Notes on Adjoint Control of Graphical Simulations

Jos Stam
Alias Systems

1 Introduction

These notes are a compilation of several documents that I wrote up while I was trying to understand the adjoint method and its implementation. This work is a collaboration with Antoine McNamara, Arien Treuille and Zoran Popovic which resulted in two SIGGRAPH publications on the control of fluids such as smoke and water [5, 3]. I hope that these notes complement those publications and that they might be useful to some.

Traditionally, computer graphics simulations are controlled "by hand" by tweaking the parameters of the model. In a physics-based animation this typically involves specifying initial conditions and external forces that drive the animation towards a specified goal. The basic idea explained in these notes is to let the computer do the tweaking for us. The problem we are trying to solve has three ingredients: (1) a simulator, (2) a set of control parameters and (3) a target behavior. In computer graphics we can think of many different applications that fit into this framework. As we will see below the simulator really can be any piece of code. So examples of (1) include a renderer, a physics-based simulator, a surface modeler, etc. The set of parameters can be considered as input variables to the simulator. For complicated simulations the effect of the parameters on the results can be unpredictable. In this case the automatic control methods are most useful. Examples of a target behaviors are: matching key-frames, minimizing the running time, maximizing smoothness, ... Any goal that can be put into a cost function really.

In these notes I will first present some theory and then try to provide enough information so that the reader can implement their own version of a controller for their application. The theory comes mostly from [2] and the implementation recipes come from [1].

2 Some Theory

The reader who hates math might as well skip this section and go straight to Section 3 which is more "hands on" and geared towards a concrete implementation.

2.1 Control Problem

Control theory is often first presented in a continuous setting and then discretized to yield implementations. We do not take this approach in these notes. Instead we start from a general discrete setting and then show that the adjoint method really is a general almost trivial result in linear algebra. This is an example of how generalization can in fact simplify a problem's solution, at least conceptually. This happens quite often in math: simplification by generalization.

In general, a simulation can be viewed as a sequence of states $q_1, q_2, \cdots, q_N$, where each state is a finite dimensional vector say of dimension n: $q_i \in \mathbf{R}^n$. In physical simulations each state is completely determined by the one preceding it. However, we will not restrict the theory to this case, but rather assume that the evolution of the states is governed by the following general equation:

$$\mathbf{E}(\mathbf{q}, u) = \mathbf{0}. \tag{1}$$

The vector $\mathbf{q} = (q_1, \cdots, q_N)$ is simply a concatenation of all the states. The variable u contains the control parameters and we assume it has size p, so that $u \in \mathbf{R}^p$. The function $\mathbf{E} : \mathbf{R}^n \times \mathbf{R}^p \to \mathbf{R}^n$ defines a relationship between the states and the control vector. The only restriction we set on this function is that it should be differentiable in its arguments. The goal in optimization is to find a set of controls which minimize a given cost function $J(\mathbf{q}, u)$. This cost function can be quite general and may depend on both the states and the controls. Again all we ask for is that it is at least once differentiable. We have now formalized all three ingredients of our control problem: (1) The simulator (Equation 1), (2) the controls u and (3) the target behavior given by the cost function $J(\mathbf{q}, u)$.

There are many techniques to solve these kind of problems. When the evolution of the states is smooth with respect to the controls it makes most sense to use an optimizer which takes into account derivatives. In fact we only need the first derivative as there are many good optimizers for which this information is sufficient to find a solution. The optimizer in this paper can be treated as a black box. It's input consists of the current control u, the cost function J and the derivative of the cost function with respect to the controls: $\frac{dJ}{du}$. The optimizer then outputs a new control value. By iterating this process hopefully we arrive at a minimum of the cost function. Treating the optimizer as a black box seems like cheating since all the hard work is done there. However, we still have to provide it with the derivative of the cost function. It is not all that obvious how to compute this derivative at first. The goal of these notes is to show exactly how this can be done in practice.

Formally we can compute the derivative of the cost function as follows:

$$\frac{dJ}{du} = \frac{\partial J}{\partial \mathbf{q}} \frac{d\mathbf{q}}{du} + \frac{\partial J}{\partial u} \in \mathbf{R}^p. \tag{2}$$

This expression involves the derivative of the state with respect to the control. An equation for it can be derived by taking the derivative of Equation 1 with respect to the control:

$$\frac{\partial \mathbf{E}}{\partial \mathbf{q}} \frac{d\mathbf{q}}{du} + \frac{\partial \mathbf{E}}{\partial u} = \mathbf{0}. \tag{3}$$

