This is a set of notes from in-class discussion (APPM 4600) on methods for non-linear systems of equations. We will be assuming that we have a set of n equations in n real variables, which can be written as $F(x) = 0$ for $x \in \mathbb{R}^n$. As is the case for one non-linear equation, we will make smoothness assumptions on this $F(x)$ sufficient for the method to converge for an initial guess x_0 near a root r. For each method, we will center our discussion to answer two key questions:

1. Under what assumptions on $F(x)$ will the method converge for x_0 near the root?
2. What is the cost per iteration (as a function of n), and how quickly does the method converge?

Recall that, for general iterative methods, answering this second question gives us an estimate of the overall cost (and so the time it will take) to solve $F(x) = 0$.

1 Fixed Point Iteration

We write down a direct generalization of the Fixed Point Iteration, which we have previously discussed for scalar rootfinding problems. The idea remains the same: given $F(x) = 0$, we come up with $G : \mathbb{R}^n \to \mathbb{R}^n$ such that r is a root of $F(x) = 0$ if and only if $G(r) = r$. This is a very general idea. We can at least consider methods where G is of the form:

$$G(x) = x + \mathbf{S}(x)F(x) \tag{1.1}$$

where matrix $\mathbf{S}(x)$ is invertible (non-singular) on a neighborhood $B_\varepsilon(r)$ of the root. This of course includes the cases where $\mathbf{S}$ does not depend on x, and where it is multiplication by a scalar ($\mathbf{S} = s\mathbf{I}$).

1.1 Error analysis and convergence

In order to analyze the convergence of FPI, we need to extend the ideas we discussed for one non-linear equation and for linear FPI. In essence, what we can show is that if $G(x)$ is *contractive* in a neighborhood $B_\varepsilon(r)$ of the root, then FPI will converge at least linearly using an argument identical to the one we used for the scalar case.

Theorem 1.1 *Let r fixed point of $G(x)$, and there exists $0 < L < 1$ such that $\forall x, y \in B_\varepsilon(r)$,*

$$||G(x) - G(y)|| \leq L||x - y|| \tag{1.2}$$

for some norm $||\cdot|| : \mathbb{R}^n \to \mathbb{R}$. Then, FPI will converge at least linearly to r for any $x_0 \in B_\varepsilon(r)$.

The proof for this statement can be copied from the scalar case: given the sequence of iterates $x_{k+1} = G(x_k)$,

$$||e_{k+1}|| = ||x_{k+1} - r|| = ||G(x_k) - G(r)|| \leq L||x_k - r|| \leq \cdots \leq L^{k+1}||e_0|| \tag{1.3}$$

And so, as we take the limit as $k \to \infty$, the error will go to zero. This also tells us that:

$$\limsup_{k\to\infty} \frac{||e_{k+1}||}{||e_k||} \leq L \tag{1.4}$$

So, convergence is linear (or better), with rate *at most* L.

So, what is left in this analysis is to ask the question: when is the map $G(x)$ contractive? Are there conditions we can check at the root or on a neighborhood of the root? If $G(x)$ is only continuous, this is generally hard to check. However, when G is at least one time continuously differentiable ($C^1(B_\varepsilon(r))$), we can produce a number of results of the form: *if* $||J_G(x)|| < 1$ *for the appropriate operator norm on a neighborhood of* r*, then* G *is contractive in that neighborhood and the conclusion of Theorem 1.1 follows.*

This is also the case if the spectral radius, that is, the largest eigenvalue in absolute value, is strictly smaller than 1. So, any result that tells us $|\lambda_i| < 1$ for all eigenvalues of $J_G(x)$ in a neighborhood of r, or that $||J_G(x)||_p < 1$ for some matrix operator norm $||\cdot||_p$ is the equivalent of asking $|g'(x)| < 1$ for the scalar case.

