A Simple Fluid Solver based on the FFT

Jos Stam

Alias|wavefront
1218 Third Ave, 8th Floor,
Seattle, WA 98101

Abstract

This paper presents a very simple implementation of a fluid solver. Our solver is consistent with the equations of fluid flow and produces velocity fields that contain incompressible rotational structures and that dynamically react to user-supplied forces. The solver is specialized for a fluid which wraps around in space. This allows us to take advantage of the Fourier transform which greatly simplifies many aspects of the solver. Indeed, given a Fast Fourier Transform, our solver can be implemented in roughly one page of readable C code. We believe that our solver is a good starting point for anyone interested in coding a basic fluid solver. The fluid solver presented is also useful as a basic motion primitive that can be used for many different applications in computer graphics.

1 Introduction

Simulating fluids is one of the most challenging problems in engineering. In fact, over the last 50 years much effort has been devoted to writing large fluid simulators that usually run on expensive supercomputers. Simulating fluids is also important to computer graphics, where there is a need to add visually convincing flows to virtual environments. However, while large production houses may be able to afford expensive hardware and large fluid solvers, most computer graphics practitioners require a simple solution that runs on a desktop PC. In this paper we introduce a fluid solver that can be written in roughly one page of C code. The algorithm is based on our Stable Fluid solver [5]. We specialize the algorithm for the case of a periodic domain, where the fluid's boundaries wrap around. In this case, we can use the machinery of Fourier transforms to efficiently solve the equations of fluid flow. Although periodic flows do not exist in Nature, we think that they can be useful in many applications. For example, the resulting fluid flow can be used as a "motion texture map." Similar texture maps based on the FFT have been used in computer graphics to model fractal terrains [8], ocean waves [2, 7], turbulent wind fields [3, 6] and the stochastic motion of flexible structures [4], for example. Also, our solver is a good starting point for someone new to fluid dynamics, since the computational fluid dynamics literature and the complexity of most existing solvers can be quite intimidating to a novice. We warn the reader, however, that our method sacrifices accuracy for speed and stability: our simulations are damped more rapidly than an accurate flow. On the other hand, our simulations do converge to the solutions of the equations of fluid flow in the limit when both the grid spacing and the time step go to zero.

2 Basic Structure of the Solver

The class of fluids that we will simulate in this paper is entirely described by a velocity field that evolves under the action of external forces. The velocity field is defined on a regular grid of N^d voxels, where d is the dimension of the domain. To each voxel's center we assign a velocity. Typically, we consider only two-dimensional and three-dimensional flows. However, our algorithm also works for flows living in an arbitrary d-dimensional space. For the sake of clarity, we describe our algorithm in a two-dimensional space where the underlying concepts are easier to visualize. We assume that our grid has the topology of a torus: the fluid is continuous across adjacent boundaries. Figure 1 illustrates this situation. Because the velocity is periodic, it can be used to tile the entire plane seamlessly. Although such fluids do not occur in Nature, they can be useful in computer graphics as a motion texture map defined everywhere in space.