This equation is actually a set of p systems of linear equations involving the same linear operator. This becomes clearer if we introduce some more standard algebraic notation:

$$\mathbf{A} = \frac{\partial \mathbf{E}}{\partial \mathbf{q}},\ \mathbf{x} = (\mathbf{x}_1, \cdots, \mathbf{x}_p) = \frac{\partial \mathbf{q}}{\partial u} \quad \text{and} \quad \mathbf{f} = (\mathbf{f}_1, \cdots, \mathbf{f}_p) = -\frac{\partial \mathbf{E}}{\partial u}.$$

Then Equation 3 actually stands for the following linear equations:

$$\mathbf{A}\mathbf{x}_k = \mathbf{f}_k, \qquad k = 1, \cdots, p. \tag{4}$$

To compute the gradient of the cost function we need to compute $\mathbf{g}^T\mathbf{x}$, where

$$\mathbf{g}^T = \frac{\partial J}{\partial \mathbf{q}} \quad \in \mathbf{R}^n.$$

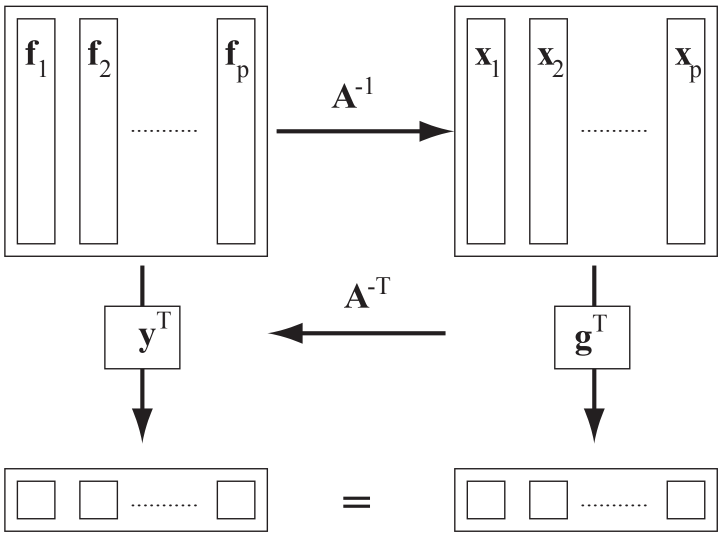

Figure 1: Illustration of the adjoint method.

At this point we have completely cast the control problem into an algebraic setting. Solving for the $\mathbf{x}$'s is known as the method of propagating derivatives forward. This operation involves solving p linear systems and is costly for many controls. This is the reason why in practice the adjoint method is used to which we turn next.

2.2 Adjoint Method

In the adjoint method, instead of solving Equation 4 we solve the following equation for the vector $\mathbf{y}$:

$$\mathbf{A}^T\mathbf{y} = \mathbf{g}. \tag{5}$$

The solution of this equation allows us to directly compute the gradient of the cost function. Indeed we have that

$$\mathbf{g}^T\mathbf{x} = (\mathbf{A}^T\mathbf{y})^T\mathbf{x} = \mathbf{y}^T\mathbf{A}\mathbf{x} = \mathbf{y}^T\mathbf{f} = (\mathbf{y}^T\mathbf{f}_1, \cdots, \mathbf{y}^T\mathbf{f}_p).$$

This simple result is really all there is to the adjoint method. The gain is immediately clear. Instead of having to solve p linear systems, we only solve one linear system for $\mathbf{y}$ and then take p dot products $\mathbf{y}^T\mathbf{f}_k$. In practice the dot product is much cheaper than the linear solve and the computational gain is enormous. In fact the cost of the method is virtually independent of the number of controls.

The reason that the adjoint method works so well for our control problem is that we do not need the entire state $\mathbf{x}$ of derivatives only its reduction by the vector $\mathbf{g}$. The adjoint method propagates this reduction vector back so that it can be applied directly to the right hand side $\mathbf{f}$. Figure 1 illustrates this situation.

2.3 Lagrange Multipliers

Another way to arrive at the adjoint equation is to view the control problem can as a constrained optimization procedure. We want to minimize the cost function under the constraint given by

Equation 1. This directly suggests including the constraint into the cost function via Lagrange Multipliers. Indeed we can define a new cost function:

$$L(\mathbf{q}, u) = J(\mathbf{q}, u) - \mathbf{p}^T \mathbf{E}(\mathbf{q}, u). \tag{6}$$

The minima of this cost function are given by

$$\frac{\partial L}{\partial \mathbf{q}} = \mathbf{0} \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{p}} = \mathbf{0}.$$

The second equation is simply our constraint, that is why we introduced the Lagrange multiplier $\mathbf{p}$ in the first place. The first equation is more interesting and provides an equation for the multipliers:

$$\left(\frac{\partial \mathbf{E}}{\partial \mathbf{q}}\right)^T \mathbf{p} = \frac{\partial J}{\partial \mathbf{q}}. \tag{7}$$

This equation is the same one as Equation 5.

