

# Notes on Adjoint Methods for 18.335

Steven G. Johnson

Created Spring 2006, updated January 14, 2021.

## 1 Introduction

Given the solution  $\mathbf{x}$  of a discretized PDE or some other set of  $M$  equations parameterized by  $P$  variables  $\mathbf{p}$  (*design parameters*, a.k.a. *control variables* or *decision parameters*), we often wish to compute some function  $g(\mathbf{x}, \mathbf{p})$  based on the parameters and the solution. For example, if the PDE is a wave equation, we might want to know the scattered power in some direction. Or, for a mechanical simulation, we might want to know the load-bearing capacity of the structure. Or for a fluid, we might wish to know the flow rate somewhere. Often, however, we want to know more than just the *value* of  $g$ —we also want to know its *gradient*  $\frac{dg}{d\mathbf{p}}$ . Adjoint methods give an efficient way to evaluate  $\frac{dg}{d\mathbf{p}}$ , with a cost *independent* of  $P$  and usually comparable to the cost of solving for  $\mathbf{x}$  *once*. (In the context of neural networks, adjoint methods are known as *backpropagation*; whereas in automatic differentiation they are known as *reverse-mode* differentiation.)

The gradient of  $g$  with respect to  $\mathbf{p}$  is extremely useful. It gives a measure of the *sensitivity* of our answer to the parameters  $\mathbf{p}$  (which may, for example, come from some experimental measurements with some associated uncertainties). Or, we may want to perform an *optimization* of  $g$ , picking the  $\mathbf{p}$  that produce some desired result; in this case the gradient indicates a useful search direction (e.g. for non-linear conjugate-gradient optimization). For large-scale optimization problems, the number  $P$  of design parameters can be hundreds, thousands, or more—this is common in *shape* or *topology optimization*, in which  $\mathbf{p}$  controls the placement and shape of arbitrary blobs of different materials constituting a given structure/design. Sometimes, this process is called *inverse design*: finding the problem that yields a given solution instead of the other way around. When  $P \gg 1$ , the amazing efficiency of adjoint methods makes inverse design possible.

I hadn't found any textbook description of adjoint methods that I particularly like, which is part of my

motivation for writing up these notes. One introduction can be found in [1], and a more general treatment can be found in [2]. Subsequently, Gil Strang wrote a nice introduction to adjoint methods in his book [3], including a discussion of the important topic of automatic differentiation (for which adjoint or “reverse” differentiation is a key idea).

## 2 Linear equations

Suppose that the column-vector  $\mathbf{x}$  solves the  $M \times M$  linear equation  $A\mathbf{x} = \mathbf{b}$  where we take  $\mathbf{b}$  and  $A$  to be real<sup>1</sup> and to depend in some way on  $\mathbf{p}$ . To evaluate the gradient directly, we would do

$$\frac{dg}{d\mathbf{p}} = g_{\mathbf{p}} + g_{\mathbf{x}}\mathbf{x}_{\mathbf{p}}$$

where the subscripts indicate partial derivatives ( $g_{\mathbf{x}}$  is a row vector,  $\mathbf{x}_{\mathbf{p}}$  is an  $M \times P$  matrix, etc.). Since  $g$  is a given function,  $g_{\mathbf{p}}$  and  $g_{\mathbf{x}}$  are presumably easy to compute. On the other hand, computing  $\mathbf{x}_{\mathbf{p}}$  is hard: evaluating it directly by differentiating  $A\mathbf{x} = \mathbf{b}$  by a parameter  $p_i$  gives  $\mathbf{x}_{p_i} = A^{-1}(\mathbf{b}_{p_i} - A_{p_i}\mathbf{x})$ . That is, we would have to solve an  $M \times M$  linear equation for  $P$  right-hand sides, once for every component of  $\mathbf{p}$ ; this is impractical if  $P$  and  $M$  are large.

