Cornell CS 322 Introduction to Scientific Computing	*out: Mon Apr 14, 2008*
Project #3: Particle Advection (or "Going With the Flow")	*due: Sat May 3, 2008*
By Jeffrey Pang (TA) and Prof. Doug James	*(before midnight)*

In this assignment, you will estimate the trajectories of particles flowing along a velocity field, $\mathbf{v}(t, \mathbf{p})$, by solving Initial Value Problems using an ODE integrator. Following the theme of "particle advection," you will investigate numerical integration of ODEs and tools for scientific visualization in Matlab:

- Section 3 investigates numerical approximations to solutions of ordinary differential equations ("ODEs").
- Section 4 explores the visualization of particulate flows, and culminates in the creative modeling of smoke, fire, etc.

We begin with some background on particle advection and its relationship to ODE integration.

1 Introduction to Advection

Advection refers to the process by which matter is moved along, or advected, by a flow. Such flows can be modeled by a velocity field, $\mathbf{v}(t, \mathbf{p}) \in \mathbb{R}^d$ which specifies the velocity at position $\mathbf{p} \in \mathbb{R}^d$ at time $t \in \mathbb{R}$. If we consider a massless particle at position $\mathbf{p}$, we can model its advection in the flow using the following first-order ordinary differential equation (ODE),

$$\dot{\mathbf{p}} = \mathbf{v}(t, \mathbf{p}) \quad \Longleftrightarrow \quad \frac{d\mathbf{p}}{dt} = \mathbf{v}(t, \mathbf{p}). \tag{1}$$

If we specify the initial position $\mathbf{p}_0$ of the particle at time $t = t_0$, then the resulting trajectory is specified by the Initial Value Problem (IVP):

$$\dot{\mathbf{p}} = \mathbf{v}(t, \mathbf{p}), \quad \mathbf{p} \in \mathbb{R}^d, \quad t \geq t_0, \tag{2}$$

subject to

$$\mathbf{p} = \mathbf{p}_0 \quad \text{at} \quad t = t_0. \tag{3}$$

In practice, one can solve these advection equations using a numerical ODE integrator, such as Euler's method. By simply advecting and rendering very many particles, a variety of special effects that mimic the behavior of smoke, water, sand, or fire can be produced (see Figure 1).

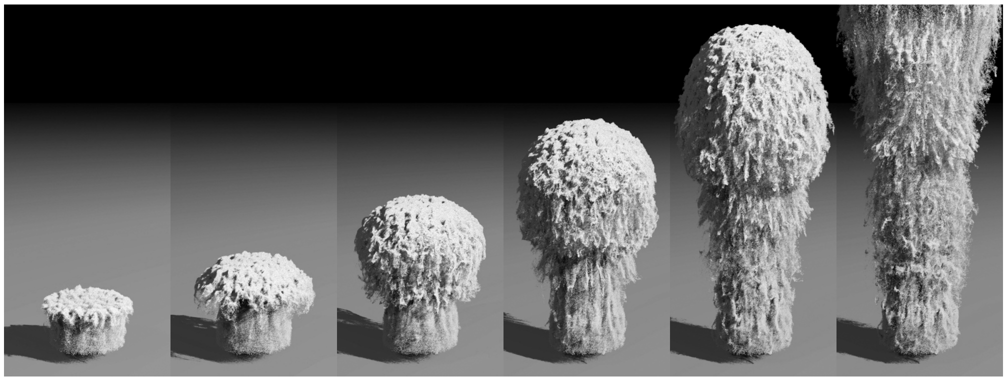

Figure 1: **Advecting particles in a turbulent velocity field (from [KTJG08]):** Particles are inserted randomly and continually at the inlet to the flow (near ground) and carried along (advected) by the velocity field. Although this 3D example is rather complex, in this assignment you will be able to create similar smoke and fire effects using numerical ODE integrators and 2D velocity field data we provide.

Advecting Particles using 2D Fluid Flow Data: In this assignment, you will advect particles in a *two-dimensional* ($d = 2$) velocity field, so that

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad \text{and} \quad \mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}, \tag{4}$$

and equation (1) is actually a system of two coupled ODEs,

$$\dot{p}_x = v_x(t, p_x, p_y) \tag{5}$$

$$\dot{p}_y = v_y(t, p_x, p_y). \tag{6}$$

You will approximate the solution to each particle's IVP (2-3) using a numerical ODE integrator similar to those you have learned about in class. Thus, given the initial conditions $(t_0, \mathbf{p}_0)$ for any particle, you will be able to approximate the particle's trajectory numerically (see Figure 2).