3 Implementation: Adjoint Code Methods

There is a huge gap between the theory presented in the preceding section and an actual implementation. This section will outline an almost automatic way to translate a simulator into it's adjoint version. I gleaned these results from the web by googling on the phrase "adjoint code." In fact there are software packages out there that will translate code into its associated adjoint. Unfortunately, these packages seem to be restricted to FORTRAN code only. I prefer to code in C. Of course I could translate all my code to FORTRAN use the tools and then convert it all back to C using NETLIB's "f2c." This however would result in rather ugly code. Alternatively, I will describe some recipes here that will help in translating C code to its adjoint counterpart. To demonstrate the usefulness of these recipes I will apply them to many algorithms found in physical simulations. The ultimate goal is to program an automatic compiler which turns any C code into it's adjoint code. Hopefully these notes will lead to such an implementation one day.

3.1 Adjoint translation

A program is composed of a set of instructions. In this note we will try to describe how to translate the instructions into their adjoint counterparts. We will denote the adjoint to an instruction I by $A(I)$. So if a piece of code is made up of the following instructions:

$$I_1, I_2, \cdots, I_n.$$

Then the associated adjoint program will be the following sequence of instructions:

$$A(I_n), A(I_{n-1}), \cdots, A(I_1).$$

The adjoint program reverses the order of the instructions. To each active variable x of the program (one that depends on one of the controls appearing in the optimization problem) we associate an adjoint variable x'. The adjoint of a single instruction is obtained by computing the differential of the instruction with respect to the variables. The equivalent adjoint instruction is then obtained by transposing the differentials. For example the instruction $I = x \leftarrow y^2$ has the following differential:

$$\begin{pmatrix} dy \\ dx \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 \\ 2y & 0 \end{pmatrix} \begin{pmatrix} dy \\ dx \end{pmatrix}.$$

So that the adjoint version of this expression becomes:

$$\begin{pmatrix} y' \\ x' \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 2y \\ 0 & 0 \end{pmatrix} \begin{pmatrix} y' \\ x' \end{pmatrix}.$$

So the corresponding code is

$$\begin{array}{rcl} y' & \leftarrow & y' + 2y\, x' \\ x' & \leftarrow & 0. \end{array}$$

The adjoint of the assigned variable is set to zero because it's previous value is lost and therefore has no influence on the cost function. Let's consider a more involved instruction like $I = x \leftarrow x^3 + \sin(y^2)z$. In this case the diferential is

$$\begin{pmatrix} dz \\ dy \\ dx \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \sin(y^2) & 2y\cos(y^2)z & 3x^2 \end{pmatrix} \begin{pmatrix} dz \\ dy \\ dx \end{pmatrix}.$$

So that the equivalent adjoint code is

$$\begin{array}{rcl} z' & \leftarrow & z' + \sin(y^2)\, x' \\ y' & \leftarrow & y' + 2y\cos(y^2)z\, x' \\ x' & \leftarrow & 3x^2\, x' \end{array}$$

Notice that we need to keep the non-adjoint values of the variables around when updating adjoint variables. We can generalize these examples for the following instruction

$$I = x \leftarrow f(x, y, \cdots, z).$$

In this case the differential is

$$\begin{pmatrix} dz \\ \vdots \\ dy \\ dx \end{pmatrix} \leftarrow \begin{pmatrix} 1 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 \\ f_z(x,y,\cdots,z) & \cdots & f_y(x,y,\cdots,z) & f_x(x,y,\cdots,z) \end{pmatrix} \begin{pmatrix} dz \\ \vdots \\ dy \\ dx \end{pmatrix}.$$

The corresponding adjoint code is

$$\begin{array}{rcl} z' & \leftarrow & z' + f_z(x,y,\cdots,z)\, x' \\ & \vdots & \\ y' & \leftarrow & y' + f_y(x,y,\cdots,z)\, x' \\ x' & \leftarrow & f_x(x,y,\cdots,z)\, x' \end{array}$$

This takes care of most assignment-like instructions occuring in a simulation. Let's now look at how to translate higher level constructs like for-loops and if-statements. The for loop is easy to translate. All we have to do is reverse the order of the loop and replace the instructions with their adjoints:

$$A\left(\begin{array}{l}\text{for } \ i=0,\cdots,n \ \text{ do}\\ \quad\quad I_i\\ \text{end for}\end{array}\right) = \begin{array}{l}\text{for } \ i=n,\cdots,0 \ \text{ do}\\ \quad\quad A(I_i)\\ \text{end for}\end{array} \quad\quad\quad .$$

If-statements are also easily handled:

$$A\left(\begin{array}{l}\text{if } \ B \ \text{ then}\\ \quad I_1\\ \text{else}\\ \quad I_2\\ \text{endif}\end{array}\right) = \begin{array}{l}\text{if } \ B \ \text{ then}\\ \quad A(I_1)\\ \text{else}\\ \quad A(I_2)\\ \text{endif}\end{array}$$

where B is some conditionial statement. If this statement depends on some of the variables then these have to be computed (or stored) before evaluating the conditional statement.