2 Newton method

We once again extend the idea from the scalar Newton method: to use the linearization of our function $F(x)$ around our current guess x_k, and set it equal to 0. The linearization now involves the Jacobian of $F(x)$, of course:

$$L(x_{k+1}) = F(x_k) + J_F(x_k)(x_{k+1} - x_k) = 0$$
$$J_F(x_k)(x_{k+1} - x_k) = -F(x_k)$$
$$x_{k+1} = x_k - J_F(x_k)^{-1}F(x_k)$$

In other words: the Newton step is $x_{k+1} = x_k + p_k$, where the step p_k is the solution to a system of linear equations $J_F(x_k)p_k = -F(x_k)$. Unless we know something special about our Jacobian, this means we will have to solve it using Gauss Elimination. This implies $O(n^3)$ cost per iteration.

As is the case for scalar Newton, given smooth $F(x)$ (we usually assume $F \in C^2(B_\delta(r))$ and $J_F(x)$ invertible in that neighborhood), we can show that there exists a neighborhood $B_\varepsilon(r)$ of r (with ε potentially smaller than δ) such that for x_0 in that neighborhood, Newton converges *quadratically.* This has the same issue as scalar Newton: we don't a priori know what this neighborhood is.

We state the result for convergence of the Newton method without proof, but note that versions of the proofs for the scalar case can be extended to achieve it.

Theorem 2.1 *Let $F(x)$ be such that it is twice continuously differentiable and $J_F(x)$ is invertible on a neighborhood $B_\delta(r)$ of the root. Then, there exists $\varepsilon > 0$ such that for all $x_0 \in B_\varepsilon(r)$, Newton converges to r* quadratically*.*

where $B_\delta(r) = \{x \mid ||x - r||_2 < \delta\}$.

Note that this is exactly the same result as in the scalar case, except that it asks that the Jacobian be invertible on a neighborhood of the root. One thing you can experiment with is what happens if the Jacobian is invertible except at the root. Does Newton retain quadratic convergence?

2.1 The problems with Newton

Now, Newton is a fantastic method for rootfinding due to its quadratic convergence. However, there are a number of well-known issues with it, and these are greatly accentuated in systems of equations, especially for large n:

- Quadratic convergence is only guaranteed for a neighborhood of r. It may take the iteration a number of steps to get into this basin of quadratic convergence, *if it converges at all.* If we do NOT have confidence in our application that x_0 is close to a solution, it is dangerous to use Newton alone. We must use some other method to *get close* to r, or we must guide the Newton iteration. A family of methods to look into to guide Newton are *Line-search algorithms.* We can also, of course, use *hybrid* methods.
- Evaluating the $n \times n$ Jacobian matrix accurately may be, for some applications, really expensive and impractical. This is usually the case when we don't have a closed formula for $\mathbf{F}(\mathbf{x})$ (e.g. it is the state of some physical system).
- Even if we have $\mathbf{J}_F(x_k)$ readily available, we must solve a linear system for it. This is generally expensive. Unless our Jacobian is really special, we usually use Gaussian Elimination, which is $O(n^3)$.

3 Quasi-Newton methods

Quasi-Newton methods arise from the goal to find Newton-like methods that do not suffer from some of the issues Newton does, namely: the need to evaluate the Jacobian once per timestep, and the high cost per iteration of the corresponding linear system solve. We want a method that:

1. Is **superlinearly convergent** for x_0 close to r. This ensures small number of iterations once we are in the basin of superlinear convergence.
2. Does *not* evaluate a Jacobian or incur in $O(n^3)$ cost per timestep. We allow *at most* one Jacobian evaluation and one $O(n^3)$ cost (e.g. LU factorization) at the *beginning* of our algorithm.

