Anderson Acceleration: Algorithms and Implementations[1]

Homer F. Walker[2]

September 2010; revised June 2011 and October 2011

The following provides outlines of Anderson acceleration in various forms, along with some ancillary algorithms and implementational outlines. The notation is that used in [6].

Suppose we want to solve $x = g(x)$ for some $g : \mathbb{R}^n \to \mathbb{R}^n$. Basic fixed-point iteration for this problem is as follows:

Algorithm FPI: Fixed-Point Iteration

Given x_0.
For $k = 0, 1, \ldots$
  Set $x_{k+1} = g(x_k)$.

For this iteration, the usual general form of Anderson acceleration is as follows:

Algorithm AA: Anderson Acceleration

Given x_0 *and* $m \geq 1$.
Set $x_1 = g(x_0)$.
For $k = 1, 2, \ldots$
  Set $m_k = \min\{m, k\}$.
  Set $F_k = (f_{k-m_k}, \ldots, f_k)$, *where* $f_i = g(x_i) - x_i$.
  Determine $\alpha^{(k)} = (\alpha_0^{(k)}, \ldots, \alpha_{m_k}^{(k)})^T$ *that solves*

$$\min_{\alpha=(\alpha_0,\ldots,\alpha_{m_k})^T} \|F_k\alpha\|_2 \text{ s. t. } \sum_{i=0}^{m_k} \alpha_i = 1.$$

  Set $x_{k+1} = \sum_{i=0}^{m_k} \alpha_i^{(k)} g(x_{k-m_k+i})$.

The constrained linear least-squares problem in Algorithm AA can be solved in a number of ways; see [5] for several alternatives. Our preference is to recast it in an unconstrained form suggested in [4], [3], and [6] that is straightforward to solve and convenient for implementing efficient updating of QR

[1] Worcester Polytechnic Institute Mathematical Sciences Department Research Report MS-6-15-50, June 2011.

[2] Mathematical Sciences Department, Worcester Polytechnic Institute, Worcester, MA 01609-2280 (walker@wpi.edu). This work was supported in part by US National Science Foundation grant DMS 0915183 and US Department of Energy award DE-SC0004880. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the Department of Energy

factorizations. We define $\Delta f_i = f_{i+1} - f_i$ for each i and set $\mathcal{F}_k = (\Delta f_{k-m_k}, \ldots, \Delta f_{k-1})$. Then the least-squares problem is equivalent to

$$\min_{\gamma=(\gamma_0,\ldots,\gamma_{m_k-1})^T} \|f_k - \mathcal{F}_k\gamma\|_2,$$

where α and γ are related by $\alpha_0 = \gamma_0$, $\alpha_i = \gamma_i - \gamma_{i-1}$ for $1 \le i \le m_k - 1$, and $\alpha_{m_k} = 1 - \gamma_{m_k-1}$.

This unconstrained least-squares problem leads to a modified form of Anderson acceleration. Denoting the least-squares solution by $\gamma^{(k)} = (\gamma_0^{(k)}, \ldots, \gamma_{m_k-1}^{(k)})^T$, we have

$$x_{k+1} = g(x_k) - \sum_{i=0}^{m_k-1} \gamma_i^{(k)}[g(x_{k-m_k+i+1}) - g(x_{k-m_k+i})] = g(x_k) - \mathcal{G}_k\gamma^{(k)},$$

where $\mathcal{G}_k = (\Delta g_{k-m_k}, \ldots, \Delta g_{k-1})$ with $\Delta g_i = g(x_{i+1}) - g(x_i)$ for each i. Then Anderson acceleration becomes

Algorithm AA: Anderson Acceleration

Given x_0 and $m \ge 1$.
Set $x_1 = g(x_0)$.
For $k = 1, 2, \ldots$
  Set $m_k = \min\{m, k\}$.
  Determine $\gamma^{(k)} = (\gamma_0^{(k)}, \ldots, \gamma_{m_k-1}^{(k)})^T$ that solves

$$\min_{\gamma=(\gamma_0,\ldots,\gamma_{m_k-1})^T} \|f_k - \mathcal{F}_k\gamma\|_2.$$

  Set $x_{k+1} = g(x_k) - \mathcal{G}_k\gamma^{(k)}$.