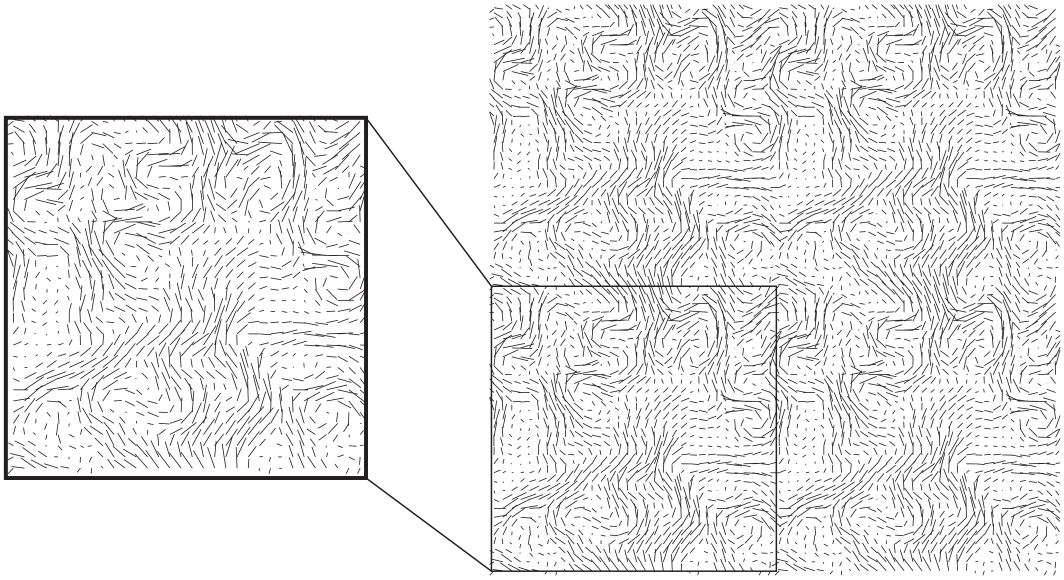

Figure 1: The fluids defined in this paper are periodic. The patch of fluid on the left can be used to seamlessly tile the entire plane.

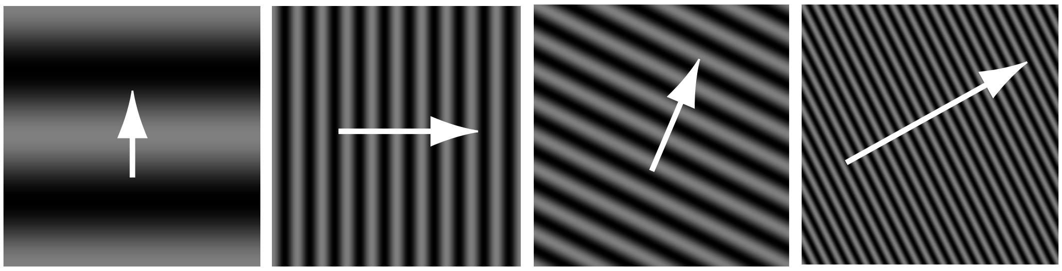

Figure 2: Four planar waves and their corresponding wavenumbers (white arrows).

Our solver updates the grid's velocities over a given time step `dt`. Consequently the simulation is a set of snapshots of the fluid's velocity over time. The simulation is driven by a user-supplied grid of external forces and a viscosity `visc`. The evolution of the fluid is entirely determined by these parameters. Subsequently, at each time step the fluid's grid is updated in four steps:

- Add force field
- Advect velocity
- Diffuse velocity
- Force velocity to conserve mass

The basic idea behind our solver is to perform the first two steps in the spatial domain and the last two steps in the Fourier domain. Before describing each step in more detail, we will say more about the Fourier transform applied to velocity fields.

3 A Fluid in the Fourier domain

The main reason to introduce the Fourier transform is that many operations become very easy to perform in the Fourier domain. This fact is well known in image and signal processing, where convolution becomes a simple multiplication in the Fourier domain. Before introducing Fourier transforms of vector fields we outline the basic ideas for single-valued scalar functions. For a more thorough treatment of the Fourier transform the reader is referred to any of the standard

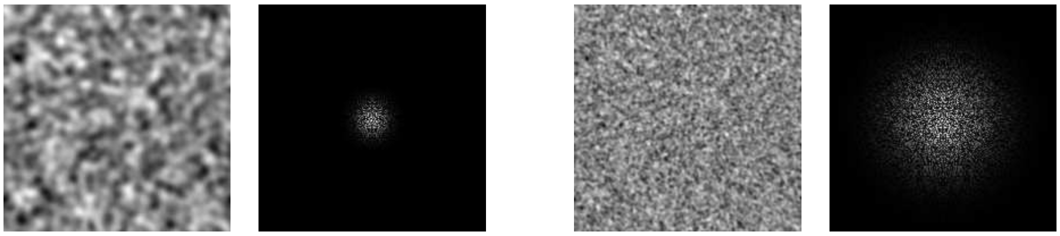

Figure 3: Two examples of a two-dimensional function and its corresponding FFT. The pair on the left is smoother than the one on the right.