More explicitly, the problematic term is:

$$g_{\mathbf{x}}\mathbf{x}_{\mathbf{p}} = \underbrace{g_{\mathbf{x}}}_{1 \times M} \underbrace{[A^{-1}(\mathbf{b}_{\mathbf{p}} - A_{\mathbf{p}}\mathbf{x})]}_{M \times M} = \underbrace{[g_{\mathbf{x}}A^{-1}]}_{1 \times M} \underbrace{(\mathbf{b}_{\mathbf{p}} - A_{\mathbf{p}}\mathbf{x})}_{M \times P},$$

where  $A_{\mathbf{p}}\mathbf{x}$  denotes the  $M \times P$  matrix with columns  $A_{p_i}\mathbf{x}$  for  $i = 1, \dots, P$ .<sup>2</sup> One way of looking at the difficulty is that in the first equation we multiply a  $M \times M$  matrix by a  $M \times P$  matrix, which costs  $O(M^2P)$

<sup>1</sup>This involves no loss of generality, since complex linear equations can always be written as real linear equations of twice the size by taking the real and imaginary parts as separate variables.

<sup>2</sup>Technically,  $A_{\mathbf{p}}$  is a rank-3 tensor or “three-dimensional matrix,” although it almost certainly isn't stored this way. For example,  $A_{p_i}\mathbf{x}$  could be computed for each  $i$  separately without saving  $A_{p_i}$ . Often,  $A_{p_i}$  will be very sparse.

work, or equivalently we have multiplications of  $A^{-1}$  by  $P$  vectors (i.e., solves of  $P$  right-hand sides, which in practice would likely use a factorization of  $A$  or an iterative solver rather than explicitly computing  $A^{-1}$ ).<sup>3</sup> However, this can be ameliorated simply by *parenthesizing in a different way* [3],<sup>4</sup> as shown in the last expression. If we multiply  $\boldsymbol{\lambda}^T = \mathbf{g}_x A^{-1}$  first, that corresponds to only a *single* solution of an *adjoint equation*<sup>5</sup>

$$A^T \boldsymbol{\lambda} = \mathbf{g}_x^T. \quad (1)$$

and then we multiply a *single* vector  $\boldsymbol{\lambda}^T$  by our  $M \times P$  matrix for only  $\theta(MP)$  work. Putting it all together, we obtain:

$$\left. \frac{dg}{d\mathbf{p}} \right|_{\mathbf{f}=0} = \mathbf{g}_p - \boldsymbol{\lambda}^T \mathbf{f}_p = \mathbf{g}_p - \boldsymbol{\lambda}^T (A_p \mathbf{x} - \mathbf{b}_p).$$

Again,  $A(\mathbf{p})$  and  $\mathbf{b}(\mathbf{p})$  are presumably specified analytically and thus  $A_p$  and  $\mathbf{b}_p$  can easily be computed (in some cases automatically, by automatic program differentiators such as ADIFOR). Note that the adjoint problem is of the same size as the original  $A\mathbf{x} = \mathbf{b}$  system, can use the same factorization (e.g. LU factorization  $A = LU$  immediately gives  $A^T = U^T L^T$ ), has the same condition number, and has the same spectrum of eigenvalues (the eigenvalues of  $A$  and  $A^T$  are identical) so iterative algorithms will have similar performance (and can use similar preconditioners)—in every sense, solving the adjoint problem should be no harder than solving the original problem.

### 3 Nonlinear equations

If  $\mathbf{x}$  satisfies some general, possibly nonlinear, equations  $\mathbf{f}(\mathbf{x}, \mathbf{p}) = 0$ , the process is almost exactly the same. Differentiating the  $\mathbf{f}$  equation, we find  $\mathbf{f}_x \mathbf{x}_p +$

<sup>3</sup>If  $M$  is sparse, then the cost might be significantly less than this  $O(M^2P)$  upper bound, but in any case solving  $P$  right-hand sides will be significantly more costly than solving a single right-hand side for the adjoint formulation.