As the algorithm proceeds, the successive least-squares problems can be solved efficiently by updating the factors in the decomposition $\mathcal{F}_k = Q_kR_k$. To describe this, we assume a "thin" QR decomposition, i.e., $Q_k \in \mathbb{R}^{n\times m_k}$ and $R_k \in \mathbb{R}^{m_k\times m_k}$, for which the solution of the least-squares problem is obtained by solving the $m_k \times m_k$ triangular system $R_k\gamma = Q_k' * f_k$, where the prime denotes transpose and "$*$" denotes matrix-vector multiplication.

The following provides a short outline of how the updating goes. Note: From here on, MATLAB-like notation is used increasingly; also, it is convenient to omit subscripts on the Q and R factors.

Updating the QR factors. Each $\mathcal{F}_k$ is $n \times m_k$ and is obtained from $\mathcal{F}_{k-1}$ by adding a column on the right and, if the resulting number of columns is greater than m, also deleting the first column on the left. If $m_{k-1} = m$, then this column deletion must occur, and a little storage and arithmetic can be saved by doing it first, i.e., before adding the column on the right. The resulting scheme for updating at the kth step is as follows:

1. When $k = 1$, set $m_1 = 1$ and initialize $\mathcal{F}_1 = QR$ with $Q = \Delta f_0 / \|\Delta f_0\|_2$ and $R = \|\Delta f_0\|_2$.
2. When $k > 1$, we have $\mathcal{F}_{k-1} = QR$, where $\mathcal{F}_{k-1}$ is $n \times m_{k-1}$, and update as follows:
   (a) If $m_{k-1} = m$, then delete the left column of $\mathcal{F}_{k-1}$, updating Q and R so that $\mathcal{F}_{k-1} \leftarrow \mathcal{F}_{k-1}(:, 2:m) = QR$ and reducing $m_{k-1} \leftarrow m - 1$.
   (b) Add a column on the right, updating Q and R so that $\mathcal{F}_k = [\mathcal{F}_{k-1}, \Delta f_{k-1}] = QR$ and setting $m_k = m_{k-1} + 1$.

We now give more details of the QR-updating procedures. The procedure for adding a column on the right is simpler, consisting of a single modified Gram–Schmidt sweep, so we describe it first. From here on, we use "End" statements to make the ends of loops and conditionals clear.

For $i = 1 : m_k - 1$
  Set $R(i, m_k) = Q(:, i)' * \Delta f_{k-1}$.
  Update $\Delta f_{k-1} \leftarrow \Delta f_{k-1} - R(i, m_k) * Q(:, i)$.
End
Set $Q(:, m_k) = \Delta f_{k-1} / \|\Delta f_{k-1}\|_2$ *and* $R(m_k, m_k) = \|\Delta f_{k-1}\|_2$.

Note that the original value of Δf_{k-1} is lost, but that's OK because it isn't needed later in an actual implementation.

We now describe the updating procedure for deleting the left column of $\mathcal{F}_{k-1}$ when $m_{k-1} = m$. Here is the general idea: If $\mathcal{F}_{k-1} = Q*R$, then $\mathcal{F}_{k-1}(:, 2:m) = Q*R(:, 2:m)$, where $R(:, 2:m) \in \mathbb{R}^{m \times (m-1)}$ is upper-Hessenberg. We determine $m \times m$ Givens rotations $J_1, \ldots, J_{m-1}$ such that $J_{m-1} * \ldots * J_1 * R(:, 2:m) \in \mathbb{R}^{m \times (m-1)}$ is upper-triangular with an all-zero bottom row. Then

$$\mathcal{F}_{k-1}(:, 2:m) = Q * R(:, 2:m) = Q * J_1' * \ldots * J_{m-1}' * J_{m-1} * \ldots * J_1 * R(:, 2:m),$$

and so we update $Q \leftarrow Q * J_1' * \ldots * J_{m-1}'(:, 1:m-1)$ and $R \leftarrow J_{m-1} * \ldots * J_1 * R(1:m-1, 2:m)$.