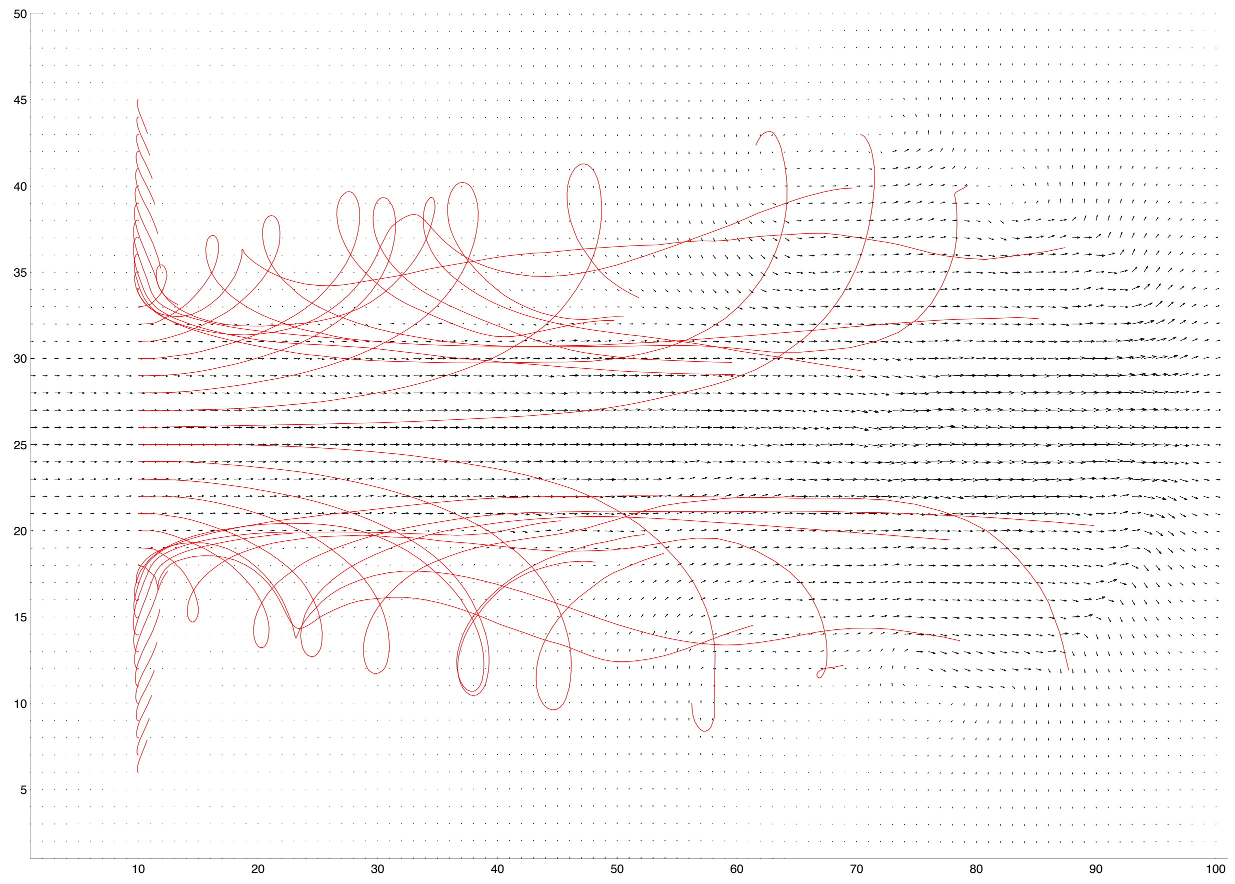

Figure 2: **Trajectories of advected particles** shown along with the velocity field at the current/final time. Observe that the final state of each trajectory is tangent to the current flow direction. (NOTE: In this assignment, we use a horizontal flow direction for plotting convenience, even though the physical flow was vertical. Feel free to rotate your plots and final renderings if you please.)

