`fpUpdate`: A Fixed Point and Equation Solver in MATLAB[*]

Alex Clymo
PSE

This repository provides a flexible and modular toolkit for solving fixed point problems of the form

$$x = f(x),$$

in MATLAB, where x is a column vector of length N. The methods are iterative, using the current guess x_k and evaluation $f(x_k)$ to build the new guess x_{k+1}, allowing the code to be implemented in a simple loop. The code implements adaptive dampening, and certain methods automatically store a history of past guesses and evaluations in order to accelerate convergence by, for example, approximating the Jacobian.

It can also be applied to solving nonlinear equations of the form $g(x) = 0$ by simply adding x to both sides, and therefore defining $f(x) = x + g(x)$

1 Overview

Solving fixed point problems is a core task in many quantitative macroeconomics applications. This toolkit is especially useful in the following scenarios:

- Embedding fixed point iterations within a broader calibration or simulation framework:
  - Wrapping a calibration loop around the solution of the steady state of your model.
  - Solving for the equilibrium price sequence following an MIT shock.
- Whenever putting your model into a MATLAB function to use `fsolve` is inconvenient.
- Testing and comparing update methods like damped iteration, Anderson acceleration, or Jacobian based methods.

The aim is practical usability: if you currently update some vector x in a loop with dampening, this can be slow. This toolkit allows you to replace that update with something more sophisticated with minimal effort.

[*]Contact: alex.clymo@psemail.eu.

2 File Descriptions

fpUpdate.m	Core function for updating the guess x using various methods. Manages internal histories needed for acceleration and Jacobian methods.
fpSetup.m	Helper function to initialize the par structure with default parameters for the chosen method.
example1_basics.m	Script demonstrating how to use the fixed point solver with example functions, comparing performance of methods and validating against MATLAB's fsolve.

3 Typical usage

The core usage updates a guess x to a new guess x_new via:

```
[x_new, par] = fpUpdate(x, fx, par);
```

where fx is the value of $f(x)$ evaluated at x (i.e. fx = f(x)). The structure par defines the update method, stores options, and automatically maintains any past data needed by the method.

To initialize par, for example with Broyden's method, call the fpSetup function at the top of your code:

```
par = fpSetup('broyden');
```

Algorithm parameters in par can then be modified manually as desired.

The typical usage then nests fpUpdate in a standard iteration loop:

```
% initial guess
x0 = ...
x = x0;

diff = 1;
tol = 1e-5;
while diff > tol

    % evaluate f(x) at current guess
    fx = ...

    % perform one update
    [x_new,par] = fpUpdate(x,fx,par);

    % compute percentage error
    diff = max( abs( (y-x)./(1e-3+abs(x)) ) );

    % update
    if diff > tol
        x = x_new;
    end

end
```

4 Methods Currently Supported

All methods optionally implement adaptive dampening, where the parameter `zeta` is lowered (raised) if the error is rising (falling).

Set the method by setting `par.method =`...

1. `fixedPoint`: Basic fixed point update with dampening. Allows the dampening parameter ζ to be a vector with different dampening for each x element.

$$x_{k+1} = \zeta \odot f(x_k) + (1 - \zeta) \odot x_k$$

   A simple update `x_new = zeta .* fx + (1 - zeta) .* x`.

2. `anderson`: Anderson Acceleration. A fixed point method which uses the history of past guesses and residuals to improve convergence. Solves a least squares problem each iteration to choose parameters α_k and updates

$$x_{k+1} = \sum_{i=0}^{m} (\alpha_k)_i f_{k-m+i}.$$

   Thus, x is updated using an adaptive weighted average of recent function evaluations. The user selects how many past evaluations to store and use. This has a smaller memory requirement than Jacobian-based methods, since only a list of evaluations are stored, not an $N \times N$ Jacobian matrix, but this still improves speed.

3. `broyden`: Broyden's Method. A quasi-Newton method that builds and stores an approximation to the inverse Jacobian of $f(x)$. Higher memory cost than Anderson, but often faster convergence.

4. `jacob_diag`: Diagonal Jacobian approximation. A quasi-Newton method that assumes the Jacobian of $f(x)$ is a diagonal matrix, i.e. each element x_i only affects the function evaluation $f(x)_i$. The Jacobian is trivially estimated each step using a finite difference approximation from the last function evaluation. Very low memory cost, but only works for problems where the Jacobian is diagonal or approximately so.