3.2 Applications

We now apply these rules to compute the adjoint versions of some well known algorithms. Let's start with the Gauss-Seidel solver with a fixed number of iterations. In one dimension the algorithm is (assuming periodic boundary conditions):

```
GaussSeidel($x, f$)
  for $k = 0, \cdots, K$ do
    for $i = 0, \cdots, n-1$ do
      $x_i \leftarrow (f_i + x_{i-1} + x_{i+1})/2$
    end for
  end for
```

Then using the recipes from the previous section we can derive the adjoint:

```
$A$(GaussSeidel($x', f'$))
  for $k = K, \cdots, 0$ do
    for $i = n-1, \cdots, 0$ do
      $x'_{i+1} \leftarrow x'_{i+1} + x'_i/2$
      $x'_{i-1} \leftarrow x'_{i-1} + x'_i/2$
      $f'_i \leftarrow f'_i + x'_i/2$
      $x'_i \leftarrow 0$
    end for
  end for
```

Now consider the Conjugate Gradient algorithm:

```
CG($x, b$)
  $r \leftarrow b - M(x)$
  for $k = 0, \cdots, K$ do
    $z \leftarrow P(r)$
    $\rho_0 \leftarrow \langle r, z \rangle$
```

```
        if $k = 0$ then
            $p \leftarrow z$
        else
            $\beta \leftarrow \rho_0/\rho_1$
            $p \leftarrow z + \beta p$
        end if
        $q \leftarrow M(p)$
        $\beta \leftarrow \langle p, q \rangle$
        $\alpha \leftarrow \rho_0/\beta$
        $x \leftarrow x + \alpha p$
        $r \leftarrow r - \alpha q$
        $\rho_1 \leftarrow \rho_0$
    end for
```

In the algorithm M is the matrix we try to invert and P is a preconditioner. Let's assume we are trying to solve Poisson's equation in two-dimensions with periodic boundary conditions. In this case:

```
$M(x)$
    for $i = 0, \cdots, n-1$ do
        for $j = 0, \cdots, n-1$ do
            $y_{i,j} \leftarrow 4x_{i,j} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$
        end for
    end for
    return $y$
```

and

```
$P(x)$
    for $i = 0, \cdots, n-1$ do
        for $j = 0, \cdots, n-1$ do
            $y_{i,j} \leftarrow x_{i,j}/4$
        end for
    end for
    return $y$
```

The adjoints of these two functions are equal to themselves because of the symmetry of the operators involved:

$$A(M) = M \quad \text{and} \quad A(P) = P.$$

The adjoint of the dot product $\langle p, q \rangle$ is

```
$AD(n', p', q')$
    for $j = n-1, \cdots, 0$ do
        for $i = n-1, \cdots, 0$ do
            $p'_{i,j} \leftarrow p'_{i,j} + q_{i,j}\ n'$
            $q'_{i,j} \leftarrow q'_{i,j} + p_{i,j}\ n'$
        end for
    end for
```

Finally here is the adjoint of the conjugate gradient algorithm.

```
$A($CG$(x', b'))$
    $\rho_0', \rho_1', \alpha, \beta \leftarrow 0$
    $p', z', q', r' \leftarrow 0$
    for $k = K, \cdots, 0$ do
        $\rho_0' \leftarrow \rho_0' + \rho_1'$
        $\rho_1' \leftarrow 0$
        $\alpha' \leftarrow \alpha' - \langle q, r' \rangle$
        $q' \leftarrow q' - \alpha\ r'$
        $\alpha' \leftarrow \alpha' + \langle p, x' \rangle$
        $p' \leftarrow p' + \alpha\ x'$
        $\beta' \leftarrow \beta' - \rho_0/\beta^2\ \alpha'$
        $\rho_0' \leftarrow \rho_0' + \alpha'\ /\beta$
        $\alpha' \leftarrow 0$
        $AD(\beta', p', q')$
        $\beta' \leftarrow 0$
        $p' \leftarrow A(M(q'))$
        $q' \leftarrow 0$
        if $k = 0$ then
            $z' \leftarrow z' + p'$
            $p' \leftarrow 0$
        else
            $\beta' \leftarrow \beta' + \langle p, p' \rangle$
            $z' \leftarrow z' + p'$
            $p' \leftarrow \beta\ p'$
            $\rho_1' \leftarrow \rho_1' - \rho_0/\rho_1^2\ \beta'$
            $\rho_0' \leftarrow \rho_0' + \beta'\ /\rho_1$
            $\beta' \leftarrow 0$
        end if
        $AD(\rho_0', r', z')$
        $\rho_0' \leftarrow 0$
        $r' \leftarrow A(P(z'))$
        $z' \leftarrow 0$
    end for
    $b' \leftarrow b' + r'$
    $x' \leftarrow x' - A(M(r'))$
    $r' \leftarrow 0$
```