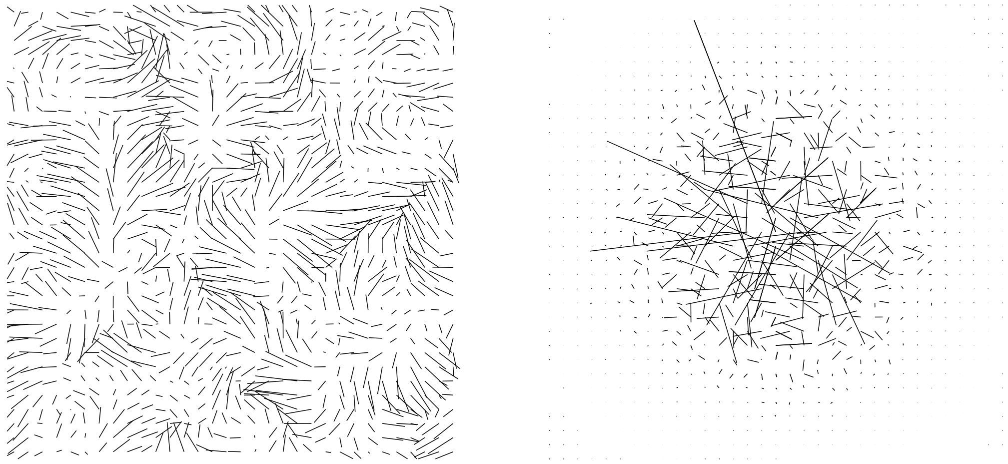

Figure 4: A velocity field (left) and the corresponding Fourier transform (right).

textbooks, e.g., [1]. Informally, a two-dimensional function can be written as a weighted sum of planar waves with different wavenumbers. In Figure 2 we depict four planar waves with their corresponding wavenumbers $\mathbf{k} = (k_x, k_y)$ (shown in white). The direction of the wavenumber is perpendicular to the crests of the wave, while the magnitude of the wavenumber $k = \sqrt{k_x^2 + k_y^2}$ is inversely proportional to the spacing between the waves. Larger wavenumbers therefore correspond to higher spatial frequencies. Any periodic two-dimensional function can be represented as a weighted sum of these simple planar waves. The Fourier transform of the signal is defined as a function that assigns the corresponding weight to each wavenumber $\mathbf{k}$. In Figure 3 we depict two pairs of functions with their corresponding Fourier transform. Notice that the smoother function on the left has a Fourier transform with weights that decay more rapidly for higher spatial frequencies than the one on the right.

To take the Fourier transform of a two-dimensional vector field, we independently take the Fourier transformations of the u components of the vectors and the v components of the vectors, each considered as a two-dimensional scalar field. Figure 4 shows a two-dimensional velocity field in the spatial domain (left) and the corresponding Fourier transform (right). The wavenumbers take values in the range $[-N/2, N/2] \times [-N/2, N/2]$, so that the origin is at the center. Since the velocity field is relatively smooth, we observe that the vectors are smaller for large wavenumbers, which is a common feature of most velocity fields. This observation directly suggests a method to account for the effects of viscosity which tend to smooth out the velocity field. Simply multiply the Fourier transform by a filter which decays along with the spatial frequency, the viscosity and the time step (see below).

As mentioned in the previous section not all operations are best performed in the Fourier domain. Therefore, we need a mechanism to switch between the spatial and the Fourier domain. Fortunately this can be done very efficiently using a Fast Fourier Transform (FFT). We will not describe an implementation of the FFT in this paper because very efficient public domain "black box" solvers are readily available. The one used in this paper is MIT's FFTW, the "Fastest Fourier Transform in the West."[1]

[1] Available at `http://www.fftw.org`.

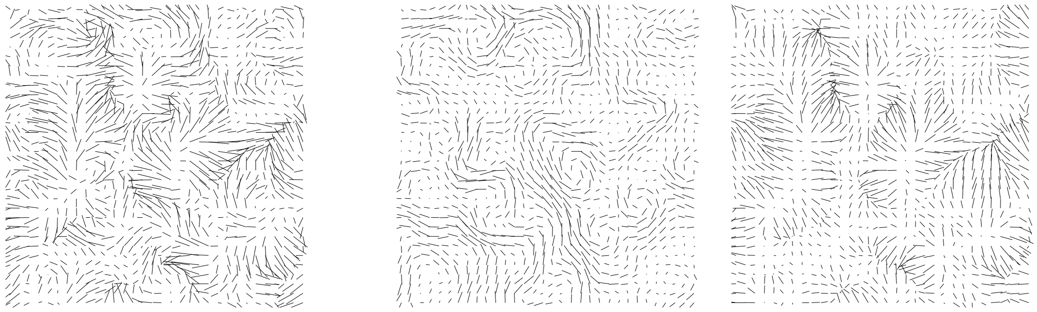

Figure 5: Any velocity field is the sum of a mass conserving field and a gradient field.