Velocity Field Dataset: To make the assignment more interesting, we will provide you with time-dependent velocity field data from a 2D simulation of incompressible flow based on the Navier-Stokes equations. The data is provided in the file `data_bare.mat`, and can be loaded using the Matlab command `load('data_bare.mat')`. An illustration of the velocity field data is shown in Figure 3. The dataset consists of a time series of 2D velocity fields sampled on a fixed-resolution spatial grid. For your convenience, we also provide an implementation of the velocity field function, $\mathbf{v}(t, \mathbf{p})$, based on tri-cubically interpolating grid data at the requested (t, p_x, p_y) position.

About "time" in the Matlab code: The data illustrated in Figure 3 shows the dataset's range of time, $t \in [1, 101]$. In the remainder of this assignment, we will consider initial value problems (IVPs) with a common $t_0 = 1$, but different initial positions, p_0. Unless otherwise specified, all trajectories are integrated out until time $t_1 = 101$.

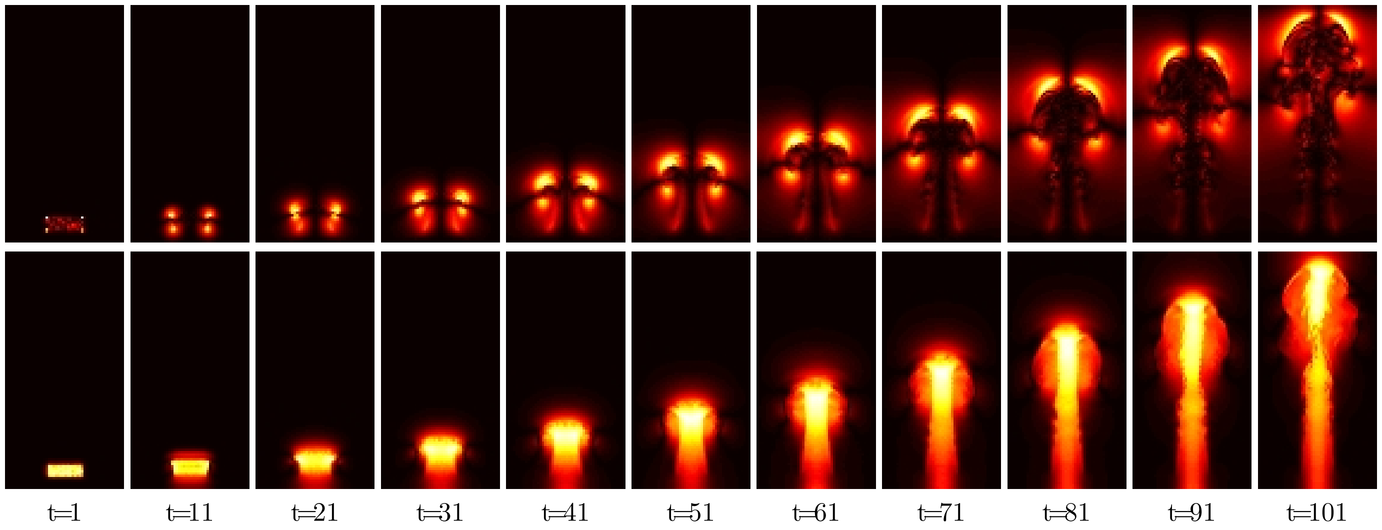

Figure 3: **Fluid flow dataset:** Snapshots of the velocity field function, $\mathbf{v}(t, \mathbf{p})$: (Top) horizontal velocity field, $v_x(t, \mathbf{p})$; (Bottom) vertical velocity field, $v_y(t, \mathbf{p})$. In each case, the absolute value of the velocity field is shown. (NOTE: Plot orientation has been changed to match the vertical direction of the physical flow.)

2 A First Step

This assignment is structured as a sequence of tasks with submission materials that you should include in your report write-up (details in §6). Each such task is indicated by a $\boxed{\text{STEP \#0}}$ box.