Since this updating procedure is rather long and ultimately appears twice in the final version of Algorithm AA, it is appropriate to describe it in subroutine form. In writing the calling statement, we follow the form of MATLAB's `qrdelete` function, which accomplishes the same thing (and should be used in a MATLAB implementation). (The "1" in the calling sequence indicates that the first column is to be deleted.) On input, $Q \in \mathbb{R}^{n \times m}$ and $R \in \mathbb{R}^{m \times m}$ are such that $\mathcal{F}_{k-1} = Q * R$; on output, $Q \in \mathbb{R}^{n \times (m-1)}$ and $R \in \mathbb{R}^{(m-1) \times (m-1)}$ are such that $\mathcal{F}_{k-1}(:, 2:m) = Q * R$. Note that $\mathcal{F}_{k-1}$ never appears in the algorithm; all we need are its Q and R factors.

[Q, R] = qrdelete(Q, R, 1)
  Given $Q \in \mathbb{R}^{n \times m}$ *and* $R \in \mathbb{R}^{m \times m}$.
  For $i = 1 : m - 1$

Set $temp = \sqrt{R(i,i+1)^2 + R(i+1,i+1)^2}$.
Set $c = R(i,i+1)/temp$ *and* $s = R(i+1,i+1)/temp$.
Update $R(i,i+1) \leftarrow temp$ *and* $R(i+1,i+1) \leftarrow 0$.
If $i < m-1$
For $j = i+2 : m$
Set $temp = c * R(i,j) + s * R(i+1,j)$.
Set $R(i+1,j) = -s * R(i,j) + c * R(i+1,j)$.
Set $R(i,j) = temp$.
End
End
For $\ell = 1 : n$
Set $temp = c * Q(\ell,i) + s * Q(\ell,i+1)$.
Set $Q(\ell,i+1) = -s * Q(\ell,i) + c * Q(\ell,i+1)$.
Set $Q(\ell,i) = temp$.
End
End
Update $Q \leftarrow Q(:, 1 : m-1)$ *and* $R \leftarrow R(1 : m-1, 2 : m)$.

The following is a "semi-final" version of Anderson acceleration that expresses the basic algorithm in an implementable form. This will be followed by a final version that incorporates additional useful features discussed below. The hope is that these versions can serve as better outlines for writing actual code than the previous, more abstract versions. Accordingly, there are some minor changes: The iterations now begin with $k = 0$ and, for clarity, m and m_k appearing in previous versions have been replaced by m_{Max} and m_{AA}, respectively. Also, we have removed iteration subscripts that would not be appropriate in an actual implementation. Finally, the number of iterations has been limited to a prescribed maximum $itmax$, a feature shared by virtually all iterative algorithms. Other possible termination criteria may vary according to circumstances and are discussed following the final version of the algorithm.

Remark. The matrices that appear in these versions are $\mathcal{G}$, Q, and R. In MATLAB, $\mathcal{G}$ should be initialized as the empty matrix; Q and R can be built up incrementally without initialization. In non-MATLAB coding, $\mathcal{G}$ and Q should be dimensioned $n \times m_{Max}$, and R should be dimensioned $m_{Max} \times m_{Max}$.

Algorithm AA: Anderson Acceleration
Given x *and* $m_{Max} \geq 1$.
Initialize $m_{AA} = 0$ *and* $\mathcal{G} = []$.
For $k = 0, 1, \ldots,$ *itmax*
Evaluate $g_{cur} = g(x)$ *and* $f_{cur} = g_{cur} - x$.
If $k > 0$
Set $\Delta f = f_{cur} - f_{old}$ *and* $\Delta g = g_{cur} - g_{old}$.