4 The Solver

We now describe in more detail the four steps of our solver. In Figure 7 we provide the corresponding implementation in C. Numbers between brackets in this section refer to the corresponding lines in the C-code. The input to our solver is the velocity of the previous time step `(u,v)`, the time step `dt`, a force field defined on a grid `(u0,v0)` and the viscosity of the fluid `visc [17-18]`. Because of the slight overhead in memory associated with the FFT, the arrays `u0` and `v0` should be of size `n*(n+2)`. Prior to calling the solver the routine `init_FFT` should be called at least once.

4.1 Adding Forces

This is the easiest step of our solver: we simply add the force grids multiplied by the time step to the velocity grid `[23-26]`. Consequently, in regions where the force field is large the fluid will move more rapidly.

4.2 Self-Advection

This step accounts for the non-linearities present in a fluid flow which makes them unpredictable and chaotic. Fluids such as air can advect (transport) many substances such as dust particles. In the latter case the particle's velocity is equal to that of the fluid at the particle's position. On the other hand, a fluid is also made up of matter. The particles that constitute this matter are also advected by the fluid's velocity. Over a time step these particles will move and transport the fluid's velocity to another location. This effect is called "self-advection": an auto-feedback mechanism within the fluid. To solve for this effect we use a very elegant technique known as a "semi-Lagrangian" scheme in the computational fluid dynamics literature `[28-38]`. Here is how it works. For each voxel of the velocity grid we trace its midpoint backwards through the velocity field over a time step `dt`. This point will end up somewhere else in the grid. We then linearly interpolate the velocity at that point from the neighboring voxels and transfer this velocity back to the departure voxel. This requires two grids, one for the interpolation `(u0,v0)` and one to store the new interpolated values `(u,v)`. To perform the particle trace, we use a simple linear approximation of the exact path. The interpolation is very easy to implement because our grids are periodic. Points that leave the grid simply reenter the grid from the opposite side.

4.3 Viscosity

We already alluded to this step in Section 3. We first transform our velocity field in the Fourier domain `[44]`. We then filter out the higher spatial frequencies by multiplying the Fourier transform by a filter whose decay depends on the magnitude of wavenumber, the viscosity and the time step `[46-60]`. A filter with all these properties which is consistent with the equations of fluid flow is given by `exp(-k^2*visc*dt)`. Since we are after visual accuracy, other possibly cheaper filters could be used.

4.4 Conservation of Mass

Most fluids encountered in practice conserve mass. Looking at a small piece of fluid, we expect the flow coming in to be equal to the flow coming out. However, after the previous three steps this is rarely the case. We'll correct this in a final step. As for viscosity, it turns out that this step is best performed in the Fourier domain. To solve for this

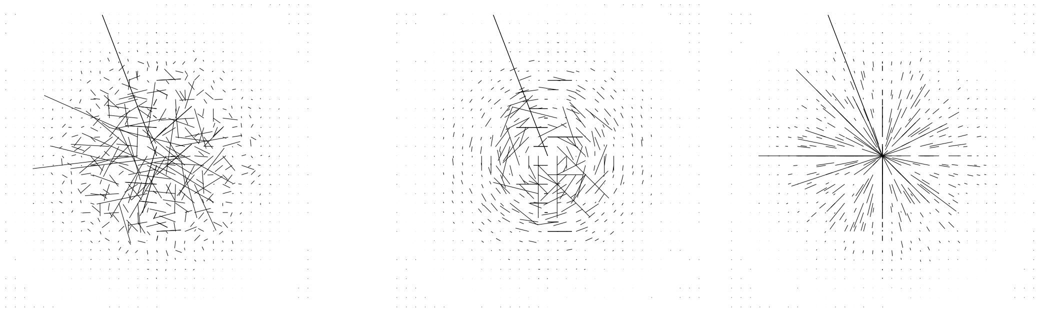

Figure 6: The corresponding decomposition of Fig. 5 in the Fourier domain.

step we use a mathematical result known as the "Helmholtz decomposition" of a velocity field: every velocity field is the sum of a mass conserving field and a gradient field. A gradient field is one that comes from a scalar field and indicates at each point in which direction the scalar field increases most. In two-dimensions the scalar field can be visualized as a height field. In this case, the gradient field simply points to the steepest upward slope. Figure 5 depicts the decomposition for an arbitrary field shown on the left. The first field on the right hand side is the mass-conserving field while the second one is the gradient field. Notice the amount of swirliness in the mass conserving field. Visually, this is clearly the type of behavior that we are seeking in computer graphics. The gradient field, on the other hand, is the worst case scenario: at almost every point the flow is either completely in-flowing or completely out-flowing.