$\boxed{\text{STEP \#1}}$ Examine the code `solve_ODE.m`. Observe that "block A" solves the ODE that governs the motion of a single particle:

$$\frac{d}{dt}(\text{p\_part}) = v(t, \text{p\_part}) \tag{7}$$

using the built-in ODE solver `ode45`[1] (with default options) at the 101 time steps `1:101`. For example the motion of a single particle could be approximated by solving the IVP with initial condition $t_0 = 1$ and $p_0 = (10, 40)$ using

p = solve_ODE(x_array, y_array, [10, 40]);

Similarly, we can also integrate trajectories associated with multiple IVPs using $t_0 = 1$ but multiple initial positions, p_0 =`[10,k]`, for $k = 11 \ldots 40$. The actual line of code that performs these multiple ODE integrations is

[1]The `ode45` method is actually a Runge-Kutta method.

p_rk = solve_ODE(x_array, y_array, [ones(30,1)*10, (11:40)']);

We denote the output by `p_rk`, and will use `p_rk` in the following Section 3. There is no need to submit anything for this first step.

3 PART I: Investigating ODE solvers in Matlab

In this section, we will investigate the forward Euler and midpoint method for solving ODEs in Matlab, as well as Matlab's own built-in solvers.

3.1 Implementing Basic ODE Integrators

$\boxed{\text{STEP \#2}}$ Implement the **forward Euler method** and the **midpoint method** in the empty files `ode_euler.m` and `ode_midpoint.m`. Use the same basic calling signature as other Matlab ODE solver functions, e.g.,

[T, P] = ode45(odefun, tspan, p0)

where the vectors `T` and `P` return particular solutions at the specific times defined by `tspan`–you can assume that the time samples are increasing. Implement your ODE solvers so that `P(i)`, the solution at `tspan(i)`, is determined using *one step* of the forward Euler/midpoint method from `P(i-1)` at `tspan(i-1)` with step size `(tspan(i)-tspan(i-1))`.

3.2 Comparing Different ODE Solvers in Matlab

In this subsection, we will explore the ODE solvers in Matlab compared to the forward Euler and the midpoint method.

$\boxed{\text{STEP \#3}}$ By modifying the file `solve_ODE.m`, run the forward Euler method for fixed step sizes of h =1, 0.5 and 0.25. Also, run the midpoint method for fixed step sizes of h =1 and 0.5, and reuse the Runge-Kutta solution `p_rk`. Rename your solutions `p_1`, `p_05`, `p_025`, `p_m1`, `p_m05` and `p_rk`. In each case, use the same multiple initial conditions as in the definition of `p_rk` (specified in §2). Compare your solutions for the forward Euler method, and midpoint method, to the Runge Kutta method by running `plot_traj` as follows:

plot_traj(p_input, [r g b]).

where `p_input` is the input solution obtained from `solve_ODE`, and the 1×3 vector, `[r g b]`, specifies the plot color. Modify trajectory plot colors to make your results clear; use the "`hold on`" command to overlay trajectory plots. *Do you see agreement between the trajectories? Describe your observations.*

$\boxed{\text{STEP \#4}}$ Compare your solutions for a single initial condition, $\mathbf{p}_0 = (10, 26)$ (at $t_0 = 1$), by running:

plot_traj(p(:,[31, 32]), [0 0 0])

Do you see strong agreement between the trajectories?

$\boxed{\text{STEP \#5}}$ Next you will run `solve_ODE` using other built-in Matlab ODE solvers (with default settings), and compare their solutions for the multiple initial conditions (of §2) to the midpoint method's solution with $h = 0.5$ (denoted `p_m05`). To compute the difference between trajectories' *final* state, $\mathbf{p}_1 = \mathbf{p}(t = t_1 = 101)$, use the Euclidean final position error estimate (at time $t = 101$),

E_final = norm(p_m05(101,:)-p(101,:))

where `p` denotes a particular ODE approximation. By sorting `E_final` values in decreasing order for the solvers `ode113`, `ode15s`, `ode23`, `ode23s`, `ode23t`, `ode23tb` and `ode45`, you should see the following results[2]:

56.4415, 51.1320, 40.9767, 36.0481, 22.4244, 13.7323, 10.4090.

Which solver has `E_final` *error corresponding to the value 10.4090?*

$\boxed{\text{STEP \#6}}$ Read the documentation of the best ODE solver in the last problem. You will find that it is a backward method: The Jacobian matrix is formed as the solver runs, and advancing the differential equation involves solving a linear equation involving the Jacobian. According to the documentation, how is the Jacobian matrix formed?