If $m_{AA} < m_{Max}$
Update $\mathcal{G} \leftarrow [\mathcal{G}, \Delta g]$.
Else
Update $\mathcal{G} \leftarrow [\mathcal{G}(:, 2 : m_{AA}), \Delta g]$.
End
Update $m_{AA} \leftarrow m_{AA} + 1$.
End
Update $f_{old} = f_{cur}$ *and* $g_{old} = g_{cur}$.
If $m_{AA} = 0$
Update $x = g_{cur}$.
Else
If $m_{AA} = 1$
Set $Q(:, 1) = \Delta f / \|\Delta f\|_2$ *and* $R(1, 1) = \|\Delta f\|_2$.
Else
If $m_{AA} > m_{Max}$
Call [Q, R] = qrdelete(Q, R, 1).
Update $m_{AA} \leftarrow m_{AA} - 1$.
End
For $i = 1 : m_{AA} - 1$
Set $R(i, m_{AA}) = Q(:, i)' * \Delta f$.
Update $\Delta f \leftarrow \Delta f - R(i, m_{AA}) * Q(:, i)$.
End
Set $Q(:, m_{AA}) = \Delta f / \|\Delta f\|_2$ *and* $R(m_{AA}, m_{AA}) = \|\Delta f\|_2$.
End
Solve $R\gamma = Q' * f_{cur}$.
Update $x \leftarrow g_{cur} - \mathcal{G} * \gamma$.
End
End

We now discuss several useful features that can be added to this basic algorithm.

Damping. Anderson's original formulation in [1] allows a more general step

$$\begin{aligned} x_{k+1} &= (1 - \beta_k) \sum_{i=0}^{m_k} \alpha_i^{(k)} x_{k-m_k+i} + \beta_k \sum_{i=0}^{m_k} \alpha_i^{(k)} g(x_{k-m_k+i}) \\ &= \sum_{i=0}^{m_k} \alpha_i^{(k)} x_{k-m_k+i} + \beta_k \left(\sum_{i=0}^{m_k} \alpha_i^{(k)} g(x_{k-m_k+i}) - \sum_{i=0}^{m_k} \alpha_i^{(k)} x_{k-m_k+i} \right), \end{aligned}$$

where $\beta_k > 0$. Usually in practice, β_k is a *damping* parameter, i.e., $0 < \beta_k \leq 1$, used to improve convergence by reducing step lengths when iterates are not near a solution. It is important to note that the damped step is from the point $x_k^{min} \equiv \sum_{i=0}^{m_k} \alpha_i^{(k)} x_{k-m_k+i}$ and not from x_k. (If the problem were linear, then x_k^{min} would give minimal fixed-point residual norm among all points in the affine subspace containing $x_{k-m_k}, \ldots, x_k$.) The points $\{x_{k-m}, \ldots, x_k\}$ are not available to compute x_k^{min} directly; however, x_k^{min} can be computed without increasing storage, as follows: In analogy with the undamped iterate expression $x_{k+1} = g(x_k) - \mathcal{G}_k\gamma^{(k)}$, we have $x_k^{min} = x_k - \mathcal{X}_k\gamma^{(k)}$, where $\mathcal{X}_k = (\Delta x_{k-m_k}, \ldots, \Delta x_{k-1})$ with $\Delta x_i = x_{i+1} - x_i$ for each i. Since $x_k = g(x_k) - f_k$ and $\mathcal{X}_k = \mathcal{G}_k - \mathcal{F}_k$, we obtain

$$\begin{aligned} x_k^{min} &= g(x_k) - f_k - (\mathcal{G}_k - \mathcal{F}_k)\gamma^{(k)} = \left(g(x_k) - \mathcal{G}_k\gamma^{(k)}\right) - \left(f_k - \mathcal{F}_k\gamma^{(k)}\right) \\ &= x_{k+1} - \left(f_k - \mathcal{F}_k\gamma^{(k)}\right), \end{aligned}$$

where $x_{k+1} = g(x_k) - \mathcal{G}_k\gamma^{(k)}$ is the undamped iterate. With $\mathcal{F}_k = QR$, this leads to the following strategy:

1. Compute the undamped iterate $x_{k+1} = g(x_k) - \mathcal{G}_k\gamma^{(k)}$.
2. Update x_{k+1} by

$$\begin{aligned} x_{k+1} \leftarrow x_k^{min} + \beta_k\left(x_{k+1} - x_k^{min}\right) &= x_{k+1} - \left(f_k - \mathcal{F}_k\gamma^{(k)}\right) + \beta_k\left(f_k - \mathcal{F}_k\gamma^{(k)}\right) \\ &= x_{k+1} - (1-\beta_k)\left(f_k - QR\gamma^{(k)}\right). \end{aligned}$$

This strategy requires no additional storage but does require some additional arithmetic, mainly to compute $QR\gamma^{(k)}$.