5 A note on solving $x = f(x)$ versus $g(x) = 0$

The code is written to solve $x = f(x)$, since this allows it to use specialised fixed point methods for any problems which converge robustly for contraction mappings. However, the code can trivially also handle general equation solving problems of the form $g(x) = 0$, since this implies that $x = g(x) + x$, or $x = g(x) - x$. Hence one can simply define $f(x) = g(x) + x$ and use the code as is. In practice, this just means remembering to modify the input `fx` in the function call `[x_new,par] = fpUpdate(x,fx,par)` to replace it with `fx=gx+x`.

For problems of the form $g(x) = 0$, the best option is to go straight for one of the Jacobian based solvers (Broyden or diagonal) since a problem $g(x) = 0$ is unlikely to be a contraction mapping. However, if you set it up carefully you might also be able use the fixed point methods. For example, the basic fixed point dampened update takes the form $x_{k+1} = \zeta f(x_k) + (1 - \zeta)x_k$. If you define $f(x) = g(x) + x$ then this becomes $x_{k+1} = \zeta g(x_k) + x_k$. We see if $\zeta > 0$ this tells the code to raise the x guess whenever $g(x) > 0$, and lower it otherwise. If $\zeta < 0$ then the code instead lowers x whenever $g(x) > 0$. If you can use economic logic or trial and error to establish which update direction is correct, you can make the fixed point option work with an appropriate sign for ζ.

6 Global settings and adaptive dampening

These settings apply to all methods:

- `par.method`: selects the method used. Current options are the list from the previous section.
- `par.xmin` and `par.xmax`: vectors setting min and max values of the updated x guesses. These are imposed using `x_new = max(min(x_new,par.xmax),par.xmin)` after the update. Note that `fpUpdate` is not a constrained solver: *these bounds must not bind in the true solution*. The bounds are just used to avoid crazy or illegal updates (e.g. negative values when you know that's impossible) during convergence.
- `par.zeta`: The main dampening parameter. If not using adaptive dampening, this value is set and not adjusted. In this case, you can allow `par.zeta` to be a vector, with different values for each element of x. If you are using adaptive dampening, `par.zeta` should be a scalar, and the initially chosen value of `par.zeta` will be updated automatically by the code during the iterations.

These are the settings related to adaptive dampening:

- `par.adaptiveDampening`: turns adaptive dampening `'on'` or `'off'`. If `'off'`, the main dampening parameter `par.zeta` remains fixed throughout the iterations. If `'on'`, `par.zeta` will be automatically adjusted based on whether the current iteration is improving convergence or not, based on comparing the current root mean squared error to the previous evaluation.
- `par.adSettings.shrinkFactor`: factor by which `par.zeta` is multiplied when the error increases and the update is deemed too aggressive. Must be ≤ 1. For example, with `shrinkFactor = 0.5`, `par.zeta` is halved in such cases.
- `par.adSettings.growFactor`: factor by which `par.zeta` is multiplied when the update is deemed too cautious (e.g. the error decreases satisfactorily). Must be ≥ 1, and in practice is often chosen $< 1/\text{shrinkFactor}$ to avoid oscillations. The default definition

  ```
  1 + 0.8*(1./shrinkFactor - 1)
  ```

  ensures the growth factor is slight below the shrink factor.
- `par.adSettings.zetaMin`: lower bound on `par.zeta` during adaptation, ensuring dampening never becomes too small to make progress.
- `par.adSettings.zetaMax`: upper bound on `par.zeta` during adaptation, ensuring updates never become too large and unstable.

The code stores some basic data in `par.iterData`:

- `par.iterData.iter`: counter for the current iteration number. Initialised to 0 before the first update and incremented automatically each iteration.

- `par.iterData.rmseList`: vector containing the root mean squared error (RMSE) at each iteration. This is used by the adaptive dampening method to decide if dampening should be increased or decreased.
- `par.iterData.zetaList`: vector recording the value of the dampening parameter `par.zeta` used at each iteration.

7 Details of fixed point methods

These methods iterate on the fixed point. They only work well for contraction mappings where such an iteration is likely to converge to the true solution. For problems which are not contraction mappings (even if they can be written as $x = f(x)$) then Jacobian methods should be used.

