \frac{0.058}{0.016} \approx 3.3$, which is a quite healthy. Given the the number of time steps has increases by a factor of 4 with doubling resolution indicates that the viscous term is most stringent for our time stepping stability. The factor 3.3 is a mix of the accuracy of the dominant diffusion term (second order accuracy would give a reduction factor of $(\frac{400}{200})^2 = 4$) and the first-order accuracy for the advection term ($(\frac{400}{200})^1 = 2$). In this case $2 + 4 \approx 3.3$.

The patient among us may want to push a bit more a do $N = 800$,

```
$ gcc -O3 -DN=800 tg.c -lm
$ ./a.out
Segmentation fault
```

This is an error raised by our operating system that does not allow to allocate so much memory for our fields on the stack. We could tell it to give us more.

```
$ ulimit -s
8192
// i.e in Kbytes default
$ su
// Enter password
# ulimit -s 100000
# ./a.out
```

This is not really a good solution. Using `malloc` is! But this would go against a core design principle of our solver, namely coding simplicity. Notice that the high-resolution runs are prohibitively expensive anyway.

Congratulations!

You have wrote a Navier-Stokes equation solver and It is therefore almost time to have some flow-related fun. First, we should make small improvements to the functionality of our code before we can really show-off some flow cases in a gallery. Lets code up some more function and an improved user interface.

11. Some improved functionalities

In this chapter we will improve our code to increase its function, and improve the user interface. These points are inspired by my wishes when composing the Gallery chapter. We consider four main topics,

1. An option to solve for a flow tracer field
2. An option for an additional body force to the Navier-Stokes equations
3. A user-interface for the time loop

(1 and 2) A tracer field and a buoyancy field

Since we already have the solving infrastructure, it would be nice to solve for the evolution of a tracer field as well. It can be implemented by adding a global field in `ns.h` called `tracer` and then update it during the `advance_ns()` step. Furthermore, it will prove useful to add an additional body-force momentum-source term to the vertical velocity (e.g. a gravity acceleration force). These two additions require very little new code to be added in `ns.h`,

```
...
double ux[N][N], uy[N][N], p[N][N];
double tracer[N][N], acceleration_y[N][N];
...

void advance_ns (double dt) {
...
    advance_scalar (uy_star, nu, dt);
    advance_scalar (tracer, kappa, dt);
    foreach()
        val(uy_star, 0, 0) += dt*val(acceleration_y, 0, 0);
    // Project and update
...
}
```

That it! See in the Gallery for its usage. But you may not recognize the clean case setup files before reading about...

(3) The User interface

I wish to implement a function called `run()` which will run the time loop (i.e. `for (t = 0; t < t_end; t += dt)`) for our solver. In its not user-friendly form we would add to `ns.h`,

```
...

double t_end = 1;

void run() {
    for (t = 0; t < t_end; t += dt) {
        // advance
        dt = dt_next(t_event);
        advance_ns(dt);
    }
}
```

were we could change `t_end` in our `.c`-case files to our desire. However, it would be nice to *optionally* break into this loop and add functions as we please. To illustrate, one may want to execute some function every iteration, but the function body is only defined in a `later.c`-case file. A method to do this is to use function *pointers*.

```

double t_end = 1;
// Function pointer
void (*every_iter)() = NULL;

void run() {
    for (t = 0; t < t_end; t += dt) {
        if (every_iter != NULL) // Do not excecute an undefined function
            every_iter();
        // advance
        dt = dt_next(t_event);
        advance_ns(dt);
    }
}

```

Using this, one may optionally assign a function to the void `every_iter()` function pointer in a `.c` file. It could look like,

```

...
void my_fun() {
    printf ("%g \n", t);
}

int main () {
    ...
    every_iter = my_fun;
    run();
}

```

This would result in the execution of the void `my_fun()` function every time step. Printing the value of the time parameter for every iteration. Similar, I would like functions that execute every `iter_interval` iterations (i.e. solver time steps) and every `t_interval` of time units. The code becomes

```

...
// Time loop variables
double t_interval = 1e8, t_end = 1;
int iter, iter_interval = 1e8;

// Function pointers
void (*every_t_interval)() = NULL;
void (*every_iter_interval)() = NULL;
void (*every_iter)() = NULL;

// A time loop
void run() {
    //(re)set
    iter = 0;
    t = 0;
    // time of next t_interval
    double t_event = 0;
    for (t = 0; t < t_end; t += dt) {
        // Call events if defined
        if (every_iter != NULL)
            every_iter();
        if ((iter % iter_interval) == 0 && every_iter_interval != NULL)
            every_iter_interval();
    }
}