We now compute the adjoint of the semi-Lagrangian advection step. In one dimension the algorithm is

```
SL$(a, b, u)$
    for $i = 0, \cdots, n-1$ do
        $x \leftarrow i - u_i$
        $k \leftarrow index(x)$
        $t \leftarrow x - k$
        $a_i \leftarrow (1-t)b_k + tb_{k+1}$
    end for
```

These steps are easily adjointed as follows:

```
$A($SL1$(a', b', u'))$
    for $i = n-1, \cdots, 0$ do
        $x', t' \leftarrow 0$
        $b'_{k+1} \leftarrow b'_{k+1} + ta'_i$
        $b'_k \leftarrow b'_k + (1-t)a'_i$
        $a'_i \leftarrow 0$
        $t' \leftarrow t' + b'_{k+1} - b'_k$
        $x' \leftarrow x' + t'$
        $t' \leftarrow 0$
        $u'_i \leftarrow u'_i - x'$
    end for
```

Or with the optimizing flag turned on we get:

```
$A($SL1$(a', b', u'))$
    for $i = n-1, \cdots, 0$ do
        $b'_{k+1} \leftarrow b'_{k+1} + ta'_i$
        $b'_k \leftarrow b'_k + (1-t)a'_i$
        $a'_i \leftarrow 0$
        $u'_i \leftarrow u'_i + b'_k - b'_{k+1}$
    end for
```

We now give the algorithm in two-dimensions:

```
SL2$(a, b, u, v)$
    for $i = 0, \cdots, n-1$ do
        for $j = 0, \cdots, n-1$ do
            $x \leftarrow i - u_{i,j}$
            $y \leftarrow j - v_{i,j}$
            $k \leftarrow index(x)$
            $l \leftarrow index(y)$
            $s \leftarrow x - k$
            $t \leftarrow y - l$
            $a_{i,j} \leftarrow (1-s)(1-t)\ b_{k,l} + (1-s)t\ b_{k,l+1} + s(1-t)\ b_{k+1,l} + st\ b_{k+1,l+1}$
        end for
    end for
```

```
$A($SL2$(a', b', u', v'))$
    for $j = n-1, \cdots, 0$ do
        for $i = n-1, \cdots, 0$ do
            $b'_{k,l} \leftarrow b'_{k,l} + (1-s)(1-t)\ a'_{i,j}$
            $b'_{k,l+1} \leftarrow b'_{k,l+1} + (1-s)t\ a'_{i,j}$
            $b'_{k+1,l} \leftarrow b'_{k+1,l} + s(1-t)\ a'_{i,j}$
        $b'_{k+1,l+1} \leftarrow b'_{k+1,l+1} + st\ a'_{i,j}$
        $a'_{i,j} \leftarrow 0$
        $u'_{i,j} \leftarrow u'_{i,j} + (1-t)(b_{k,l} - b_{k+1,l}) + t(b_{k,l+1} - b_{k+1,l+1})$
        $v'_{i,j} \leftarrow v'_{i,j} + (1-s)(b_{k,l} - b_{k,l+1}) + s(b_{k+1,l} - b_{k+1,l+1})$
    end for
end for
```