<sup>4</sup>Another way of looking at this, and the source of the  $\boldsymbol{\lambda}$  notation, is to think of sort of a “Lagrange multiplier” process: replace  $g$  with  $\tilde{g} = g - \boldsymbol{\lambda}^T \mathbf{f}$  by adding a multiple  $\boldsymbol{\lambda}$  of  $\mathbf{f} = 0$ , and then choose  $\boldsymbol{\lambda}$  is a clever way to cancel the annoying derivative term. This gives the same result, and may be easier to generalize to some more complicated circumstances, however, such as differential-algebraic equations [2].

<sup>5</sup>For complex-valued  $\mathbf{x}$  and  $A$  and real  $g$ , instead of the transpose  $A^T$  one typically obtains the adjoint  $A^\dagger = A^{T*}$  (the conjugate transpose).

$\mathbf{f}_p = 0$  and thus  $\mathbf{x}_p = -\mathbf{f}_x^{-1} \mathbf{f}_p$ . Hence, we write

$$\frac{dg}{d\mathbf{p}} = \mathbf{g}_p + \mathbf{g}_x \mathbf{x}_p = \mathbf{g}_p - \underbrace{\mathbf{g}_x}_{1 \times M} \underbrace{[\mathbf{f}_x^{-1}]}_{M \times M} \underbrace{\mathbf{f}_p}_{M \times P} = \mathbf{g}_p - \underbrace{[\mathbf{g}_x \mathbf{f}_x^{-1}]}_{1 \times M} \underbrace{\mathbf{f}_p}_{M \times P}.$$

We solve for  $\mathbf{x}$  by whatever method, then solve for  $\boldsymbol{\lambda}$  from

$$\mathbf{f}_x^T \boldsymbol{\lambda} = \mathbf{g}_x^T, \quad (2)$$

and finally obtain

$$\left. \frac{dg}{d\mathbf{p}} \right|_{\mathbf{f}=0} = \mathbf{g}_p - \boldsymbol{\lambda}^T \mathbf{f}_p. \quad (3)$$

The only difference is that the adjoint equation (2) is not simply the adjoint of the equation for  $\mathbf{x}$ . Still, it is a single  $M \times M$  linear equation for  $\boldsymbol{\lambda}$  that should be of comparable (or lesser) difficulty to solving for  $\mathbf{x}$  (being equivalent to the cost of one Newton iteration for  $\mathbf{f} = 0$ ).

### 4 Eigenproblems

As a more complicated example illustrating the use of equations (2) and (3) from the previous sections, let us suppose that we are solving a linear eigenproblem  $A\mathbf{x} = \alpha\mathbf{x}$  and looking at some function  $g(\mathbf{x}, \alpha, \mathbf{p})$ . For simplicity, assume that  $A$  is real-symmetric and that  $\alpha$  is simple (non-degenerate; i.e.,  $\mathbf{x}$  is the only eigenvector for  $\alpha$ ).<sup>6</sup> In this case, we now have  $M + 1$  unknowns described by the column vector:

$$\tilde{\mathbf{x}} = \begin{pmatrix} \mathbf{x} \\ \alpha \end{pmatrix}.$$

The eigenequation  $\mathbf{f} = A\mathbf{x} - \alpha\mathbf{x}$  only gives us  $M$  equations and doesn’t completely determine  $\tilde{\mathbf{x}}$ , for two reasons. First, of course, there are many possible eigenvalues, but let’s assume that we have picked one in some fashion (e.g. the smallest  $\alpha$ , or the  $\alpha$  closest to  $\pi$ , or the third largest  $|\alpha|$ , or ...). Second, the eigenequation does not determine the length  $|\mathbf{x}|$ ; let’s arbitrarily pick  $|\mathbf{x}| = 1$  or  $\mathbf{x}^T \mathbf{x} = 1$ . This gives us  $M + 1$  equations  $\tilde{\mathbf{f}} = 0$  where:

$$\tilde{\mathbf{f}} = \begin{pmatrix} \mathbf{f} \\ \mathbf{x}^T \mathbf{x} - 1 \end{pmatrix}.$$

<sup>6</sup>Problems involving degenerate eigenvalues occur surprisingly often in optimization of eigenvalues (e.g. when maximizing the minimum eigenvalue of some system), and must be treated with special care. In that case, a generalization of the gradient is required to determine sensitivities or the steepest-descent direction [4], a more elaborate version of what is called *degenerate perturbation theory* in quantum mechanics [?].