```

```

    if (fabs(t - t_event) < 1e-8) { // Allow small binary-representation error
        if (every_t_interval != NULL)
            every_t_interval();
        // update t_interval
        t_event = min(t_end, t_event + t_interval);
    }
    // advance
    dt = dt_next(t_event);
    advance_ns(dt);
    iter++;
}
}

```

Finally,

I have added a `noise()` macro to `common.h` which computes a random value between -1 and 1.

```

#include <stdlib.h>
#define noise() ((double)((rand()%2000) - 1000)/1000.)

```

We will use this for the cases in the gallery

12. A gallery of fluid motion

On this page we solve some example flow problems, present their case file (.c) contents and the resulting movies. For a dummy file case.c the workflow is.

```
$ FILE=case
$ gcc -O3 -DN=250 $FILE.c -lm
$ ./a.out
...
$ convert $FILE-*.ppm $FILE.mp4.png
$ rm $FILE-*.ppm
```

A Kelvin-Helmholtz instability

A shear layer can roll up when it is perturbed. For our case, we consider a jet in the x direction, within which we initialize a tracer field.



Figure 15: Evolution of the tracer field

```
#include "ns.h"
#include "visual.h"
// The jet is two-thirds of L0 wide
#define JET (fabs(y) < L0/6.)

void output_frame() {
    double omega[N][N];
    vorticity (omega);
    char fname[99];
    sprintf (fname, "kh-%04d.ppm", iter);
    output_ppm (tracer, fname, -1, 1, false);
    // Log progress
    printf ("%d %g\n", iter, t);
}
```



```

int main() {
    L0 = 5;
    Y0 = -L0/2.;
    nu = 2e-4;
    t_end = 20;
    t_interval = .2;
    // Overload function
    every_t_interval = output_frame;
    // Initialize
    foreach() {
        val (ux, 0, 0) = JET ? 0.5 : -0.5;
        val (uy, 0, 0) += 1e-3*noise();
        val (tracer, 0, 0) = JET ? 1. : 0.;
    }
    // call solver..
    run();
}

```

Decaying two-dimensional turbulence

Two-dimensional turbulence is characterized by a cascade of small vortices towards larger ones over time. In our case, we initialize a Taylor-Green solution and perturb it such that we can see the unstable deformation and the resulting chaos.

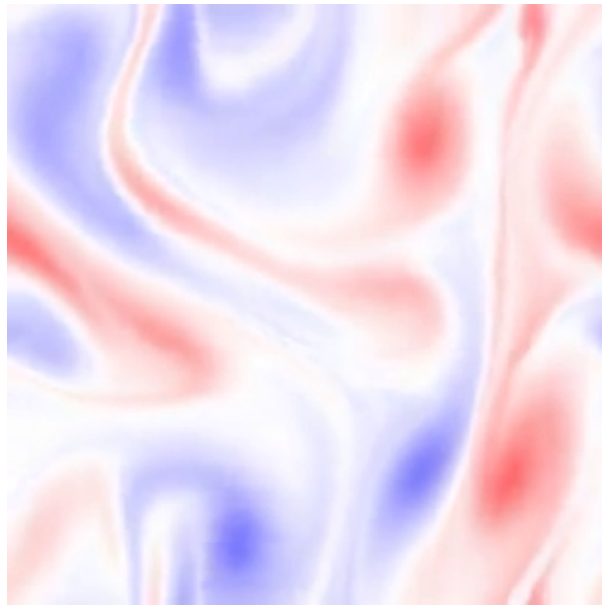


Figure 16: Evolution of the vorticity field

```

#include "ns.h"
#include "visual.h"

void output_frame() {
    double omega[N][N];
}

