RANDOM VORTEX METHOD IN 2D

PAUL DWORZANSKI

Abstract. The purpose of this report is to help the author better understand mathematical fluids, specifically the random vortex method as discussed in Chapter 6 of Majda's textbook [1].

1. Introduction

The first section will discuss three versions of Navier-Stokes equation. Section two will discuss random vortex methods to approximate solutions to the Navier-Stokes equations. Section three will present the algorithm. Section four will discuss an the implementation of the random vortex algorithm.

2. Fluid Equations

2.1. **Navier-Stokes Equations.** We begin by defining a fluid as an incompressible deformable continuum, for which motion is completely defined by a velocity field, $\mathbf{v}(\mathbf{x}, t)$, and a pressure field, $p(\mathbf{x}, t)$. Assuming constant density and ignoring temperature, one can apply Newton's second law ($mass \cdot acceleration = \sum Forces$) to an infinitesimal particle to derive the Navier-Stokes equations,

$$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} = -\nabla p + \nu \Delta \mathbf{v} \tag{2.1}$$

$$\nabla \cdot \mathbf{v} = 0 \tag{2.2}$$

where ν is the constant coefficient for viscosity of Newtonian fluids. The case when $\nu = 0$ is called Euler's equation.

2.2. **Vorticity Equations.** By taking the *curl* of 2.1, and employing linear algebra identities, one can rewrite the 2d Navier-Stokes equations in their vorticity form,

$$\frac{\partial \omega}{\partial t} + (\mathbf{v} \cdot \nabla)\omega = \nu \Delta \omega \tag{2.3}$$

where $\omega(\mathbf{x}, t) = \nabla \times \mathbf{v}$ is the vorticity. Intuitively, vorticity is the local rotation of an infinitesimal particle. Noteworthy is that for the Euler case ($\nu = 0$), vorticity is conserved by individual particles.

2.3. **Integral Differential Equations.** Using Feynman-Kac theorem, the vorticity equation 2.3 can be rewritten as the Forward Kolmogorov (or Fokker-Planck) equation

$$\mathbf{X}(\alpha, t) = \alpha + \int_{\mathbb{R}^2} \mathbf{K}[\mathbf{X}(\alpha, t) - \mathbf{X}(\alpha', t)]\omega_0(\alpha')d\alpha' + \sqrt{2\nu}\mathbf{W}(t) \tag{2.4}$$

where the deterministic position $\mathbf{x}$ changed to stochastic process $\mathbf{X}$, $\alpha = \mathbf{X}(\alpha, 0)$ is the initial position of the particle $\mathbf{X}(\alpha, t)$, $\mathbf{W}(t)$ is a Wiener process, and the kernel $\mathbf{K}(\mathbf{x}) = \frac{1}{2\pi|\mathbf{x}|^2}(-x_2, x_1)$. All vectors are 2d. This can be viewed as a system of interacting convection-diffusion processes. This is the equation that is implemented in computer code.

3. Random Vortex Method

The idea is to release a finite number of vortex cores into space with initial positions α and initial vorticities $\omega_0(\alpha)$, and allow them to interact with respect to 2.4. At each time step, their initial position, α, moves through the nonlocal integral operator in 2.4, and then randomly perturbed by a Wiener process. This algorithm is repeated for each time step. In an ensemble, the distribution of voriticity should approximate that given by 2.3. Pseudo-code for the algorithm follows.

```
set initial positions α and vorticities ω_0(α)
for number of time steps do
  for each vortex i do
    for each other vortex j do
      add vortex j's contribution to integral operator in 2.4 for vortex i
    end for
    add result of above loop to vortex i's current position
    perturb vortex i's position by a Wiener process
  end for
end for
```

4. Implementation of Algorithm

4.1. **Design.** A computer program was written to implement the random vortex algorithm. The following design considerations were made.

- The C programming language was used for speed and flexibility.
- Second order midpoint Runge-Kutta was used for forward time integration of the integral operator of 2.4.
- Data arrays were dynamically allocated; keeping sequential data next to each other, lowering latency of memory caching.

4.2. **Analysis.** The algorithm is $O(n^2)$ where n is the number of vortex cores. Each vortex core interacts with all other cores through the integral operator of 2.4. Perhaps this algorithm cab be sped up by grouping nearby vortexes to act as one vortex.