4 Adjoint Stable Fluids

It is now time to show some actual code. I have chosen to adjoint my simple implementation of the Stable Fluids Algorithm given in [4]. The code is also available on the course notes CDROM. All the hard work was done in the preceding section where we derived the adjoint versions of a simple linear solver and of the semi-Lagrangian solver. Here is the code of the solver:

```
#define IX(i,j) ((i)+(N+2)*(j))
#define SWAP(x0,x) {float * tmp=x0;x0=x;x=tmp;}
#define FOR_EACH_CELL for ( i=1 ; i<=N ; i++ ) { for ( j=1 ; j<=N ; j++ ) {
#define END_FOR }}

void add_source ( int N, float * x, float * s, float dt )
{
   int i, size=(N+2)*(N+2);
   for ( i=0 ; i<size ; i++ ) x[i] += dt*s[i];
}

void set_bnd ( int N, int b, float * x )
{
   int i;

   for ( i=1 ; i<=N ; i++ ) {
      x[IX(0 ,i)] = b==1 ?  -x[IX(1,i)] :  x[IX(1,i)];
      x[IX(N+1,i)] = b==1 ?  -x[IX(N,i)] :  x[IX(N,i)];
      x[IX(i,0 )] = b==2 ?  -x[IX(i,1)] :  x[IX(i,1)];
      x[IX(i,N+1)] = b==2 ?  -x[IX(i,N)] : x[IX(i,N)];
   }
   x[IX(0 ,0 )] = 0.5f*(x[IX(1,0 )]+x[IX(0 ,1)]);
   x[IX(0 ,N+1)] = 0.5f*(x[IX(1,N+1)]+x[IX(0 ,N)]);
   x[IX(N+1,0 )] = 0.5f*(x[IX(N,0 )]+x[IX(N+1,1)]);
   x[IX(N+1,N+1)] = 0.5f*(x[IX(N,N+1)]+x[IX(N+1,N)]);
}

void lin_solve ( int N, int b, float * x, float * x0, float a, float c )
{
   int i, j, k;
```

```
    for ( k=0 ; k<20 ; k++ ) {
        FOR_EACH_CELL
            x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
        END_FOR
        set_bnd ( N, b, x );
    }
}

void diffuse ( int N, int b, float * x, float * x0, float diff, float dt )
{
    float a=dt*diff*N*N;
    lin_solve ( N, b, x, x0, a, 1+4*a );
}

void advect ( int N, int b, float * d, float * d0, float * u, float * v, float dt )
{
    int i, j, i0, j0, i1, j1;
    float x, y, s0, t0, s1, t1, dt0;

    PUSH(d0); PUSH(u); PUSH(v);
    dt0 = dt*N;
    FOR_EACH_CELL
        x = i-dt0*u[IX(i,j)]; y = j-dt0*v[IX(i,j)];
        if (x<0.5f) x=0.5f; if (x>N+0.5f) x=N+0.5f; i0=(int)x; i1=i0+1;
        if (y<0.5f) y=0.5f; if (y>N+0.5f) y=N+0.5f; j0=(int)y; j1=j0+1;
        s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;
        d[IX(i,j)] = s0*(t0*d0[IX(i0,j0)]+t1*d0[IX(i0,j1)])+
            s1*(t0*d0[IX(i1,j0)]+t1*d0[IX(i1,j1)]);
    END_FOR
    set_bnd ( N, b, d );
}

void project ( int N, float * u, float * v, float * p, float * div )
{
    int i, j;
    FOR_EACH_CELL
        div[IX(i,j)] = -0.5f*(u[IX(i+1,j)]-u[IX(i-1,j)]+v[IX(i,j+1)]-v[IX(i,j-1)])/N;
        p[IX(i,j)] = 0;
    END_FOR
    set_bnd ( N, 0, div ); set_bnd ( N, 0, p );

    lin_solve ( N, 0, p, div, 1, 4 );

    FOR_EACH_CELL
        u[IX(i,j)] -= 0.5f*N*(p[IX(i+1,j)]-p[IX(i-1,j)]);
        v[IX(i,j)] -= 0.5f*N*(p[IX(i,j+1)]-p[IX(i,j-1)]);
    END_FOR
    set_bnd ( N, 1, u ); set_bnd ( N, 2, v );
}

void dens_step ( int N, float * x, float * x0, float * u, float * v, float diff, float dt )
{
    add_source ( N, x, x0, dt );
    SWAP ( x0, x ); diffuse ( N, 0, x, x0, diff, dt );
    SWAP ( x0, x ); advect ( N, 0, x, x0, u, v, dt );
}

void vel_step ( int N, float * u, float * v, float * u0, float * v0, float visc, float dt )
{
    add_source ( N, u, u0, dt ); add_source ( N, v, v0, dt );
    SWAP ( u0, u ); diffuse ( N, 1, u, u0, visc, dt );
    SWAP ( v0, v ); diffuse ( N, 2, v, v0, visc, dt );
    project ( N, u, v, u0, v0 );
    SWAP ( u0, u ); SWAP ( v0, v );
    advect ( N, 1, u, u0, u0, v0, dt ); advect ( N, 2, v, v0, u0, v0, dt );
    project ( N, u, v, u0, v0 );
}
```