3.3 Running ODEs Backwards

Recall the differential equation $\dot{\mathbf{p}} = \mathbf{v}(t, \mathbf{p})$. Given the starting time t_0, ending time t_1, where $t_1 > t_0$, and initial condition $\mathbf{p}_0$, we can find a numerical approximation to the solution to the ODE, $\mathbf{p}(t)$ for $t \in [t_0, t_1]$. Here, $\mathbf{p}(t)$ satisfies $\mathbf{p}(t_0) = \mathbf{p}_0$, and usually, $\mathbf{p}_1 := \mathbf{p}(t_1)$ is of interest.

$\boxed{\text{STEP \#7}}$ In the above description, $\mathbf{p}_1$ is obtained when t_0, t_1 and $\mathbf{p}_0$ are given and $\mathbf{v} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is a known function. One can solve the ODE backward in time allows us to obtain $\mathbf{p}_0$ when t_0, t_1 and $\mathbf{p}_1$ are given instead. A simple transformation of variables allows us to solve a certain ODE forward in time instead. Find that ODE, stating clearly the initial conditions and time interval.

$\boxed{\text{STEP \#8}}$ Take the array `p_m05` to be the most accurate solution to the ODE we have. We now have an approximation of the end state $(t_1, \mathbf{p}_1)$ corresponding to a number of initial positions at $t_0 = 1$: $\mathbf{p}_0 = (10, 11), (10, 12), (10, 13), \ldots, (10, 40)$. Theoretically, if we integrate any particular initial condition $(t_0, \mathbf{p}_0)$ forward in time to $(t_1, \mathbf{p}_1)$, then solving the ODE backwards in time from the end state $(t_1, \mathbf{p}_1)$ should arrive back at the original initial condition, $(t_0, \mathbf{p}_0)$. You will now investigate this numerically as follows:

- Use the midpoint method ($h = 0.5$) to estimate the end state $(t_1, \mathbf{p}_1)$ corresponding to the initial condition $\mathbf{p}_0 = (10, 14)$ (at $t_0 = 1$). Now reverse integrate from the end state $(t_1, \mathbf{p}_1)$ back to the initial condition, at time $t_0 = 1$. You should get $(10.6775, 13.9142)$ as the approximation to $\mathbf{p}_0$.
- Repeat this process for the trajectory with initial condition $\mathbf{p}_0 = (10, 26)$, and report your approximation to $\mathbf{p}_0$. What does this say about the numerical solution of ODEs in general?

4 PART II: Flow Visualization

In this second part you will investigate different methods for visualizing particles advected by the flow. In addition to exploring some of Matlab's visualization tools, you will also get to make some pretty pictures and movies of smoke and fire.

4.1 Basics: Making Fire with `colormap`

$\boxed{\text{STEP \#9}}$ Examine the code `fire_movie.m`. As it is, the code prints out trajectory points in black, which you can check by running:

```
fire_movie([0 50 0 101], 0.001, p_rk);
```

where `p_rk` is from STEP #1 in equation (7). Amend the code in the highlighted block so that `fire_color` is the colormap in the `hot` color palette. Upon running the command above again, you should see a plot that looks something like a fire. *Include a picture of the color plot in your submission.*

[2] If you do not see these numbers, and have verified that you have not made a mistake, please contact Jeffrey Pang (TA). Note that different Matlab versions may cause discrepancies. Be sure to verify that you are computing the error in the final position using `norm(p_m05(101,:)-p(101,:))` and not just `norm(p_m05-p)`.

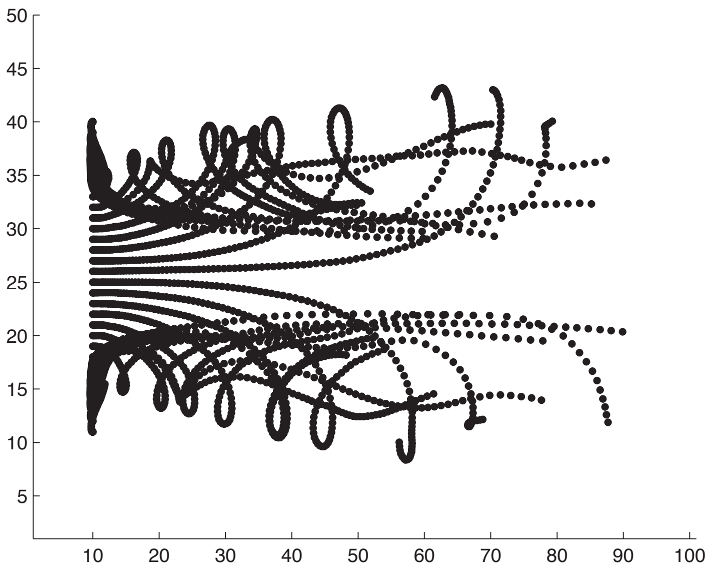

Figure 4: **A bad fire rendering:** Can you do better?