Condition control. In practice, there is often a danger that $\mathcal{F}_k$ will become ill-conditioned as the iterations proceed. With $\mathcal{F}_k = QR$, the conditioning of $\mathcal{F}_k$ can be monitored by monitoring the condition number of R, which can be done inexpensively, e.g., using *incremental condition estimation* [2] or, in MATLAB programming, the `cond` command. Acceptable conditioning can be maintained by monitoring the condition number of R and, if it exceeds a prescribed threshhold, updating Q and R (implicitly deleting columns of $\mathcal{F}_k$ on the left) using the `qrdelete` procedure discussed previously until the condition number drops below the threshhold.

Delayed acceleration start. In some applications, it may be advantageous to delay the start of Anderson acceleration until the underlying fixed-point method has gone through some number of initial iterations. For example, this may be the case if the underlying method has strong global convergence properties and the initial iterations may help to bring the iterates closer to a solution before starting the acceleration. Such a delayed start is easily achieved by prescribing an iteration number at which acceleration is to begin.

The following is our final version of Anderson acceleration. In this, the additional features discussed above are incorporated into the algorithm, with new optional inputs as follows: $droptol$, a threshhold for deleting columns to maintain acceptable conditioning; $beta$, a damping parameter to allow damping with $\beta_k = beta$ for each k (this can be a function such that $\beta_k = beta(k)$); and $AAstart$, an iteration number at which to begin acceleration. The condition number of R is denoted by $cond(R)$.

Algorithm AA: ANDERSON ACCELERATION

Given x, $m_{Max} \geq 1$, and, optionally, droptol (default 1.e10), beta (default 1), and AAstart (default 0).
Initialize $m_{AA} = 0$ and $\mathcal{G} = []$.
For $k = 0, 1, \ldots,$ itmax
Evaluate $g_{cur} = g(x)$ and $f_{cur} = g_{cur} - x$.
If $k >$ AAstart
Set $\Delta f = f_{cur} - f_{old}$ and $\Delta g = g_{cur} - g_{old}$.
If $m_{AA} < m_{Max}$
Update $\mathcal{G} \leftarrow [\mathcal{G}, \Delta g]$.
Else
Update $\mathcal{G} \leftarrow [\mathcal{G}(:, 2 : m_{AA}), \Delta g]$.
End
Update $m_{AA} \leftarrow m_{AA} + 1$.
End
Update $f_{old} = f_{cur}$ and $g_{old} = g_{cur}$.
If $m_{AA} = 0$
Update $x = g_{cur}$.
Else
If $m_{AA} = 1$
Set $Q(:, 1) = \Delta f / \|\Delta f\|_2$ and $R(1, 1) = \|\Delta f\|_2$.
Else
If $m_{AA} > m_{Max}$
Call [Q, R] = qrdelete(Q, R, 1).
Update $m_{AA} \leftarrow m_{AA} - 1$.
End
For $i = 1 : m_{AA} - 1$
Set $R(i, m_{AA}) = Q(:, i)' * \Delta f$.
Update $\Delta f \leftarrow \Delta f - R(i, m_{AA}) * Q(:, i)$.
End
Set $Q(:, m_{AA}) = \Delta f / \|\Delta f\|_2$ and $R(m_{AA}, m_{AA}) = \|\Delta f\|_2$.
End
While cond(R) > droptol and mAA > 1

Call [Q, R] = qrdelete(Q, R, 1).
Update $m_{AA} \leftarrow m_{AA} - 1$.
End
Solve $R\gamma = Q' * f_{cur}$.
Update $x \leftarrow g_{cur} - \mathcal{G} * \gamma$.
If $beta > 0$ *and* $beta \neq 1$
Update $x \leftarrow x - (1-beta) * (f_{cur} - Q * R * \gamma)$.
End
End
End

