Properties of the adjoint equations

The adjoint equations have a reputation for being counterintuitive. When told the adjoint equations run backwards in time, this can strike the novice as bizarre. Therefore, it is worth taking some time to explore these properties, until it is *obvious* that the adjoint system should run backwards in time, and (more generally) reverse the propagation of information. In fact, these supposedly confusing properties are induced by nothing more exotic than simple transposition.

The adjoint reverses the propagation of information

A simple advection example

Suppose we are are solving a one-dimensional advection-type equation on a mesh with three nodes, at $x_0 = 0$, $x_1 = 0.5$, and $x_2 = 1$. The velocity goes from left to right, and so we impose an inflow boundary condition at the left-most node x_0. A simple sketch of the linear system that might describe this configuration could look as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ a & b & 0 \\ c & d & e \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},$$

where a, b, c, d and e are some coefficients of the matrix arising from a discretisation of the equation. The equation for u_1 does not depend on u_2, as information is flowing from left to right. The structure of the matrix dictates the propagation of information of the system: first u_0 is set to the boundary condition value, then u_1 may be computed, and then finally u_2. The *lower-triangular nature of the matrix reflects the rightward propagation of information.*

Notice that u_0 is *prescribed*: that is, the value of u_0 does not depend on the values at any other nodes; all off-diagonal entries on the row for u_0 are zero. Notice further that the value u_2 is *diagnostic*: no other nodes depend on its value; all off-diagonal entries on its column are zero.

Now suppose that we take the adjoint of this system with respect to some functional $J(u)$. The operator is linear (no entry in the matrix depends on u), and so the adjoint of this system is just its transpose:

$$\begin{pmatrix} 1 & a & c \\ 0 & b & d \\ 0 & 0 & e \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} \partial J/\partial u_0 \\ \partial J/\partial u_1 \\ \partial J/\partial u_2 \end{pmatrix},$$

where λ is the adjoint variable corresponding to u. Observe that transposing the forward system yields an upper-triangular adjoint system: *the adjoint propagates information from right to left, in the opposite sense to the propagation of the forward system*. To solve this system, one would first solve for λ_2, then compute λ_1, and finally λ_0.

Further notice that λ_2 is now prescribed: it can be computed directly from the data, with no dependencies on the values of other adjoint variables; all of the off-diagonal entries in its row are zero. λ_0 is now diagnostic: no other variables depend on its value; all off-diagonal entries in its column are zero.

The adjoint of the advection equation

If the forward equation is $u \cdot \nabla T$, where u is the advecting velocity and T is the advected tracer, then its corresponding adjoint term is $-u \cdot \nabla \lambda$. The adjoint advection equation is itself an advection equation, with the reverse of the forward velocity.

Prescribed and diagnostic variables

This is a general pattern. Variables that are prescribed in the forward model are diagnostic in the adjoint; variables that are diagnostic in the forward model are prescribed in the adjoint.

A time-dependent example

Now consider a time-dependent system. For convenience, we assume the system is linear, but the result holds true in exactly the same way for nonlinear systems. We start with an initial condition f_0 for u_0 (where the subscript denotes the timestep, rather than the node). We then use this information to compute the value at the next timestep, u_1. This information is then used to compute u_2, and so on. This temporal structure can be represented as a *block-structured* matrix:

$$\begin{pmatrix} I & 0 & 0 \\ A & B & 0 \\ C & D & E \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix},$$

where I is the identity operator, A, B, C, D and E are some operators arising from the discretisation of the time-dependent system, f_0 is the initial condition for u_0, and f_n is the source term for the equation for u_n.

Again, the *temporal propagation of information forward in time is reflected in the lower-triangular structure of the matrix*. This reflects the fact that it is possible to timestep the system, and solve for parts of the solution u at a time. If the discrete operator were not lower-triangular, all timesteps of the solution u would be coupled, and would have to be solved for together.

Notice again that the value at the initial time u_0 is prescribed, and the value at the final time u_2 is diagnostic.

Now let us take the adjoint of this system. Since the operator has been assumed to be linear, the adjoint of this system is given by the block-structured matrix

$$\begin{pmatrix} I & A^* & C^* \\ 0 & B^* & D^* \\ 0 & 0 & E^* \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} \partial J/\partial u_0 \\ \partial J/\partial u_1 \\ \partial J/\partial u_2 \end{pmatrix},$$

where λ is the adjoint variable corresponding to u. Observe that the adjoint system is now upper-triangular: *the adjoint propagates information from later times to earlier times, in the opposite sense to the propagation of the forward system*. To solve this system, one would first solve for λ_2, then compute λ_1, and finally λ_0.