We'll need  $M + 1$  adjoint variables  $\tilde{\boldsymbol{\lambda}}$ :

$$\tilde{\boldsymbol{\lambda}} = \begin{pmatrix} \boldsymbol{\lambda} \\ \beta \end{pmatrix}.$$

The adjoint equations (2) then give:

$$(A - \alpha)\boldsymbol{\lambda} = g_{\mathbf{x}}^T - 2\beta\mathbf{x}, \quad (4)$$

$$-\mathbf{x}^T\boldsymbol{\lambda} = g_{\alpha}. \quad (5)$$

The first equation, at first glance, seems to be problematic:  $A - \alpha$  is singular, with a null space of  $\mathbf{x}$ . It's, okay, though! First, we have to choose  $\beta$  so that solutions of equation (4) *exist*: the right-hand side must be orthogonal to  $\mathbf{x}$  so that it is not in the null space of  $A - \alpha$ . That is, we must have  $\mathbf{x}^T(g_{\mathbf{x}}^T - 2\beta\mathbf{x}) = 0$ , and thus  $\beta = \mathbf{x}^T g_{\mathbf{x}}^T / 2$  (since  $\mathbf{x}^T \mathbf{x} = 1$ ), and therefore  $\boldsymbol{\lambda}$  satisfies:

$$(A - \alpha)\boldsymbol{\lambda} = (1 - \mathbf{x}\mathbf{x}^T)g_{\mathbf{x}}^T = P g_{\mathbf{x}}^T \quad (6)$$

where  $P = 1 - \mathbf{x}\mathbf{x}^T$  is the projection operator into the space orthogonal to  $\mathbf{x}$ . This equation then has a solution, and in fact it has infinitely many solutions: we can add any multiple of  $\mathbf{x}$  to  $\boldsymbol{\lambda}$  and still have a solution. Equivalently, we can write  $\boldsymbol{\lambda} = \boldsymbol{\lambda}_0 + \gamma\mathbf{x}$  for  $\mathbf{x}^T \boldsymbol{\lambda}_0 = 0$  and some  $\gamma$ . Fortunately,  $\gamma$  is determined by (5):  $\gamma = -g_{\alpha}$ . Finally, with  $\boldsymbol{\lambda}_0$  determined by (6),<sup>7</sup> we can find the desired gradient via (3):

$$\left. \frac{dg}{d\mathbf{p}} \right|_{\mathbf{f}=0} = g_{\mathbf{p}} - \boldsymbol{\lambda}^T A_{\mathbf{p}} \mathbf{x} = g_{\mathbf{p}} - \boldsymbol{\lambda}_0^T A_{\mathbf{p}} \mathbf{x} + g_{\alpha} \mathbf{x}^T A_{\mathbf{p}} \mathbf{x}. \quad (7)$$

If we compare with  $\frac{dg}{d\mathbf{p}} = g_{\mathbf{p}} + g_{\mathbf{x}}\mathbf{x}_{\mathbf{p}} + g_{\alpha}\alpha_{\mathbf{p}}$ , we immediately see that  $\alpha_{\mathbf{p}} = \mathbf{x}^T A_{\mathbf{p}} \mathbf{x}$ . This is a well-known result from quantum physics and perturbation theory, where it is known as the Hellman-Feynman theorem.