7.1 `fixedPoint`: Basic fixed point update

Basic fixed point update with dampening. Allows the dampening parameter ζ to be a vector with different dampening for each x element.

$$x_{k+1} = \zeta \odot f(x_k) + (1 - \zeta) \odot x_k$$

A simple update `x_new = zeta .* fx + (1 - zeta) .* x`. No data needs to be stored to use this method, which just uses the current `x` and `fx` to make the new `x_new`. There are no method options apart from the dampening parameter and optional adaptive dampening.

7.2 `anderson`: Anderson Acceleration

Anderson Acceleration is a technique for accelerating fixed-point iterations $x_{k+1} = f(x_k)$ by using a linear combination of multiple past iterates. It is particularly effective when a simple damped fixed-point update converges slowly due to oscillations or poor contraction properties.

Let N denote the dimension of x, and let $m = \texttt{par.Ma} - 1$ be the number of past iterations used beyond the current one. At iteration k, we maintain:

- $\mathbf{x}_i \in \mathbb{R}^N$: the i-th past iterate.
- $\mathbf{f}_i = f(\mathbf{x}_i)$: function evaluations at those iterates.
- $\mathbf{r}_i = \mathbf{f}_i - \mathbf{x}_i$: the residual vectors.

The residual history matrix is

$$R_k = \begin{bmatrix} \mathbf{r}_{k-m} & \mathbf{r}_{k-m+1} & \dots & \mathbf{r}_k \end{bmatrix} \in \mathbb{R}^{N \times (m+1)}.$$

7.2.1 Update Formula

The Anderson update solves the following constrained least squares problem:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^{m+1}} \|R_k \boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \mathbf{1}^\top \boldsymbol{\alpha} = 1. \tag{1}$$

Without dampening, the new iterate would then given by

$$\mathbf{x}_{k+1} = \sum_{i=0}^{m} \alpha_i \mathbf{f}_{k-m+i}. \tag{2}$$

We add dampening, and instead compute the update as

$$\mathbf{x}_{k+1} = \zeta \sum_{i=0}^{m} \alpha_i \mathbf{f}_{k-m+i} + (1-\zeta)\mathbf{x}_k \tag{3}$$

where $\zeta =$ `par.zeta` is a scalar or element-wise vector in $(0, 1]$.

7.2.2 Tikhonov Regularisation

When the columns of R_k are nearly linearly dependent, problem (1) becomes ill-conditioned. This occurs when the iterates are close to the fixed point, as the residuals become very similar. We stabilise the problem by solving the *regularised Anderson problem*:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^{m+1}} \|R_k \boldsymbol{\alpha}\|_2^2 + \lambda^2 \|\boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \mathbf{1}^\top \boldsymbol{\alpha} = 1, \tag{4}$$

where $\lambda \geq 0$ is the regularisation parameter.

Interpretation. When $\lambda = 0$, (4) reduces to the standard Anderson problem. As $\lambda \to \infty$, the solution approaches equal weights $\alpha_i = 1/(m+1)$.

7.2.3 Choosing λ from a Condition Number Target

Let $\sigma_1 \geq \cdots \geq \sigma_{m+1} > 0$ be the singular values of R_k. The condition number of the regularised normal equations matrix $R_k^\top R_k + \lambda^2 I$ is

$$\kappa_\lambda = \frac{\sigma_1^2 + \lambda^2}{\sigma_{m+1}^2 + \lambda^2}.$$

We impose a user-specified maximum condition number $\kappa_{\max} =$ `par.maxCondR`. If the unregularised problem has $\kappa(R_k) \leq \kappa_{\max}$, we set $\lambda = 0$. Otherwise, we choose λ to achieve $\kappa_\lambda = \kappa_{\max}^2$:

$$\lambda^2 = \frac{\sigma_1^2 - \kappa_{\max}^2 \sigma_{m+1}^2}{\kappa_{\max}^2 - 1}. \tag{5}$$

The squared condition number bound $\kappa_{\max}^2$ is applied to $R^\top R$ since $\text{cond}(R^\top R) = \text{cond}(R)^2$. The least-squares problem (4) is solved via `lsqlin` in MATLAB with the constraint $\mathbf{1}^\top \alpha = 1$.

7.2.4 Summary of Parameters

- `par.Ma`: memory length $(m+1)$.
- `par.zeta0`: dampening factor used in the initial few iterations when $M <$ `Ma` (i.e. insufficient history).
- `par.maxCondR`: maximum allowed condition number of R_k before regularisation.