```

```

vorticity (omega);
char fname[99];
sprintf (fname, "turbulence-%04d.ppm", iter);
output_ppm (omega, fname, -.2, .2, false);
// Log progress
printf ("%d %g\n", iter, t);
}

int main() {
    L0 = 2.*pi;
    t_end = 500;
    t_interval = 2.;
    // Overload function
    every_t_interval = output_frame;
    // Initialize
    foreach() {
        val (ux, 0, 0) = sin(5*x)*cos(5*y);
        val (uy, 0, 0) = -sin(5*y)*cos(5*x);
        val (uy, 0, 0) += 0.001*noise();
    }
    // call solver..
    run();
}

```

Colliding dipolar vortices

Two vortex dipoles can collide and exchange vortices. Instead of following the previous Lamb-Chaplygin model, we project two localized Gaussian jets, targeted for a head-on collision



Figure 17: Evolution of the vorticity field

```

#include "ns.h"
#include "visual.h"

void output_frame() {
    double omega[N][N];
    vorticity (omega);
    char fname[99];
    sprintf (fname, "dip-%04d.ppm", iter);
    output_ppm (omega, fname, -.5, .5, false);
    // Log progress
    printf ("%d %g\n", iter, t);
}

int main() {
    L0 = 15;
    X0 = Y0 = -L0/2.;
    t_end = 50;
    t_interval = .5;
    // Overload function
    every_t_interval = output_frame;
    // Initialize
    foreach()
        val (ux, 0, 0) = exp(-sq(x + 3) - sq(y)) - exp(-sq(x - 3) - sq(y));
    projection (ux, uy, p, 1);
    // reset p
    foreach()
        val(p, 0, 0) = 0;
    // call solver..
    run();
}

```

Vortex rebound from a wall

A dipole may also find a solid obstacle on its path. A crude way to implement such a condition is to prescribe the solution of the stationary wall every timestep.

```

#include "ns.h"
#include "visual.h"
// The wall is located at the lhs of the domain
// such that it appears at the rhs
#define WALL (x < (X0 + L0/10.))

void output_frame() {
    double omega[N][N];
    vorticity (omega);
    char fname[99];
    sprintf (fname, "dip-wall-%04d.ppm", iter);
    output_ppm (omega, fname, -.1, .1, false);
    // Log progress
    printf ("%d %g\n", iter, t);
}

```

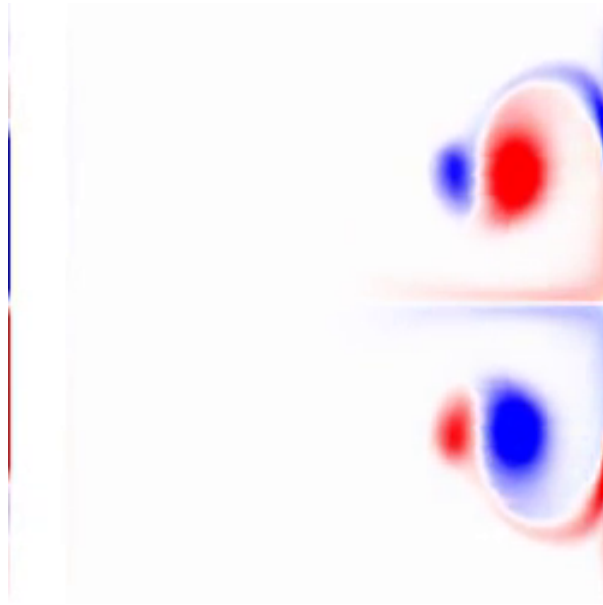


Figure 18: Evolution of the vorticity field

```

void boundary() {
    foreach() {
        if (WALL) {
            val (ux, 0, 0) = 0;
            val (uy, 0, 0) = 0;
        }
    }
}

int main() {
    L0 = 20;
    X0 = Y0 = -L0/2.;
    t_end = 250;
    t_interval = .5;
    nu = 1e-5;
    // Overload functions
    every_t_interval = output_frame;
    every_iter = boundary;
    // Initialize
    foreach()
        val (ux, 0, 0) = exp(-sq(x - 2) - sq(y));
    projection (ux, uy, p, 1);
    // reset p
    foreach()
        val(p, 0, 0) = 0;
    // call solver..
    run();
}