## 5 Example inverse design

As a more concrete example of an inverse-design problem, let's consider the Schrodinger eigen-equation in one dimension,

$$\left[ -\frac{d^2}{dx^2} + V(x) \right] \psi(x) = E\psi(x),$$

with periodic boundaries  $\psi(x+2) = \psi(x)$ . Normally, we take a given  $V(x)$  and solve for  $\psi$  and  $E$ . Now, however, we will specify a particular  $\psi_0(x)$  and

<sup>7</sup>Since  $P$  commutes with  $A - \alpha$ , we can solve for  $\boldsymbol{\lambda}_0$  easily by an iterative method such as conjugate gradient: if we start with an initial guess orthogonal to  $\mathbf{x}$ , all subsequent iterates will also be orthogonal to  $\mathbf{x}$  and will thus converge to  $\boldsymbol{\lambda}_0$  (except for roundoff, which can be corrected by multiplying the final result by  $P$ ).

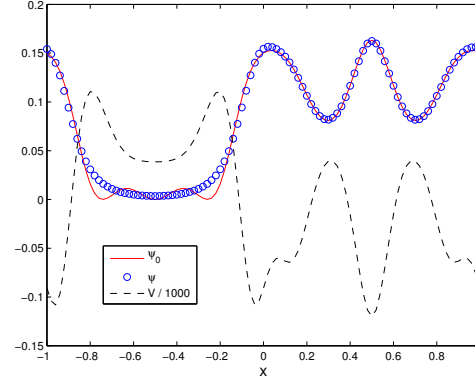


Figure 1: Optimized  $V(x)$  (scaled by 1/1000) and  $\psi(x)$  for  $\psi_0(x) = 1 + \sin[\pi x + \cos(3\pi x)]$  after 500 cg iterations.

find the  $V(x)$  that gives  $\psi(x) \approx \psi_0(x)$  for the ground-state eigenfunction (i.e. for the smallest eigenvalue  $E$ ). In particular, we will find the  $V(x)$  that minimizes

$$g = \int_{-1}^1 |\psi(x) - \psi_0(x)|^2 dx.$$

To solve this numerically, we will discretize the interval  $x \in [-1, 1]$  with  $M$  equally-spaced points  $x_n = n\Delta x$  ( $\Delta x = \frac{2}{M+1}$ ), and solve for the solution  $\psi(x_n)$  at these points, denoted by the vector  $\boldsymbol{\psi}$ . That is, to compare with the notation of the previous sections, we have the eigenvector  $\mathbf{x} = \boldsymbol{\psi}$ , the eigenvalue  $\alpha = E$ , and the parameters  $V(x_n)$  or  $\mathbf{p} = \mathbf{V}$ . If we discretize the eigenoperator with the usual center-difference scheme, we get  $A\boldsymbol{\psi} = E\boldsymbol{\psi}$  for:

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & -1 \\ -1 & 2 & -1 & 0 & \cdots & \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \vdots & & & \ddots & & \\ & & & & -1 & 2 & -1 \\ -1 & 0 & \cdots & 0 & -1 & 2 \end{pmatrix} + \text{diag}(\mathbf{V}).$$

As before, we normalize  $\boldsymbol{\psi}$  (and  $\boldsymbol{\psi}_0$ ) to  $\boldsymbol{\psi}^T \boldsymbol{\psi} = 1$ ,<sup>8</sup> giving a projection operator  $P = 1 - \boldsymbol{\psi}\boldsymbol{\psi}^T$  (or  $P = 1 - |\psi\rangle\langle\psi|$ , in Dirac notation). The discrete version of  $g$  is now

$$g(\boldsymbol{\psi}, \mathbf{V}) = (\boldsymbol{\psi} - \boldsymbol{\psi}_0)^T (\boldsymbol{\psi} - \boldsymbol{\psi}_0) \Delta x$$

where  $\boldsymbol{\psi}_0$  is  $\psi_0(x_n)$ , our target eigenfunction. Therefore,  $g_{\boldsymbol{\psi}} = 2(\boldsymbol{\psi} - \boldsymbol{\psi}_0)^T \Delta x$  and thus, by eq. (6), we

<sup>8</sup>We also have an arbitrary choice of sign, which we fix by choosing  $\int \psi dx > 0$ .