7.2.5 MATLAB Implementation Notes

The algorithm maintains and updates two history matrices:

$$\begin{aligned} &\texttt{par.x\_hist} \in \mathbb{R}^{N\times h}, \quad \text{past } x \text{ guesses,} \\ &\texttt{par.f\_hist} \in \mathbb{R}^{N\times h}, \quad \text{past } f(x) \text{ evaluations.} \end{aligned}$$

At each iteration:

1. Append the current (x, f) to the history.
2. If history length $M <$ `Ma`, perform damped fixed-point update with ζ_0.
3. Otherwise:
   (a) Form R_k from last $m+1$ residuals.
   (b) Compute SVD to determine condition number.
   (c) Choose λ via (5) if regularisation needed.
   (d) Solve (4) via `lsqlin`.
   (e) Compute $\mathbf{x}_{k+1}$ via (2) and apply dampening.

8 Details of Jacobian-Based Methods

Many algorithms for solving nonlinear equations and fixed-point problems rely on Jacobian information to accelerate convergence. These methods approximate Newton's method, but avoid computing or inverting the full Jacobian directly. This section presents two such approaches: Broyden's method and a rank-1 Steffensen-inspired method.

8.1 General Framework

We consider fixed-point problems of the form

$$x = f(x),$$

which may equivalently be written as a root-finding problem:

$$g(x) := f(x) - x = 0.$$

The Jacobian matrix of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^n$ is defined elementwise as:

$$[Df(x)]_{ij} := \frac{\partial f_i(x)}{\partial x_j}, \quad \text{for } i, j = 1, \ldots, n.$$

Letting $g(x) = f(x) - x$, the Jacobian of the residual function is:

$$Dg(x) = Df(x) - I.$$

A Newton-type update is then:

$$x_{k+1} = x_k - [Dg(x_k)]^{-1} g(x_k).$$

Quasi-Newton methods avoid computing $Dg(x_k)$ explicitly and instead construct approximations. Let $A_k \approx Dg(x_k)$, then the general update becomes:

$$x_{k+1} = x_k - A_k^{-1} g(x_k) = x_k - A_k^{-1}(f(x_k) - x_k).$$

8.2 `broyden`: Broyden's Method

Broyden's method is a quasi-Newton algorithm that updates an approximation of the inverse Jacobian $H_k \approx [Dg(x_k)]^{-1}$ using only function evaluations and previous iterates.

Let:

$$
\begin{aligned}
s_k &= x_k - x_{k-1}, \\
y_k &= g(x_k) - g(x_{k-1}) = (f(x_k) - x_k) - (f(x_{k-1}) - x_{k-1}).
\end{aligned}
$$

The update seeks a matrix H_{k+1} satisfying the secant condition:

$$H_{k+1} y_k = s_k,$$

while remaining as close as possible to the previous approximation H_k. This is formalized as the solution to the optimization problem:

$$H_{k+1} = \arg\min_H \|H - H_k\|_F^2 \quad \text{subject to } H y_k = s_k.$$

This yields the rank-1 update:

$$H_{k+1} = H_k + \frac{(s_k - H_k y_k) y_k^\top}{y_k^\top y_k}.$$

At each step:

(i) Evaluate $f_k = f(x_k)$, $g_k = f_k - x_k$

(ii) Compute the Newton-type step: $x_{k+1} = x_k - \zeta H_k g_k$ where ζ is the dampening

(iii) Form $s_k = x_{k+1} - x_k$, $y_k = g(x_{k+1}) - g_k$

(iv) Update $H_k \to H_{k+1}$ via the rank-1 formula (skipped or reset if problem poorly conditioned)

The initial guess for the inverse Jacobian is $H_0 = \alpha I$, where I is the identity matrix. This method converges superlinearly under mild conditions and is well suited to small or medium-scale problems where storing and updating the full matrix $H_k \in \mathbb{R}^{n \times n}$ is tractable.

8.2.1 Summary of Parameters

- `par.H0scale`: α in initial H_0 guess. Smaller number generates smaller initial steps and less diagonal Jacobian. Can be positive or negative.
- `par.tolDenom`: If denominator $y_k^\top y_k$ less than this number, then Jacobian update is considered unstable and skipped, leaving Jacobian at previous guess.
- `par.maxCondH`: maximum allowed condition number of Jacobian. If goes above this number then Jacobian is considered ill conditioned and reset back to H_0.

8.3 `jacob_diag`: Diagonal Jacobian Approximation

A simple Jacobian-based approach assumes that the Jacobian of the residual function $g(x) = f(x) - x$ is diagonal. This corresponds to the assumption that each component $g_i(x)$ depends only on the corresponding variable x_i, and not on the others. This approximation can be especially useful in large-scale problems or when variables are nearly decoupled.