Notice once more that the prescribed-diagnostic relationship applies. In the forward model, the initial condition is prescribed, and the solution at the final time is diagnostic. In the adjoint model, the solution at the final time is prescribed (a so-called *terminal condition*, rather than an initial condition), and the solution at the beginning of time is diagnostic. This is why when the adjoint of a continuous system is derived, the formulation always includes the specification of a terminal condition on the adjoint system.

The adjoint equation is linear

As noted in the previous section, the operator of the tangent linear system is the linearisation of the operator about the solution u; therefore, the adjoint system is always linear in λ.

This has two major effects. The first is a beneficial effect on the computation time of the adjoint run: while the forward model may be nonlinear, *the adjoint is always linear, and so it can be much cheaper to solve than the forward model*. For example, if the forward model employs a Newton solver for the nonlinear problem that uses on average 5 linear solves to converge to machine precision, then a rough estimate for the adjoint computation is that it will take $1/5$ the runtime of the forward model.

The second major effect is on the storage requirements of the adjoint run. Unfortunately, this effect is not beneficial. The adjoint operator is a linearisation of the nonlinear operator about the solution u: therefore, *if the forward model is nonlinear, the forward solution must be available to assemble the adjoint system*. If the forward model is steady, this is not a significant difficulty: however, *if the forward model is time-dependent, the entire solution trajectory through time must be available*.

Unconverged nonlinear iterations

Note that the nonlinear iteration *has to converge* for the linearisation about the solution at that timestep to be valid. If the model does not drive the nonlinear problem to convergence (perhaps it only does a fixed number of Picard iterations, say), then it is not consistent to see the nonlinear solve as one equation, and to trade it for a linear solve in the adjoint. In other words, if the nonlinear solve does not converge, then each iteration of the *unconverged* nonlinear solve induces a linear solve in the adjoint system, and so the adjoint will take approximately the same runtime as the forward model.

Converging your nonlinear problem is not only more accurate, it makes the adjoint relatively much more efficient!

The obvious approach to making the entire solution trajectory available is to store the value of every variable solved for. This approach is the simplest, and it is the most efficient option if enough storage is available on the machine to store the entire solution at once. However, for long simulations with many degrees of freedom, it is usually impractical to store the entire solution trajectory, and therefore some alternative approach must be implemented.

The space cost of storing all variables is linear in time (double the timesteps, double the storage) and the time cost is constant (no extra recomputation is required). The opposite strategy, of storing nothing and recomputing everything

when it becomes necessary, is quadratic in time and constant in space. A *checkpointing algorithm* attempts to strike a balance between these two extremes to control both the spatial requirements (storage space) and temporal requirements (recomputation).

Checkpointing algorithms have been well studied in the literature, usually in the context of algorithmic differentiation [4M-Gri92] [4M-HS05] [4M-SW10] [4M-WMI09]. There are two categories of checkpointing algorithms: *offline* algorithms and *online* algorithms. In the offline case, the number of timesteps is known in advance, and so the optimal distribution of checkpoints may be computed a priori (and hence "offline"), while in the online case, the number of timesteps is not known in advance, and so the distribution of checkpoints must be computed during the run itself. Of particular note is the revolve software of Griewank and Walther, which achieves logarithmic growth of both space *and* time [4M-GW00]. This algorithm is provably optimal for the offline case [4M-GPRS96].

Checkpointing in dolfin-adjoint

Libadjoint, the library that is the backbone of dolfin-adjoint, embeds the revolve algorithm of Griewank and Walther. Activating checkpointing is a simple matter of adding two function calls. For more details, see the manual section on checkpointing.

Summary

Now that the adjoint and tangent linear equations have been introduced, and some of their properties discussed, let us see in more detail the applications of these concepts. This is discussed in the next section.

References

[4M-Gri92] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992. doi:10.1080/10556789208805505.

[4M-GW00] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000. doi:10.1145/347837.347846.

[4M-GPRS96] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, 95–106. Philadelphia, PA, 1996. SIAM.

[4M-HS05] M. Hinze and J. Sternberg. A-revolve: an adaptive memory-reduced procedure for calculating adjoints; with an application to computing adjoints of the instationary Navier–Stokes system. *Optimization Methods and Software*, 20(6):645–663, 2005. doi:10.1080/10556780410001684158.

[4M-SW10] P. Stumm and A. Walther. New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing*, 32(2):836–854, 2010. doi:10.1137/080742439.

[4M-WMI09] Q. Wang, P. Moin, and G. Iaccarino. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing*, 31(4):2549–2567, 2009. doi:10.1137/080727890.