find  $\lambda$  via:

$$(A - E)\lambda = 2P(\psi - \psi_0)\Delta x, \quad (8)$$

with  $P\lambda = 0$  ( $\lambda = \lambda_0$  since  $g_E = 0$ ).  $g_V$  and  $g_E$  are both 0. Moreover,  $A_{V_n}$  is simply the matrix with 1 at  $(n, n)$  and 0's elsewhere, and thus from (7):

$$\frac{dg}{dV_n} = -\lambda_n \psi_n$$

or equivalently  $\frac{dg}{dV} = -\lambda \odot \psi$  where  $\odot$  is the point-wise product (`.*` in Matlab).

Whew! Now how do we solve these equations numerically? This is illustrated by the Matlab function `schrodinger_fd_adj` given below. We set up  $A$  as a sparse matrix, then find the smallest eigenvalue and eigenvector via the `eigs` function (which uses an iterative Arnoldi method). Then we solve (8) for  $\lambda$  via the Matlab `pcg` function (preconditioned conjugate-gradient, although we don't bother with a preconditioner).

Then, given  $g$  and  $\frac{dg}{dV}$ , we then just plug it into some optimization algorithm. In particular, nonlinear conjugate gradient seems to work well for this problem.<sup>9</sup>

## 5.1 Optimization results

In this section, we give a few example results from running the above procedure (nonlinear cg optimization) for  $M = 100$ . As the starting guess for our optimization, we'll just use  $V(x) = 0$ . That is, we are doing a *local optimization* in a *100-dimensional space*, using the adjoint method to get the gradient. It is somewhat remarkable that this works—in a few seconds on a PC, it converges to a very good solution!

We'll try a couple of example  $\psi_0(x)$  functions. To start with, let's do  $\psi_0(x) = 1 + \sin[\pi x + \cos(3\pi x)]$ . (Note that the ground-state  $\psi$  will never have any nodes, so we require  $\psi_0 \geq 0$  everywhere.) This  $\psi_0(x)$ , along with the resulting  $\psi(x)$  and  $V(x)$  after 500 cg iterations, are shown in figure 1. The solution  $\psi(x)$  matches  $\psi_0(x)$  very well except for a couple of small ripples, and  $V(x)$  is quite complicated—not something you could easily guess!

Oh, but that  $\psi_0$  was too easy! Let's try one with discontinuities:  $\psi_0(x) = 1 - |x|$  for  $|x| < 0.5$  and 0 otherwise (which looks a bit like a “house”). This

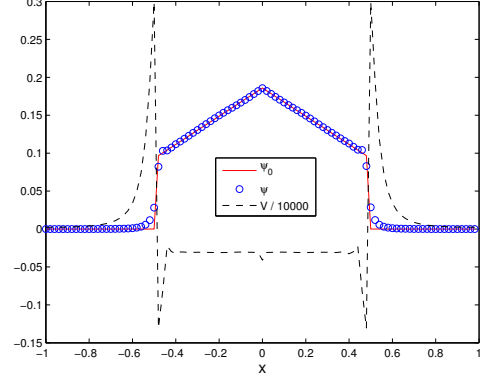


Figure 2: Optimized  $V(x)$  (scaled by 1/10000) and  $\psi(x)$  for  $\psi_0(x) = 1 - |x|$  for  $|x| < 0.5$ , after 5000 cg iterations.

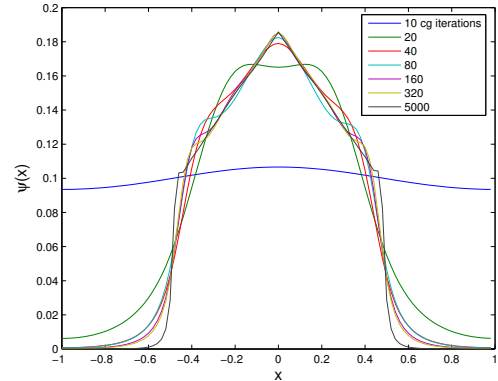


Figure 3: Optimized  $\psi(x)$  for  $\psi_0(x) = 1 - |x|$  for  $|x| < 0.5$ , after various numbers of nonlinear conjugate-gradient iterations (from 10 to 10000).