Let x_k and $f_k = f(x_k)$ denote the current iterate and its function evaluation, and let x_{k-1}, f_{k-1} be the previous ones. We define the component-wise secant approximation of the Jacobian diagonals as:

$$d_i^{(k)} = \frac{g_i^{(k)} - g_i^{(k-1)}}{x_i^{(k)} - x_i^{(k-1)}}$$

where $g_k = f_k - x_k$. These diagonal elements approximate $\partial g_i / \partial x_i$, and the Jacobian $Dg(x_k)$ is thus approximated by a diagonal matrix $D_k = \text{diag}(d_i^{(k)})$. Since the Jacobian is diagonal, its inverse is also diagonal with entries d_i^{-1}, and the Newton-like update becomes:

$$x_i^{(k+1)} = x_i^{(k)} - \zeta \frac{g_i^{(k)}}{\max(d_i^{(k)}, d_{\text{min}})}.$$

Where ζ is the dampening parameter. To ensure numerical stability, we additionally apply a minimum value of the derivative d_{min} which effectively imposes a maximum step size to avoid large steps if the derivative is badly estimated.

This method is very memory efficient, requiring only two stored vectors of size n. By assuming independence, we only need one step to estimate the diagonal Jacobian. It still provides a form of curvature correction that can accelerate convergence compared to standard fixed-point iteration. But, it only works for problems where variables are independent or only weakly coupled. If off-diagonal Jacobian terms are important, this method might fail.

8.3.1 Summary of Parameters

- `par.zeta0`: Fixed dampening for first iteration before you can compute the Jacobian. Method takes a simple fixed point step, and `par.zeta0` should be a small number.
- `par.dmin`: minimum derivative parameter d_{min}. Set to zero if don't want to use, but recommended for stability.

Appendix

A Methods in progress

These are some other methods I might potentially add. The derivations below need double checking, refining, implementing, and testing.

A.1 One-Shot Inverse Broyden Method

This idea is a variant of Broyden's "good" update applied to the inverse Jacobian, but with a crucial simplification: instead of maintaining and updating a stored inverse Jacobian matrix across iterations, we start each iteration from a fixed baseline guess for the inverse Jacobian and apply a single rank-1 Broyden update using the most recent secant information. This allows us to incorporate curvature information from the latest step direction, without the memory and computational cost of storing a full $N \times N$ matrix.

Setup. We wish to solve a fixed-point problem

$$x = f(x), \quad x \in \mathbb{R}^N,$$

which can be written equivalently as a nonlinear equation $g(x) = 0$, where

$$g(x) \equiv f(x) - x.$$

Let $J_k \equiv \nabla g(x_k)$ denote the Jacobian of g at iteration k, and $H_k \equiv J_k^{-1}$ its inverse.

Baseline approximation. We begin each iteration with a simple diagonal baseline guess for the inverse Jacobian:

$$H_k^{(0)} = \alpha I,$$

where $\alpha > 0$ is a scalar parameter chosen by the user. This choice corresponds to assuming the Jacobian is $\frac{1}{\alpha} I$ in the absence of other information.

Secant information. Let

$$s_k = x_k - x_{k-1}, \quad y_k = g(x_k) - g(x_{k-1}),$$

be the changes in the iterate and residual between the last two iterations. The secant equation for the inverse Jacobian reads:

$$H_k y_k = s_k.$$

The Broyden "good" update finds a rank-1 correction to $H_k^{(0)}$ that satisfies this secant equation while remaining as close as possible (in the Frobenius norm) to the baseline $H_k^{(0)}$.

Rank-1 update. The solution to this constrained minimization problem is:

$$H_k = H_k^{(0)} + \frac{(s_k - H_k^{(0)} y_k) y_k^\top}{y_k^\top y_k}.$$

In our case $H_k^{(0)} = \alpha I$, so:

$$H_k = \alpha I + \frac{(s_k - \alpha y_k)y_k^\top}{y_k^\top y_k}.$$

Iteration formula. Given the current iterate x_k and function evaluation $F_k = f(x_k)$, we have $g(x_k) = F_k - x_k$. The update step is:

$$p_k = -H_k g(x_k) = -\alpha g(x_k) - (s_k - \alpha y_k)\frac{y_k^\top g(x_k)}{y_k^\top y_k}.$$

We then form the next iterate with an optional scalar damping factor $\zeta \in (0, 1]$:

$$x_{k+1} = x_k + \zeta\, p_k.$$

Interpretation. This method can be seen as a *memoryless quasi-Newton* approach: it discards all but the latest secant information, and each iteration starts from a scaled identity guess for H. It is particularly useful when N is large and storing/updating a full inverse Jacobian is infeasible, but one still wishes to capture curvature along the most recent step direction.