The code for the adjoint solver is almost as simple. I found this actually a lot of fun to code and did it one line at the time using the recipes. It involves quite some mental gymnastics where you have to revert all the instructions and compute derivatives on the fly. After this while this becomes almost automatic. Of course it would be nicer if the process was just automated. Notice that for the semi-Lagrangian step the values of the forward simulation are required. I used a very simple technique to have these values around when doing the adjoint simulation. When simulating forward I simply "push" the values on an a heap and then "pop" them when needed. This is simply implemented using two macros. First I allocate a heap of size `9*(N+2)*(N+2)*k`, where `N` is the grid size and `k` is the number of time steps taken. Then the two macros are simply:

```
#define PUSH(a) { int l; for ( l=0 ; l<(N+2)*(N+2) ; l++ ) heap[heap_top++] = a[l]; }
#define POP(a) { int l; for ( l=(N+2)*(N+2)-1 ; l>=0 ; l-- ) a[l] = heap[heap_top--]; }
```

This technique of course can become very costly in terms of memory usage for long simulation and or large grids. In fact, there is some memory wasted in the above implementation. Another option is to run the simulation forward uptil the point where we need the values. But that is very costly in terms of CPU time. Ideally a tradeoff between these two extremes is desireable.

```
#define IX(i,j) ((i)+(N+2)*(j))
#define SWAP(x0,x) {float * tmp=x0;x0=x;x=tmp;}
#define AFOR_EACH_CELL for ( j=N ; j>=1 ; j-- ) { for ( i=N ; i>=1 ; i-- ) {
#define END_FOR }}

static void a_add_source ( int N, float * ax, float * as, float dt )
{
    int i;
    for ( i=(N+2)*(N+2)-1 ; i>=0 ; i-- ) as[i] += dt*ax[i];
}

static void a_set_bnd ( int N, int b, float * ax )
{
    int i;

    ax[IX(N+1,N)] += 0.5f*ax[IX(N+1,N+1)];
    ax[IX(N,N+1)] += 0.5f*ax[IX(N+1,N+1)];
    ax[IX(N+1,N+1)] = 0.0f;

    ax[IX(N+1,1)] += 0.5f*ax[IX(N+1,0 )];
    ax[IX(N,0 )] += 0.5f*ax[IX(N+1,0 )];
    ax[IX(N+1,0 )] = 0.0f;

    ax[IX(0 ,N)] += 0.5f*ax[IX(0 ,N+1)];
    ax[IX(1,N+1)] += 0.5f*ax[IX(0 ,N+1)];
    ax[IX(0 ,N+1)] = 0.0f;

    ax[IX(0 ,1)] += 0.5f*ax[IX(0 ,0 )];
    ax[IX(1,0 )] += 0.5f*ax[IX(0 ,0 )];
    ax[IX(0 ,0 )] = 0.0f;

    for ( i=N ; i>=1 ; i-- ) {
        ax[IX(i,N)] += b==2 ?  -ax[IX(i,N+1)] :  ax[IX(i,N+1)]; ax[IX(i,N+1)] = 0.0f;
        ax[IX(i,1)] += b==2 ?  -ax[IX(i,0 )] :  ax[IX(i,0 )]; ax[IX(i,0 )] = 0.0f;
        ax[IX(N,i)] += b==1 ?  -ax[IX(N+1,i)] :  ax[IX(N+1,i)]; ax[IX(N+1,i)] = 0.0f;
        ax[IX(1,i)] += b==1 ?  -ax[IX(0 ,i)] :  ax[IX(0 ,i)]; ax[IX(0 ,i)] = 0.0f;
    }
}

static void a_lin_solve ( int N, int b, float * ax, float * ax0, float a, float c )
{
    int i, j, k;

    for ( k=19 ; k>=0 ; k-- ) {
        a_set_bnd ( N, b, ax );
        AFOR_EACH_CELL
            ax[IX(i,j+1)] += a*ax[IX(i,j)]/c;
            ax[IX(i,j-1)] += a*ax[IX(i,j)]/c;
            ax[IX(i+1,j)] += a*ax[IX(i,j)]/c;
            ax[IX(i-1,j)] += a*ax[IX(i,j)]/c;
            ax0[IX(i,j)] += ax[IX(i,j)]/c;
            ax[IX(i,j)] = 0.0f;
        END_FOR
```

```
  }
}

void a_diffuse ( int N, int b, float * ax, float * ax0, float diff, float dt )
{
  float a=dt*diff*N*N;
  a_lin_solve ( N, b, ax, ax0, a, 1+4*a );