<sup>9</sup>I used the nonlinear conjugate-gradient Matlab `conj_grad` routine from:

<http://www2.imm.dtu.dk/~hbn/Software/>

$\psi_0(x)$ , along with the resulting  $\psi(x)$  and  $V(x)$  after 500 cg iterations, are shown in figure 2. Amazingly, it still captures  $\psi_0$  pretty well, although it has a bit more trouble with the discontinuities than with the slope discontinuity. This time, we let it converge for 5000 cg iterations to give it a bit more time. Was this really necessary? In figure 3, we plot  $\psi(x)$  for 10, 20, 40, 80, 160, 320, and 5000 cg iterations. It gets the rough shape pretty quickly, but the discontinuous features are converging fairly slowly. (Presumably this could be improved if we found a good preconditioner, or perhaps by a different optimization method or objective function.)

## 5.2 Matlab code

The following code solves for  $g$  and  $\frac{dg}{dV}$ , not to mention the eigenfunction  $\psi$  and the corresponding eigenvalue  $E$ , for a given  $V$  and  $\psi_0$ .

```
% Usage: [g,gp,E,psi] = schrodinger_fd_adj(x, V, psi0)
%
% Given a column-vector x(:) of N equally spaced x points and
% V of the potential V(x) at those points, solves Schrodinger
% [ -d^2/dx^2 + V(x) ] psi(x) = E psi(x)
% with periodic boundaries for the lowest "ground state" eigen
% wavefunction psi.
%
% Furthermore, it computes the function g = integral |psi - p
% the gradient gp = dg/dV (at each point x).

function [g,gp,E,psi] = schrodinger_fd_adj(x, V, psi0)
    dx = x(2) - x(1);
    N = length(x);
    A = spdiags([ones(N,1), -2 * ones(N,1), ones(N,1)], -1:1, N
    A(1,N) = 1;
    A(N,1) = 1;
    A = - A / dx^2 + spdiags(V, 0, N,N);

    opts.disp = 0;
    [psi,E] = eigs(A, 1, 'sa', opts);
    E = E(1,1);
    if sum(psi) < 0
        psi = -psi; % pick sign; note that psi' * psi = 1 from ei
    end

    gpsi = psi - psi0;
    g = gpsi' * gpsi * dx;
    gpsi = gpsi * 2*dx;

    P = @(x) x - psi * (psi' * x); % projection onto direction

    [lambda,flag] = pcg(A - spdiags(E*ones(N,1), 0, N,N), P(gps
    lambda = P(lambda);
    gp = -real(conj(lambda) .* psi);

    disp(g);
```

## 6 Initial-value problems

So far, we have looked at  $\mathbf{x}$  that are determined by “simple” algebraic equations (which may come from a PDE, etcetera). What if, instead, we are determining  $\mathbf{x}$  by integrating a set of equations in *time*? The simplest example of this is an initial-value problem for a linear, time-independent, homogeneous set of ODEs:

$$\dot{\mathbf{x}} = B\mathbf{x}$$

whose solution after a time  $t$  for  $\mathbf{x}(0) = \mathbf{b}$  is formally:

$$\mathbf{x} = \mathbf{x}(t) = e^{Bt}\mathbf{b}.$$

This, however, is exactly a linear equation  $A\mathbf{x} = \mathbf{b}$  with  $A = e^{-Bt}$ , so we can just quote our results from earlier! That is, suppose we are optimizing (or evaluating the sensitivity) of some function  $g(\mathbf{x}, \mathbf{p})$  based on the solution  $\mathbf{x}$  at time  $t$ . Then we find the adjoint vector  $\boldsymbol{\lambda}$  via (1):