A.2 Rank-1 Fixed Point Update (Steffensen-Inspired)

This method is a simplified and memory-efficient approach to approximate Newton-type steps in fixed-point problems of the form $x = f(x)$. It is inspired by Steffensen's method and designed for large-scale systems where storing or updating full Jacobians is impractical.

Notation: Define the residual function:

$$g(x) := f(x) - x,$$

and let:

$$\begin{aligned} x_k &\quad \text{(current iterate)} \\ x_{k-1} &\quad \text{(previous iterate)} \\ f_k := f(x_k), \quad f_{k-1} := f(x_{k-1}) & \\ g_k := f_k - x_k, \quad g_{k-1} := f_{k-1} - x_{k-1} & \end{aligned}$$

Define the consistent secant pair:

$$\Delta x := x_k - x_{k-1}, \qquad \Delta g := g_k - g_{k-1} = (f_k - x_k) - (f_{k-1} - x_{k-1}) = \Delta f - \Delta x.$$

Goal: Approximate the residual Jacobian $Dg(x)$ by a rank-1 matrix:

$$A_k = I + u\Delta x^\top,$$

such that:

$$A_k \Delta x \approx \Delta g.$$

Optimization Problem: We determine $u \in \mathbb{R}^n$ as the solution to the minimization:

$$\min_u \|A_k \Delta x - \Delta g\|^2 \quad \text{subject to } A_k = I + u\Delta x^\top.$$

This yields the explicit solution:

$$u = \frac{\Delta g - \Delta x}{\|\Delta x\|^2},$$

so the Jacobian approximation becomes:

$$A_k = I + \frac{(\Delta g - \Delta x)}{\|\Delta x\|^2}\Delta x^\top.$$

Update Step: We avoid storing or inverting large matrices by using the Sherman–Morrison formula:

$$A_k^{-1} = I - \frac{u\Delta x^\top}{1 + \Delta x^\top u}.$$

With this, the update to the new iterate is:

$$x_{k+1} = x_k - A_k^{-1} g_k = x_k - g_k + \frac{u(\Delta x^\top g_k)}{1 + \Delta x^\top u}.$$

Remarks:

- Unlike Broyden's method, this method builds a rank-1 approximation to the Jacobian using only vector operations.
- The secant pair $(\Delta x, \Delta f)$ comes from actual consecutive iterates and their function evaluations, making the approximation consistent.
- It is especially useful in large systems, such as dynamic macroeconomic models, where storing and manipulating $n \times n$ Jacobians is infeasible.
- But looks like it might rely heavily on the identity matrix, hence is not suited for problems where the off diagonal Jacobian terms dominate.

Intuition behind the rank-1 Jacobian approximation. The matrix $A_k = I + u\Delta x^\top$ is constructed to serve as a low-memory approximation of the Jacobian of the residual function $g(x) = f(x) - x$. The idea is to ensure that A_k reproduces the observed change in the residual function along the most recent update direction. Specifically, we want:

$$A_k \Delta x \approx \Delta g,$$

where $\Delta x = x_k - x_{k-1}$ and $\Delta g = g_k - g_{k-1}$. This ensures that the approximation is locally consistent with the behavior of the function in the direction we actually stepped. In other words, we enforce secant consistency in one direction, while assuming that the Jacobian is the identity in all orthogonal directions.

This leads to a computationally efficient strategy: the matrix A_k is only a rank-1 modification of the identity, and its inverse can be computed using the Sherman–Morrison formula.

This allows us to apply a Newton-like correction step at each iteration without storing or inverting an $n \times n$ matrix.

In summary, the method builds a Jacobian approximation that is *exactly correct* along the last step direction and uses the identity elsewhere. This strikes a balance between curvature information and minimal memory usage, making it suitable for large-scale fixed point problems.

A.3 Limited memory Broyden (van de Rotten and Verduyn Lunel, 2003)

See their method here. Adapts Broyden to only store a list of past function evaluations rather than the whole Jacobian.