Other termination criteria. In an actual implementation, there are very likely to be termination criteria in addition to whether the number of iterations has reached $itmax$. One is whether the problem has been sufficiently well solved. This is likely to be based on whether the fixed-point residual norm $\|f_{cur}\|$ is sufficiently small, where $\|\cdot\|$ is an appropriate norm, possibly a weighted norm that accounts for different scaling among the components of f. Such a test is likely to be placed in the algorithm just after the evaluation of g_{cur} and f_{cur}. Another possibility is a criterion based on whether the norm of some auxiliary function is sufficiently small. For example, this may be the case when the underlying fixed-point iteration is Picard iteration determined by some PDE problem, and it is appropriate to terminate on the PDE residual norm instead of (or in addition to) the fixed-point residual norm. A test of this type will probably be placed before the evaluation of g_{cur} and f_{cur}. Finally, there is likely to be a criterion based on whether the iterates are making insufficient progress. This will likely involve a test of whether the norm (possibly a weighted norm) of the difference between successive iterates is less than some prescibed tolerance.

References

[1] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.

[2] C. H. Bischof. Incremental condition estimation. *SIAM J. Matrix Anal. Appl.*, 11:312–322, 1990.

[3] H. Fang and Y. Saad. Two classes of multisecant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.

[4] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Computational Materials Sci.*, 6:15–50, 1996.

[5] P. Ni and H. F. Walker. A linearly constrained least-squares problem in electronic structure computations. Technical Report MS-1-13-46, Worcester Polytechnic Institute Mathematical Sciences Department, January 2010.

[6] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM J. Numer. Anal.*, 49:1715–1735, 2011.

Appendix. A MATLAB implementation.