4.2 Basics: Making a Movie

STEP #10 In this step, you will examine the code `vector_field_movie.m` and modify it to animate flow trajectories. The program still requires some editing, so follow the instructions therein to update the sequence of plots to create an animation. Test your modified function by running:

```
F = vector_field_movie(x_array, y_array, [1 101 1 50], 0.01, p_rk);
movie(F);
```

What matlab commands can be used to make a video file out of the structure `F`? (Do not submit a video file for *this* part since it is large.)

4.3 Smokey Particles

Smoke can be modeled by advecting numerous tiny black "soot" particles or larger "blobbies." In practice, drawing a simple hard-edged circle may not be visually pleasing, especially if you don't have very many particles. Instead, it can be better to use "blurry particles," that both increase in radius, and fade away with time (see Figure 5).

STEP #11 Running the code `diffusion.m` should produce a movie of advected dark circles that expand as time goes on. Rewrite the code in `stencil.m` so that:

1. the density of the circles is higher in the middle than at the edges, and
2. the shading gets less dense as the circles get bigger over time.

Explore such Matlab plotting options to make the smoke particles look as good as you can.

4.4 Creating Particles Randomly at the Inlet

Creating particles at fixed inlet locations leads to an unnatural looking stream of particles, as you probably observed in the previous sections, e.g., see Figure 4. To get a more natural flow, you should use Monte Carlo

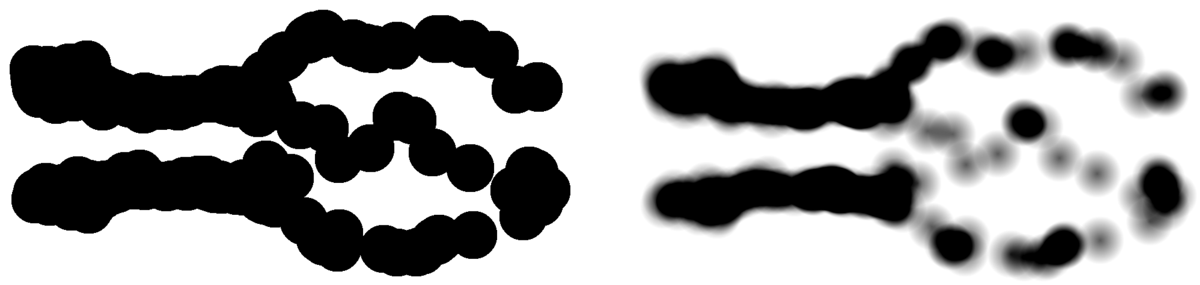

Figure 5: **Modeling smoke with particles:** (Left) smoke particles rendered as solid circles have a harsher appearance than (Right) "blurry" particles with soft edges. Such renderings of smoke can look significantly better when numerous tiny particles are used.

sampling to generate, at each time step, additional new particles with suitably randomized initial positions and times (see Figure 1). You can achieve this effect by exploiting Monte Carlo sampling in three ways:

1. **Spatial Sampling:** Positions should not be chosen as fixed grid locations. Instead, sample initial positions uniformly from the inlet line (assuming horizontal orientation), e.g., $\{(x,y)|x = 2,\ y \in [y_{min}, y_{max}]\}$ for some choice of $y_{min/max}$. Feel free to modify your inlet location to suit your taste.

2. **Temporal Sampling:** To avoid having all particles look like they were released from the inlet line at the exact same time instant (which will lead to aliasing or banding (see Figure 4)), you should randomly sample the time τ they were released on an interval, $\tau \in [t - \Delta t,\ t]$. Particles created at time t fall on the inlet line, whereas particles created slightly earlier (by an amount $\delta t = (t - \tau) \geq 0$) can sample the velocity on the line position $\mathbf{p}(t)$ at time t, let's call it $\mathbf{v}(t)$, but then use the back-tracked position
$$\mathbf{p}(\tau) \approx \mathbf{p}(t) - \delta t\ \mathbf{v}(t), \tag{8}$$
so that they will appear slightly before the inlet and then "move on screen."