```

Rising warm plume

We can couple a tracer to the acceleration field to describe a buoyancy force with the Boussinesq approximation. In this case, we initialize a warm (buoyant) Gaussian plume



Figure 19: Evolution of the buoyancy field

```
#include "ns.h"
#include "visual.h"

void output_frame() {
    char fname[99];
    sprintf (fname, "plume-%04d.ppm", iter);
    output_ppm (tracer, fname, -.1, .1, false);
    // Log progress
    printf ("%d %g\n", iter, t);
}

void buoyancy() {
    foreach()
        val(acceleration_y, 0, 0) = val(tracer, 0, 0);
}

int main() {
    L0 = 20;
    X0 = Y0 = -L0/2.;
    t_end = 20;
    t_interval = .1;
    nu = 1e-5;
    // Overload functions
    every_t_interval = output_frame;
    every_iter = buoyancy;
    // Initialize
```

```

foreach()
    val (tracer, 0, 0) = exp(-sq(x) - sq(y + 5));
    // call solver..
    run();
}

```

A Rayleigh-Taylor instability

A heavy fluid resting on top of a lighter fluid creates an unstable stratification. Using the Boussinesq approximation, we initialize a sharp interface between a heavier and a lighter region.

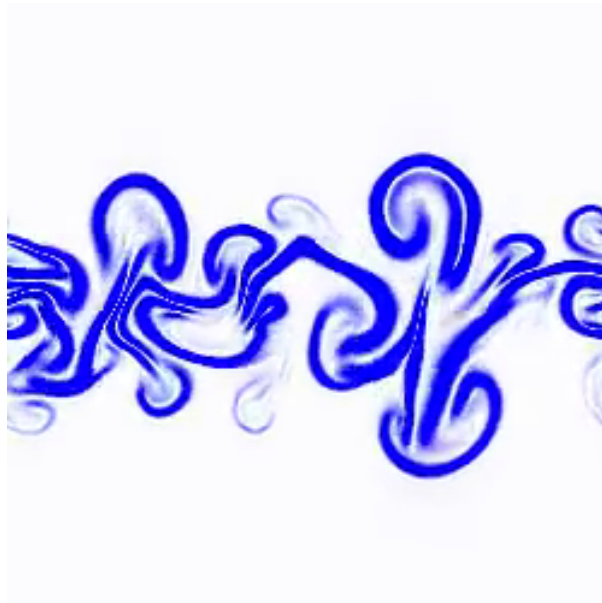


Figure 20: Evolution of the sharp interface, defined as the region with high gradients in the tracer field (i.e. $(\nabla s)^2$) from small to larger values displayed as white to blue, respectively

```

#include "ns.h"
#include "visual.h"

void output_frame() {
    double grad[N][N];
    foreach() {
        val(grad, 0, 0) = sq((val(tracer, 1, 0) - val(tracer, -1, 0))/(2*Delta));
        val(grad, 0, 0) += sq((val(tracer, 0, 1) - val(tracer, 0, -1))/(2*Delta));
        val(grad, 0, 0) = -(val (grad, 0, 0));
    }
    char fname[99];
    sprintf (fname, "rt-%04d.ppm", iter);
    output_ppm (grad, fname, -100, 100, false);
    // Log progress
    printf ("%d %g\n", iter, t);
}

void buoyancy() {

```

```

foreach()
    val (acceleration_y, 0, 0) = val (tracer, 0, 0);
}

int main() {
    L0 = 1;
    Y0 = -L0/2.;
    t_end = 5;
    t_interval = .05;
    nu = kappa = 1e-5;
    // Overload functions
    every_t_interval = output_frame;
    every_iter = buoyancy;
    // Initialize
    double sqN = 1;
    foreach() {
        val (tracer, 0, 0) = (y < 0 ? L0/2*sqN : - L0/2.*sqN) + sqN*y;
        val (uy, 0, 0) = 0.01*noise();
    }
    // Finding the hydrostatic pressure field requires some additional sweeping
    max_sweeps = 1e5;
    // call solver..
    run();
}

```

Because I cant help myself. I also discuss the good, the bad and the Ugly of our resulting solver here.

13. The good, the bad and the ugly

Here I share some thoughts on our solver design.