3.1 Lazy and Approximate Newton methods

The first thing we tried is to fix the initial Jacobian and commit to it for the entirety of the iteration. This is informally known as "Lazy Newton" (more formally known as the *chord iteration*). The step then becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}_F(\mathbf{x}_0)^{-1} F(x_k) \tag{3.1}$$

The advantage we gain is: given an LU or another such factorization of $J_F(x_0)$, we can calculate this step in at most $O(n^2)$ cost (2 triangular solves). However, because we have given up on this matrix changing, we can show that this is a Fixed Point Iteration where the matrix S does not change. In practice, this means convergence of the chord iteration is *linear, at best.*

Other ideas

- **Shaminskii method:** We can generalize the chord iteration by updating the Jacobian *every m iterations* instead of committing to the initial Jacobian. This improves convergence (potentially making it superlinear), but the cost per timestep goes up, so it might not be worth it.
- **Inaccurate or approximate Newton:** We can use an innaccurate or sparsified version of the Jacobian to compute the Newton step. This might, in some instances, alleviate the cost of evaluating the Jacobian and somewhat reduce the cost of the Newton step.

For these methods, what we must show is that the innacurate Newton step $q_k \simeq -\tilde{\mathbf{J}}_F(x_k)^{-1}F(x_k)$ is close enough to the exact Newton step $p_k = -\mathbf{J}_F(x_k)^{-1}F(x_k)$. An exercise for the reader is to show that if $\|p_k - q_k\| \leq \eta\|F(x_k)\|$, then there exist $C, M > 0$ constants such that:

$$\|e_{k+1}\| \leq C\|e_k\|^2 + \eta M\|e_k\|$$

What this implies in practice is that the convergence of inexact Newton is quadratic *until the error is proportional to a multiple of* η. If we ask for more precision, it slows down to linear convergence (and would eventually plateau).

3.2 Quasi-Newton method: Broyden

The methods proposed above don't satisfy both of our asks, or they do only under certain conditions. Quasi-Newton methods like **Broyden** (there is a whole zoo of them, especially for optimization), on the other hand, are the real deal: they retain superlinear convergence while providing formulas that allow us to compute the "Quasi-Newton step" in at most $O(n^2)$ cost, if not faster. For this reason, they are widely used in modern rootfinding and smooth optimization routines and packages.

The ideas behind Quasi-Newton methods go back to the work pioneered by Broyden and his collaborators in the 60s (the method we discuss below was described in 1965). They are, in a sense, an extension of the secant method. Assume we have an initial guess $\mathbf{x}_0$, and an initial matrix $\mathbf{B}_0$ (usually $\mathbf{B}_0 \simeq J_F(x_0)$, but this isnt' required by the method). We then take one "Quasi-Newton step":

$$x_1 = x_0 - \mathbf{B}_0^{-1}F(x_0)$$

We now want to update the matrix from $\mathbf{B}_0$ to $\mathbf{B}_1$. What we want from $\mathbf{B}_1$ is:

1. To satisfy the **Secant equation** for x_1 and x_0. That is,

$$F(x_1) - F(x_0) = \mathbf{B}_1(x_1 - x_0) \tag{3.2}$$
$$\Delta F_0 = \mathbf{B}_1 \Delta x_0 \tag{3.3}$$

   where we denote $\Delta F_0 = F(x_1) - F(x_0)$, $\Delta x_0 = x_1 - x_0$. This should remind us of the slope of the secant line in one dimension.

2. **Rank 1 update:** $\mathbf{B}_1 = \mathbf{B}_0 + \mathbf{u}\mathbf{v}^T$. We can justify this by arguing that these matrices are imitating Jacobian matrices, which are continuous. $\mathbf{B}_1$ should thus be a minimal update of $\mathbf{B}_0$.

3. **Best rank 1 update:** Out of all possible rank 1 updates, $\mathbf{B}_1$ should be the closest to $\mathbf{B}_0$ in Frobenius norm. That is:

$$\mathbf{B}_1 = \text{argmin}\|\mathbf{B_1} - \mathbf{B_0}\|_F^2 = \text{argmin}\|\mathbf{u}\|_2^2\|\mathbf{v}\|_2^2 \tag{3.4}$$

   where the minimum is taken over $\mathbf{B}_1 = \mathbf{B}_0 + \mathbf{u}\mathbf{v}^T$.