4.3. **Use.**

4.3.1. *Installation Guide & Tutorial.* The program depends on `libc`, a `c` compiler (`gcc` was used), and `gnuplot` for optional plotting. The program is compiled with a simple `make` command. The program is run in default mode with a simple `./main.exec`, or with arguments from the command line `./main.exec -ns 100 -ft 90 -nv 8 -c 2 -gp 0`. The command line arguments are listed below.

```
-ns  number of time steps
-ft  final time
-nv  number of vortexes
```

```
-nu  coefficient of vorticity
-c   test case, 1=Random, 2=Circle, 3=Double circle
-gp  toggle on/off gnuplot
-gps gnuplot plot size
```

Some results are shown in Figure 1.

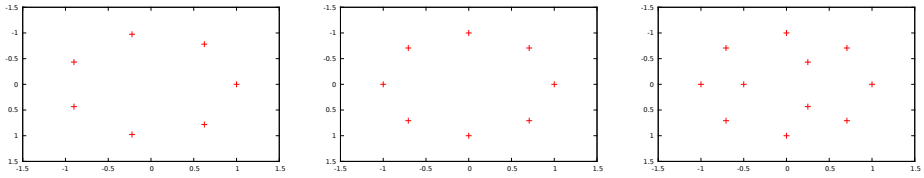

Figure 1. A sanity check. When $\nu = 0$, vortex cores should remain in stable rotation for $n <= 7$. Stable rotation is illustrated in (a) for 7 vortex cores; eight vertex cores (b) is unstable after many time steps; exotic configurations such as (c) are also known to be stable.

4.3.2. *Reference Manual.* A user interface is provided, with which the user can populate initial position and vorticity data for finite number of vortex cores, and propogate their interaction forward in time. The dynamics routines are declared in `dynamics.h` (listing 1). Optional plotting routines are declared in `gnuplot.h` (listing 2).

4.3.3. *Developer Guide.* The interface is employed in a user program found in listing 3. The user program can be modified to test specific cases, or expanded with additional functionality. A prerequisite is experience with the `c` programming language.

References

1. A.J. Majda and A.L. Bertozzi, *Vorticity and incompressible flow*, Cambridge Texts in Applied Mathematics Series, Cambridge University Press, 2002.

Listing 1. dynamics.h

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5  #include <time.h>
6
7  //data for each vortex, and other dynamics data
8  typedef struct {
9    double * positions; //position data
10   double * midPos; //temporary position data for RK2
11   double * velocities; //velocity data
12   double * vorticities; //vorticity data
13   int numVortexes; //number of vortexes used
14 } dynamics_data;
15
16 //init dynamics data structure
17 int dynamics_init(dynamics_data * d, int numVortexes);
18
19 //populate dynamics data
20 int dynamics_populate(dynamics_data * d, int width, int height, int c);
21
22 //print dynamics data
23 void dynamics_print(dynamics_data * d);
24
25 //BiotSavart eqn for vorticities, O(n^2) time where n=#vortexes
26 void biot_savart(double *pos, double *velocities, double *vorticities,
27                  int numVortexes);
28
29 //standard Euler's method to step forward in time
30 void euler(double *pos, double *vel, double dt, double *out,
31            int numVortexes);
32
33 //generate random gaussian
34 double gaussrand();
35
36 //main interface to user, steps forward in time
37 int compute_one_time_step(dynamics_data * d, double nu, double dt);
```

Listing 2. gnuplot.h

```
1
2  //data for gnuplot
3  typedef struct {
4    FILE *gp;
5  } gnuplot_data;
6
7  //initialize gnuplot data structure
8  int gnuplot_init(gnuplot_data *gd);
9
10 //plot data
11 int gnuplot(gnuplot_data *gd, dynamics_data * d, double dt, double size);
```

Listing 3. snippet user code main.c

```
1
2    //initialize dynamics data arrays
3    dynamics_data dynamics;
4    dynamics_init(&dynamics, numVortexes);
5    dynamics_populate(&dynamics, width, height, test_case);
6
7    //initialize gnuplot
8    gnuplot_data gd;
9    gnuplot_init(&gd);
10
11   //print and plot initial dynamics data
12   if (printFlag) dynamics_print(&dynamics);
13   if (gnuplotFlag) gnuplot(&gd, &dynamics, dt, gnuplotSize);;
14
15   //main loop
16   int Tstep;
17   for(Tstep=0;Tstep<numTimeSteps;++Tstep){
18     compute_one_time_step(&dynamics,nu, dt);
19     if (printFlag) printf("%f\n",Tstep*dt);
20     if (gnuplotFlag) gnuplot(&gd, &dynamics,dt,gnuplotSize);
21   }
```