}

static void a_advect ( int N, int b,
  float * d0, float * u, float * v,
  float * ad, float * ad0, float * au, float * av,
  float dt )
{
  int i, j, i0, j0, i1, j1;
  float x, y, s0, t0, s1, t1, dt0;

  POP(v); POP(u); POP(d0);

  dt0 = dt*N;
  a_set_bnd ( N, b, ad );
  AFOR_EACH_CELL
    x = i-dt0*u[IX(i,j)]; y = j-dt0*v[IX(i,j)];
    if (x<0.5f) x=0.5f; if (x>N+0.5f) x=N+0.5f; i0=(int)x; i1=i0+1;
    if (y<0.5f) y=0.5f; if (y>N+0.5f) y=N+0.5f; j0=(int)y; j1=j0+1;
    s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;

    ad0[IX(i1,j1)] += s1*t1*ad[IX(i,j)];
    ad0[IX(i1,j0)] += s1*t0*ad[IX(i,j)];
    ad0[IX(i0,j1)] += s0*t1*ad[IX(i,j)];
    ad0[IX(i0,j0)] += s0*t0*ad[IX(i,j)];

    au[IX(i,j)] += dt0*(t0*(d0[IX(i0,j0)]-d0[IX(i1,j0)]) + t1*(d0[IX(i0,j1)]-d0[IX(i1,j1)]))*ad[IX(i,j)];
    av[IX(i,j)] += dt0*(s0*(d0[IX(i0,j0)]-d0[IX(i0,j1)]) + s1*(d0[IX(i1,j0)]-d0[IX(i1,j1)]))*ad[IX(i,j)];

    ad[IX(i,j)] = 0.0f;
  END_FOR
}

static void a_project ( int N, float * au, float * av, float * ap, float * adiv )
{
  int i, j;

  a_set_bnd ( N, 2, av ); a_set_bnd ( N, 1, au );
  AFOR_EACH_CELL
    ap[IX(i,j-1)] -= -0.5f*N*av[IX(i,j)];
    ap[IX(i,j+1)] -= 0.5f*N*av[IX(i,j)];
    ap[IX(i-1,j)] -= -0.5f*N*au[IX(i,j)];
    ap[IX(i+1,j)] -= 0.5f*N*au[IX(i,j)];
  END_FOR

  a_lin_solve ( N, 0, ap, adiv, 1, 4 );

  a_set_bnd ( N, 0, ap ); a_set_bnd ( N, 0, adiv );
  AFOR_EACH_CELL
    ap[IX(i,j)] = 0.0f;
    av[IX(i,j-1)] += 0.5f*adiv[IX(i,j)]/N;
    av[IX(i,j+1)] += -0.5f*adiv[IX(i,j)]/N;
    au[IX(i-1,j)] += 0.5f*adiv[IX(i,j)]/N;
    au[IX(i+1,j)] += -0.5f*adiv[IX(i,j)]/N;
    adiv[IX(i,j)] = 0.0f;
  END_FOR
}

void a_dens_step ( int N, float * x, float * x0, float * u, float * v,
  float * ax, float * ax0, float * au0, float * av0, float diff, float dt )
{
  a_advect ( N, 0, x0, u, v, ax, ax0, au, av, dt );
  SWAP ( ax0, ax );
  a_diffuse ( N, 0, ax, ax0, diff, dt );
  SWAP ( ax0, ax );
  a_add_source ( N, ax, ax0, dt );
}

void a_vel_step ( int N, float * u, float * v, float * u0, float * v0,
  float * au, float * av, float * au0, float * av0, float visc, float dt )
{
  a_project ( N, au, av, au0, av0 );
  a_advect ( N, 2, v0, u0, v0, av, av0, au0, av0, dt );
  a_advect ( N, 1, u0, u0, v0, au, au0, au0, av0, dt );
  SWAP ( av0, av ); SWAP ( au0, au );
  a_project ( N, au, av, au0, av0 );
  a_diffuse ( N, 2, av, av0, visc, dt ); SWAP ( av0, av );
  a_diffuse ( N, 1, au, au0, visc, dt ); SWAP ( au0, au );
  a_add_source ( N, av, av0, dt ); a_add_source ( N, au, au0, dt );
}
```

References

[1] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.

[2] M. B. Giles and N. A. Pierce. An introduction to the adjoint method to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000.

[3] A. McNamara, A. Treuille, Z. Popovic, and J. Stam. Fluid control using the adjoint method. *ACM Transactions on Graphics. Special issue: Proceedings of ACM SIGGRAPH 2004*, 23(3):(to appear), 2004.

[4] J. Stam. Real-Time Fluid Dynamics for Games. In *Proceedings of the Game Developer Conference*, March 2003.

[5] A. Treuille, A. McNamara, Z. Popovic, and J. Stam. Keyframe control of smoke simulations. *ACM Transactions on Graphics. Special issue: Proceedings of ACM SIGGRAPH 2003*, 22(3):716–723, 2003.