$$e^{-B^T t} \boldsymbol{\lambda} = g_{\mathbf{x}}^T.$$

Equivalently,  $\boldsymbol{\lambda}$  is the exactly the solution  $\boldsymbol{\lambda}(t)$  after a time  $t$  of its *own* adjoint ODE:

$$\dot{\boldsymbol{\lambda}} = B^T \boldsymbol{\lambda}$$

with initial condition  $\boldsymbol{\lambda}(0) = g_{\mathbf{x}}^T$ . We should have expected this by now: solving for  $\boldsymbol{\lambda}$  always involves a task of similar complexity to finding  $\mathbf{x}$ , so if we found  $\mathbf{x}$  by integrating an ODE then we find  $\boldsymbol{\lambda}$  by an ODE too! Of course, we need not solve these ODEs by matrix exponentials; we can use Runge-Kutta, forward Euler, or (if  $B$  comes from a PDE) whatever scheme we deem appropriate (e.g. Crank-Nicolson).

One important property to worry about is *stability*, and here we are in luck. The eigenvalues of  $B$  and  $B^T$  are the same, and so if one is stable (eigenvalues with absolute values  $\leq 1$ ) then the other is!

Finally, we can write down the gradient  $\frac{dg}{d\mathbf{p}}$  via equation (3):

$$\frac{dg}{d\mathbf{p}} = g_{\mathbf{p}} - \boldsymbol{\lambda}^T (A_{\mathbf{p}}\mathbf{x} - \mathbf{b}_{\mathbf{p}}).$$

Now, since  $A = e^{-Bt}$ , one might be tempted to write  $A_{\mathbf{p}} = -B_{\mathbf{p}}t \cdot A$ , but this is not true except in the *very* special case where  $B_{\mathbf{p}}$  commutes with  $B$ ! Unfortunately, the general expression for differentiating a matrix exponential turns out to be more complicated:  $A_{\mathbf{p}} = -\int_0^t e^{-Bt'} B_{\mathbf{p}} e^{-B(t-t')} dt'$ , and so,

$$\frac{dg}{d\mathbf{p}} = g_{\mathbf{p}} + \int_0^t \boldsymbol{\lambda}^T(t-t') B_{\mathbf{p}} \mathbf{x}(t') dt' + \boldsymbol{\lambda}^T \mathbf{b}_{\mathbf{p}}.$$

This is especially unfortunate because it usually means that we have to *store*  $\mathbf{x}(t')$  at all times  $0 \leq t' \leq t$  in order to compute the integral. Adjoint methods are storage-intensive for time-dependent problems!

More generally, of course, one might wish to include time-varying  $A$ , nonlinearities, inhomogeneous (source) terms, etcetera, into the equations to integrate. A very general formulation of the problem, for differential-algebraic equations (DAEs), can be found in [2]. A similar general principle remains, however: the adjoint variable  $\boldsymbol{\lambda}$  is determined by integrating a similar (adjoint) DAE, using the *final* value of  $\mathbf{x}(t)$  to compute the *initial* condition of  $\boldsymbol{\lambda}(0)$ . In fact, the  $\boldsymbol{\lambda}(t)$  equation is actually often interpreted as being integrated *backwards* in time from  $t$  to 0. Alternatively, one can consider a “discrete-time” situation of *recurrence* equations, in which case the adjoint problem is a recurrence “backward in time”—see my online notes on adjoint methods for recurrences.

## References

- [1] R. M. Errico, “What is an adjoint model?,” *Bulletin Am. Meteorological Soc.*, vol. 78, pp. 2577–2591, 1997.
- [2] Y. Cao, S. Li, L. Petzold, and R. Serban, “Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution,” *SIAM J. Sci. Comput.*, vol. 24, no. 3, pp. 1076–1089, 2003.
- [3] G. Strang, *Computational Science and Engineering*. Wellesley, MA: Wellesley-Cambridge Press, 2007.
- [4] A. P. Seyranian, E. Lund, and N. Olhoff, “Multiple eigenvalues in structural optimization problems,” *Structural Optimization*, vol. 8, pp. 207–227, 1994.