

Dan Morris's notes on:

“Fluid Control Using the Adjoint Method”
A McNamara, A Treuille, Z Popovic, J Stam
SIGGRAPH 2004

Note that this summary skips the hard math, but explains the overall setting and formulation of the relevant equations...

- **Motivation**

- Lots of work has been done in simulating fluids (smoke, fire, water, etc.)
- Almost every simulator described in the literature takes an initial state and simulates what happens to infinity. This makes it difficult for an artist to use, since artists typically have a very specific vision in mind, and a “real” simulation might not look good, and it’s usually impossible to find the right set of initial conditions to make the water look “just right”.
- So this paper approaches the problem of *constrained* simulation, where an artist might say that the 227th frame should look more or less like *this*, and the simulation provides the “most realistic” path from the initial state to that frame, which probably isn’t exactly the “real” path that the water would “want” to take.
- This is much more computationally demanding than the unconstrained problem. Others have approached this problem, but this paper presents an optimization that makes constrained simulation feasible for larger meshes and for water (previously constrained simulation was applied only to smoke).

So the goal is to take an initial state describing some water, and a set of frames where the water should have a particular configuration, and find the most realistic behavior to get from A to B.

- **Imaginary tools I can get from the web**

- The first thing I do is go on the web and download a water simulator. In other words, this paper is not about how to simulate water. (Also, water is more interesting than smoke, so this sentence is the last time I’ll mention smoke.)
 - I assume this simulator uses level sets defined over a regular, constant grid (although that’s only one specific application of their method). In other words (level sets in one-half of one sentence), at each frame, each grid point stores a velocity vector and a single value that represents its distance from the water surface.
 - When I’m using this simulator, I can modify the state of the water at every timestep, after the “real” simulation has occurred. I can modify the velocity of water at each grid point, and I can modify the position of the water surface. The more modifications I make to the simulation’s results, the less “real” the final product will be.
 - They take all the “changes” they’re going to make for a whole simulation (lots of timesteps) and call them **u**. They don’t specify exactly how they encode **u**,

but a naïve assumption is that if I have t timesteps and a simulation grid that's $a \times b \times c$, they fill \mathbf{u} up with all the velocity and level set modifications for each grid point at each timestep, so \mathbf{u} would be a vector of length $(a*b*c*t)(3+1)$, (assuming that a velocity modification requires three elements and a level set modification requires one element). This is probably absurd, but it conceptually illustrates what \mathbf{u} contains.

- I also go on the web and download a nonlinear solver. A NLS's job is to find the value of each of a set of variables that minimizes a function.
 - So I tell him how many variables I have, and I give him a function that he can call with a vector full of candidate values for each variable. I probably also give him a reasonable guess about starting values for each variable.
 - Whenever he calls my function, he gives me a value for each of my variables, and I give him the value of the function that he's trying to minimize. Critically, I also give him the *gradient* of the function for the values he gave me (which is a matrix, since the input to the function is a vector of values). Which means I have to know *how* to compute the gradient.

- **The objective function**

- So the next big question is... what function are we trying to minimize? I want a function that defines how “good” my results are, for a given set of manipulations that might apply to the underlying simulation.
- What makes my final video “good”?
 - Well, matching closely to the specified keyframes is important.
 - Also, not sacrificing physically plausible behavior to get to the keyframes is important.
- They wrap these two things up in eq (2):

$$\varphi(\mathbf{u}) = \frac{1}{2} \sum_{t=0}^n \left(\|W_t(\gamma(\mathbf{q}_t) - \gamma(\mathbf{q}_t^*))\|^2 + \alpha \|M_t \mathbf{u}\|^2 \right). \quad (2)$$

- Here \mathbf{u} is our familiar control vector, \mathbf{q}_t is the state of the system (positions, velocities) at time t , \mathbf{q}_t^* is the state described by each keyframe. But of course there aren't keyframes for every timestep (that would be brute-force manual animation), so we introduce \mathbf{W}_t , a weight matrix that specifies where constraints actually exist in a given timestep. For a timestep with no constraints at all (probably most timesteps), \mathbf{W}_t is all zeros. α is the parameter that says “how much do I care about realistic behavior vs. matching my keyframes”. Note that alpha scales the “size” of \mathbf{u} . In other words, the “bigger” \mathbf{u} is (the more I have to mess with the physical simulation), the worse my result is. A small α says “I don't care much about messing with the physical simulation, just hit my keyframes”.

It's very straightforward to compute $\phi(\mathbf{u})$ given the simulator I downloaded from the web. I can run the simulator for each frame, and make the (straightforward) modifications specified in \mathbf{u} , then plug my final results into eq. (2). But the solver also wants the *gradient* of $\phi(\mathbf{u})$, aka $d\phi/d\mathbf{u}$. This is the computation that makes constrained simulation really hard, and this is what this paper is really about.

- **The hard math**

- Really I'm going to skip the hard math, because we reviewed this paper two weeks or so ago and I'm behind on my summaries. But the heart of the math is the Adjoint Method, which is actually a very clever way of reformulating a certain class of linear equations to require much less computation than the obvious approach would have. Sort of like the way you learn in your numerical analysis class that even though solving $Ax=b$ by inverting A is intuitive, you don't really want to invert A .

It turns out that the derivative being computed – $d\phi/d\mathbf{u}$ – can be formulated as exactly the type of equation that the adjoint shortcut optimizes. In fact it reduces the computation of the overall $d\phi/d\mathbf{u}$ to the computation of the *partial* derivatives at each timestep (much less computationally-intensive).

Therefore the last few pages of the paper describe specifically how to compute the relevant *partial* derivatives for each of the operations involved in fluid simulation (advection, diffusion, etc). The novel contribution here comes when we get up to the level-set redistancing operation, not something that has an obvious derivative. They make the argument that despite its being a highly nonlinear and unusual operation, it is locally continuous, and can be differentiated... again, I skip the details.

The point is they find out how to compute relevant partials for each operation in their simulation, which lets them compute $d\phi/d\mathbf{u}$ via the adjoint method, which lets them happily run their solver on their simulator, which lets them compute the correct (optimal) constraints \mathbf{u} to apply to the simulation to meet the keyframes according to the specified tolerance constants. Hooray!

- **Ideas for improvements or related projects**

- Higher-level, non-image-space constraints (e.g. make x follow some trajectory, or keep the water below n turbulence, etc.)