In Figure 6 we show the corresponding fields in the Fourier domain. Notice how these fields have a very simple structure. The velocity of the mass conserving field is always perpendicular to the corresponding wavenumber, while the gradient field is always parallel to its wavenumber. Therefore, in the Fourier domain we force our field to be mass conserving by simply projecting each velocity vector onto the line (or plane in three dimensions) perpendicular to the wavenumber `[53-58]`. This is a very simple operation. The mathematical reason why the mass-conserving and gradient fields are so simple in the Fourier domain follows from the fact that differentiation in the spatial domain corresponds to a multiplication by the wavenumber in the Fourier domain.

Finally after this step we transform the velocity field back into the spatial domain `[62]` and normalize it `[64-67]`.

4.5 Parameters

Typical values for the parameters of the fluid solver are `dt=1`, `visc=0.001` and force values having a magnitude of roughly 10. These values should give nice animations of fluids.

5 Conclusions

We have shown that a simple fluid solver can be coded in roughly one page of C code. The solver as in [5] can be advanced at any speed and is unconditionally stable. This is highly desirable in computer graphics applications. Although the code is only given for a two-dimensional fluid, it can easily be extended to handle three-dimensional or even higher dimensional fluids. Applications of fluid dynamics to dimensions higher than three are unknown to the author. Perhaps the velocity (rate of change) of a set of d parameters might be governed by a fluid equation. We leave this as a topic for future research.

Web Information

The C-code in this paper is available at `http://www.acm.org/jgt/papers/Stam01`.

References

[1] R. N. Bracewell. *The Fourier Transform and its Applications (2nd Edition)*. McGrawHill, New York, 1978.

[2] G. A. Mastin, P. A. Watterberg, and J. F. Mareda. Fourier Synthesis of Ocean Scenes. *IEEE Computer Graphics and Applications*, 7(3):16–23, March 1987.

[3] M. Shinya and A. Fournier. Stochastic Motion - Motion Under the Influence of Wind. In *Proceedings of Eurographics '92*, pages 119–128, September 1992.

[4] J. Stam. Stochastic Dynamics: Simulating the Effects of Turbulence on Flexible Structures. *Computer Graphics Forum (EUROGRAPHICS'97 Proceedings)*, 16(3):159–164, 1997.

[5] J. Stam. Stable Fluids. In *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, pages 121–128, August 1999.

[6] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *SIGGRAPH 93 Conference Proceedings, Annual Conference Series*, pages 369–376, August 1993.

[7] J. Tessendorf. Simulating Ocean Water. In *SIGGRAPH'2000: Course Notes 25: Simulating Nature: From Theory to Practice*. 2000.

[8] R. F. Voss. Fractals in nature: From characterization to simulation. In *The Science of Fractal Images*, pages 21–70. Springer-Verlag, New York Berlin Heidelberg, 1988.

```c
#include <srfftw.h>

static rfftwnd_plan plan_rc, plan_cr;

void init_FFT ( int n )
{
   plan_rc = rfftw2d_create_plan ( n, n, FFTW_REAL_TO_COMPLEX, FFTW_IN_PLACE );
   plan_cr = rfftw2d_create_plan ( n, n, FFTW_COMPLEX_TO_REAL, FFTW_IN_PLACE );
}

#define FFT(s,u)\
if (s==1) rfftwnd_one_real_to_complex ( plan_rc, (fftw_real *)u, (fftw_complex *)u );\
else rfftwnd_one_complex_to_real ( plan_cr, (fftw_complex *)u, (fftw_real *)u )

#define floor(x) ((x)>=0.0?((int)(x)):(-((int)(1-(x)))))

void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ ) {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n ) {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n ) {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                       s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                       s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }

   FFT(1,u0); FFT(1,v0);

   for ( i=0 ; i<=n ; i+=2 ) {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ ) {
         y = j<=n/2 ?  j :  j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,u0); FFT(-1,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j]; }
}
```

Figure 7: Our fluid solver coded in C.