The good

I consider minimalism to be a good thing when doing a coding project and since I did my best to achieve this feature, I do believe our solver is a minimalist design for the functionality put on display in the gallery. It nicely served to highlight the concepts of the steps typically also tackled by more useful solvers. Further, I think it helps to better appreciate alternative methods. The user interface is also relatively good as I do think the codes of the Gallery are quite intuitive.

Another good thing is that we have written our solver in the C programming language. This is a proper language for high-performance computing and may be the only thing that you could take from this solver if you are to code a better one.

The bad

Our solver is time inefficient. The discrete approximations are not very accurate and therefore require relatively small discretization elements (time steps and mesh-element sizes) to achieve an acceptable solution. To make things worse, our implementation of the Poisson-problem solver is highly inefficient and the number of required sweeps for a given spatial problem actually increases with the resolution, making the solver not well behaved. For this reason, virtually none of the choices we have made in this regard are used in more useful solver designs.

The ugly

We have already encountered the problem where we cannot run our code with a large number of grid points. Memory allocation for the solution fields on the stack is obviously not the way to go.

In an earlier chapter, I wrote about the modular design of this solver for the purpose of future improvement. However, if almost every thing needs improving, a complete rewrite makes more sense.

What do other solvers do different?

As mentioned in the introduction, there is no consensus on how to solve flow problems most efficiently and numerical methods are often developed with a specific type of flow problems in mind. Aside the from coding details, I will list some solver-design choice alternatives which could have some relevant depending on the application.

Functions and features

Solvers typically advertise their merits by listing the type of flow problems they can handle. Examples include; multi-phase flows, flow in complex geometries, coupling to other solvers (multi physics), etc.

Method of Lines

The method of lines, where the mesh only discretizes the spatial dimensions, is often used in any solver that is not specifically investigating alternatives.

Finite differences

The finite difference method is not particularly unpopular. However, the finite volume method (FVM) is often preferred for solving problems described by a flux-divergence evolution equation. the FVM is

formulated such that the approximate solution to a conservation equation inherits this conservation property *exactly*. A second prominent alternative is the finite element method. Apart from the mathematical rigor and the grid-node layout flexibility, I can't think of many fundamental advantages for it in this context.

Co-located variables

Virtually all finite difference codes use a staggered grid, where the velocity components are defined at the faces in between grid cells. In practice, this allows to easily do an *exact* projection.

Cartesian grid

Our Cartesian grid is excellent for meshing the square domain. However, it would be nice if we could focus the cells a bit towards the regions of interest. This could be achieved with stretching and squeezing the mesh, but if you are doing many computations in complex geometries where only certain parts of the domain require a high resolution, an unstructured mesh would be more flexible. Noting that a quadtree structure may represent the best of both worlds.

Forward Euler time advancement

The forward Euler method is prone to instabilities and requires many small time steps for an accurate solution. It can thus become a computationally expensive option. Popular time-advancement schemes (both ex and implicit) can typically be casted in a Runge-Kutta formulation. Special attention is often given to the formulations that require least storage in memory. For some reason, linear multistep methods are also still around.

First-order accurate upwind advection

Although our first-order method has some excellent theoretical properties, the first order accuracy introduces large (diffusive) errors. There really exists a full zoo of advection-scheme alternatives. I like to highlight so-called compact numerical schemes as they can have excellent dispersion properties for a given degree of accuracy.

2nd-order accurate diffusion

This was quite an OK choice, although its explicit stability is a huge issue for viscous problems. For this reason, there are quite a few solvers that employ a so-called implicit + explicit (IMEX) scheme, where only the viscous term is treated implicitly.

Iterative Poisson solver

Iterative Poisson solving is often synonymous for also using multigrid acceleration because it does not really work well without it (as we have seen). Further, the sweeping design can be altered in many ways. Most notably, under relaxation may help to improve the convergence rate. Apart from the iterative method, on periodic grids, a (fast) Fourier transform would be a better option. This is often referred to as a spectral method, and we could actually have done all our differencing approximations in spectral space. Finally, (direct and indirect) matrix solver are also a popular choice to tackle linear problems.

Chorin's projection method

Chorin's projection method is quite popular but other operator splitting methods exist.