3.3 Derivation of Broyden

We can use the 3 conditions above to derive the formula for Broyden. This is included in these notes for completion, but is beyond our syllabus. In practice, all that we need to implement is the formula obtained at the end of this subsection.

We substitute the rank 1 update formula into the secant equation. This gives us:

$$(\mathbf{B}_0 + \mathbf{u}\mathbf{v}^T)\Delta \mathbf{x}_0 = \Delta \mathbf{F}_0 \tag{3.5}$$

$$\mathbf{u}(\mathbf{v}^T \Delta x_0) = \Delta \mathbf{F}_0 - \mathbf{B}_0 \Delta \mathbf{x}_0 = \mathbf{r}_0 \tag{3.6}$$

$$\mathbf{u} = \frac{1}{(\mathbf{v}^T \Delta x_0)} \mathbf{r}_0 \tag{3.7}$$

where $\mathbf{r}_0 = \Delta \mathbf{F}_0 - \mathbf{B}_0 \Delta \mathbf{x}_0$ is a residual vector for the Secant Equation *applied to* $\mathbf{B}_0$. We note two things: if this residual is zero, this formula tells us to keep using $\mathbf{B}_0$. If it is non-zero, it gives us a formula for $\mathbf{u}$ as a function of $\mathbf{v}$.

The only condition we have not used is that the update should be minimal Frobenius norm:

$$\|\mathbf{B}_1 - \mathbf{B}_0\|_F^2 = \|\mathbf{u}\mathbf{v}^T\|_F^2 = \|\frac{\mathbf{r}_0}{(\mathbf{v}^T \Delta \mathbf{x}_0)} \mathbf{v}^T\|_F^2 \tag{3.8}$$

The Frobenius norm squared is the sum of all the matrix entries squared. For an exterior product $\mathbf{u}\mathbf{v}^T$ (rank 1 matrix), this is simply the product of the norms of each vector squared (exercise: show this). So, we must minimize:

$$\min_v \|\frac{\mathbf{r}_0}{(\mathbf{v}^T \Delta \mathbf{x}_0)} \mathbf{v}^T\|_F^2 = \min_v \|\frac{\mathbf{r}_0}{(\mathbf{v}^T \Delta \mathbf{x}_0)}\|_2^2 \|\mathbf{v}\|_2^2 \tag{3.9}$$

$$= \|\mathbf{r}_0\|_2^2 \min_v \frac{1}{|\mathbf{v}^T \Delta \mathbf{x}_0|^2} \|\mathbf{v}\|_2^2 \tag{3.10}$$

$$= \|\mathbf{r}_0\|_2^2 \min_v \frac{1}{\|\mathbf{v}\|_2^2 \|\Delta \mathbf{x}_0|\|_2^2 \cos^2\theta} \|\mathbf{v}\|_2^2 \tag{3.11}$$

$$= \frac{\|\mathbf{r}_0\|_2^2}{\|\Delta \mathbf{x}_0|\|_2^2} \min_v \frac{1}{\cos^2\theta} \tag{3.12}$$

Where θ is the angle between $\mathbf{v}$ and Δx_0. Clearly, this is minimized when this cosine is maximized, meaning $\theta = 0$, and $\mathbf{v} = \alpha \Delta x_0$ for some non-zero α. Putting everything together:

$$\mathbf{B}_1 = \mathbf{B}_0 + \frac{\mathbf{r}_0}{(\Delta \mathbf{x}_0)^T \Delta \mathbf{x}_0} \Delta \mathbf{x}_0^T \tag{3.13}$$

And so, the **Broyden update formula** reads:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{r}_k}{(\Delta \mathbf{x}_k)^T \Delta \mathbf{x}_k} \Delta \mathbf{x}_k^T \tag{3.14}$$

where $\Delta \mathbf{F}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$, $\Delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{r}_k = \Delta \mathbf{F}_k - \mathbf{B}_k \Delta \mathbf{x}_k$.