The following is a MATLAB implementation of the final version of Anderson acceleration. Note that if the input mMax is zero, then the code performs fixed-point iteration without acceleration.

```
function [x,iter,res_hist] = AndAcc(g,x,mMax,itmax,atol,rtol,droptol,beta,AAstart)

% This performs fixed-point iteration with or without Anderson
% acceleration for a given fixed-point map g and initial
% approximate solution x.
%
% Required inputs:
%   g  = fixed-point map (function handle); form gval = g(x).
%   x  = initial approximate solution (column vector).
%
% Optional inputs:
%   mMax    = maximum number of stored residuals (non-negative integer).
%             NOTE: mMax = 0 => no acceleration.
%   itmax   = maximum allowable number of iterations.
%   atol    = absolute error tolerance.
%   rtol    = relative error tolerance.
%   droptol = tolerance for dropping stored residual vectors to improve
%             conditioning: If droptol > 0, drop residuals if the
%             condition number exceeds droptol;  if droptol <= 0,
%             do not drop residuals.
%   beta    = damping factor: If beta > 0 (and beta ~= 1), then the step is
%             damped by beta; otherwise, the step is not damped.
%             NOTE: beta can be a function handle; form beta(iter), where iter is
%             the iteration number and 0 < beta(iter) <= 1.
%   AAstart = acceleration delay factor: If AAstart > 0, start acceleration
%             when iter = AAstart.
%
% Output:
%   x        = final approximate solution.
%   iter     = final iteration number.
%   res_hist = residual history matrix (iteration numbers and residual norms).
%
% Homer Walker (walker@wpi.edu), 10/14/2011.
```

```
% Set the method parameters.
if nargin < 2, error('AndAcc requires at least two arguments.'); end
if nargin < 3, mMax = min{10, size(x,1)}; end
if nargin < 4, itmax = 100; end
if nargin < 5, atol = 1.e-10; end
if nargin < 6, rtol = 1.e-10; end
if nargin < 7, droptol = 1.e10; end
if nargin < 8, beta = 1; end
if nargin < 9, AAstart = 0; end

% Initialize the storage arrays.
res_hist = [];   % Storage of residual history.
DG = [];         % Storage of g-value differences.

% Initialize printing.
if mMax == 0
  fprintf('\n No acceleration.');
elseif mMax > 0
  fprintf('\n Anderson acceleration, mMax = %d \n',mMax);
else
  error('AndAcc.m: mMax must be non-negative.');
end
fprintf('\n iter    res_norm  \n');

% Initialize the number of stored residuals.
mAA = 0;

% Top of the iteration loop.
for iter = 0:itmax

  % Apply g and compute the current residual norm.
  gval = g(x);
  fval = gval - x;
  res_norm = norm(fval);
  fprintf('  %d    %e  \n', iter, res_norm);
  res_hist = [res_hist;[iter,res_norm]];

  % Set the residual tolerance on the initial iteration.
  if iter == 0, tol = max(atol,rtol*res_norm); end
```

```
% Test for stopping.
if res_norm <= tol,
  fprintf('Terminate with residual norm = %e \n\n', res_norm);
  break;
end

if mMax == 0 || iter < AAstart,
  % Without acceleration, update x <- g(x) to obtain the next
  % approximate solution.
  x = gval;
else
  % Apply Anderson acceleration.

  % Update the df vector and the DG array.
  if iter > AAstart,
    df = fval-f_old;
    if mAA < mMax,
      DG = [DG gval-g_old];
    else
      DG = [DG(:,2:mAA) gval-g_old];
    end
    mAA = mAA + 1;
  end
  f_old = fval;
  g_old = gval;

  if mAA == 0
    % If mAA == 0, update x <- g(x) to obtain the next approximate solution.
    x = gval;
  else
    % If mAA > 0, solve the least-squares problem and update the
    % solution.
    if mAA == 1
      % If mAA == 1, form the initial QR decomposition.
      R(1,1) = norm(df);
      Q = R(1,1)\df;
    else
      % If mAA > 1, update the QR decomposition.
```

```
  if mAA > mMax
    % If the column dimension of Q is mMax, delete the first column and
    % update the decomposition.
    [Q,R] = qrdelete(Q,R,1);
    mAA = mAA - 1;
    % The following treats the qrdelete quirk described below.
    if size(R,1) ~= size(R,2),
      Q = Q(:,1:mAA-1); R = R(1:mAA-1,:);
    end
    % Explanation: If Q is not square, then qrdelete(Q,R,1) reduces the
    % column dimension of Q by 1 and the column and row
    % dimensions of R by 1. But if Q *is* square, then the
    % column dimension of Q is not reduced and only the column
    % dimension of R is reduced by one. This is to allow for
    % MATLAB's default "thick" QR decomposition, which always
    % produces a square Q.
  end
  % Now update the QR decomposition to incorporate the new
  % column.
  for j = 1:mAA - 1
    R(j,mAA) = Q(:,j)'*df;
    df = df - R(j,mAA)*Q(:,j);
  end
  R(mAA,mAA) = norm(df);
  Q = [Q,R(mAA,mAA)\df];
end
if droptol > 0
  % Drop residuals to improve conditioning if necessary.
  condDF = cond(R);
  while condDF > droptol && mAA > 1
    fprintf('    cond(D) = %e, reducing mAA to %d \n', condDF, mAA-1);
    [Q,R] = qrdelete(Q,R,1);
    DG = DG(:,2:mAA);
    mAA = mAA - 1;
    % The following treats the qrdelete quirk described above.
    if size(R,1) ~= size(R,2),
      Q = Q(:,1:mAA); R = R(1:mAA,:);
    end
    condDF = cond(R);
```

```
        end
      end
      % Solve the least-squares problem.
      gamma = R\(Q'*fval);
      % Update the approximate solution.
      x = gval - DG*gamma;
      % Apply damping if beta is a function handle or if beta > 0
      % (and beta ~= 1).
      if isa(beta,'function_handle'),
        x = x - (1-beta(iter))*(fval - Q*R*gamma);
      else
        if beta > 0 && beta ~= 1,
          x = x - (1-beta)*(fval - Q*R*gamma);
        end
      end
    end
  end
end
% Bottom of the iteration loop.

if res_norm > tol && iter == itmax,
  fprintf('\n Terminate after itmax = %d iterations. \n', itmax);
  fprintf(' Residual norm = %e \n\n', res_norm);
end
```