3. **Particle Creation Rate, r:** You can use a particle creation rate parameter, r (units of particles/time), to determine how many particles should be made during any given time step. The real-valued number of particles to create will be $N_{\Delta t} = r\ \Delta t$, which may be greater or less than one. You can interpret this by creating $\lfloor N_{\Delta t} \rfloor$ particles, then creating one additional particle with probability $(N_{\Delta t} - \lfloor N_{\Delta t} \rfloor) \in [0, 1)$.

STEP #12 Using these three techniques, create a more realistic flow of particles; **to illustrate your approach, submit an image of the most impressive flow you can create.** You may combine this step with the following step ("creative artifact"). You are free to generate and store the trajectories of each of the particles in any way you see fit.

TIP: In many large-scale production applications that use thousands or millions of particles, particles are created/advected one frame at a time, and rendered immediately, to avoid storing all particles' trajectories in memory.

4.5 Putting Things Together: Make a Creative Artifact

STEP #13 Many computer animated movies rely heavily on the creative use of numerical computing. In this step, you will use the visualization methods you have explored to make a creative artifact. *Submit a high-resolution image (or short video) that illustrates the most realistic or otherwise interesting smoke, fire, or other effect you can model with the provided flow dataset.* See Figures 1 and 6 for inspiration.

Figure 6: **A better rendering of fire and smoke (from [RNGF03])** created by advecting many "fire and smoke particles" in a complex turbulent flow, as well as a sophisticated lighting simulation. For sophisticated 3D renderings it is not uncommon to use several million particles.

5 Matlab Variable Names

Here are some basic comments about variables that may be helpful in understanding the various function arguments of the Matlab code:

1. The function `v_full` is a nonlinear function in terms of its variables, mapping from $\mathbb{R}^3$ to $\mathbb{R}^2$. This can be viewed as two functions mapping from $\mathbb{R}^3$ to $\mathbb{R}$. The x-coordinate is formed by interpolating on a $50 \times 100 \times 101$ array called `x_array`, while the y-coordinate is formed by interpolating another array of the same size called `y_array`.
2. Whenever we need to plot a movie, we might need to pause so that the frames appear properly. The variable `delay`, often set to `0.001`, is the time in seconds to pause at each frame.
3. The bounds of plotting the diagrams are sometimes represented by the variable `bounds`.

6 What to Submit?

You should submit the following in a `zip` file via CMS:

1. **Checklist STEPs** for this assignment were denoted clearly by a sequence of "STEP #" tasks. You should provide code and an explanatory text with supporting images/plots (e.g., this document was created with pdflatex). These can be included in your ...
2. **A brief report** describing your findings for the checklist STEP items, including any images/plots needed to support your findings. PDF documents are preferred, but can be made using any editor/program you choose, e.g., LaTeX/pdflatex.
3. **Your Matlab implementations, complete with directory structure** necessary to run your programs that implement the checklist steps. If you need to add any additional functions, you may do so in additional files (please give them meaningful names and comments).
4. **Code Documentation:** As in the first assignment, your program should be documented *thoroughly*. Numerical code without good documentation can be very hard to read, so it is important that you explain to us what you are doing via documentation in your program. If there is any point in your solution at which is is not completely clear *what* your are doing or *why* you are doing it, explain it using comments. If your approach has limitations or potential shortcomings, explain it using comments. If you do something clever in your program to speed it up, or make it more accurate, or make it more robust, explain it using comments. Part of your mark is based on how easy your code is to understand.

5. **Do not submit huge files:** You can submit images and short videos as supporting evidence, however upload sizes are limited. Please try to keep your overall submission below 50MB.

Acknowledgement: A special thanks to Theodore Kim for generating the fluid flow data used in this assignment.

References

[KTJG08] Theodore Kim, Nils Thürey, Doug L. James, and Markus Gross. Wavelet Turbulence for Fluid Simulation. *ACM Transactions on Graphics*, August 2008. (to appear).

[RNGF03] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald P. Fedkiw. Smoke Simulation for Large-Scale Phenomena. *ACM Transactions on Graphics*, 22(3):703–707, July 2003.