Using the Sherman-Morrison formula, we can produce a formula *for the inverse update*, which is what is actually implemented in Broyden methods:

$$\mathbf{B}_{k+1}^{-1} = \mathbf{B}_k^{-1} + \frac{\Delta \mathbf{x}_k - \mathbf{B}_k^{-1} \Delta \mathbf{F}_k}{(\Delta \mathbf{x}_k)^T \mathbf{B}_k^{-1} \Delta \mathbf{F}_k} ((\mathbf{B}_k^{-1})^T \Delta \mathbf{x}_k)^T \tag{3.15}$$

We note that the most expensive operation in this rank 1 update formula for the inverse involves computing $\mathbf{B}_k^{-1} \Delta \mathbf{F}_k$ and $(\mathbf{B}_k^{-1})^T \Delta \mathbf{x}_k$.

3.4 Implementation details for Broyden

We assume that we have a function that, given a vector v, computes $\mathbf{B}_0^{-1}$ (that is, solves the system $\mathbf{B}_0\mathbf{x} = \mathbf{b}$) in at most $O(n^2)$ work (e.g. we have computed an LU decomposition of $\mathbf{B}_0$). The pseudocode for Broyden will read as follows:

```
Given $\mathbf{x}_0$, $\mathbf{B}_0$ :
$k = 0$;
$np_k = 1$;
while $(np_k \geq \varepsilon$ and $k \leq k_{max})$ :
    $\mathbf{p}_k = -(\mathbf{B}_0^{-1} + \mathbf{U}\mathbf{V}^T)\mathbf{F}(\mathbf{x}_k)$
    $np_k = \|\mathbf{p}_k\|_\infty$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$

    $\mathbf{y}_k = (\mathbf{B}_0^{-1} + \mathbf{U}\mathbf{V}^T)\Delta\mathbf{F}_k$
    $\mathbf{z}_k = ((\mathbf{B}_0^{-1})^T + \mathbf{V}\mathbf{U}^T)\Delta\mathbf{x}_k$

    $\mathbf{U}(:, k+1) = \frac{\Delta\mathbf{x}_k - \mathbf{y}_k}{(\Delta\mathbf{x}_k)^T\mathbf{y}_k}$
    $\mathbf{V}(:, k+1) = \mathbf{z}_k$
    $k = k + 1$;
```

The key to implementing Broyden and other Quasi-Newton methods in practice, as can be seen from this pseudocode, is to write the rank 1 update to $\mathbf{B}_k^{-1}$ as a rank $k+1$ update to $\mathbf{B}_0^{-1}$. We store this as $\mathbf{U}\mathbf{V}^T$ for $n \times (k+1)$ matrices $\mathbf{U}, \mathbf{V}$

Key results

Broyden has the following properties:

- Broyden still is only guaranteed to converge superlinearly for $\mathbf{x}_0$ in a neighborhood of the root. This also depends on our choice of $\mathbf{B}_0$. Intuitively, the closer it is to $J_F(x_0)$, the more it will behave like Newton.
- Like Newton, this method needs to be guided to become more robust. We can use the same hybrid methods or *line-search algorithms* that work for Newton.
- Broyden's cost per iteration amounts to two solves for $\mathbf{B_0}$ plus $O(nk)$ work to apply the rank k update for the k-th iteration. Worst case scenario, this is $O(n^2)$. If we want to live dangerously, we can try $B_0 = sI$, which is $O(n)$.
- We don't have to compute the Jacobian during the iteration. This can be a major savings in computational cost.

4 Summary

Similar to our summary for linear solvers, we compile a table summarizing what we know about each of our methods to solve nonlinear systems of n equations in n variables $F(x) = 0$, with $J_F(x)$ the $n \times n$ Jacobian Matrix. Once again, we can ask:

- For what $F(x)$ and what initial values x_0 is this guaranteed to work?
- For iterative methods, what is the cost per iteration?
- For iterative methods, what do we know about convergence?

Let's recall what the step for each of the iterative methods looks like:

- **Fixed Point Iteration:** $x_{k+1} = G(x_k)$, where typically $G(x) = x - \mathbf{S}(x)F(x)$ for $\mathbf{S}(x)$ non-singular around the root.
- **Newton:** $x_{k+1} = x_k - \mathbf{J_F}^{-1}(x_k)F(x_k)$, where $\mathbf{J_F^{-1}}(x)$ non-singular around the root.
- **Lazy Newton:** $x_{k+1} = x_k - \mathbf{J_F}^{-1}(x_0)F(x_k)$, with similar assumptions as Newton.
- **Broyden:** $x_{k+1} = x_k - \mathbf{B_k}^{-1}F(x_k)$, with similar assumptions as Newton.

Method	Assumptions	Cost per Iteration	Convergence
Fixed Point	Contractive in $B_\varepsilon(r)$ x_0 near r	$F(x_k)$ eval + applying $\mathbf{S}$	Linear (at least)
Newton	$F \in C^2(B_\varepsilon(r))$ x_0 near r	$F(x_k), \mathbf{J}_F(x_k)$ eval Solve $\mathbf{J}_F(x_k)p_k = -F(x_k)$ $O(n^3)$	Quadratic!
Lazy Newton	Newton	Solve $\mathbf{J}_F(x_0)p_k = -F(x_k)$ $O(n^2)$	Linear
Approximate Newton	Newton + $\|q_k - p_k\|$ control	Approx solve $\mathbf{J}_F(x_k)p_k = -F(x_k)$ (complexity depends)	Quadratic until approx error
Broyden	Newton	Solve $\mathbf{B}_0\mathbf{p} = \mathbf{F}$ $O(n^2)$	Superlinear

5 Bonus: Newton and Quasi-Newton for smooth optimization

A huge area of application of non-linear systems of equations is smooth unconstrained optimization. That is, we are trying to solve a problem of the form $\min q(\mathbf{x})$ for $q : \mathbb{R}^n \to \mathbb{R}$, and by that, we mean to find at least a local minimizer of this function. For this to exist, we just need this function to be locally convex. In other words, we need to find a critical point $\mathbf{r}$:

$$\nabla q(\mathbf{r}) = 0 \tag{5.1}$$

and the Hessian matrix (matrix of second derivatives) should be Symmetric Positive Definite (SPD) in a neighborhood of the critical point, so that it is a local minima.

We may notice that the problem of finding a critical point is, by itself, typically a system of non-linear equations. We may think to use the methods discussed above, as well as a generalization of the steepest descent algorithm, to solve this problem. At first, the fact that we don't want *any critical point, but a particular kind* (a local max or a saddle point won't do) sounds like a complication, but it is in fact a boon, because it *gives us a way to guide the steepest descent, Newton and Quasi-Newton iterations.* This makes them more reliable methods.

As we discussed in class, relating systems of non-linear equations (rootfinding) with smooth optimization gives us a powerful relationship that can help us solve these two types of problems. If the problem we want to solve is of the form $F(\mathbf{x}) = 0$, we can use an optimization solver (e.g. gradient descent) to find the minimum of the function

$$q(\mathbf{x}) = \sum_{j=1}^{n} F_j(\mathbf{x})^2 = ||F(\mathbf{x})||_2^2$$

we derived a formula for the gradient of this function: $\nabla q(\mathbf{x}) = \mathbf{J_F}(\mathbf{x})^T F(\mathbf{x})$. So, much as is the case for fixed point, if we have a rootfinding problem, we can solve it using optimization solvers, and if we have an optimization problem, we can solve it using rootfinding methods.

5.1 Steepest (Gradient) descent method

From multivariate calculus we know that given a function $q(\mathbf{x})$, we can produce a contour plot and then, for a given input x_0, we can ask: what is the direction of steepest ascent? What is the direction of steepest descent?

In other words, we can calculate the directional derivative of moving away from $\mathbf{x}_0$ in the direction of $\mathbf{p}$. If $q(x)$ is differentiable, we find that this derivative is equal to $\nabla q(x_0)^T \mathbf{d}$. In other words, if we define $\phi(\alpha) = q(\mathbf{x}_0 + \alpha \mathbf{p})$, then $\phi'(0) = \nabla q(\mathbf{x}_0)^T \mathbf{d}$. We then conclude that the direction that leads to the steepest descent (most negative value) is $\mathbf{p} = -\nabla q(\mathbf{x}_0)$.

Once we have chosen a direction to move in, and this is a direction in which *the value of q decreases, at least for some range of α*, we would like to pick α such that this descent is optimal. However, it is impractical to pick the best possible α.

Backtracking line-search algorithm

Instead, we start with $\alpha = 1$, and check what is known as "sufficient descent" conditions. That is, we check if our function value has gone down by a sufficient amount (Armijo condition), and we check that the slope of the tangent line at the new iterate is smaller in absolute value (Wolfe condition) by a sufficient amount. If so, we accept this α. If not, we reject and cut α by a factor of 0.5. We cut α until we either accept the step or we reach a maximum number of attempts (at which point we accept the tiny step).

In class and in our homework, we see that this implementation of gradient descent (with a backtracking linesearch) reliably converges to a local minimum (or to a root if we are applying this method to rootfinding) *linearly*. The rate can, however, be quite slow (close to 1), and for challenging functions, the iterations might "zig-zag" to the solution.

We include below an introduction to applying other rootfinding methods to smooth optimization:

5.2 Newton method

The full Newton step for the problem of finding a critical point becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}q(\mathbf{x}_k)^{-1}\nabla q(\mathbf{x_k}) \tag{5.2}$$

where $\mathbf{H}q(\mathbf{x}_k)$ is the Hessian matrix. That is, we take a step in the direction p_k where

$$\mathbf{p}_k = -\mathbf{H}q(\mathbf{x}_k)^{-1}\nabla q(\mathbf{x_k}) \tag{5.3}$$

To make Newton more rubust, we implement the exact same backtracking line-search algorithm we used for steepest descent. That is, the Newton step becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{5.4}$$

where α_k is determined by the backtracking line-search to provide sufficient descent. This makes Newton converge from further away. Once our iterates get into the "basin of quadratic convergence", the method generally accepts $\alpha = 1$ and quickly converges to the solution to high accuracy.

5.3 Quasi-Newton methods

There are a number of Quasi-Newton methods specialized to smooth optimization. Broyden will not do very well, and for one key reason: it is not designed to make $\mathbf{B}_k$ an SPD matrix. Even if $\mathbf{B}_0$ is SPD, the updates are not guaranteed to remain SPD. Intuitively: the matrix we are imitating is now the Hessian. If $\mathbf{B}_k$ stops being SPD, the associated quadratic model for $f(\mathbf{x})$ is no longer locally convex, and our algorithm stops descending.

Here are some famous Quasi-Newton methods for optimization. They all involve either rank 1 or rank 2 updates for the Hessian-like matrix $\mathbf{B}_k$ every iteration:

- **BFGS** (Broyden, Fletcher, Goldfarb, Shanno)
- **DFP** (Davidon, Fletcher, Powell)
- **SR1** (Symmetric rank one)

Because they all involve low rank updates, they can all be implemented using exactly the same ideas that we used to implement Broyden, plus of course the same line-search algorithm to improve global convergence.

Out of these, BFGS and DFP are the most competitive; they each have their advantages in terms of performance. There are large-scale implementations of BFGS known as "limited memory BFGS" that tackle enormous optimization problems.